# Databricks Certified Data Engineer Associate

# Delta Lake

# Delta Lake

- **Optimized storage layer that provides foundation for storing data and tables in Databricks Lakehouse Platform**
- **Open source software that extends,**

1. Parquet data files with a file-based transaction log for ACID transactions
2. And scalable metadata handling

# Delta Lake

- **ACID transactions : Atomicity, Consistency, Isolation, Durability**

1. **Atomicity:** All transactions either succeed or fail completely
2. **Consistency:** Guarantees relate to how a given state of the data is observed by simultaneous operations
3. **Isolation:** Refers to how simultaneous operations potentially conflict with one another
4. **Durability:** Means that committed changes are permanent

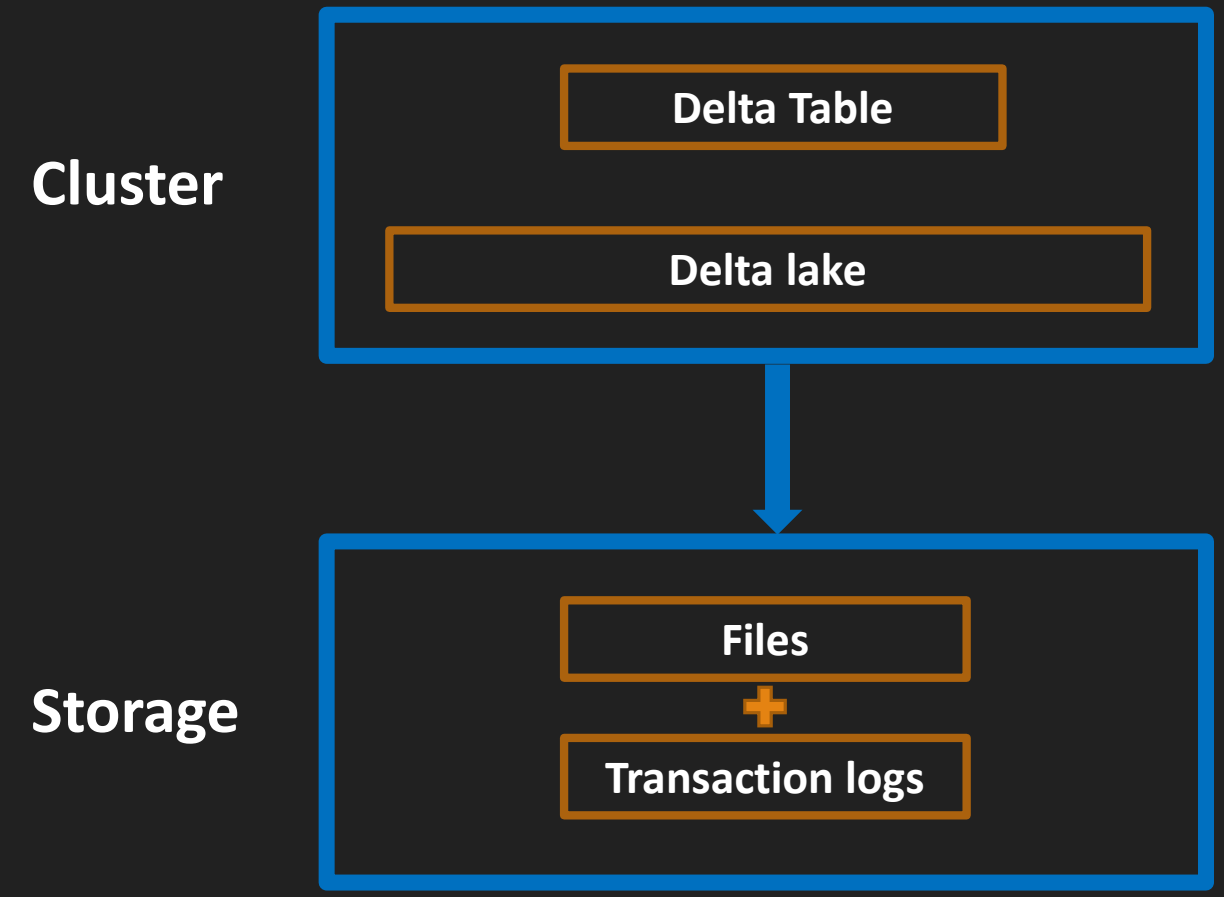
# Delta Lake

- Fully compatible with Apache Spark APIs, developed for tight integration with Structured Streaming
- Allows you to easily use a single copy of data for both batch and streaming operations, provides incremental processing at scale
- Delta Lake is the **default storage format for all operations on Databricks,** unless specified, all tables on Databricks are Delta tables
- Databricks originally developed Delta Lake protocol & continues to actively contribute to open source project
- Many of the optimizations and products in the Databricks Lakehouse Platform build upon the guarantees provided by Apache Spark and Delta Lake

# Delta Lake

- Open source technology, not proprietary technology
- Storage framework/layer, not storage format/medium
- Enabling building lakehouse, not data warehouse/database service

# Delta Lake



# Transaction log

- **Delta Lake transaction log (DeltaLog):** An ordered record of every transaction that has ever been performed on a Delta Lake table
- Delta Lake transaction log **serves as a single source of truth** - the central repository that tracks all changes that users make to the table
- **Implementation of Atomicity on Delta Lake:** One of the four properties of ACID transactions (atomicity), guarantees that operations (INSERT or UPDATE) performed on data lake either complete fully, or don't complete at all

# How Does the Transaction Log Work?

- **Breaking Down Transactions Into Atomic Commits:** Whenever a user performs an operation to modify a table (INSERT, UPDATE or DELETE), Delta Lake breaks that operation down into a series of discrete steps composed of one or more of the actions below,

1. Add file - adds a data file
2. Remove file - removes a data file
3. Update metadata - Updates the table's metadata (e.g., changing the table's name, schema or partitioning)
4. Set transaction - Records that a structured streaming job has committed a micro-batch with the given ID
5. Change protocol - enables new features by switching the Delta Lake transaction log to the newest software protocol
6. Commit info - Contains information around the commit, which operation was made, from where and at what time

# How Does the Transaction Log Work?

- **Commits:** Those actions are then recorded in the transaction log as ordered, atomic units known as commits

- **For example,** suppose a user creates a transaction to add a new column to a table plus add some more data to it. Delta Lake would break that transaction down into its component parts, and once the transaction completes, add them to the transaction log as the following commits:

  Update metadata - change the schema to include the new column

  Add file - for each new file added

# Delta Lake Advantages

- Brings ACID transactions to object storage (Atomicity, Consistency, Isolation, Durability)
- Handle scalable metadata
- Full audit trail of all changes
- Builds upon standard data formats: Parquet + Json

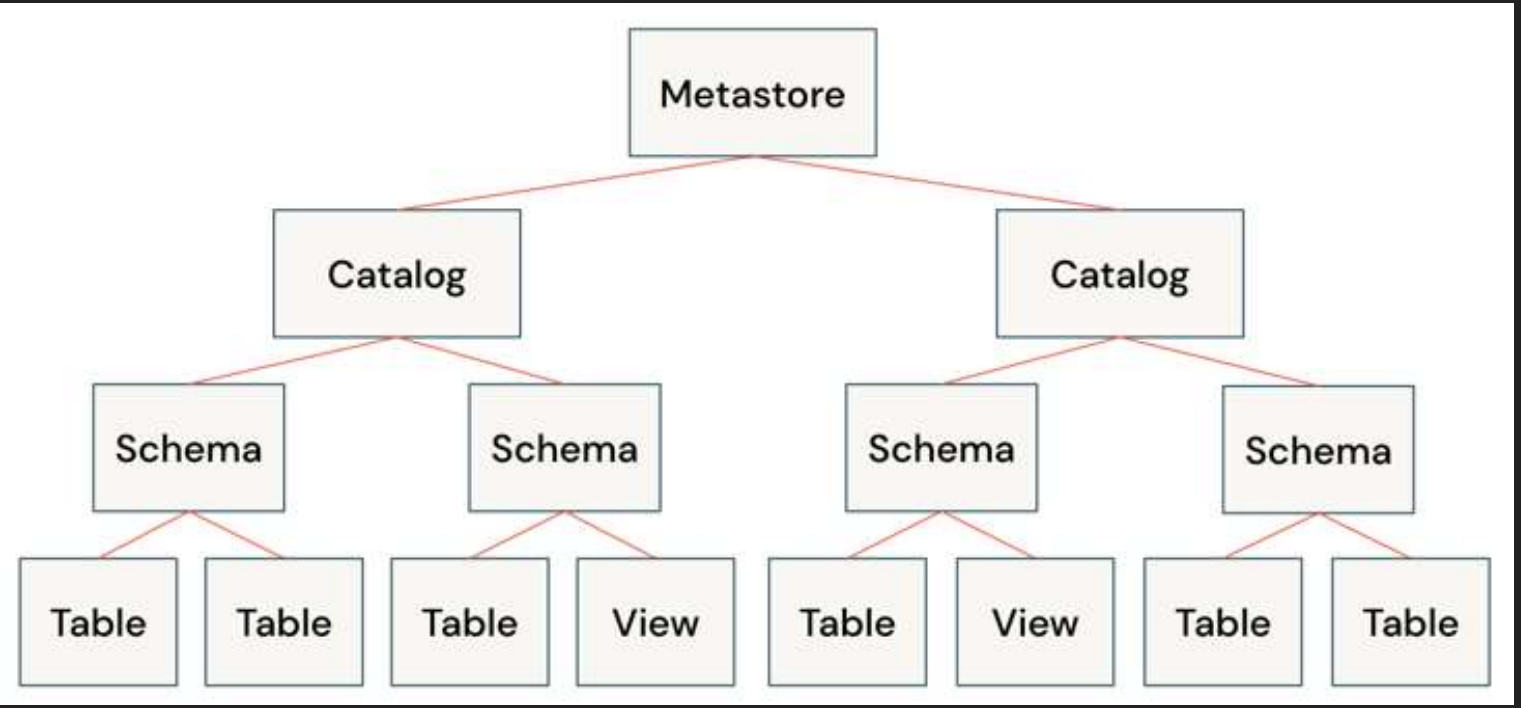
# Data objects in Databricks Lakehouse

# What are the data objects in Databricks Lakehouse?

- **The Databricks Lakehouse architecture combines data stored with the Delta Lake protocol in cloud object storage with metadata registered to a metastore**
- There are five primary objects in the Databricks Lakehouse:

1. **Catalog:** Grouping of databases
2. **Database or schema:** Grouping of Database in a catalog. Databases contain tables, views, and functions
3. **Table:** Collection of rows and columns stored as data files in object storage
4. **View:** Saved query typically against one or more tables or data sources
5. **Function:** Saved logic that returns a scalar value or set of rows

# What are the data objects in Databricks Lakehouse?



# What is a metastore?

- **The metastore contains all of the metadata that defines data objects in the lakehouse**
- Databricks provides the following metastore options:

1. **Unity Catalog:** you can create a metastore to store & share metadata across multiple Databricks workspaces. Unity Catalog is managed at the account level
2. **Hive metastore:** Databricks stores all metadata for built-in Hive metastore as a managed service. An instance of the metastore deploys to each cluster and securely accesses metadata from a central repository for each customer workspace
3. **External metastore:** you can also bring your own metastore to Databricks

- Regardless of the metastore used, Databricks stores all data associated with tables in object storage configured by customer in their cloud account

# What is a catalog?

- A catalog is the highest abstraction (or coarsest grain) in the Databricks Lakehouse relational model. Every database will be associated with a catalog & catalogs exist as objects within a metastore
- The built-in Hive metastore only supports a single catalog, hive_metastore

# What is a database?

- **A database is a collection of data objects,** such as tables, views & functions
- In Databricks, the terms “schema” and “database” are used interchangeably (whereas in many relational systems, a database is a collection of schemas)

# What is Table?

- **The Databricks table is a collection of structured data.** A Delta table stores data as a directory of files on cloud object storage and registers table metadata to the metastore within a catalog and schema
- As Delta Lake is the default storage provider for tables created in Databricks, all tables created in Databricks are Delta tables, by default
- **Note that,** it is possible to create tables on Databricks that are not Delta tables. These tables are not backed by Delta Lake, and will not provide the ACID transactions and optimized performance of Delta tables
- **There are two kinds of tables in Databricks,**

1. Managed tables
2. Unmanaged (or external) tables

# Managed Table

- Databricks manages **both metadata and data** for a managed table; when you drop a table, you also delete the underlying data
- Managed tables are the default when creating a table, the data for a managed table resides in the LOCATION of the database it is registered to
- There is relationship between the data location and the database means that in order to move a managed table to a new database, you must rewrite all data to the new location

# Ways to create managed tables

- **SQL:** CREATE TABLE table_name AS SELECT * FROM another_table
- **SQL:** CREATE TABLE table_name (field_name1 INT, field_name2 STRING)
- **Python:** df.write.saveAsTable("table_name")

# Unmanaged Table

- Databricks **only manages metadata** for unmanaged (external) tables;
- when you drop a table, you do not affect the underlying data
- Unmanaged tables will always specify a LOCATION during table creation;
- you can either register an existing directory of data files as a table or provide a path when a table is first defined
- Because data and metadata are managed independently, you can rename a table or register it to a new database without needing to move any data
- Data engineers often prefer unmanaged tables and for flexibility they provide for production data

# Ways to create unmanaged tables

**1) SQL:**

```
CREATE TABLE table_name
USING DELTA
LOCATION '/path/to/existing/data’
```

**2) SQL:**

```
CREATE TABLE table_name
(field_name1 INT, field_name2 STRING)
LOCATION '/path/to/empty/directory'
```

**3) Python:** df.write.option("path", "/path/to/empty/directory").saveAsTable("table_name")

# What is view?

**A view stores the text for a query** typically against one or more data sources or tables in the metastore

# Views

# Views

- Stores the text for a query typically against one or more data sources or tables in metastore
- Logical query against source tables
- **Types of views:**

1. Views (Stored/classical)
2. Temporary views
3. Global Temporary views

# Views (Stored/Classical)

- **Persisted objects:** Like tables, stored views are persisted in the database

- **Create view:** CREATE VIEW view_name
  AS query

# Temporary view

- **Session-scoped view,** tied to spark session and dropped when the session ends
- **Create view:**

```
CREATE TEMP VIEW view_name
AS query
```

- **When a new spark session is created in databricks?**

1. Opening a new notebook
2. Detaching and reattaching a notebook to a cluster
3. Installing a python package
4. Restarting a cluster

# Global Temporary views

- **Cluster-scoped view,** tied to the cluster
- As long as cluster is running, any notebook attached to the cluster can access its global temporary views
- **Create view:**

```
CREATE GLOBAL TEMP VIEW view_name
AS query
```

- **SELECT * FROM global_temp.view_name:** Global temporary views are added to a cluster's temporary database called global_temp,
- So when you query this view in a SELECT statement, you need to use the global_temp database qualifier

# Views Comparison

| (Stored) Views | Temp views | Global Temp views |
|---|---|---|
| Persisted in DB | Session-scoped | Cluster-scoped |
| Dropped only by DROP VIEW | dropped when session ends | dropped when cluster restarted |
| CREATE VIEW | CREATE TEMP VIEW | CREATE GLOBAL TEMP VIEW |

# THANK YOU