Databricks

- Databricks is a unified analytics platform designed to provide data engineers, data scientists, and analysts with the tools they need to collaborate on large-scale data processing projects.
- Databricks Overview
    - Unified Analytics Platform: Combines data engineering, data science, and business analytics.
    - Built on Apache Spark: Managed environment for running Spark applications.
    - Collaborative Notebooks: Support Python, Scala, SQL, and R for interactive data processing.
    - Delta Lake: Provides ACID transactions for data lakes.
    - MLflow: Manages the end-to-end machine learning lifecycle.
- Key Features
    1. Apache Spark Integration:
        - Distributed computing system.
        - Simplifies large-scale data processing.
    2. Collaborative Notebooks:
        - Interactive, shareable.
        - Supports multiple languages.
    3. Delta Lake:
        - Ensures data reliability and consistency.
        - ACID transactions.
    4. MLflow:
        - Experiment tracking.
        - Model packaging and deployment.
    5. Scalability and Performance:
        - Automatic resource scaling.
        - High performance and cost-efficient.
    6. Integration with Azure and AWS:
        - Managed services: Azure Databricks and Databricks on AWS.
        - Seamless cloud service integration.
- Use Cases
    - Data Engineering: Data pipelines for processing and transforming datasets.
    - Data Science: Exploratory analysis, building ML models, advanced analytics.
    - Business Intelligence: Dashboards, visualizations for decision-making.
    - Real-time Analytics: Processing and analyzing streaming data.
- Advantages
    - Ease of Use: Simplifies big data infrastructure management.
    - Collaboration: Facilitates teamwork across roles.
    - Flexibility: Supports multiple languages and data sources.
    - Performance: Optimized for large-scale processing and ML.
- Disadvantages of Databricks
    1. Cost:
        - Expensive, especially for large-scale deployments.
        - Costs can escalate with cloud platform integration.
    2. Learning Curve:

        - Requires time and effort to master complex workflows.
        - Skill acquisition for effective use.
    3. Vendor Lock-in:
        - Tied to specific cloud providers (Azure, AWS).
        - Limited flexibility in choosing alternatives.
    4. Resource Management:
        - Automatic scaling may lead to unexpected costs.
        - Requires careful monitoring and management.
    5. Performance Tuning:
        - Optimizing for specific use cases can be challenging.
        - Knowledge of Spark configurations needed.
    6. Dependency on Connectivity:
        - Requires internet access for using Databricks notebooks.
        - Limits offline or restricted environment use.
    7. Limited Infrastructure Control:
        - Managed service reduces control over infrastructure.
        - Less customization compared to self-hosted setups.
    8. Support and Customization:
        - Customization may be more constrained.
        - Dependence on vendor support for complex issues.

Data Lakehouse

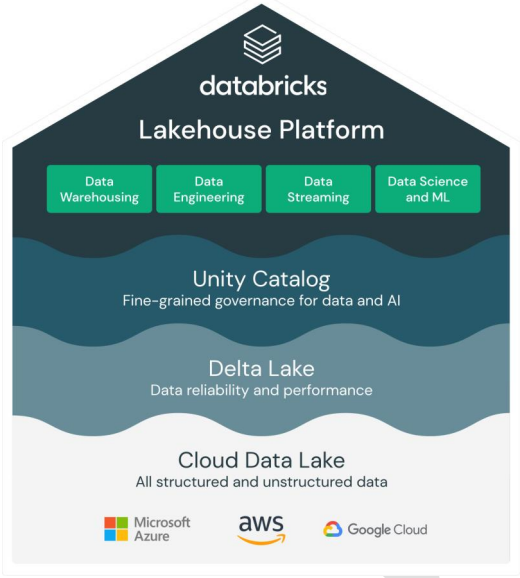
- A Data Lake House is a modern data architecture that combines the best features of both data lakes and data warehouses.
- Data Lake House Overview
    - Integration of Data Lake and Data Warehouse: Combines scalable storage with structured querying.
    - Unified Platform: Stores and analyzes structured and unstructured data.
    - Supports Diverse Workloads: Batch processing, interactive queries, ML, and real-time analytics.
- Key Features
    - Scalable Storage: Utilizes data lake for cost-effective storage of raw and processed data.
    - Schema Enforcement: Enforces schemas for efficient querying.
    - Data Quality and Governance: Manages data quality and ensures governance.
    - Supports Multiple Data Types: Handles structured, semi-structured, and unstructured data.
    - Advanced Analytics: Enables machine learning and AI with flexible data processing.
- Advantages
    - Flexibility: Adapts to various data types and analytics tasks.
    - Scalability: Scales horizontally for large data volumes.
    - Cost Efficiency: Uses cloud storage for economical data management.
- Performance: Combines fast query processing with complex analytics capabilities.

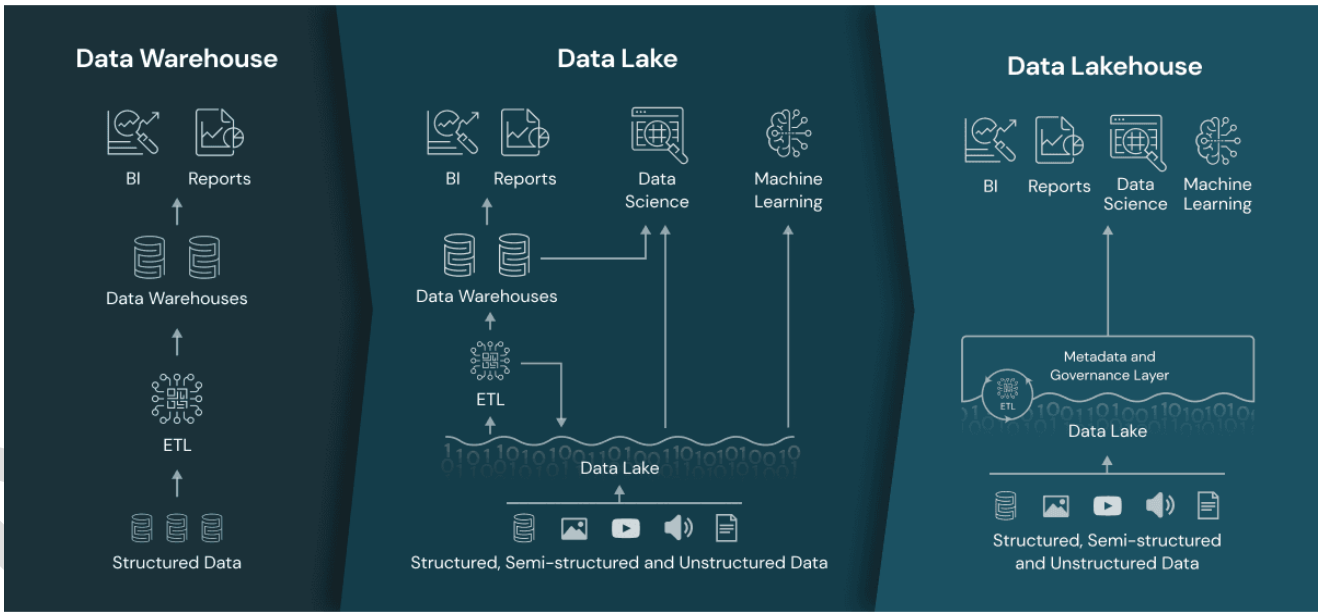
- Use Cases
    - Enterprise Data Lakes: Enhances existing data lake environments.
    - Real-time Analytics: Supports immediate data ingestion and analysis.
    - Data Science and ML: Facilitates data exploration and model deployment.
- Example Technologies
    - Delta Lake: Ensures ACID transactions for data consistency.
    - Apache Spark: Powers data processing and analytics in Data Lake House architectures.

Data Lakehouse Vs Data Warehouse Vs Data Lake

Feature	Data Lakehouse	Data Warehouse	Data Lake
Primary Purpose	Unified platform for storing and analyzing structured and unstructured data. Combines features of both data lake and warehouse.	Optimized for structured data storage and fast query performance.	Centralized repository for storing raw and structured data in its native format.
Storage Model	Utilizes scalable cloud storage (data lake) with schema enforcement and indexing (warehouse).	Centralized structured storage optimized for fast read/write operations.	Stores data in its native format (often in files or object storage) without enforcing a schema.
Schema Enforcement	Yes, supports schema enforcement for efficient querying and data governance.	Yes, strictly enforces schemas to optimize query performance.	No strict schema enforcement; supports schema-on-read approach.
Data Types	Supports structured, semi-structured, and unstructured data types.	Primarily structured data, less support for unstructured data.	Supports a wide range of data types including structured, semi-structured, and unstructured data.
Query Performance	Efficient query processing with scalable storage capabilities.	Highly optimized for complex queries and analytics.	Generally slower query performance due to schema-on-read and less indexing.
Analytics	Supports advanced analytics including machine learning and AI.	Designed for business intelligence, reporting, and complex analytics.	Used for exploratory data analysis, big data processing, and batch analytics.
Use Cases	Real-time analytics, data science, machine learning, and diverse analytics workloads.	Business intelligence, reporting, ad-hoc queries, and analytics.	Big data processing, IoT data storage, data lake analytics, and data exploration.
Technology Examples	Delta Lake, Apache Spark.	Amazon Redshift, Google BigQuery, Snowflake.	Hadoop, Apache HBase, Amazon S3, Azure Data Lake Store.

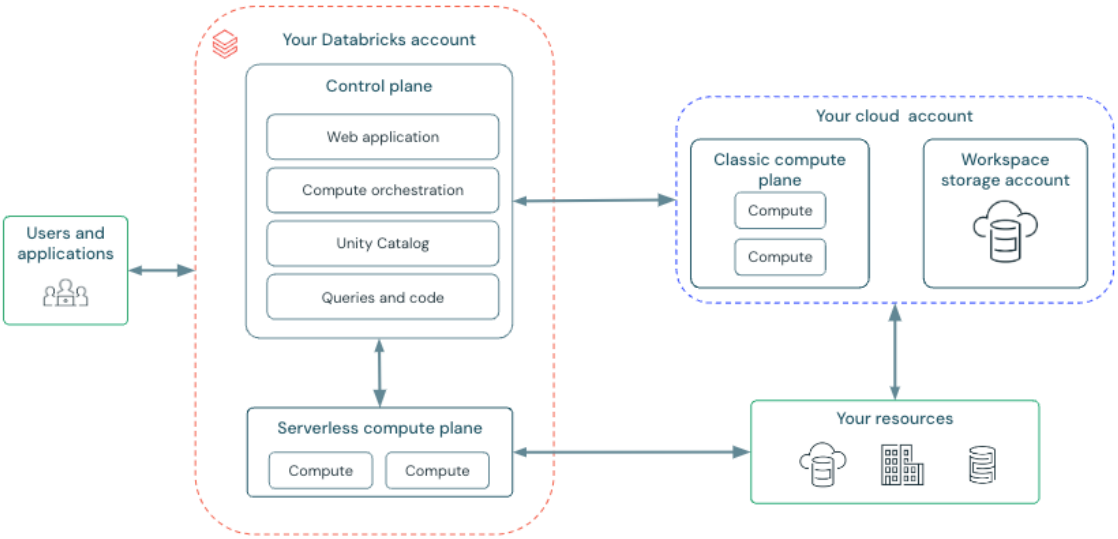
Flexibility	Flexible schema and data types support.	Limited flexibility in schema changes and data types.	Highly flexible in terms of data types and schema evolution.
Cost Efficiency	Cost-effective storage with cloud scalability.	Higher cost due to optimized performance and structured storage.	Cost-effective for storing large volumes of raw data, but may incur costs for processing and querying.
Example Platforms	Databricks (Delta Lakehouse), AWS Glue.	Amazon Redshift, Google BigQuery, Snowflake.	Hadoop ecosystem (HDFS, Hive), Azure Data Lake.





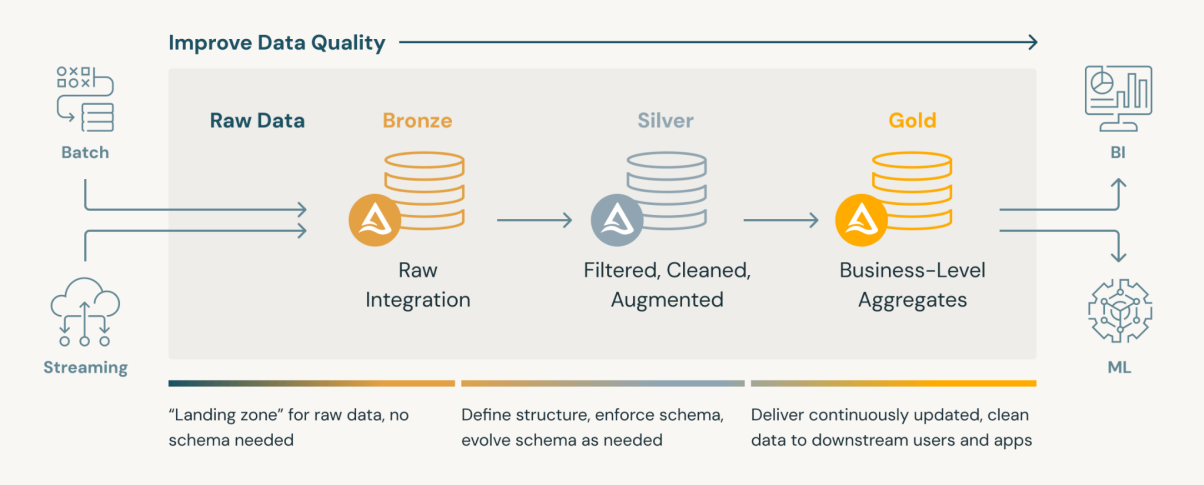
Azure Databricks Architecture

- Azure Databricks is a managed Apache Spark-based platform optimized for cloud-based big data processing and analytics.
- Components:
    1. Workspace:
        - Manages environment settings, user access, and collaboration using notebooks.
        - Central hub for data engineers, data scientists, and analysts.
    2. Clusters:
        - Managed compute resources for Apache Spark.
        - Auto-scaling based on workload demands.
        - Executes Spark jobs and tasks.
    3. Notebooks:
        - Interactive environment for code development and data exploration.
        - Supports Python, Scala, SQL, R, and more.
        - Version control and collaboration features.
    4. Jobs:
        - Scheduled execution of notebooks or scripts.
        - Integration with ETL workflows and batch processing.
    5. Libraries:
        - Installation of additional libraries for custom data processing.
        - Integrates with machine learning frameworks like TensorFlow and scikit-learn.
    6. Data Sources:
        - Connects to Azure Storage, Azure SQL Database, Cosmos DB, etc.
        - Optimized data access using connectors and Delta Lake for efficient processing.
    7. Security:
        - Azure Active Directory integration for authentication.
        - Data encryption in transit and at rest.
    8. Integration with Azure Services:
        - Azure ML for model deployment and management.
        - Azure Synapse Analytics for data warehousing.
- Advantages
    - Scalability: Automatic scaling of compute resources.
    - Performance: Optimized for large-scale data processing with Spark.
    - Collaboration: Shared notebooks and version control.
    - Integration: Seamless integration with Azure services.
- Use Cases
    - Data Engineering: ETL processes, data pipelines.
    - Data Science: Machine learning model development.
    - Business Intelligence: Interactive querying, reporting.



Lakehouse Medallion Architecture

- The Lakehouse Medallion Architecture is a data management and processing architecture that combines the best features of data lakes and data warehouses. It organizes data into three layers or "medallions" - bronze, silver, and gold - to progressively refine data quality and structure, facilitating efficient analytics and machine learning.



- Key Features:

Feature	Description
Unified Data Governance	Centralized governance for data assets, managing access controls and data policies through a single interface.
Fine-Grained Access Control	Role-based access control (RBAC) at table, view, and column levels, ensuring data security and compliance.

Auditing and Compliance	Comprehensive auditing to track data access and modifications, ensuring regulatory compliance.
Data Lineage	Tracks the origin and movement of data, providing a visual representation of data flow.
Collaboration	Facilitates collaboration among data engineers, data scientists, and analysts by providing a unified view of data assets.
Scalability	Designed to handle large-scale data environments with high performance, supporting multi-cloud and hybrid architectures.

- Layers of the Medallion Architecture:

Attribute	Bronze Layer	Silver Layer	Gold Layer
Purpose	Store raw, unprocessed data	Store cleaned, refined, partially processed data	Store fully processed, aggregated, ready-for-use data
Characteristics	- Ingested from various sources - Contains raw, unvalidated, and potentially noisy data	- Data is transformed and cleaned - Standardized and enriched with additional context	- Data is fully processed and aggregated - Optimized for reporting, BI, and advanced analytics
Usage	- Historical data storage - Source for initial data validation and cleansing	- Intermediate storage for analytics and reporting - Source for more refined data processing and transformation	- End-user reporting and dashboarding - Source for machine learning models and decision-making processes
Example	- Raw log files - Sensor data - Transactional records	- Cleaned customer records - Deduplicated event logs	- Monthly sales reports - Customer segmentation data

- Benefits of Lakehouse Medallion Architecture:

Benefit	Description
Scalability	Efficiently handles large volumes of data leveraging data lake scalability.
Flexibility	Supports various data types (structured, semi-structured, unstructured) and integrates with different data sources.
Cost Efficiency	Reduces storage costs by keeping raw data in cheaper storage layers and processing data incrementally.
Improved Data Quality	Progressive data refinement ensures higher quality data in the gold layer.
Enhanced Analytics	Structured approach to data transformation and analysis optimizes data layers for efficient querying.

Unified Data Management	Combines benefits of data lakes and data warehouses, simplifying data governance and management.

- Implementation Example:

Step	Description
Data Ingestion	Raw data from various sources is ingested into the bronze layer using ETL/ELT processes.
Data Transformation	Data from the bronze layer is cleaned, standardized, and enriched, then moved to the silver layer.
Data Aggregation	Data from the silver layer is aggregated, processed, and optimized for specific use cases, then moved to the gold layer.
Data Consumption	End-users and applications access the gold layer for reporting, analytics, and machine learning.

- Summary:
    - The Lakehouse Medallion Architecture effectively organizes and processes data in a layered approach, leveraging the scalability of data lakes and the performance of data warehouses. It ensures high data quality, cost efficiency, and enhanced analytics, making it a powerful architecture for modern data management and processing.

Databricks Cluster

- A Databricks cluster is a fundamental component of the Databricks platform, designed to execute data processing tasks using Apache Spark.
- Databricks Cluster
    1. Definition and Purpose:
        - A Databricks cluster is a managed set of virtual machines (VMs) in the cloud that runs Apache Spark. It provides computational resources for processing large-scale data and executing distributed computing tasks.
        - Databricks clusters are scalable and can automatically adjust their size (scale up or down) based on workload demands, ensuring efficient resource utilization.
    2. Components:
        - Driver Node: Controls the execution of the Spark application and coordinates tasks across worker nodes.
        - Worker Nodes: Execute tasks in parallel, processing data stored in distributed storage systems (e.g., Azure Blob Storage, AWS S3).
    3. Key Features:
        - Automatic Scaling: Databricks clusters can automatically scale the number of worker nodes based on the workload. This elasticity helps optimize performance and manage costs.

        - Instance Types: Users can choose different instance types for driver and worker nodes, depending on their computational and memory requirements.
        - Cluster Policies: Define policies for auto-scaling, instance types, and other configurations to tailor clusters for specific tasks or workloads.
    4. Cluster Modes:
        - Standard Mode: Each notebook attached to the cluster has its own driver node, providing isolation and control over resources.
        - High Concurrency Mode: Multiple notebooks share a single pool of resources, optimizing resource usage for environments with many users.
    5. Cluster Libraries:
        - Users can install additional libraries (e.g., Python packages, JAR files) on Databricks clusters. These libraries extend the functionality of Apache Spark and enable custom data processing and analytics tasks.
    6. Lifecycle:
        - Creation: Users can create clusters through the Databricks UI, APIs, or programmatically.
        - Management: Databricks manages the cluster lifecycle, including provisioning, scaling, and termination, to simplify operations for users.
    7. Integration:
        - Databricks clusters integrate seamlessly with other Databricks features such as notebooks, jobs, and data sources. They also support integration with cloud storage services and databases for data ingestion and processing.
- Use Cases
    - Data Engineering: Running ETL (Extract, Transform, Load) jobs, data cleansing, and transformation tasks.
    - Data Science: Training machine learning models, performing exploratory data analysis, and running statistical computations.
    - Business Intelligence: Executing SQL queries for interactive data analysis and generating reports.
- Advantages
    - Scalability: Automatically adjusts cluster size based on workload demands.
    - Performance: Harnesses parallel processing capabilities of Apache Spark for high-speed data processing.
    - Ease of Use: Simplifies cluster management with automated provisioning and configuration.
    - Flexibility: Supports multiple instance types and configurations to meet varying computational requirements.

Compute Options in Azure Databricks

Compute Option	Definition & Features	Use Cases
All-purpose Compute	General-purpose clusters for interactive data analytics, exploration, and development. - Suitable for notebooks. - Flexible library installations. - Autoscaling.	- Interactive data analysis - ML model development - Ad-hoc querying
Job Compute	Clusters for scheduled jobs, ETL processes, and batch tasks. - Auto-creation and termination. - Optimizes resource usage. - Configurable.	- Scheduled ETL pipelines - Batch processing - Automated workflows
SQL Warehouses	Compute for running SQL queries and BI workloads. - High concurrency and performance. - Photon engine for fast execution. - BI tool integration.	- Interactive dashboards - Ad-hoc SQL querying - Real-time analytics
Vector Search	Specialized compute for high-performance vector similarity searches. - Optimized for vector data and similarity computations. - Supports large-scale AI/ML inference.	- Image/text similarity searches - Recommendation systems - AI model inference
Pools	Pre-configured clusters of VMs to reduce start-up time and manage costs. - Warm pool of reusable instances. - Reduces cluster start-up latency. - Cost optimization.	- Frequent cluster start-ups - Development/testing environments - Reducing start-up time
Policies	Governance and control settings for cluster creation and management. - Enforces best practices. - Limits cluster sizes and configurations. - Ensures compliance.	- Enterprise governance - Cost management - Standardizing configurations

Configuration Options for All-purpose Compute Clusters

Option Category	Configuration Option	Description
Cluster Basics	Cluster Name	A user-friendly name for the cluster.
	Cluster Mode	Select between Standard and High Concurrency modes.
	Databricks Runtime Version	Choose the runtime version, including standard, ML, and GPU versions.
	Spark Version	Select the version of Apache Spark to run on the cluster.
Cluster Configuration	Worker Type	Select the VM instance type for worker nodes.
	Driver Type	Select the VM instance type for the driver node.
	Min Workers	Set the minimum number of worker nodes (for autoscaling).
	Max Workers	Set the maximum number of worker nodes (for autoscaling).
	Auto Termination	Automatically terminate the cluster after a specified period of inactivity.
	Enable Autoscaling	Enable or disable auto scaling of the cluster based on workload.
Advanced Options	Spot Instances	Use spot instances to reduce costs (available in some regions).
	Driver and Worker Tags	Add tags to the driver and worker nodes for identification and management.
	Custom Tags	Add custom tags for resource organization and cost management.
	Init Scripts	Specify scripts to run on cluster startup for custom initialization.
Security Options	Cluster Log Path	Define a path for storing cluster logs.
	Enable Secure Cluster Connectivity	Enable secure communication between the cluster and Azure Databricks.
	Credential Passthrough	Enable Azure Active Directory credential passthrough for accessing data.
Libraries	Library Installation	Install additional libraries (e.g., Python packages, JAR files) to extend cluster functionality.
Permissions	Cluster Owner	Specify the users or groups who can manage the cluster.
	Access Control	Set permissions for who can attach notebooks, run jobs, and manage the cluster.

Configuration Options for Job Compute

Option Category	Configuration Option	Description
Cluster Basics	Cluster Name	A user-friendly name for the cluster.
	Cluster Mode	Select the mode for the cluster (Standard or High Concurrency).
	Databricks Runtime Version	Choose the runtime version, including standard, ML, and GPU versions.
	Spark Version	Select the version of Apache Spark to run on the cluster.
Cluster Configuration	Worker Type	Select the VM instance type for worker nodes.
	Driver Type	Select the VM instance type for the driver node.
	Min Workers	Set the minimum number of worker nodes (for autoscaling).
	Max Workers	Set the maximum number of worker nodes (for autoscaling).
	Enable Autoscaling	Enable or disable auto scaling of the cluster based on workload.
	Auto Termination	Automatically terminate the cluster after a specified period of inactivity.
Advanced Options	Spot Instances	Use spot instances to reduce costs (availability varies by region).
	Init Scripts	Specify scripts to run on cluster startup for custom initialization.
	Custom Tags	Add custom tags for resource organization and cost management.
	Driver and Worker Tags	Add tags to the driver and worker nodes for identification and management.
Security Options	Cluster Log Path	Define a path for storing cluster logs.
	Enable Secure Cluster Connectivity	Enable secure communication between the cluster and Azure Databricks.
	Credential Passthrough	Enable Azure Active Directory credential passthrough for accessing data.
Libraries	Library Installation	Install additional libraries (e.g., Python packages, JAR files) to extend cluster functionality.
Permissions	Cluster Owner	Specify the users or groups who can manage the cluster.
	Access Control	Set permissions for who can attach notebooks, run jobs, and manage the cluster.
Job Configuration	Schedule	Define the schedule for the job (e.g., daily, hourly).
	Job Tasks	Specify the tasks to be performed by the job,

		including task dependencies.
	Job Libraries	Libraries specific to the job that need to be installed.
	Cluster Specification	Choose a pre-defined cluster specification for the job, if applicable.
Notifications	Success Notifications	Configure notifications for job success (e.g., email, webhook).
	Failure Notifications	Configure notifications for job failure (e.g., email, webhook)

Notebooks in Azure Databricks

- In Azure Databricks (ADB), a notebook is an interactive computing environment that enables data engineers, data scientists, and analysts to perform data exploration, data analysis, machine learning model development, and collaboration.
- Interactive computing environments for data exploration, analysis, and collaboration.
- Features:
    - Multi-language Support: Python, SQL, R, Scala, Markdown.
    - Cells: Code, Markdown, visualizations.
    - Interactive Execution: Immediate feedback on code execution.
    - Visualization: Integration with libraries like Matplotlib, Seaborn.
    - Markdown: Formatted text, headers, links, images.
    - Collaboration: Real-time sharing and editing.
    - Version Control: Track changes, revert to previous versions.
- Use Cases:
    - Data exploration, statistical analysis.
    - Machine learning model development.
    - Data visualization and reporting.
    - Collaborative projects and team workflows.
    - Scheduled jobs for automation.
- Best Practices:
    - Modularize code for readability.
    - Document with Markdown for clarity.
    - Reuse code and functions.
    - Version notebooks for history and recovery.
    - Optimize performance with cluster settings.
    - Ensure security and compliance with access controls.
- Integration: Part of Databricks Workspace, managed alongside clusters and jobs.
- Benefits: Enhances productivity, supports iterative development, facilitates team collaboration, and enables efficient data analysis and reporting.

Magic Commands

Command	Description
%sql	Executes SQL queries on Spark SQL.
%python	Executes Python code.
%scala	Executes Scala code.
%r	Executes R code.
%md	Renders Markdown text.
%fs	Accesses the Databricks File System (DBFS) for file operations.
%sh	Executes shell commands.
%run	Runs another notebook.
%pip	Installs Python packages.

%classpath	Manages Java/Scala classpaths for libraries.
%spark	Executes code with specific Spark configurations.
%sqlContext	Accesses the SQL context for running SQL queries.
%config	Configures notebook settings and parameters.
%lsmagic	Lists all available magic commands.
%%sql	Executes multi-line SQL queries.
%%python	Executes multi-line Python code.

Mount

- Mounting in Azure Databricks refers to connecting external storage systems (like Azure Blob Storage, Azure Data Lake Storage, or other cloud storage services) to Databricks' File System (DBFS).
- Purpose:
  - Seamless access to data stored in external cloud storage systems.
  - Simplified data access and management.
  - Reduced need to handle authentication and connection details repeatedly.
  - Integration with various storage services without complex configurations.
- Benefits:
  - Simplifies data access and management.
  - Supports integration with various storage services.
  - Provides familiar file system paths in DBFS.
- Supported Storage Services:
  - Azure Blob Storage
  - Azure Data Lake Storage (ADLS) Gen1 and Gen2
  - Amazon S3 (with appropriate credentials)
- Mount Point:
  - A directory in DBFS that maps to the external storage location (e.g., /mnt/my-mount).

Code

- Mounting Azure Data Lake Storage (ADLS) Gen2

```
configs = {"fs.azure.account.auth.type": "OAuth",
           "fs.azure.account.oauth.provider.type":
"org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
           "fs.azure.account.oauth2.client.id":
dbutils.secrets.get(scope = "<scope-name>", key = "<client-id>"),
           "fs.azure.account.oauth2.client.secret":
dbutils.secrets.get(scope = "<scope-name>", key = "<client-secret>"),
```

```
          "fs.azure.account.oauth2.client.endpoint":
"https://login.microsoftonline.com/<tenant-id>/oauth2/token"}

dbutils.fs.mount(
  source = "abfss://<file-system-name>@<storage-account-
name>.dfs.core.windows.net/",
  mount_point = "/mnt/<mount-name>",
  extra_configs = configs
)
```

- Accessing Mounted Storage
    - List files in the mount point:

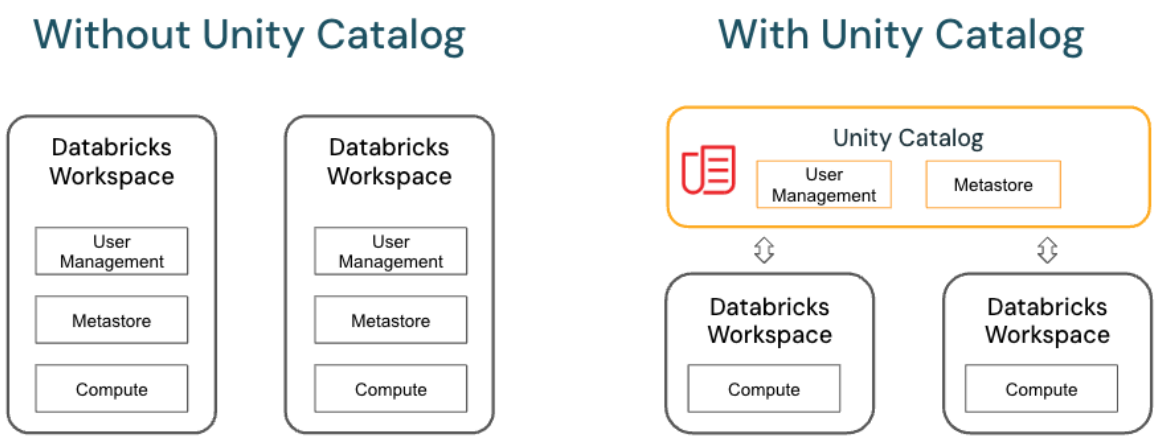
```
display(dbutils.fs.ls("/mnt/<mount-name>"))
```

- Unmounting Storage
    - Unmount storage:

```
dbutils.fs.unmount("/mnt/<mount-name>")
```

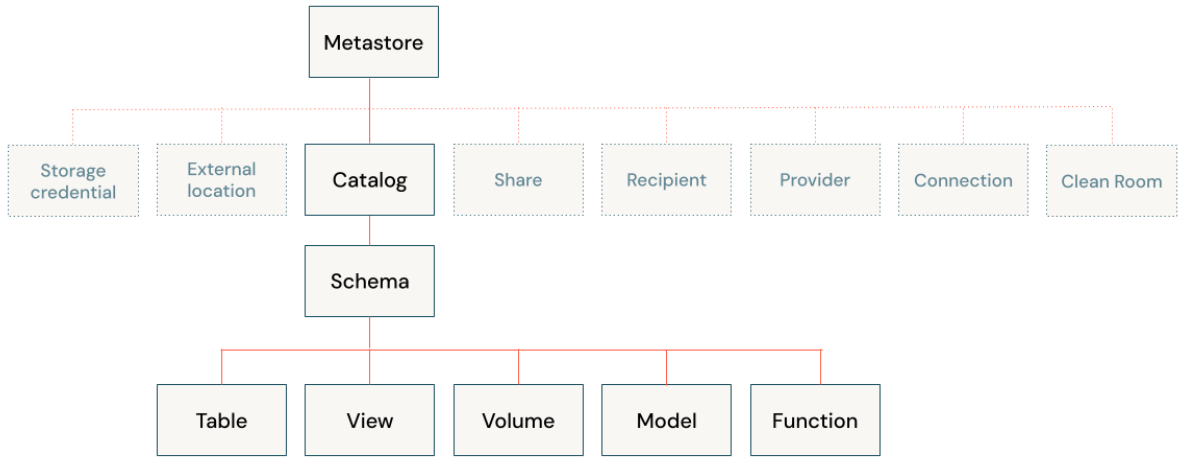
Unity Catalog

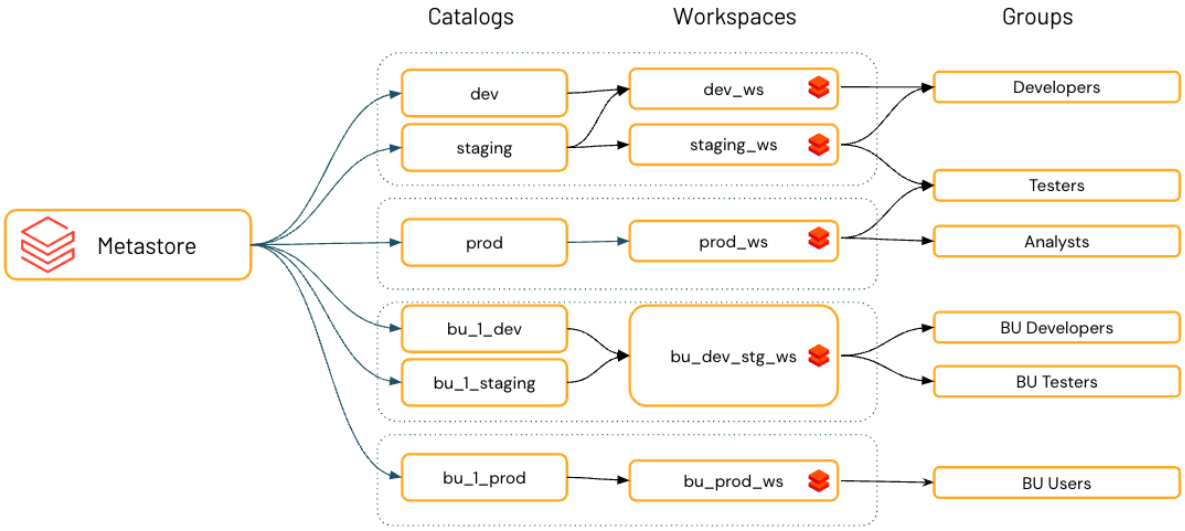
- Unity Catalog is a unified data governance solution in Azure Databricks that provides centralized management for data governance, security, and compliance across all data assets within Databricks.



- Key Features:
    1. Centralized Data Governance:
        - Single interface for managing access controls and policies.

        - Role-Based Access Control (RBAC) for fine-grained security at table, view, and column levels.
        - Global and local permissions for controlling access across different levels.
    2. Data Lineage:
        - Automatic tracking of data origin, movement, and transformations.
        - Visual lineage graph for understanding data flow and dependencies.
    3. Auditing and Compliance:
        - Detailed audit logs for tracking data access and actions.
        - Compliance reporting to meet regulatory requirements (e.g., GDPR, CCPA, HIPAA).
    4. Collaboration:
        - Unified view of data assets to facilitate collaboration among teams.
        - Simplified data discovery and sharing with governance policies in place.
    5. Scalability:
        - Designed for large-scale data environments with support for multi-cloud and hybrid architectures.
        - Seamless integration with other Databricks features, such as Delta Lake.
- Use Cases:
    1. Data Security and Compliance:
        - Ensures that only authorized users access sensitive data, with detailed logging for auditing.
    2. Data Lineage Tracking:
        - Provides full visibility into the lifecycle of data, from ingestion to processing to consumption.
    3. Collaboration Across Teams:
        - Enables secure collaboration among data engineers, scientists, and analysts.
    4. Streamlining Data Management:
        - Reduces administrative overhead by centralizing data governance and access management.
- Summary:
    - Unity Catalog in Azure Databricks centralizes data governance, security, and compliance management, enabling organizations to securely manage, audit, and collaborate on data across large-scale environments.
- Unity Catalog object model
    - In Unity Catalog, all metadata is registered in a metastore.
    - The hierarchy of database objects in any Unity Catalog metastore is divided into three levels, represented as a three-level namespace (*catalog.schema.table-etc*) when you reference tables, views, volumes, models, and functions.





Metastore

- The Metastore in Azure Databricks is a central repository that stores metadata about data objects like tables, columns, and schemas. It helps in managing and organizing data efficiently within the Databricks environment.
- Key Functions:
    1. Metadata Management:
        - Stores metadata, including table structures, column data types, and relationships.
        - Tracks the location of data files, enabling seamless data management.
    2. Schema Management:
        - Manages the creation, alteration, and deletion of schemas and tables.
        - Ensures data consistency through schema enforcement.
    3. Data Governance:
        - Works with Unity Catalog to enforce access controls and data lineage.

        - Helps maintain compliance by managing user permissions.
    4. Query Optimization:
        - Stores statistics and metadata used by the query engine for performance optimization.
        - Facilitates efficient data retrieval by providing insights on data distribution and partitioning.
    5. Integration with Hive Metastore:
        - Supports integration with an external Apache Hive metastore for compatibility with existing Hive-based systems.
        - Enables interoperability and migration between Databricks and traditional big data environments.
- Types of Metastore:
    1. Databricks Managed Metastore:
        - Default, fully managed by Databricks.
        - Ideal for users who prefer an out-of-the-box solution without managing infrastructure.
    2. External (Hive) Metastore:
        - Allows connection to an external Hive metastore for users who have existing Hive metadata or prefer self-management.
        - Provides more control but requires additional maintenance.
- Summary:
    - The Metastore in Azure Databricks is essential for metadata management, schema enforcement, query optimization, and data governance. It can be either managed by Databricks or integrated with an external Hive metastore, offering flexibility and control over metadata.

Metastore Components

1. Volumes
    - Volumes are logical containers or storage units for managing and accessing data within Databricks.
    - Purpose: They provide a way to manage and organize data storage locations and can be used to access data stored in various formats.
    - Usage: Volumes are typically used to abstract and manage storage locations, enabling easier data management and access.
2. Tables
    - Tables are collections of structured data organized into rows and columns. They represent the fundamental data objects in the metastore.
    - Purpose: Store and manage data for querying and analysis.
    - Types:
        i. Managed Tables: Data and metadata are managed by Databricks. Dropping the table removes both data and metadata.
        ii. External Tables: Metadata is managed by Databricks, but data resides in an external storage location. Dropping the table removes only the metadata.
    - Usage: Used to store large volumes of data and perform queries and transformations.

3. Views
    - Views are virtual tables created by querying one or more existing tables.
    - Purpose: Provide a simplified or customized representation of data without duplicating it.
    - Types:
        i. Temporary Views: Exist only within the session or notebook and are not persisted beyond that scope.
        ii. Permanent Views: Persist in the metastore and can be accessed across different sessions or notebooks.
    - Usage: Useful for abstracting complex queries and presenting data in a specific format.
4. Functions
    - Functions are user-defined routines that perform specific operations or transformations on data.
    - Purpose: Extend SQL capabilities by allowing custom operations and transformations.
    - Types:
        i. User-Defined Functions (UDFs): Allow users to define custom functions that can be used in SQL queries or data transformations.
        ii. Built-in Functions: Predefined functions provided by Databricks or SQL that perform common operations.
    - Usage: Enhance data processing by enabling custom logic and operations.
5. Models
    - Models refer to machine learning models that are stored and managed within the Databricks environment.
    - Purpose: Facilitate the deployment, management, and serving of machine learning models.
    - Usage: Used for applying machine learning algorithms to data, making predictions, and integrating model results into data pipelines.

Component	Definition	Purpose	Usage
Volumes	Logical containers for managing storage locations	Manage and organize data storage	Access and manage data in various formats
Tables	Collections of structured data (rows and columns)	Store and manage data	Perform queries and transformations on data
Views	Virtual tables created by querying existing tables	Simplify or customize data representation	Abstract complex queries and present data
Functions	User-defined routines for data operations	Extend SQL capabilities with custom operations	Enhance data processing and transformations
Models	Machine learning models stored in Databricks	Deploy, manage, and serve machine learning models	Apply algorithms to data, make predictions, and integrate model results

Metastore Components Related to Data Access and Sharing

1. Storage Credentials
    - Authentication credentials used to access data stored in external storage systems, such as cloud storage (e.g., Azure Blob Storage, AWS S3).
    - Purpose: Facilitate secure access to external storage locations where data is stored.
    - Usage: Storage credentials are configured to enable Databricks to read from or write to external storage systems. These credentials include keys, tokens, or other authentication methods.
    - Example:
        i. Azure Storage Credential: Typically involves a storage account key or SAS token for accessing Azure Blob Storage.
2. External Locations
    - References to external storage locations where data is stored.
    - Purpose: Define the paths to external data storage systems, allowing Databricks to interact with data stored outside of its managed environment.
    - Usage: External locations are configured to map to specific paths in external storage systems. This setup enables Databricks to perform operations on data stored externally.
    - Example:
        i. External Location Path:

```
abfss://my-container@my-storage-account.dfs.core.windows.net/path/to/data/
```

3. Connections
    - Network and data connections used to integrate Databricks with external systems and databases.
    - Purpose: Establish secure communication between Databricks and external data sources or systems.
    - Usage: Connections may include configurations for JDBC, ODBC, or API-based integrations with external databases or services.
    - Example:
        i. JDBC Connection: Configuration for connecting Databricks to an external relational database using JDBC.
4. Clean Rooms
    - Secure environments that facilitate data sharing and collaboration without exposing sensitive information.
    - Purpose: Allow multiple parties to collaborate on data analysis and processing in a controlled and secure manner.
    - Usage: Clean rooms are used for data collaboration while maintaining strict privacy and security controls.
    - Example:
        i. Secure Data Sharing: Collaborating on data analysis projects while ensuring data privacy and compliance.
5. Shares

- Mechanisms for sharing data between Databricks workspaces or with external organizations.
- Purpose: Enable data sharing and collaboration across different environments or with external partners.
- Usage: Shares define the data access permissions and the scope of data being shared.
- Example:
    i. Data Share: Configuring a share to provide access to specific datasets with another Databricks workspace or external organization.

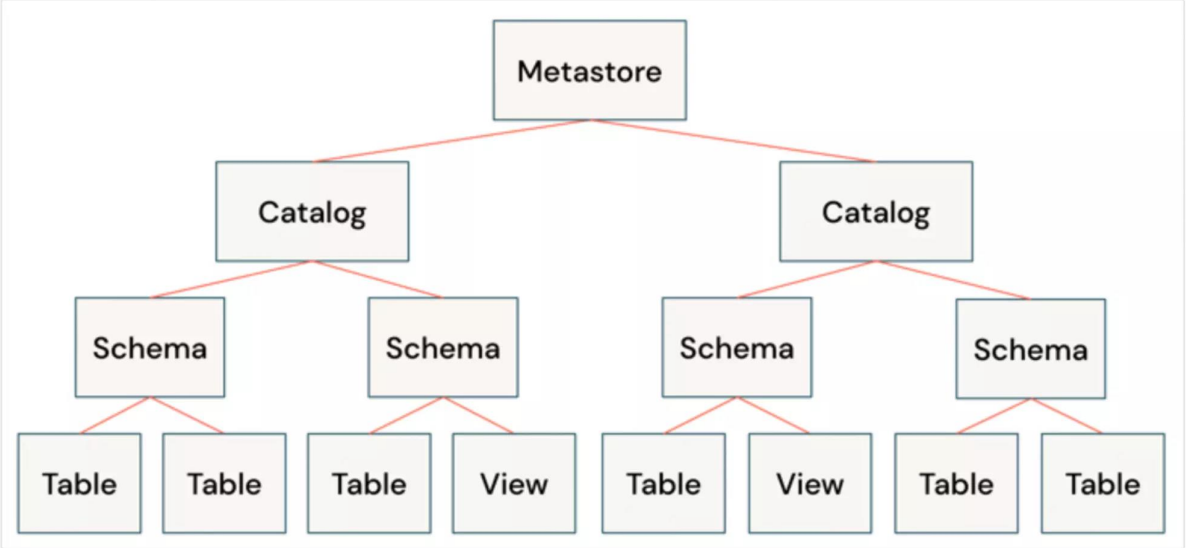
6. Recipients
    - Entities or users that receive shared data.
    - Purpose: Manage and control who has access to the data that is shared from Databricks.
    - Usage: Recipients are specified when configuring data shares to grant access to data.
    - Example:
        i. Recipient Configuration: Specifying a Databricks workspace or external user who will receive shared data.
7. Providers
    - Data sources or services that provide data to Databricks.
    - Purpose: Define and manage the data sources from which Databricks can consume data.
    - Usage: Providers include external databases, data lakes, and other data services integrated with Databricks.
    - Example:
        i. Data Provider: An external database or cloud data service that supplies data to be used within Databricks.

Component	Definition	Purpose	Usage
Storage Credentials	Authentication credentials for external storage systems	Secure access to external data storage	Configure access to external storage locations
External Locations	References to external storage paths	Define paths to external data	Map paths to interact with data in external storage
Connections	Network and data connections	Integrate Databricks with external systems	Configure integrations like JDBC, ODBC
Clean Rooms	Secure collaboration environments	Facilitate data sharing and collaboration securely	Collaborate on data analysis while maintaining privacy
Shares	Mechanisms for sharing data	Enable data sharing and collaboration	Configure data access permissions for sharing
Recipients	Entities receiving shared data	Manage access to shared data	Specify who receives shared data
Providers	Data sources or services	Define sources of data for Databricks	Integrate with external data sources

Object hierarchy in the metastore

- The metastore in Azure Databricks organizes metadata using a hierarchical structure. Understanding this hierarchy is crucial for managing data effectively and ensuring proper governance.
- The main components of this hierarchy include:
    1. Catalogs
        - The top-level container for managing databases and tables.
        - Purpose: Acts as a namespace for organizing and categorizing databases.
        - Example: default, my_catalog.
    2. Databases (Schemas)
        - Containers within catalogs that organize tables and views.
        - Purpose: Helps in structuring and managing tables and views within a specific namespace.
        - Example: sales, marketing.
    3. Tables
        - Collections of data organized in rows and columns.
        - Purpose: Stores the actual data and schema information.
        - Types:
        - Managed Tables: Fully managed by Databricks, where both data and metadata are controlled by Databricks.
        - External Tables: Metadata is managed by Databricks, but data is stored in an external location (e.g., cloud storage).
        - Example: customers, transactions.
    4. Views
        - Virtual tables created by querying one or more tables.
        - Purpose: Provides a way to simplify complex queries and present data in a specific format.
        - Types:
        - Temporary Views: Exists only during the session or notebook runtime.
        - Permanent Views: Stored in the metastore and persist beyond session or notebook scope.
        - Example: monthly_sales_summary, top_customers.
    5. Functions
        - User-defined functions (UDFs) that can be used in queries to perform operations.
        - Purpose: Extends SQL capabilities by allowing custom operations and transformations.
        - Example: calculate_age, format_date.
    6. Indexes
        - Structures that improve query performance by allowing faster data retrieval.
        - Purpose: Optimizes search and retrieval operations.
        - Example: Indexes on frequently queried columns.
- Summary

1. Catalogs are the highest level, organizing databases and tables.
2. Databases group tables and views under a specific namespace.
3. Tables hold the actual data and can be managed or external.
4. Views offer virtual tables for simplified querying.
5. Functions extend SQL capabilities with custom operations.
6. Indexes improve query performance by optimizing data retrieval.



Managed Tables & External Tables

- Managed Tables:
    - Tables where both the data and metadata are managed by Databricks.
    - Storage Location: Data is stored in Databricks-managed storage.
    - Lifecycle Management: Databricks manages both data and metadata. Dropping a table deletes both.
    - Schema Management: Enforced and managed by Databricks.
    - Data Management: Handled by Databricks (e.g., backups, optimization).
    - Access Control: Managed by Databricks and integrated with Unity Catalog.
    - Example

```
CREATE TABLE my_managed_table (
  id INT,
  name STRING
) USING DELTA;
```

- External Tables:
    - Tables where metadata is managed by Databricks, but data is stored externally.
    - Storage Location: Data is stored in external storage systems like cloud storage.

- Lifecycle Management: Databricks manages only metadata. Dropping a table does not affect the external data.
- Schema Management: Users manage schema consistency and evolution.
- Data Management: Users are responsible for managing data in external locations.
- Access Control: Managed through Unity Catalog or other governance solutions.
- Example

```
CREATE TABLE my_external_table (
  id INT,
  name STRING
) USING DELTA
LOCATION 'abfss://my-container@my-storage-
account.dfs.core.windows.net/path/to/data/';
```

Aspect	Managed Tables	External Tables
Definition	Data and metadata managed by Databricks	Metadata managed by Databricks; data stored externally
Storage Location	Databricks-managed storage	External storage (e.g., cloud storage)
Lifecycle Management	Databricks manages both data and metadata	Databricks manages metadata only; users manage data
Schema Management	Managed by Databricks	Users manage schema consistency and evolution
Data Management	Managed by Databricks (e.g., backups, optimization)	Managed by users (e.g., storage, backups)
Access Control	Managed by Databricks, integrated with Unity Catalog	Managed through Unity Catalog or other solutions

---

Delta Lake

- Delta Lake is an open-source storage layer that enhances data lakes with features for improved reliability, performance, and management. It integrates seamlessly with Apache Spark and Databricks, and can also be used with other data processing engines.
- Key Features
    1. ACID Transactions
        - Definition: Ensures atomicity, consistency, isolation, and durability of transactions.
        - Purpose: Guarantees reliable and consistent operations, preventing data corruption.
        - Usage: Supports concurrent reads and writes, maintaining data integrity.
    2. Scalable Metadata Handling
        - Definition: Efficient management of metadata, which includes details about data files and schema.
        - Purpose: Handles large metadata volumes effectively, supporting high-performance queries.
        - Usage: Enables fast querying and modification of large datasets.
    3. Data Versioning
        - Definition: Allows access to and rollback of previous versions of data.
        - Purpose: Provides historical data access and the ability to revert changes.
        - Usage: Maintains a transaction log for time travel queries and historical data retrieval.
    4. Schema Enforcement and Evolution
        - Definition: Ensures data adheres to a specified schema and supports schema changes.
        - Purpose: Maintains data quality and allows schema modifications without disruption.
        - Usage: Enforces schema consistency and allows flexible schema evolution.
    5. Data Compaction and Optimization
        - Definition: Optimizes data storage for better performance and management.
        - Purpose: Reduces the number of small files and improves query performance.
        - Usage: Supports operations like vacuuming to remove obsolete data and optimize file layout.
    6. Unified Batch and Streaming
        - Definition: Supports both batch and streaming data processing.
        - Purpose: Facilitates processing of real-time and historical data consistently.
        - Usage: Enables simultaneous batch and streaming queries on the same dataset.

Feature	Description	Purpose	Usage
ACID Transactions	Ensures reliable processing of database operations	Guarantees data integrity and consistency	Prevents data corruption and inconsistencies
Scalable Metadata Handling	Efficient management of metadata	Supports high-performance querying and data management	Handles large volumes of metadata
Data Versioning	Ability to access and revert to previous data versions	Provides historical access and rollback capability	Query historical data and roll back changes
Schema Enforcement and Evolution	Ensures data adheres to schema and allows schema changes	Maintains data quality and flexibility in schema changes	Prevents schema mismatch and supports schema evolution
Data Compaction and Optimization	Processes to improve data storage and query performance	Optimizes data layout and performance	Compacts files and removes obsolete data
Unified Batch and Streaming	Support for both batch and streaming data processing	Simplifies data processing workflows	Processes real-time and historical data together

Creating a Delta Table

1. 1. Creating a Delta Table with SQL

```
CREATE TABLE table_name (
  column1_name column1_type,
  column2_name column2_type,
  ...
) USING DELTA
[OPTIONS (key1 = 'value1', key2 = 'value2', ...)]
```

```
CREATE TABLE IF NOT EXISTS dev_cat.demo_db.flight_time_tbl (
    FL_DATE DATE,
    OP_CARRIER STRING,
    OP_CARRIER_FL_NUM INT,
    ORIGIN STRING,
    ORIGIN_CITY_NAME STRING,
    DEST STRING,
    DEST_CITY_NAME STRING,
    CRS_DEP_TIME INT,
    DEP_TIME INT,
    WHEELS_ON INT,
    TAXI_IN INT,
    CRS_ARR_TIME INT,
    ARR_TIME INT,
```

```
    CANCELED STRING,
    DISTANCE INT
) USING DELTA
```

2. Creating a Delta Table with DataFrame in PySpark

```
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("DeltaLakeExample").getOrCreate()

# Create DataFrame
data = spark.createDataFrame([
  (1, "Alice", "2024-08-21 10:00:00"),
  (2, "Bob", "2024-08-21 11:00:00")
], ["id", "name", "timestamp"])

# Write DataFrame to Delta table
data.write.format("delta").save("/mnt/delta/delta_table")
```

   a. Additional Step to Register Table:

```
# Register the Delta table in the metastore
spark.sql("CREATE TABLE delta_table USING DELTA LOCATION
'/mnt/delta/delta_table'")
```

3. Creating a Delta Table from Existing Data
   a. If you have existing data in a file (e.g., CSV, Parquet) and want to convert it into a Delta table:

```
flight_schema_ddl = """FL_DATE DATE, OP_CARRIER STRING,
OP_CARRIER_FL_NUM INT, ORIGIN STRING,
          ORIGIN_CITY_NAME STRING, DEST STRING, DEST_CITY_NAME STRING,
CRS_DEP_TIME INT, DEP_TIME INT,
          WHEELS_ON INT, TAXI_IN INT, CRS_ARR_TIME INT, ARR_TIME INT,
CANCELED STRING, DISTANCE INT"""

flight_time_df = (spark.read.format("json")
                    .schema(flight_schema_ddl)
                    .option("dateFormat", "M/d/y")
                    .load(f"/mnt/stadls/input-data/flight-time.json")
                  )
flight_time_df.write.format("delta").mode("append").saveAsTable("dev_cat
.demo_db.flight_time_tbl")
```

Register Table:

```
# Register the Delta table
```

```
spark.sql("CREATE TABLE delta_table USING DELTA LOCATION
'/mnt/delta/delta_table'")
```

4. Creating a Delta Table with Schema Evolution
    a. If you want to allow schema changes when writing data:

```
data.write.format("delta") \
  .option("mergeSchema", "true") \
  .mode("append") \
  .save("/mnt/delta/delta_table")
```

Note: The mergeSchema option allows the schema of the Delta table to evolve to include new columns.

5. Creating a Delta Table with Delta Lake's DeltaTable API

```
from delta import DeltaTable

(DeltaTable.createOrReplace(spark)
    .tableName("dev.demo_db.flight_time_tbl")
    .addColumn("id", "INT")
    .addColumn("FL_DATE", "DATE")
    .addColumn("OP_CARRIER", "STRING")
    .addColumn("OP_CARRIER_FL_NUM", "INT")
    .addColumn("ORIGIN", "STRING")
    .addColumn("ORIGIN_CITY_NAME", "STRING")
    .addColumn("DEST", "STRING")
    .addColumn("DEST_CITY_NAME", "STRING")
    .addColumn("CRS_DEP_TIME", "INT")
    .addColumn("DEP_TIME", "INT")
    .addColumn("WHEELS_ON", "INT")
    .addColumn("TAXI_IN", "INT")
    .addColumn("CRS_ARR_TIME", "INT")
    .addColumn("ARR_TIME", "INT")
    .addColumn("CANCELED", "STRING")
    .addColumn("DISTANCE", "INT")
    .execute()
)
```

Method	Description	Usage
SQL	Define table schema and create Delta table using SQL syntax.	Ideal for straightforward table creation in SQL.
DataFrame in PySpark	Create DataFrame and write it as a Delta table.	Commonly used for programmatic data processing.
Existing Data	Convert existing data files into Delta format.	Useful for migrating data into Delta Lake format.

Schema Evolution	Allow schema changes when writing data.	Enables flexibility in managing evolving data schemas.
DeltaTable API	Programmatic way to create and manage Delta tables.	Advanced API for managing Delta tables and schema.

Schema Validation in Delta Lake

- Schema validation in Delta Lake ensures that the data written to a Delta table conforms to the table's defined schema. This feature is crucial for maintaining data quality and consistency. It works by enforcing rules on the data being inserted or updated, ensuring it matches the schema of the Delta table.
- How Schema Validation Works
    1. Validation on Write: When you write data to a Delta table, Delta Lake checks if the schema of the incoming data matches the schema of the existing table. This includes checking data types, column names, and column order.
    2. Enforcement: If schema validation is enabled, Delta Lake will enforce schema compliance and reject any data that does not conform to the table's schema.
    3. Schema Evolution: Delta Lake supports schema evolution, allowing you to modify the schema of a Delta table and automatically accommodate changes such as new columns.
- When Schema Validation Will Work
    1. Exact Schema Match
        - Scenario: Data being written exactly matches the schema of the Delta table.
        - Outcome: Schema validation passes, and data is written successfully.
    2. Schema Evolution Enabled
        - Scenario: Schema evolution is enabled (mergeSchema option set to true), and the schema of the incoming data includes new columns not present in the existing table schema.
        - Outcome: Delta Lake will update the table schema to include the new columns and write the data.
    3. Data Type Compatibility
        - Scenario: Data types of the incoming data are compatible with the defined schema (e.g., writing integers to an integer column).
        - Outcome: Schema validation passes, and data is written successfully.
- When Schema Validation Will Not Work
    1. Schema Mismatch with mergeSchema Disabled
        - Scenario: Data being written does not match the table schema, and schema evolution is not enabled.
        - Outcome: Schema validation fails, and the write operation is rejected.
    2. Incorrect Data Types
        - Scenario: Incoming data contains values that do not match the expected data types (e.g., writing a string to an integer column).
        - Outcome: Schema validation fails, and the write operation is rejected.

3. Missing Required Columns
    - Scenario: Data being written does not include columns that are defined as required in the table schema.
    - Outcome: Schema validation fails if the columns are mandatory and not nullable.
4. Column Order Mismatch
    - Scenario: Data being written has columns in a different order compared to the schema definition.
    - Outcome: Schema validation may fail if column order is enforced, though Delta Lake generally does not enforce column order strictly.
5. Unsupported Schema Changes
    - Scenario: Schema evolution attempts to make changes that are not supported (e.g., changing a column's data type).
    - Outcome: Schema evolution may fail, resulting in write operations being rejected.
6. Incompatible Data with Evolved Schema
    - Scenario: Schema evolution occurs, but the data being written is incompatible with the newly evolved schema (e.g., new column data does not meet the schema requirements).
    - Outcome: Schema validation might fail depending on the nature of the incompatibility.

Scenario	Description	Outcome
Exact Schema Match	Data matches the table schema exactly.	Schema validation passes; data is written successfully.
Schema Evolution Enabled	Schema evolution is enabled, and incoming data includes new columns.	Schema is updated, and data is written successfully.
Data Type Compatibility	Data types of incoming data are compatible with table schema.	Schema validation passes; data is written successfully.
Schema Mismatch with mergeSchema Disabled	Data schema does not match the table schema, and schema evolution is disabled.	Schema validation fails; write operation is rejected.
Incorrect Data Types	Incoming data contains incompatible data types.	Schema validation fails; write operation is rejected.
Missing Required Columns	Data lacks columns defined as required in the table schema.	Schema validation fails; write operation is rejected.
Column Order Mismatch	Data columns are in a different order from the schema definition.	Schema validation may fail, though order is not strictly enforced.
Unsupported Schema Changes	Schema evolution attempts unsupported changes.	Schema evolution may fail; write operation is rejected.
Incompatible Data with Evolved Schema	Data does not meet the requirements of the newly evolved schema.	Schema validation might fail depending on the incompatibility.

Schema Evolution in Delta Lake

- Schema Evolution is a feature in Delta Lake that allows a Delta table to adapt to changes in its schema over time. This feature is essential for managing evolving data requirements without disrupting existing data processing workflows.
- How Schema Evolution Works
    1. Automatic Schema Evolution
        - Definition: When you write data to a Delta table, Delta Lake can automatically update the table schema to include new columns that are present in the incoming data but not in the existing schema.
        - Activation: This is controlled by the mergeSchema option.
        - Usage: When mergeSchema is set to true, Delta Lake will automatically merge the new schema with the existing schema, adding new columns as needed.
    2. Manual Schema Evolution
        - Definition: Allows explicit modification of the table schema using DDL commands or Delta Lake APIs.
        - Activation: Changes are applied manually through commands or programmatic updates.
        - Usage: You can use SQL commands or Delta Lake APIs to add, drop, or modify columns in the schema.
- Key Points of Schema Evolution
    1. New Columns: Automatically added to the table schema if the mergeSchema option is enabled.
    2. Column Type Changes: Changing data types of existing columns is generally not supported; schema evolution is more focused on adding new columns.
    3. Backward Compatibility: Schema evolution is backward compatible. Older data and queries that do not use the new columns will continue to function as expected.
    4. Data Validation: The new columns must adhere to the data types and constraints defined in the table schema.

Feature	Description	Example Usage
Automatic Schema Evolution	Automatically updates schema to include new columns.	Set mergeSchema to true when writing data.
Manual Schema Changes	Allows explicit modification of the schema.	Use SQL ALTER TABLE commands to add/drop columns.
New Columns	Automatically added if mergeSchema is enabled.	Write data with new columns, and they will be added.
Column Type Changes	Generally not supported; focus on adding new columns.	Schema evolution does not typically support type changes.
Backward Compatibility	Older queries and data are not affected by new columns.	Existing queries continue to work without new columns.

Practical Implementation Schema Evolution

Original Table Code

```
CREATE OR REPLACE TABLE dev_cat.demo_db.people_tbl
(
  id INT,
  firstName STRING,
  lastName STRING
) USING DELTA;
```

1. Manual schema evolution -
    i. New column at the end

```
ALTER TABLE dev_cat.demo_db.people_tbl ADD COLUMNS (birthDate STRING);
```

    ii. New column in the middle-

```
ALTER TABLE  dev_cat.demo_db.people_tbl ADD COLUMNS (phoneNumber STRING
after lastName);
```

2. Automatic schema evolution
    i. At Session level

Config Below Code Than use Code

```
SET spark.databricks.delta.schema.autoMerge.enabled = true
```

```
INSERT INTO dev_cat.demo_db.people_tbl_new
SELECT id, fname firstName, name lastName, dob birthDate
FROM json.`/mnt/stadls/input-data/people_2.json`
```

    ii. Using API (Recommend)

```
from pyspark.sql.functions import to_date

people_2_schema = "id INT, fname STRING, lname STRING, dob STRING"

people_2_df =  (spark.read.format("json").schema(people_2_schema)
                    .load("/mnt/stadls/input-data/people_2.json")
                    .toDF("id", "firstName", "lastName", "birthDate"))

(people_2_df.write
      .format("delta")
      .mode("append")
      .option("mergeSchema", "true")
      .saveAsTable("dev_cat.demo_db.people_tbl_new_API")
)
```

Components of a Delta Table

1. Table Metadata
    - Definition: Metadata about the Delta table, including schema, partitioning information, and table properties.
    - Location: Stored in the _delta_log directory as JSON files.
    - Details: Includes the table schema, partitioning columns, and metadata such as table properties and version history.
2. Transaction Log (_delta_log)
    - Definition: A directory within the Delta table that contains JSON files describing all changes made to the table.
    - Files: Includes JSON files (000000000000.json, 000000000001.json, etc.) and possibly checkpoint files (_delta_log/_last_checkpoint).
    - Purpose: Keeps track of all transactions, including additions, deletions, updates, and schema changes.
3. Data Files
    - Definition: Parquet files where the actual data of the Delta table is stored.
    - Location: Located in the Delta table directory (e.g., /mnt/delta/delta_table/).
    - Details: Data files are managed by Delta Lake, and their organization is determined by the table schema and partitioning.
4. Checkpoint Files
    - Definition: Files that help optimize performance by summarizing the state of the table at a specific point in time.
    - Location: Located in the _delta_log directory.
    - Purpose: Used to speed up the process of reconstructing the table's state by reducing the number of log files that need to be processed.
5. Schema Information
    - Definition: The schema defines the structure of the data, including column names, data types, and constraints.
    - Location: Stored in the Delta table’s metadata in the _delta_log.
6. Partitioning
    - Definition: A method to divide the table into smaller, manageable pieces based on column values.
    - Purpose: Improves query performance by reducing the amount of data scanned.
    - Location: Partitions are reflected in the directory structure where data files are stored (e.g., /mnt/delta/delta_table/year=2024/month=08/).
7. Delta Table Properties
    - Definition: Custom properties that can be set on a Delta table for configuration and management.
    - Examples: Properties like delta.autoOptimize.optimizeWrite and delta.autoOptimize.autoCompact.

Component	Description	Location
Table Metadata	Schema, partitioning, and properties of the Delta table.	_delta_log directory
Transaction Log	JSON files tracking all changes and updates.	_delta_log directory

Data Files	Parquet files containing the actual data.	Delta table directory
Checkpoint Files	Summary files optimizing read performance.	_delta_log directory
Schema Information	Defines column names, types, and constraints.	Metadata in _delta_log
Partitioning	Directory-based organization of data for performance.	Delta table directory
Delta Table Properties	Custom configuration settings for table management.	Metadata in _delta_log

Time Travel in Delta Lake

- Time Travel is a powerful feature in Delta Lake that allows you to access and query the historical versions of your Delta tables. This feature enables you to "travel back in time" to see the state of your data at any previous point.
- Key Features of Time Travel
    1. Version Control: Delta Lake keeps track of all changes made to a table, allowing you to access older versions of the table.
    2. Historical Queries: You can query the data as it existed at a specific timestamp or version number.
    3. Data Auditing: Enables auditing changes to the data over time, which is useful for compliance and debugging.
    4. Data Recovery: You can recover or revert the table to a previous state in case of accidental deletions or updates.
- How Time Travel Works
    1. Delta Lake maintains a transaction log (in the _delta_log directory) that records every change made to the table. This log allows Delta Lake to reconstruct the table's state at any given point in time by applying or ignoring specific transactions.
    2. You can access historical data using two methods:
        - Using Version Number: Specify a specific version of the table.
            - Example: Query data from version 3.

```
SELECT * FROM my_delta_table VERSION AS OF 3;
```

        - Using Timestamps: Specify a timestamp to query data as it existed at that point in time.
            - Example: Query data as of a specific timestamp.

```
SELECT * FROM my_delta_table TIMESTAMP AS OF '2024-08-21 00:00:00';
```

- Scenario Examples
    1. Data Auditing
        - Scenario: You want to audit the changes made to your data on a specific date.
        - Action: Query the table using a timestamp to see the data before and after the changes.

```
-- Before change
```

```
SELECT * FROM my_delta_table TIMESTAMP AS OF '2024-08-20 10:00:00';

-- After change
SELECT * FROM my_delta_table TIMESTAMP AS OF '2024-08-20 12:00:00';
```

2. Recover Data
    - Scenario: You accidentally deleted important records from your table.
    - Action: Use time travel to restore the table to its state before the deletion.

```
-- Query data before deletion
SELECT * FROM my_delta_table VERSION AS OF 5;
```

3. Compare Data Across Versions
    - Scenario: You want to compare data between two versions to understand what changed.
    - Action: Query the data from both versions and compare the results.

```
-- Version 5
SELECT * FROM my_delta_table VERSION AS OF 5;

-- Version 6
SELECT * FROM my_delta_table VERSION AS OF 6;
```

4. Historical Analysis
    - Scenario: Perform analysis on data from a specific point in the past.
    - Action: Use a timestamp to query and analyze the data as it existed at that time.

```
SELECT * FROM my_delta_table TIMESTAMP AS OF '2024-07-15 08:00:00';
```

- Configuration Options
    1. Retention Period: Delta Lake automatically cleans up older versions of the table according to a retention period. The default retention period is 30 days, but this can be configured using the delta.logRetentionDuration and delta.deletedFileRetentionDuration properties.
        - Example:

```
ALTER TABLE my_delta_table SET TBLPROPERTIES (
  'delta.logRetentionDuration' = '60 days',
  'delta.deletedFileRetentionDuration' = '60 days'
);
```

    2. Vacuum: To permanently delete older data that is no longer needed, you can use the VACUUM command.

```
VACUUM my_delta_table RETAIN 168 HOURS;  -- Retains 7 days of history
```

- Key Benefits
    1. Flexibility: Allows querying and recovering data from different points in time.
    2. Data Integrity: Ensures that historical data is preserved and auditable.
    3. Disaster Recovery: Provides a mechanism to revert accidental changes.
    4. Compliance: Facilitates compliance with data retention and auditing policies.

Feature	Description	Example
Version Control	Access table data from a specific version.	`SELECT * FROM my_table VERSION AS OF 3;`
Timestamp Queries	Access table data as of a specific timestamp.	`SELECT * FROM my_table TIMESTAMP AS OF '2024-08-21';`
Data Auditing	Review changes made at different points in time.	`SELECT * FROM my_table TIMESTAMP AS OF '2024-08-20 10:00:00';`
Data Recovery	Restore the table to a previous state before an error.	`SELECT * FROM my_table VERSION AS OF 5;`
Retention Configuration	Adjust the retention period for historical data.	`ALTER TABLE SET TBLPROPERTIES ('delta.logRetentionDuration' = '60 days');`
Vacuum	Permanently remove old data to reclaim storage.	`VACUUM my_table RETAIN 168 HOURS;`