inceptez
Web: Inceptez.com Mail: info@inceptez.com Call: 7871299810, 7871299817

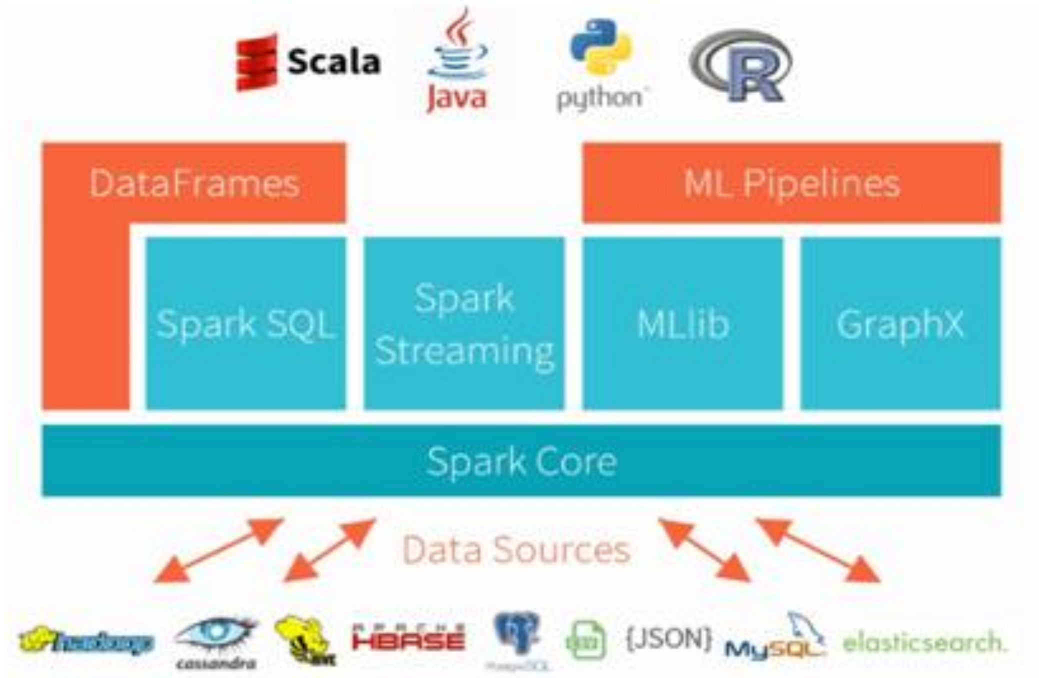
Databricks

GitHub Integration

Apache Spark

About Spark

- Open-source framework to process large volume of data
- Provides high-level APIs in Scala, Java , Python and R
- 10x(Disk) and 100x(In-Memory) faster than Hadoop
- Spark offer different types of data processing
- Unified data access
- Runs on multiple cluster
- Fast growing open-source and large support community base
- Written in Scala,Java,Python and Other languages
- Developed by Matei Zaharia at UC Berkeley's AMPLab in 2009, and released as apache open source in 2014



Why Spark

Speed (In-Memory Processing)

Spark keeps data in memory (RAM) instead of writing to disk between every operation like MapReduce.

Unified System

Supports **batch, streaming, SQL, machine learning, and graph processing** within a single framework.

Combine Processing Types in One Program

You can mix **SQL + ML + Streaming** in the same application.

Example: query streaming data, enrich with ML model, then store in a warehouse.

Code Reuse

Write logic once and reuse it across batch jobs, streaming pipelines, or ML workflows.

One System to Learn & Maintain

Reduces complexity: developers, analysts, and data scientists can work on a common platform.

Supports All Processing Modes

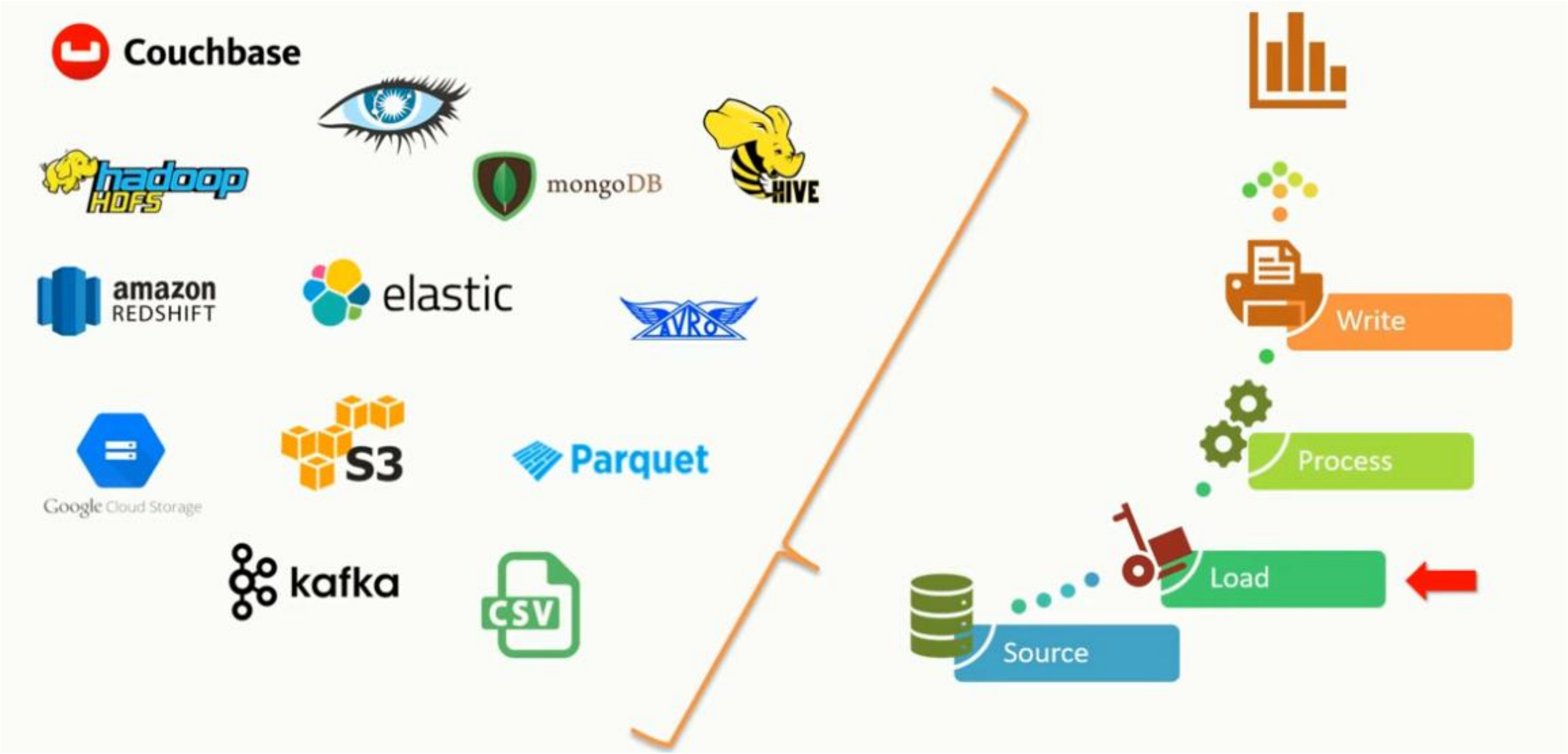
Realtime → Spark Structured Streaming

Iterative → ML algorithms that loop efficiently

In-Memory → Faster than disk-based MapReduce

Batch → Large-scale ETL and data warehouse jobs

Supported Data Sources



Pillars of Spark

Resilient Distributed Dataset(RDD): Fundamental unit of data in spark

Resilient – If data in memory lost, it recreated

Distributed - Store data in memory across the cluster

Dataset – Collection of data

Directed Acyclic Graph(DAG): graph denoting the sequence of operations that performed on the RDD

Directed – Directly connected from one node to another. This creates a sequence i.e. each node is in linkage from earlier to later in the appropriate sequence.

Acyclic – Defines that there is no cycle or loop available. Once a transformation takes place it cannot returns to its earlier position.

Graph – From graph theory, it is a combination of vertices and edges. Those pattern of connections together in a sequence is the graph.

Properties of RDD

Immutability

- Once RDD is created, it **cannot be changed**.
- Any operation on an RDD produces a **new RDD**.

Distributed

- Data is **split across multiple nodes** in a cluster, allowing parallel processing.
- Each partition of the RDD is processed independently.

Lazy Evaluation

- Transformations (like map, filter) are **not executed immediately**.
- They are recorded as a lineage graph, and only executed when an **action** (like count, collect, save) is called.

Resilient (Fault-Tolerant)

- RDDs can **recompute lost data** using lineage information.
- If a partition is lost due to a node failure, Spark rebuilds it using the original transformations.

In-Memory Computation

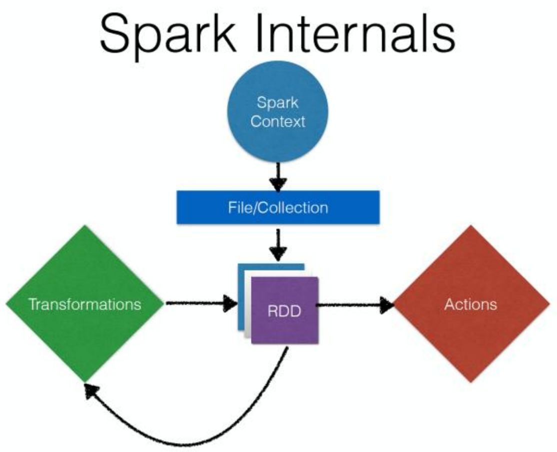
- RDDs can be **cached or persisted** in memory for faster re-use in iterative operations.
- Supports multiple storage levels (MEMORY_ONLY, MEMORY_AND_DISK, etc.).

Partitioned

- Data inside an RDD is divided into **partitions**.
- Partitions are the unit of parallelism and distribution in Spark.

Spark RDD Operations

- In Apache Spark, an operation is an action you perform on an RDD or DataFrame to process data.
- Spark operations are the building blocks of computation.
- Every Spark job is made up of a series of operations.
- Operations are classified into two main types:
  - **Transformations**
  - **Actions**



1. Transformations

Operations that **create a new RDD/DataFrame** from an existing one.

Lazy evaluation

Spark doesn’t execute immediately; it just builds a **lineage/DAG**.

Examples:

map() → apply function to each element

filter() → filter elements by condition

flatMap() → flatten the results

union() → combine datasets

groupByKey() → group values by key (wide dependency → shuffle)

reduceByKey() → aggregate values by key

2. Actions

Operations that **trigger the execution** of transformations and return a value to the driver or write data to storage.

Eager evaluation → causes Spark to run the job.

Examples:

- collect() → bring data to driver
- count() → count records
- first() → first record
- take(n) → first n records
- saveAsTextFile() → save data
- reduce() → reduce dataset

Spark SQL

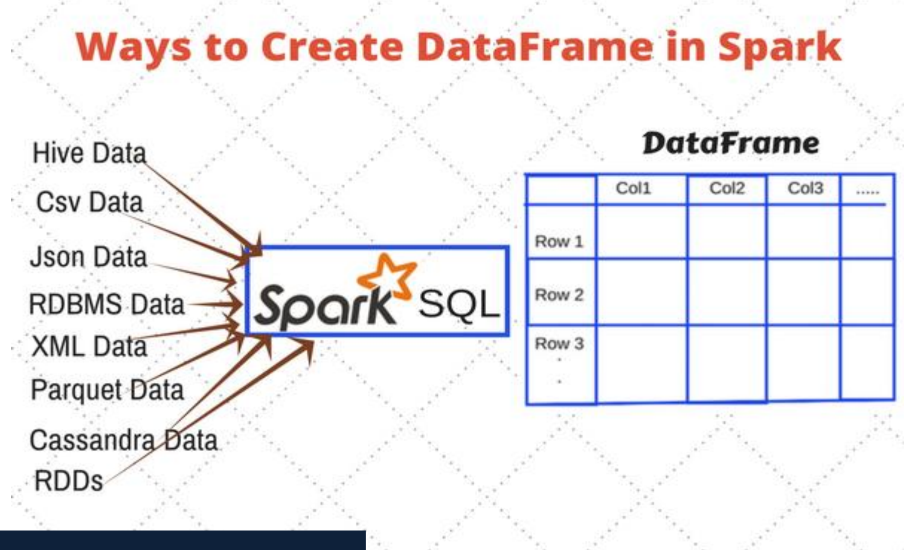
- Spark SQL is a module in Apache Spark for structured data processing.
- It lets you use SQL queries alongside Spark’s DataFrame and Dataset APIs.
- In Databricks, Spark SQL is tightly integrated into the notebook environment, allowing you to mix SQL + Python/Scala/R seamlessly.

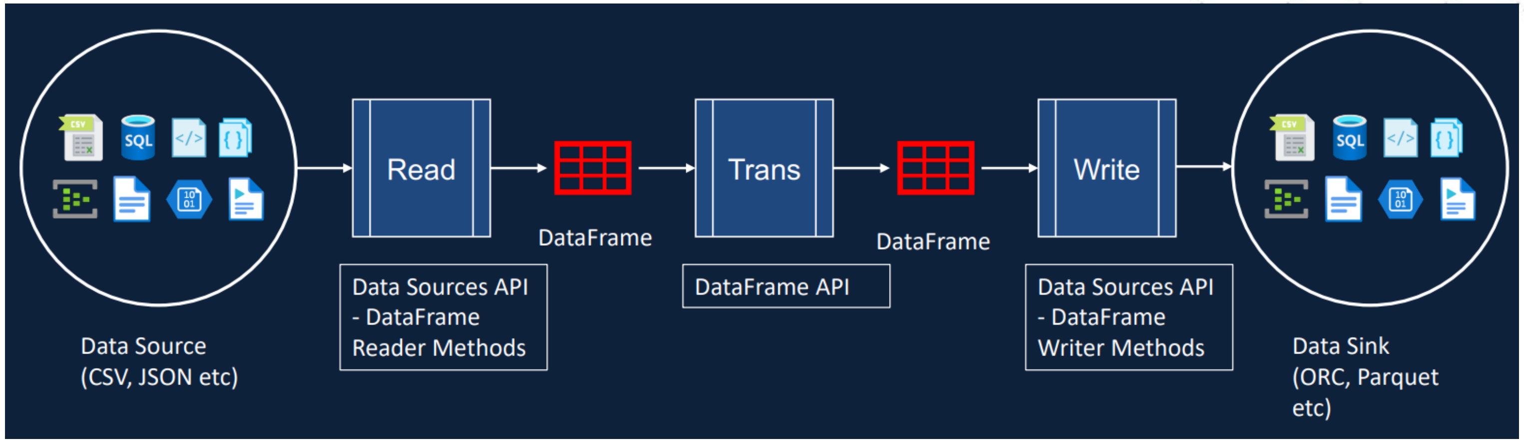
<u>Key Features of Spark SQL in Databricks</u>

1. **Unified API** → Use SQL, DataFrames interchangeably.
2. **Catalog & Metadata** → Integrates with **Unity Catalog** or **Hive Metastore** for table management.
3. **Performance Optimizations**
    1. Catalyst Optimizer (query optimization)
    2. Tungsten Engine (in-memory execution).
4. **Delta Lake Support** → Read/write to Delta tables with ACID transactions.
5. **Integration with BI Tools** → Power BI, Tableau, Looker, etc.

Spark SQL - DataFrame

- Spark SQL uses a programming abstraction called DataFrame.
- It is a distributed collection of data organized in named columns.
- DataFrame is equivalent to a database table but provides a much finer level of optimization.
- Data Frame is similar to a table in a relational database.
- DataFrame supports structured and semi structured data.
- DataFrame read, process and write data from different sources.
- DataFrame created using SparkSession, SQLContext or HiveContext





Databricks – Unity Catalog Overview

- Unity Catalog is a unified governance solution for data and AI assets on Databricks Lakehouse.
- Unity Catalog provides centralized access control, auditing, lineage, and data discovery capabilities across Databricks workspaces.

Unity Catalog Object Model

3-level namespace:

`catalog.schema.object`

Where **object** can be a table, view, function, model, etc.

Metastore

  └──> *Catalog*

    └──> *Schema*

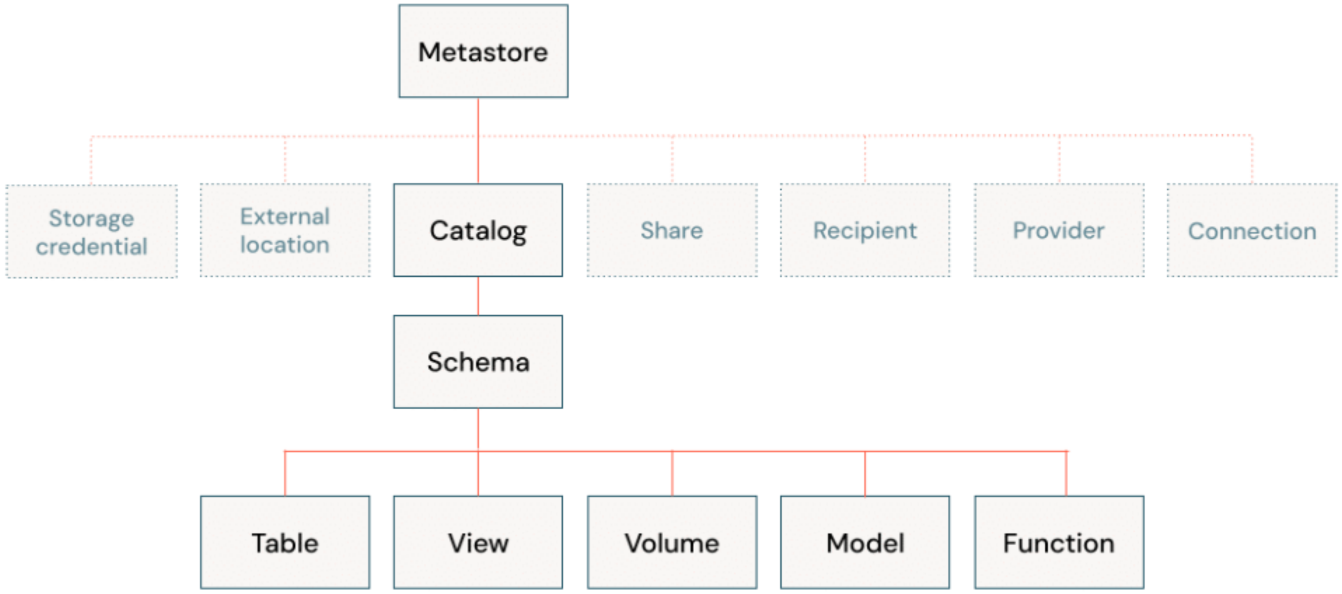
      |──> *Tables (Managed / External / Delta / Views)*

      |──> *Views (Temporary / Materialized)*

      |──> *Functions (UDFs)*

      |──> *Volumes (file collections)*

      |──> *Models (MLflow models)*



Databricks – Binary Formats in Spark

Binary formats are file formats that store data in a **machine-readable binary form** rather than human-readable text.

- Binary formats (Parquet, ORC, Delta, Avro) are much faster and storage-efficient compared to text formats.
- Columnar formats (Parquet, ORC, Delta) are ideal for analytics and aggregation queries.
- Row-based formats (CSV, Avro) are simpler for data interchange, but slower for analytics.
- Delta Lake adds ACID support and time travel, making it the best choice for production data lakes.
- Text formats are suitable only for small datasets or simple data exchange.

Key Benefits

- Compact Storage – Data is stored efficiently, often with built-in compression.
- Faster I/O – Spark can read/write binary files more quickly than parsing text.
- Supports Schema – Many binary formats (Parquet, ORC, Avro) store schema information with the data.
- Columnar or Row-based – Binary formats can be columnar (Parquet, ORC, Delta) or row-based (Avro).
- Predicate Pushdown – Some formats allow filtering at the storage level to reduce I/O.

Databricks – Binary Formats Comparison

Feature / Format	CSV / JSON / TXT (Text)	ORC (Binary)	Parquet (Binary)	Delta Lake (Binary)
Storage Type	Row-based, plain text	Columnar	Columnar	Columnar (built on Parquet)
Compression	Low or none	Very High	High (Snappy, GZIP, etc.)	High (inherits Parquet)
Read Performance	Slow	Fast	Fast	Very Fast
Write Performance	Medium	Fast	Fast	Fast
Predicate Pushdown	No	Yes	Yes	Yes
Schema Support	No	Yes	Yes	Yes
Schema Evolution	No	Yes	Yes	Yes
ACID Transactions	No	No	No	Yes
Best Use Case	Small datasets, simple storage	OLAP workloads	Large analytical datasets	Data lake, streaming, time travel

Databricks – ORC vs Parquet vs Delta

Feature	ORC	Parquet	Delta
📁 Type	Columnar file format	Columnar file format	Transactional layer **on Parquet**
⚙️ Ecosystem	Hive, Presto	Spark, Databricks	Spark, Databricks
📊 Read Performance	Very fast (Hive)	Fast	Fast + optimized
✍️ Write Performance	Slower	Faster	Slightly slower (logs)
🪶 File Size	Larger	Smaller	Slightly larger
🧠 Schema Evolution	Limited	Good	✅ Excellent
🔍 Predicate Pushdown	✅ Yes	✅ Yes	✅ Yes
🧪 ACID Transactions	❌ No	❌ No	✅ Yes
⏪ Time Travel	❌ No	❌ No	✅ Yes
🔁 Merge/Upsert	❌ No	❌ No	✅ Yes
🧰 Use Case	Hive batch queries	General Spark ETL	Lakehouse, real-time, analytics

Databricks – Partitioning in Spark

In Databricks, **data partitioning** means organizing large datasets into **directory structures** based on column values so that Spark can **read only what it needs** instead of scanning everything.

- A data partition is a subdirectory of your dataset created based on the distinct values of a column.
- Spark uses these folders to skip irrelevant data when you query (known as partition pruning).

Key Benefits of Partitioning

- **Faster queries** – Spark reads only relevant partitions instead of scanning the entire dataset.
- **Reduced I/O** – Minimizes data read from storage, saving time and resources.
- **Faster incremental loads** – Enables quick updates or inserts for specific partitions.
- **Better parallelism** – Allows Spark to process partitions independently for faster execution.
- **Optimized filtering** – Queries with partition column filters become highly efficient.

/Volumes/inceptez_catalog/outputdb/customerdata/

profession=Pilot/

profession=Teacher/

profession=Lawyer/

profession=Firefighter/

…

Best Practices in Partitioning

- Use low-cardinality columns – Partition on columns with relatively few unique values (e.g., year, region).
- Keep partition size balanced – Aim for 100 MB to 1 GB per partition file for optimal performance.
- Limit the number of partitions – Too many partitions can increase metadata overhead and slow queries.
- Use dynamic partition overwrite – Overwrite only affected partitions during incremental loads.
- Optimize and vacuum regularly – Compact small files and clean up old data in Delta tables.
- Partition based on query patterns – Choose columns that are frequently used in filters.
- Avoid over-partitioning small datasets – Small datasets do not benefit from heavy partitioning.

Databricks – TempView

- A Temp View is a virtual table created from a DataFrame.
- It does not store data physically but allows you to query the DataFrame using Spark SQL.
- Two types:
  - Local Temp View – **createOrReplaceTempView** – session-scoped.
  - Global Temp View – **createOrReplaceGlobalTempView** – cluster-scoped (global_temp database prefix).

Feature	Local Temp View	Global Temp View
Scope	Current Spark session only	All Spark sessions in cluster
Lifetime	Until session ends	Until Spark application ends
SQL Reference	view_name	global_temp.view_name
Shareable Across Notebooks	No	Yes

Databricks – Managed vs External Table

- In Databricks Spark SQL, there are two main types of tables
  - **Managed Table**
    - Spark manages both the metadata and the data.
    - Data is stored inside the warehouse directory .
    - Dropping the table that deletes both metadata and data.
  - **External Table**
    - Spark manages only the metadata, not the data.
    - Data is stored in a user-defined location (e.g., DBFS, S3, ADLS).
    - Dropping the table that deletes only the metadata, keeps the data.

Use Managed Tables for:

- Staging and internal pipeline tables
- Tables where Spark manages lifecycle

Use external tables when:

- Data is shared across systems.
- Data is already present and managed elsewhere.
- Need to preserve data even if the table is dropped.

Feature	Managed	External
Data storage	Inside warehouse directory	User-specified path
Data deletion on DROP	✅ Yes	❌ No
Requires LOCATION	❌ No	✅ Yes
Best for	Internal pipeline data	Shared or existing data
Metadata in metastore	✅ Yes	✅ Yes

Databricks – Persistent Views

A **Persistent View** (also called a *Permanent View*) is a **saved SQL query** stored in the **Databricks metastore or Unity Catalog**.

Unlike temporary views, it **persists across sessions, clusters, and users** until you explicitly drop it.

It acts like a **virtual table** — it doesn't store physical data, only the SQL logic used to fetch the data.

Regular (Non-Materialized) View

- Stores only the query definition.
- Data is fetched live from base tables when queried.
- The standard view type in Databricks.
- Always reflects current data from the base tables.
- Best for dynamic data and ad hoc analytics.

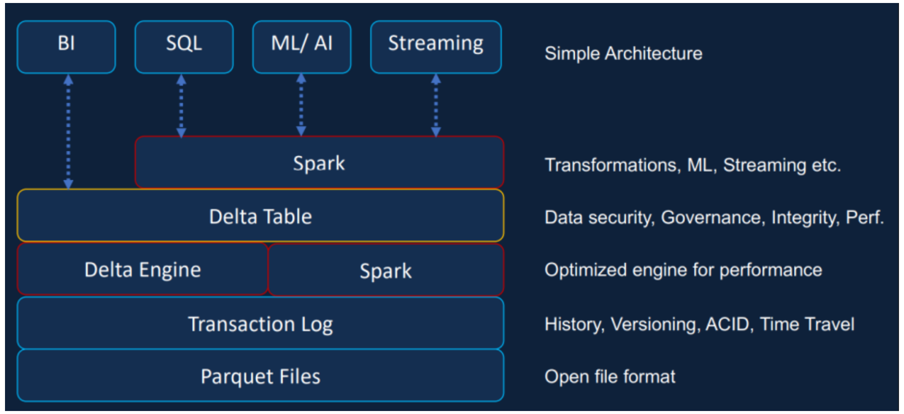
Materialized View

- A precomputed and cached view.
- Used for large datasets or frequently accessed aggregations.
- Data is physically stored and automatically refreshed on schedule.
- Improves performance and reduces compute cost.

Scenario	Recommended View Type
Frequently changing base data	Regular View
Static or slowly changing data	Materialized View
Dashboard or reporting performance optimization	Materialized View
Exploratory / ad-hoc analysis	Regular View

Databricks – Delta Lake

Delta Lake is an **open-source storage layer** that sits on top of data lake (like **Azure Data Lake**, **S3**, or **GCS**) and brings **ACID transaction support**, **schema enforcement**, and **data versioning** to **Apache Spark and Databricks**.



Delta Lake Architecture

A Delta table has two main components:

1. **Data Files** – Stored as Parquet files in object storage.
2. **Transaction Log (_delta_log)** – JSON and checkpoint files that record all table changes.

Feature	Description
ACID Transactions	Guarantees Atomicity, Consistency, Isolation, and Durability, even with concurrent writes.
Schema Enforcement & Evolution	Ensures data adheres to the expected schema, while allowing safe schema updates.
Time Travel	Lets query older versions of data (for debugging, audit, rollback).
Upserts & Deletes	Supports MERGE, UPDATE, and DELETE directly on large datasets.
Data Versioning	Every change creates a new version tracked in the _delta_log.
Optimized Performance	Uses file compaction, data skipping, and caching for faster reads/writes.

Databricks – Optimization on Delta Lake

Concept	Purpose / Function	When to Use	Example Syntax
OPTIMIZE	Rewrites small files into larger contiguous files	Improve query performance	OPTIMIZE delta_table
Z-ORDER	Co-locates related data physically for faster filtering	After table creation / updates	OPTIMIZE delta_table ZORDER BY (customer_id)
CLUSTER BY	Organizes data physically during table creation or insert	At table creation / insert	CREATE TABLE t CLUSTER BY (customer_id) AS SELECT * FROM s
VACUUM	Deletes old/unneeded files to free storage	Periodic cleanup	delta_table.vacuum(retentionHours=168)
CTAS	Create table from query results	When creating new table	CREATE TABLE new_table AS SELECT * FROM old_table
CLONE	Full copy of table (data + metadata)	Duplicate table fully	CREATE TABLE clone_table CLONE original_table
SHALLOW CLONE	Copies only metadata; points to same underlying files	Lightweight duplication	CREATE TABLE shallow_clone_table SHALLOW CLONE original_table
CHECKPOINT	Create a compact version of the delta log for faster recovery	More number of log files	ALTER TABLE delta.`/Volumes/inceptez_catalog/inputdb/employee/emp_chkpoint` SET TBLPROPERTIES ('delta.checkpointInterval' = '1');

Databricks – Delta Lake

Action	Delta Lake Behavior
Write	Stores data as Parquet + creates _delta_log/0000.json
Append	New Parquet + new JSON version
Transaction Log	Tracks add/remove files (ACID)
Checkpoint	Periodic Parquet snapshot for faster reads
Read	Combines Parquet + latest log for consistent view
Update/Delete	Marks old files as removed, writes new files
Optimize	Merges small files for performance
Zorder	Co-locate and sort the data in the file physically
Vacuum	Removes old files no longer needed
Schema Evolution	Automatically adapts schema
Liquid Clustering	Co-locate the data in the file physically. Define during table creation

Databricks – Clustering or Z-Ordering

- Clustering is a general concept of organizing rows in partitions for better performance.
- Z-Ordering is a specific type of clustering for high-cardinality columns using multi-dimensional ordering.
- Both complement partitioning in Delta Lake for maximum query efficiency.
- Z-Ordering works only on Delta tables in Databrick
- Table must exist before you can Z-Order it

Concept	Partitioning	Clustering (Z-Ordering)
Definition	Divides a table into separate **folders/directories** based on column values.	Organizes or **reorders data within files** so similar rows are physically close together.
Scope	Physical directory level	Within files (blocks)
Purpose	Reduce the amount of data scanned by Spark via **partition pruning**	Improve query performance for **high-cardinality filter columns** by minimizing block reads

Databricks – %run vs dbutils.notebook.run()

%run is a Databricks notebook command (called a cell magic) used to import and execute another notebook inline, as if its code was written directly in the current notebook.

- It is similar to a Python import — all variables, functions, and classes defined in the child notebook become available in the parent.
- The child notebook runs in the same Spark session, same context, and shares variables.

dbutils.notebook.run() is a Databricks utility function that executes another notebook as a separate job or task, usually with parameters and can return a result.

- It runs the target notebook in a new, isolated context.
- We can pass arguments and get a return value (as a string) using dbutils.notebook.exit() in the child.
- It's used for orchestrating multi-step pipelines.

Feature	%run	dbutils.notebook.run()
Execution context	Same session & Spark context.	New, isolated session & Spark context.
Variable sharing	Shared – variables and functions become available.	Not shared – must pass via parameters.
Parameter passing	Not supported directly (can use variable injection hack).	Supported – pass dictionary of arguments.
Return value	Cannot return values.	Can return a string result.
Use case	Share configs, functions, and constants.	Build multi-step workflows or pipelines.

Databricks – Lakeflow Jobs

- LakeFlow Jobs are an enhanced, unified orchestration framework in Databricks for building, scheduling, and managing data pipelines and workflows.

LakeFlow Jobs: Core Building Blocks

- **Job -** The overall workflow definition.
- **Task**

  A single unit of work (e.g., Python script, SQL query, ingestion).
- **DAG (Directed Acyclic Graph)**

  Defines dependencies and execution order.
- **Schedule/Trigger** - Controls when the job runs.
- **Monitoring** - Built-in dashboards, lineage, logs, and alerts.

Task Type	Description
Notebook	Run a Databricks notebook (PySpark, Scala, SQL, etc.)
Python Script	Run a .py file directly
Delta Live Table (DLT)	Trigger a DLT pipeline
SQL Query	Execute a SQL statement or script
LakeFlow Connect Task	Ingest data from a source (Postgres, Snowflake, etc.)
Webhook Task	Trigger external APIs
dbt Task	Run a dbt project
Custom Task	Shell or REST command

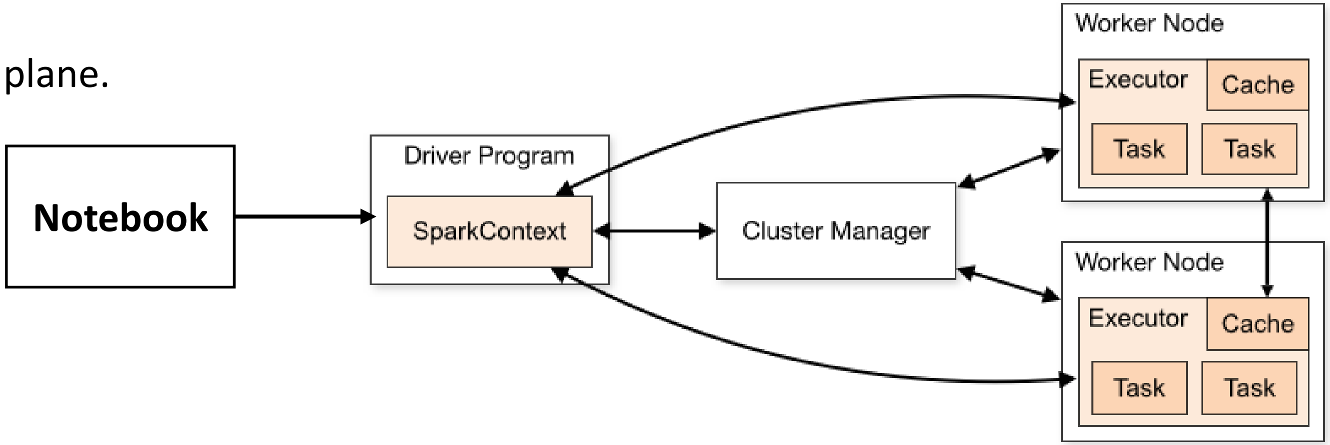
Lakeflow Jobs – Job-Level Options

Option	Description	Example
Name	Name of the workflow	Daily Txn ETL
Description	Detailed description	Ingest, transform, and aggregate daily transaction data
Tags	Key-value metadata for filtering/searching	env=prod, team=data-eng
Owner	Assigned user or service principal	data-team@databricks.com
Permissions	Control who can view/edit/trigger	Viewer, Editor, Owner
Max concurrent runs	Prevent overlapping runs	e.g., 1
Timeout (mins)	Kill job if exceeds limit	e.g., 180
Retry policy	Global retry for job	e.g., max 3 retries, interval: 10m
Notifications	Email/Slack/PagerDuty on success/failure	on_failure: Slack channel
Git Integration	Link code to a Git repo/branch	main branch in GitHub

Spark Architecture

Notebook UI (browser)

- Just a **frontend** where we write code (Python/SQL/Scala/R).
- Sends commands to the cluster through the Databricks control plane.



Driver Program

- Runs on the **driver node** of the attached cluster.
- Responsible for:
  - Translating your notebook code into a Spark job
  - Building the logical/physical execution plan
  - Coordinating execution across executors

```
df = spark.read.csv("dbfs:/databricks-datasets/airlines/part-00000", header=True)
df.groupBy("carrier").count().show()
```

Executors (worker nodes)

- Run the tasks assigned by the driver.
- Do the heavy lifting:
  - filtering, joining, aggregations, reading/writing data.

1. Notebook cell → code sent to the **driver node**.
2. Driver program (on driver node):
3. Uses **Catalyst Optimizer** to create a logical & physical plan.
4. Breaks into **tasks**.
5. Cluster manager assigns tasks to **executors**.
6. Executors run the tasks (parsing CSV, counting, grouping).
7. Results go back to the **driver** → notebook UI shows them (.show() table).

Databricks – DBFS vs Parquet vs Delta Lake

- DBFS (Databricks File System) - store and access files (CSV, JSON, Parquet, Delta, images, models, etc.) in **cloud object storage (S3, ADLS, GCS)** as if it's a local fs

- Parquet – Columnar file format and efficient storage of structured data with compression and fast read

/dbfs/mnt/data/sales_parquet/
part-00000-1f23c9c9.parquet
part-00001-7a9d8d12.parquet
part-00002-8b7d3e4a.parquet
...

- Delta Lake - **open-source storage layer** built on top of Parquet + transaction logs (_delta_log). Makes Parquet files **reliable and production-ready** with **ACID transactions, schema enforcement, and time travel**.

```
/dbfs/mnt/data/sales_delta/
  part-00000-1f23c9c9.snappy.parquet
  part-00001-7a9d8d12.snappy.parquet
  part-00002-8b7d3e4a.snappy.parquet
  ...
  _delta_log/
    00000000000000000000.json
    00000000000000000001.json
    00000000000000000002.json
    ...
```

- DBFS is the storage container.
- Inside DBFS, you can save data as Parquet (efficient files) or Delta Lake (transactional tables).
- Delta Lake uses Parquet under the hood but adds _delta_log to make it reliable and queryable like a database.

Feature	**DBFS**	**Parquet**	**Delta Lake**
Type	File system	File format	Storage layer (built on Parquet)
Stores what?	Any file (CSV, Parquet, Delta, etc.)	Tabular data in columnar format	Parquet + transaction log
ACID transactions	❌ No	❌ No	✅ Yes
Schema enforcement	❌ No	❌ No	✅ Yes
Schema evolution	❌ No	❌ Limited	✅ Yes
Time travel (versioning)	❌ No	❌ No	✅ Yes
Updates / Deletes	❌ No		✅ Yes
Best use case	General file storage	Raw/intermediate analytics data	Production-grade tables

Databricks – Common UseCase

Enterprise Data Lakehouse - Combines data lakes and warehouses into a single platform, providing a unified source of truth and reducing complexity in data management.

ETL and Data Engineering - Enables efficient extraction, transformation, and loading (ETL) with Delta Lake, Auto Loader, and Lakeflow Pipelines for clean, reliable, and timely data delivery.

Machine Learning, AI, and Data Science - Provides integrated tools like MLflow and Databricks Runtime for ML to build, train, and deploy machine learning models efficiently.

Data Warehousing, Analytics, and BI - Offers SQL-based analytics, scalable compute (SQL Warehouses), and notebooks for interactive queries, dashboards, and business intelligence.

Data Governance and Secure Sharing - Uses Unity Catalog for centralized governance, fine-grained permissions, and secure internal/external data sharing with Delta Sharing.

DevOps, CI/CD, and Task Orchestration - Simplifies development lifecycles with version control, automation, job scheduling, and Git integration for ETL, ML, and analytics workflows.

Real-time and Streaming Analytics - Processes streaming data using Apache Spark Structured Streaming and Delta Lake for low-latency, incremental analytics.

Lakehouse


Data Lake
- Open
- Flexible
- ML support
Lakehouse
One platform that unify all of your data engineering, analytics, and AI workloads
Data warehouse
- Reliable
- Strong governance
- Performance

Lakehouse

Aspect	Hive Metastore (Legacy)	Unity Catalog (Catalogs)
Scope	One per **workspace**	One per **region per account** (shared across workspaces)
Namespace	database.table (2 levels)	catalog.schema.table (3 levels)
Governance	Limited, table-level ACLs	Fine-grained RBAC (catalog → schema → table → column/row)
Lineage & Audit	Not available	Built-in lineage and auditing
Sharing	Manual copy / external storage	Native Delta Sharing (cross-org, cross-region)
Supported Assets	Only databases/tables/views	Tables, views, files (volumes), ML models, etc.

Window/Analytical Functions

Window functions perform **calculations across a set of table rows** - called a **window**.

Category	Functions	Purpose
Ranking	ROW_NUMBER(), RANK(), DENSE_RANK(), NTILE()	Assign position or rank
Aggregate	SUM(), AVG(), MIN(), MAX(), COUNT()	Compute rolling or grouped aggregates
Value / Navigation	LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE()	Access previous, next, or first/last rows in the window

Window Functions

ROW_NUMBER() → Strict order (no ties)

RANK() → Ties allowed, gaps exist

DENSE_RANK() → Ties allowed, no gaps

Function	Purpose	How It Handles Ties	Gaps in Ranking
`ROW_NUMBER()`	Assigns a unique sequence number to each row	❌ Each row gets a unique number (no ties)	❌ No gaps
`RANK()`	Assigns same rank to tied values	✅ Same rank for equal values	✅ Gaps after ties
`DENSE_RANK()`	Assigns same rank to ties	✅ Same rank for equal values	❌ No gaps
NTILE()			
LAG() & LEAD()			

Cache

In **Databricks**, **cache** is used to **store DataFrame or table data in memory** (RAM) - so that repeated access becomes **faster**.

Type	Description	Trigger	Scope	Control
1. Default (Automatic) Cache - Delta Cache	Databricks automatically caches data **on local SSDs** when reading from **Delta or Parquet** files.	Happens automatically when reading data from storage.	**Node-level** (local SSD cache).	Managed by Databricks (you don't control it).
2. Explicit Cache — Using CACHE TABLE or df.cache()	You explicitly ask Spark to **store data in cluster memory (RAM)** for reuse.	Triggered Manually (CACHE TABLE, .cache(), .persist()).	**Cluster-wide** (memory across executors).	Fully user-controlled — you can cache/unpersist manually.

Auto Cache

When query a **Delta table**, Databricks automatically caches **data files** on:

- Local **SSD / NVMe disk** of the cluster nodes (when using Photon / SQL Warehouse)
- Or in **memory** when using interactive notebooks

This speeds up repeated queries because:

- Data is **indexed** and **optimized**
- Subsequent queries **read from local cache instead of cloud storage** (S3 / ADLS / GCS)

Table Format	Auto Cache Enabled?	Reason
Delta Table	✅ **Yes**	Optimized layout + metadata + file format supports caching
Parquet / ORC / CSV / JSON	❌ **No**	Not controlled by Delta metadata → no automatic caching
External Hive tables	❌ No	Client-side caching only if **manual** CACHE TABLE used

Explicit Cache

Explicit Cache manually tells **Spark/Databricks to cache a table or DataFrame**, instead of relying on automatic caching (which only applies to Delta tables).

We use explicit caching to **speed up repeated queries** on **any table or DataFrame**, including:

- Parquet / CSV / External Tables
- Complex DataFrame transformations
- Results of expensive joins / aggregations

When to Use Explicit Cache

Use explicit caching when:

- The same data/table/DataFrame is accessed **multiple times**
- The data **fits within the cluster memory**
- Want to avoid re-reading from storage / recalculating transformations

Avoid caching when:

- The data is **very large and used only once**
- Cluster memory is limited (may cause eviction & slow down)

Cache using SQL:

```
CACHE TABLE my_table;
UNCACHE TABLE my_table;
```

Cache using pyspark code:

```
df = spark.read.table("sales")
df.cache()
df.count()  # <--- triggers caching

df.unpersist()

from pyspark import StorageLevel
df.persist(StorageLevel.MEMORY_ONLY)
```

Cache – Storage Level

When we **cache** or **persist** a DataFrame, Spark needs to know **where** and **how** to store the data.

Storage Level	Description	Memory Usage	Speed	Fault Tolerance
MEMORY_ONLY	Store RDD/DataFrame as deserialized Java objects in memory only. If not enough memory → recompute partitions.	High	Fastest	❌ No
MEMORY_ONLY_SER	Same as MEMORY_ONLY, but stores data in serialized form. Uses less memory but more CPU.	Medium	Moderate	❌ No
MEMORY_AND_DISK	Keep in memory; spill remaining partitions to disk if not enough memory.	Medium	Fast	✅ Yes
MEMORY_AND_DISK_SER	Store serialized data in memory and disk if necessary.	Low	Moderate	✅ Yes
DISK_ONLY	Store data only on disk.	Very Low	Slow	✅ Yes
MEMORY_ONLY_2, MEMORY_AND_DISK_2	Same as above but replicate each partition on 2 nodes.	High	Fast	✅ High

Databricks Workspace

A **workspace** is the *development and execution environment* in Databricks.

It provides:

- Notebooks (Python, SQL, Scala)
- Repos (Git integration)
- Clusters and SQL warehouses (compute)
- Jobs & Workflows (ETL pipelines)
- MLflow tracking
- Workspace folders, dashboards
- Role-based access for compute & workspace objects

Workspace controls:

- Compute access
- Notebook permissions
- Job execution
- Library installation

Workspace does NOT control:

- Table permissions
- Data access governance
- Storage credentials
- These belong to Unity Catalog.

Databricks Unity Catalog

Unity Catalog (UC) is the **central data governance and security layer** for Databricks.

It provides:

- Centralized data access control
- Catalog → schema → table organization
- Permissions for SELECT/INSERT/UPDATE/DELETE
- Row/column-level security
- Data lineage
- Metadata management
- External storage governance

UC controls:

- Catalogs (databases)
- Schemas
- Tables & views
- External locations (ADLS/S3/GCS)
- Storage credentials
- Row/column masking
- Auditing & lineage

UC does NOT handle:

- Compute
- Notebook execution
- Workspace operations

Databricks Unity Catalog & Workspace Rules

- Workspace can attach to only **one UC metastore**
- One UC metastore can attach to **multiple workspaces**
- Multiple UC metastores can live in **one region**
- Workspace controls compute; UC controls data
- Data access = Workspace access + UC permissions
- UC catalog objects are global across attached workspaces
- Cross-metastore queries are not allowed
- Workspace Admin is not a Unity Catalog Admin

UC-Metastore-DEV
- Workspace-DEV
- Workspace-POC

UC-Metastore-UAT
- Workspace-UAT

UC-Metastore-PROD
- Workspace-PROD

Databricks Identities/Principals

In Databricks, **principals** are the *identities* that can be assigned permissions to access data, compute, and workspace resources.

Users

A *user* is an individual person with a Databricks login.

A user has:

- Access to the workspace
- Ability to run clusters (if allowed)
- Ability to access repos
- Access to catalogs, schemas, and tables (if granted)

Service Principals

A *service principal* is a **non-human identity** used by:

- CI/CD pipelines
- Data pipelines
- Azure DevOps / GitHub Actions
- Applications accessing Databricks APIs

Groups

A **group** is a collection of users (and sometimes service principals).

Groups are the **primary way to control permissions** in Databricks.

Groups are used for:

- Workspace access
- Cluster policies
- Catalog permissions (Unity Catalog)
- Workflow/Jobs permissions
- Table read/write permissions
- Repo access permissions

Roles and Permissions in Databricks

Databricks roles fall into **three layers**:

1. Account-Level Roles (Top Level)
2. Workspace-Level Roles
3. Unity Catalog Roles

Account-Level Roles

(Managed in *Databricks Account Console*)

- Account Admin
  - Highest privilege
  - Full control across the **entire Databricks account**

Roles and Permissions in Databricks

Workspace-Level Roles (Managed inside a single workspace)

Access Type / Entitlement	Primary Function	Typical User Persona	Default Status (Non-Admin)
Workspace Admin	Full management of a single workspace (users, settings, resources).	Platform Engineers, Databricks Admins	Not granted
Workspace Access	Access to Data Science & Engineering, and ML features (notebooks, jobs).	Data Scientists, Data Engineers	Granted by default
Databricks SQL Access	Access to Databricks SQL environment (queries, dashboards, SQL warehouses).	Data Analysts, BI Users	Granted by default
Consumer Access	Simplified, read-only access for consuming shared assets (dashboards).	Business Stakeholders, Executives	Not granted (must be explicitly assigned)
Unrestricted Cluster Creation	Ability to create clusters with any configuration (no policy required).	Workspace Admins, Platform Admins	Not granted (must be explicitly assigned)

Roles and Permissions in Databricks

Unity Catalog Roles

Account Admin (Highest Level):

- Manages the entire Databricks account (users, groups, service principals).
- Can create and manage **Unity Catalog Metastores** and link them to workspaces.
- Can assign the **Metastore Admin** role.

Metastore Admin (UC Central Control):

- Has full control over the **Unity Catalog Metastore**.
- Can manage **Storage Credentials** and **External Locations**.
- Can grant top-level privileges like CREATE CATALOG on the metastore.
- Has the ability to read and write all data governed by the metastore (if they grant themselves the necessary privileges).

Workspace Admin (Workspace Control):

- Manages users, clusters, and jobs within a **single workspace**.
- In an auto-enabled UC workspace, they are granted default privileges

  on the attached workspace catalog (e.g., CREATE EXTERNAL LOCATION, ownership of the workspace catalog).

Roles and Permissions in Databricks

Level	Object	Description	Key Privileges
Metastore	METASTORE	The top-level container for all metadata, centrally managed across the account.	CREATE CATALOG, CREATE EXTERNAL LOCATION, CREATE STORAGE CREDENTIAL
1st Level	CATALOG	The first layer, typically used to organize data by organizational unit, environment (e.g., dev, prod), or team.	USE CATALOG, CREATE SCHEMA, SELECT (inherited)
2nd Level	SCHEMA	(Also called Database) The second layer, used to organize related data assets within a catalog.	USE SCHEMA, CREATE TABLE, CREATE VOLUME, SELECT (inherited)
3rd Level	TABLE / VIEW / VOLUME / FUNCTION / MODEL	The lowest level, representing the actual data or compute logic.	SELECT, MODIFY, READ VOLUME, WRITE VOLUME, EXECUTE (on functions/models)