Databricks

Databricks is a cloud-based data and AI platform built on top of Apache Spark that provides an all-in-one environment for big-data engineering, data science, machine learning, and analytics.

Databricks is a fully managed Spark environment. You don't install or maintain clusters—Databricks automatically handles infrastructure, scaling, and Spark version upgrades.

Why don't we create a SparkSession manually in Databricks?

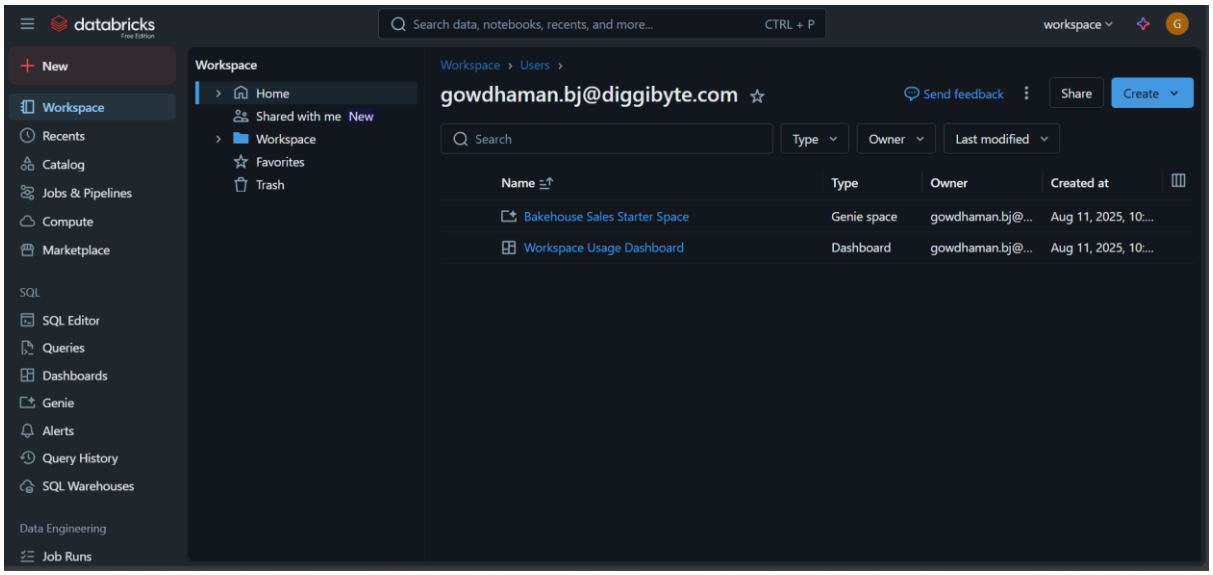
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("MyApp").getOrCreate()

- When a Databricks cluster starts, it automatically launches a Spark application on the driver node and pre-creates a global SparkSession object named **spark**. (When you attach a notebook to a Databricks cluster, Databricks starts a **driver process** with a Spark application already running.)
- This object is automatically available to every notebook that attaches to that cluster. Because the cluster is already running Spark with all configurations applied, we can directly use spark for reads, writes, and transformations without calling SparkSession.builder.
- If we need custom settings, we can either set them at the cluster level or create a new session from the existing one.

1.1 Workspace UI

The **Workspace UI** is the main interface you see when you log in to Databricks. It organizes all your assets—like notebooks, datasets, jobs, and clusters—into a navigable folder structure.



1.2 Overview of UI Components

Key components visible in the Databricks Workspace UI:

- **Sidebar Navigation** – Quick access to Workspace, Data, Compute, Workflows, and other resources.
- **Main Panel** – Displays content (e.g., notebook editor, cluster configuration, data explorer).

Sidebar **(Left panel)**

- **New** – Create new notebooks, jobs, dashboards, etc.
- **Workspace** – Your files, notebooks, and shared items.
- **Recents** – Recently accessed notebooks/tables.
- **Catalog** – Data management (Unity Catalog if enabled).
- **Jobs & Pipelines** – Workflow automation and scheduling.
- **Compute** – Cluster management.
- **Marketplace** – Access third-party datasets and solutions.

SQL Section – SQL Editor, Queries, Dashboards, Genie, Alerts, Query History, SQL Warehouses.

Data Engineering Section – Job Runs, Data Ingestion etc.

Main Content Area

- Shows your **Workspace folder structure** and files.
- Current path: /Workspace/Users/[your_user]
- Example files:
  - **Bakehouse Sales Starter Space** (Genie space)
  - **Workspace Usage Dashboard** (Dashboard)

Top Bar

- **Search bar** – Find data, notebooks, dashboards, etc.
- **User profile menu** – Settings, account info.
- **Create button** – Add a new notebook, job, query, etc.
- **Share button** – Share current workspace folder or files.
- **Send feedback** – Provide feedback to Databricks.

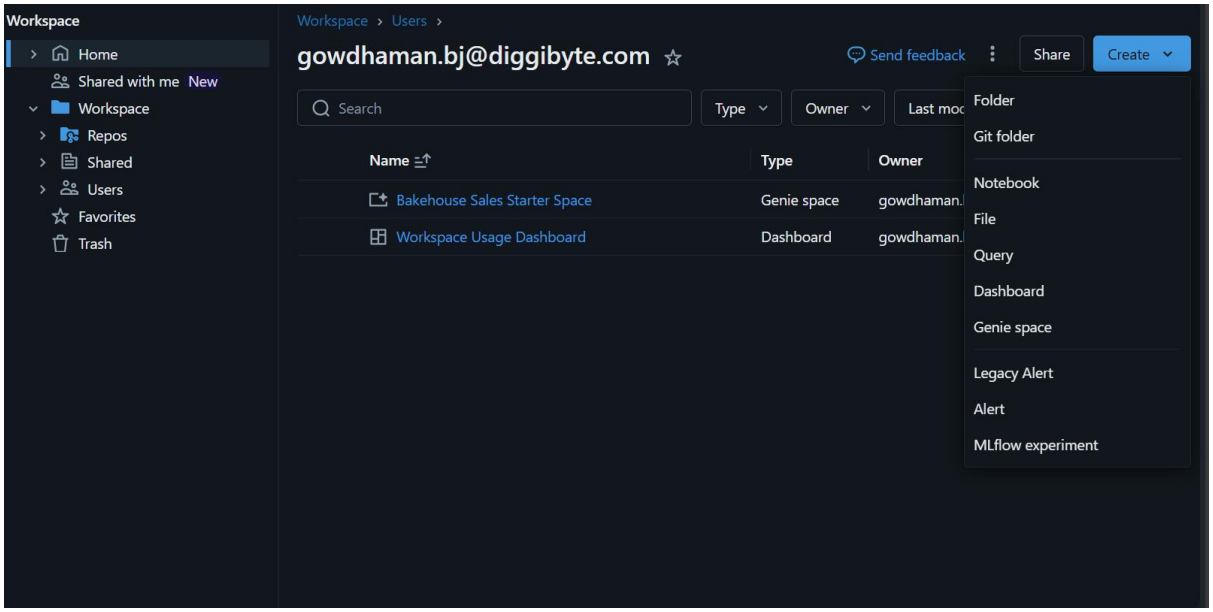
Filters (Above file list)

- **Type, Owner, Last Modified** – Filter workspace items.

1.3 Workspaces

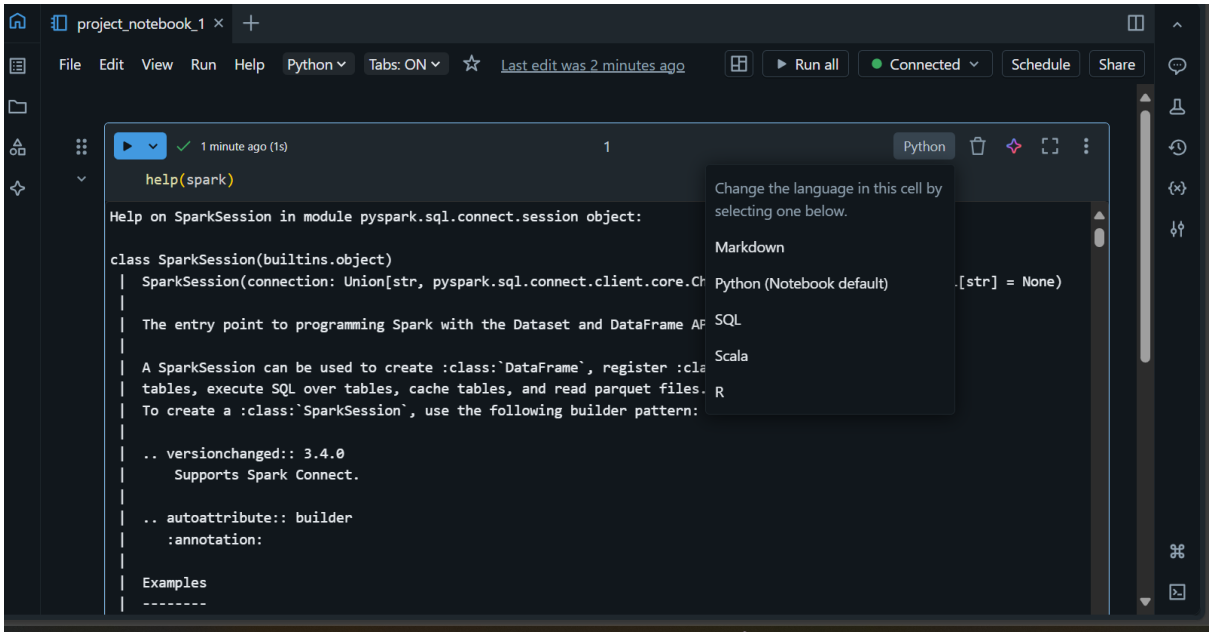
A **Workspace** is the environment in Databricks where users store and organize their projects.

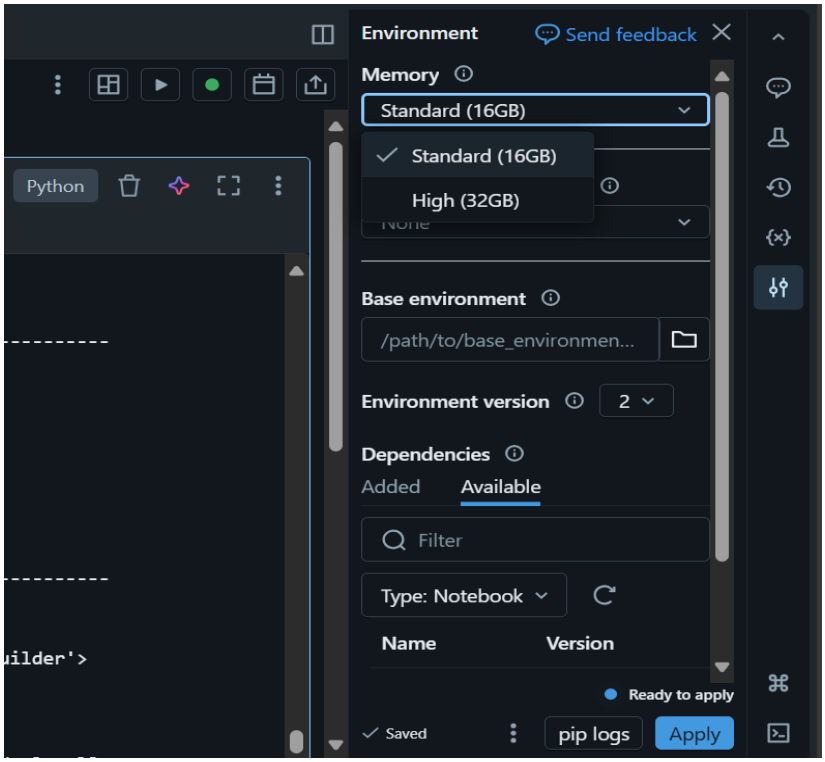
- Contains folders and items such as notebooks, libraries, dashboards, and experiments.
- Supports collaborative development — multiple users can work together.
- Can be personal (only visible to you) or shared (visible to the team).
- Can have access controls for fine-grained permissions.



1.4 Notebooks

- Interactive documents for writing and running code (supports **Python, SQL, R, Scala**).
- Supports **mixing code and markdown** for documentation.
- Can run in **interactive mode** or as a **scheduled job**.
- Attached to a **cluster** for execution.
- Key features:
    - Rich text formatting (Markdown)
    - Visualization (charts, tables)
    - Version history
    - Collaboration (multiple users editing)





1.5 Libraries:

In your current Databricks **Free Edition**, the **Libraries** section isn't visible as a separate menu item because:

- In **Databricks**, libraries are not listed in the left sidebar.
- They are **managed inside a cluster** (or job) configuration.

Where to find Libraries:

1. Go to **Compute** in the left sidebar.
2. Click the **cluster** you want to use.
3. Inside the cluster page, you'll see a **Libraries** tab.
4. From there, you can:
    - Install from **PyPI** (Python packages)
    - Install from **Maven** (Java/Scala)
    - Upload JAR/Wheel/Egg files from **DBFS** or local

1.6 Data

Used to explore and upload data in CSV, Parquet, Delta formats.

- Browse databases and tables.
- Upload files from local or cloud storage.
- Create Delta Tables.
- Query using SQL editor.
- Central location for managing datasets.
- Supports:
    - **Tables** (Delta, Parquet, CSV, etc.)
    - **Databases** (schemas)
    - **External sources** (S3, Azure Blob, GCS, JDBC)
- Provides:

    - Data Explorer to browse and preview tables.
    - Metadata & schema info.
    - Create tables from files directly in UI.
    - Data permissions via Unity Catalog.

1.7 Clusters

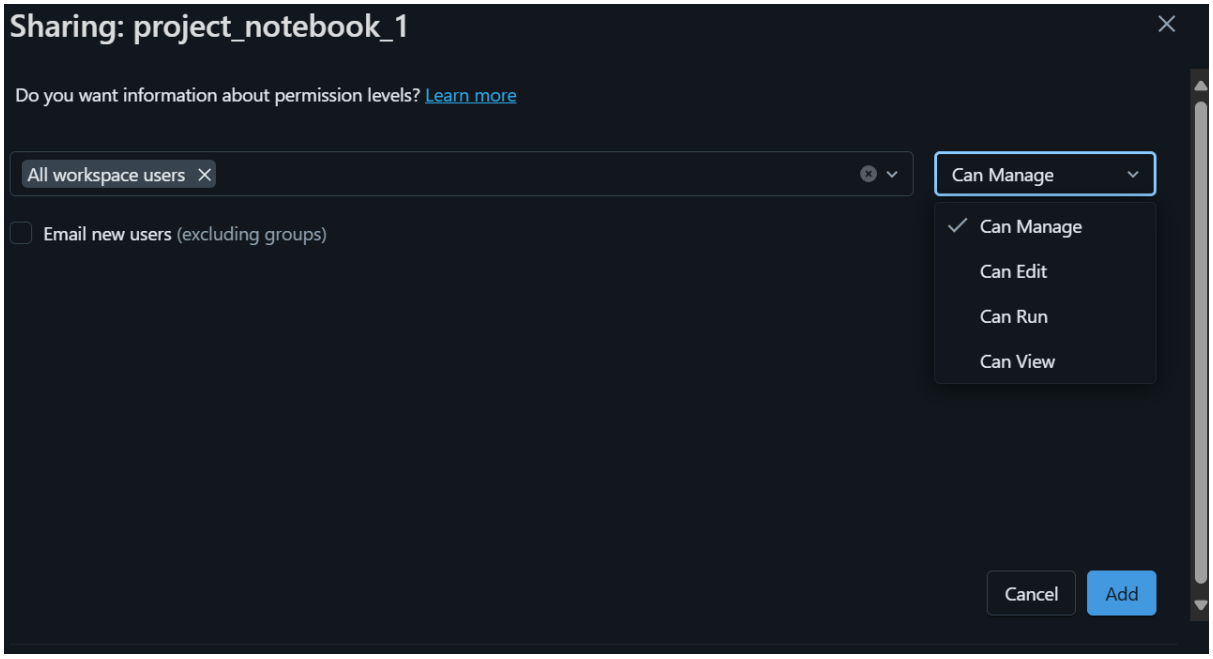
- **Clusters** are collections of compute resources used to execute notebooks, jobs, and queries in Databricks.
- Virtual machines where your code executes.
- Types:
    - **Interactive Clusters** (for development/testing)
    - **Job Clusters** (spawned automatically for scheduled jobs)
- Configurable settings:
    - Node type & size
    - Autoscaling
    - Termination after inactivity
- Attached to:
    - Notebooks
    - Jobs
    - Workflows
- Supports installing libraries directly.

Managing Clusters

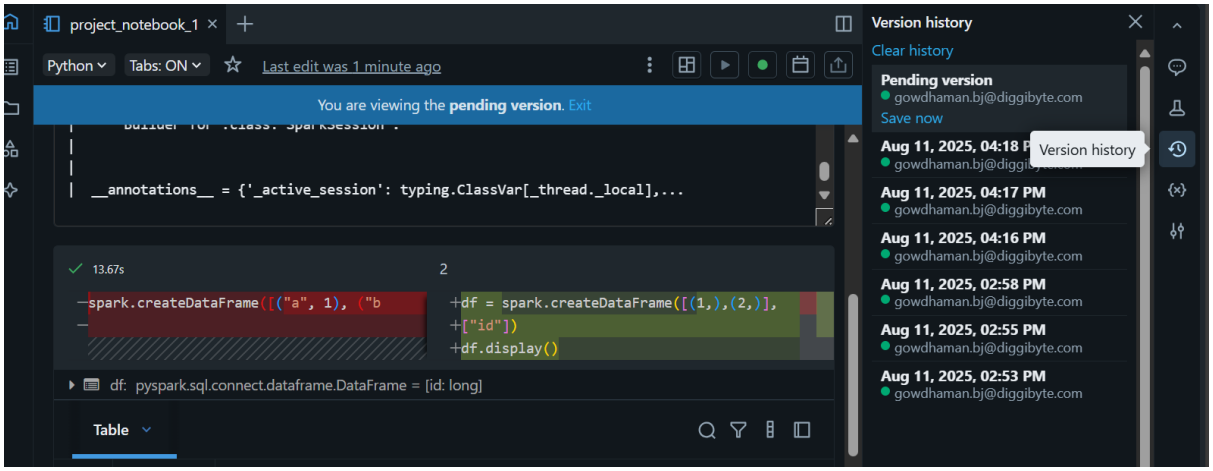
- **Clusters** are compute resources that execute notebooks and jobs.
- **Key management tasks:**
    - Create clusters with specific node types, sizes, and configurations.
    - Set **auto-scaling** and **auto-termination** to control costs.
    - Restart or terminate when needed.
    - Attach/detach notebooks for execution.
- **Types:** Interactive (development) and Job clusters (scheduled/automated).

Sharing Notebooks

- Notebooks can be shared with specific users or groups.
- Permissions:
    - Read – View only.
    - Edit – Modify code and markdown.
    - Manage – Edit and change permissions.
- Share via workspace path or export as HTML, IPython (.ipynb), or source file.



Version: Databricks automatically keeps **version checkpoints** for notebooks



3. Cluster Creation

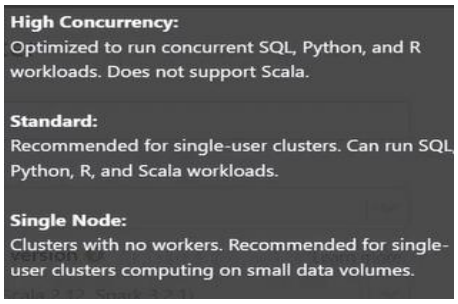
1. A **cluster** in Databricks is a set of virtual machines (driver + worker nodes) used to run notebooks, jobs, and queries.
2. **Steps to create a cluster:**
    1. Go to **Compute** in the Databricks workspace.
    2. Click **Create Cluster**.
    3. Choose configuration options (name, type, worker nodes, runtime version).
    4. Click **Create** — Databricks provisions the resources automatically.

3.1 Cluster Configuration

- **Key settings during cluster creation:**
    - **Cluster name** – For identification in the workspace.

- **Databricks Runtime version** – Determines available libraries and optimizations (e.g., Spark version, ML, GPU).
- **Node types** – Defines hardware specs (CPU, memory, GPU).
- **Termination settings** – Auto-shutdown after inactivity to save cost.
- **Environment variables & init scripts** – For custom setup before starting.

3.2 Cluster Type (Standard, High Concurrency)



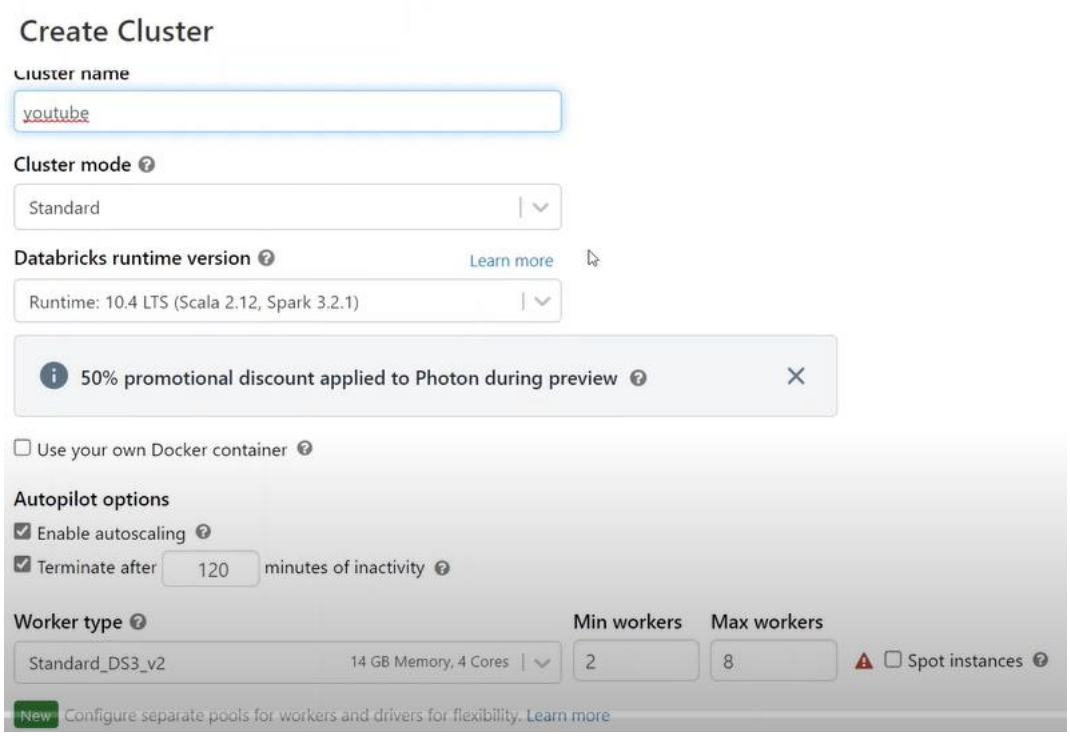
- **Standard Cluster**
  - Single-user or small-team workloads.
  - Supports Python, Scala, SQL, R.
  - Best for ETL, batch jobs, and single-team development.
- **High Concurrency Cluster**
  - Optimized for serving multiple users simultaneously.
  - Better for BI tools, dashboards, and SQL analytics.
  - Uses **fine-grained resource sharing** for better concurrency.

3.3 Worker Node Types

- Worker nodes perform the actual data processing.
- Choose based on workload:
  - **General Purpose** – Balanced CPU & memory (ETL, general analytics).
  - **Memory Optimized** – Large memory for heavy joins, caching, ML.
  - **Compute Optimized** – Higher CPU for processing-heavy workloads.
  - **GPU Enabled** – For deep learning, image processing, NLP models.

3.4 Auto Scaling Options

- Allows cluster to **scale up or down** based on workload.
- **Benefits:**
  - Saves cost when demand is low.
  - Handles spikes in workload automatically.
- **Settings:**
  - **Min Workers** – Lowest number of worker nodes.
  - **Max Workers** – Highest number of worker nodes allowed.
  - Spark dynamically adds or removes nodes as needed.



Catalog

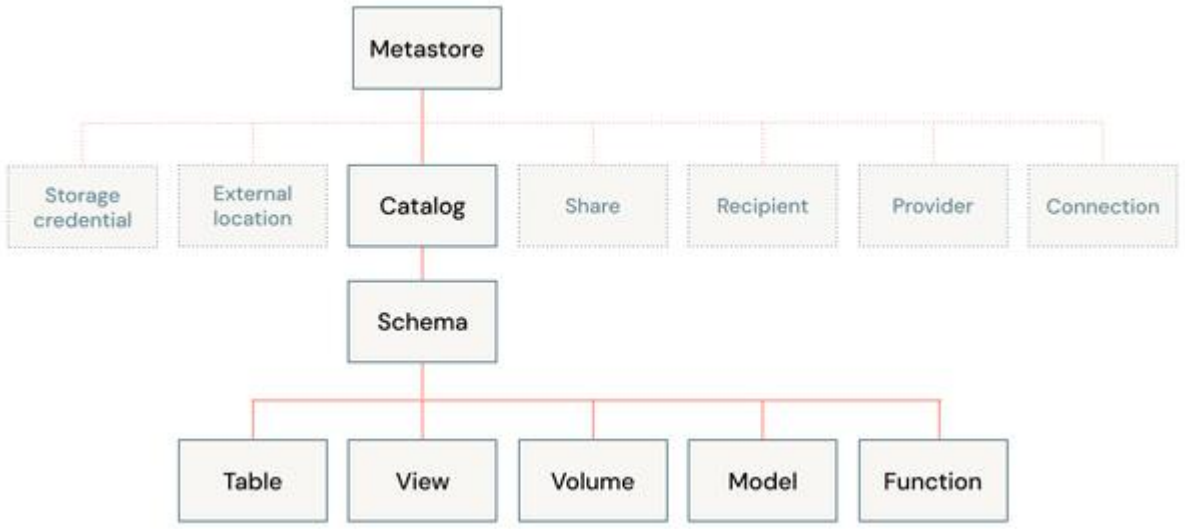
What is Unity Catalog?

Unity Catalog is Databricks' unified data governance layer. It provides centralized management of data assets (tables, files, views, and machine learning models) across all Databricks workspaces and cloud storage.

Why Unity Catalog?

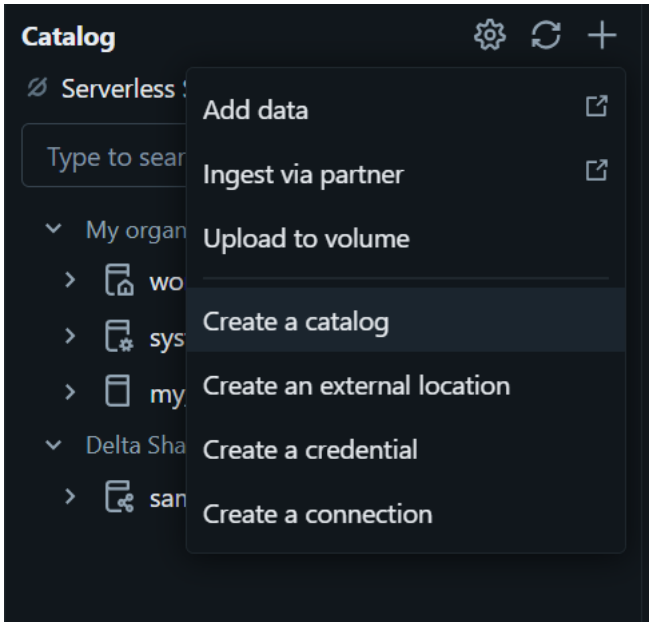
- Before Unity Catalog, each workspace managed its own metadata and permissions.
- This caused fragmented access control, duplicated datasets, and security gaps.
- Unity Catalog standardizes governance across an organization, making compliance easier.

Feature	Description
Centralized Governance	Manage permissions for all data assets in one place.
Fine-Grained Access Control	Grant access at table, column, or row level, not just entire databases.
Cross-Workspace Sharing	Share data securely across multiple Databricks workspaces or teams.
Audit & Lineage	Track who accessed what data and how it changed over time.
Support for Delta Lake	Works seamlessly with Delta tables for ACID transactions and time travel.



Create catalog → create schema → create table or create view or create volume or model or function.

In free edition, we have only access to create table, volume and model



What Can Be Stored in Each?

(a) Table

- Structured data in **rows & columns**.
- Supports formats: Delta, Parquet, CSV, JSON, etc.
- Examples:
  - Sales transactions
  - Customer master data
  - Aggregated analytics output
- Stored in Databricks-managed storage (managed tables) or external storage (external tables).

(b) Volume

- A **folder or directory** in Unity Catalog for storing **unstructured or semi-structured data**.
- Think of it as a **managed file store**.
- Can store:
  - Images
  - PDFs
  - Raw CSV/JSON files
  - Sensor logs
- Useful for ML projects that need raw files, not just tabular data.

(c) Model

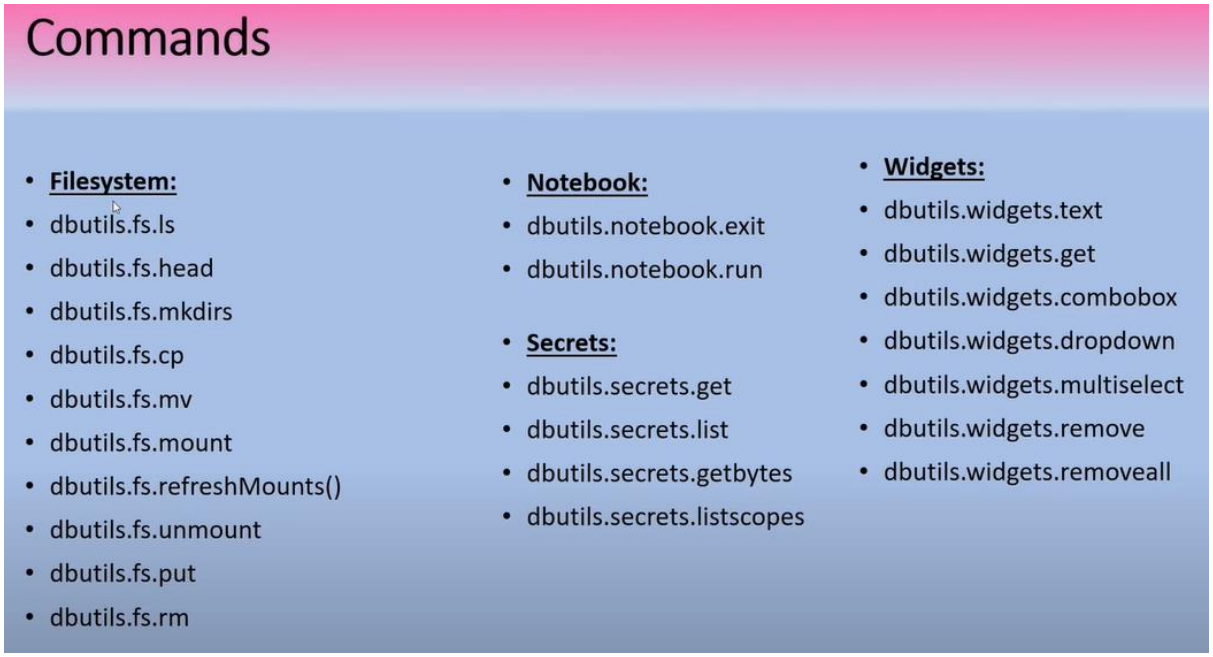
- Stores **machine learning models** registered in the **Databricks Model Registry**.
- Contains:
  - Model artifacts (trained weights, serialized files like .pkl or MLflow model format)
  - Versioning metadata
  - Associated training parameters and metrics
- Example:
  - A trained XGBoost fraud detection model
  - A PyTorch image classification model
  - A Scikit-learn regression model

dbutils

Databricks Utilities (dbutils) is a powerful tool for interacting with Databricks workspaces, managing files, and performing various operations. Below are some commonly used dbutils commands categorized by their functionalities:

Module	Purpose
dbutils.fs	File system operations (DBFS)
dbutils.secrets	Access secrets from secret scopes
dbutils.widgets	Create interactive widgets in notebooks
dbutils.library	Install or uninstall libraries
dbutils.notebook	Call and run other notebooks

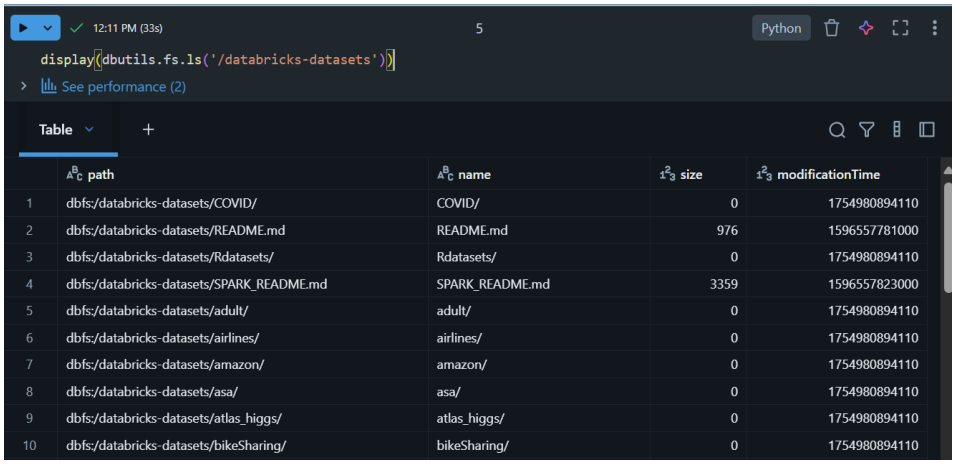
- **fs: DbfsUtils** -> Manipulates the Databricks filesystem (DBFS) from the console.
- **jobs: JobsUtils** -> Utilities for leveraging jobs features.
- **library: LibraryUtils** -> Utilities for session isolated libraries.
- **meta: MetaUtils** -> Methods to hook into the compiler (EXPERIMENTAL).
- **notebook: NotebookUtils** -> Utilities for the control flow of a notebook (EXPERIMENTAL).
- **preview: Preview** -> Utilities under preview category.
- **secrets: SecretUtils** -> Provides utilities for leveraging secrets within notebooks.
- **widgets: WidgetsUtils** -> Methods to create and get bound value of input widgets inside notebooks.



File System Utilities (dbutils.fs)

Used for file and directory operations.

1. **List files** in a directory: dbutils.fs.ls("path/to/directory")
2. **Copy files**: dbutils.fs.cp("source/path", "destination/path", recurse=True)
3. **Move files**: dbutils.fs.mv("source/path", "destination/path", recurse=True)
4. **Remove files or directories**: dbutils.fs.rm("/path/to/file_or_directory", recurse=True)
5. **Mount a storage**:
   dbutils.fs.mount(
   source="s3://bucket-name",
   mount_point="/mnt/mount-name",
   extra_configs={"fs.s3a.access.key": "ACCESS_KEY", "fs.s3a.secret.key": "SECRET_KEY"}
)



Accessing dbutils in Notebooks:

- Open a Databricks notebook cell (Python or Scala).

- Type dbutils and call functions directly.
- Works only when the notebook is attached to a **running cluster** or **SQL warehouse**

```
dbutils.notebook.run("/Workspace/Users/gowdhaman.bj@diggibyte.com/sample1/project_notebook_1", timeout_seconds=60,
arguments={"key": "value"})
```

Notebook Workflows

Start time	End time	Notebook path	Duration	Status	Error code	Run parameters
Aug 11, 2025, 05:24 PM	Aug 11, 2025, 05:25 PM	...ct_notebook_1	36s	Succeeded		key: value

```
dbutils.notebook.exit("Processing complete")
```

```
Notebook exited: Processing complete
```

dbutils.widgets

dbutils.widgets provides utilities for working with notebook widgets. You can create different types of widgets and get their bound value.

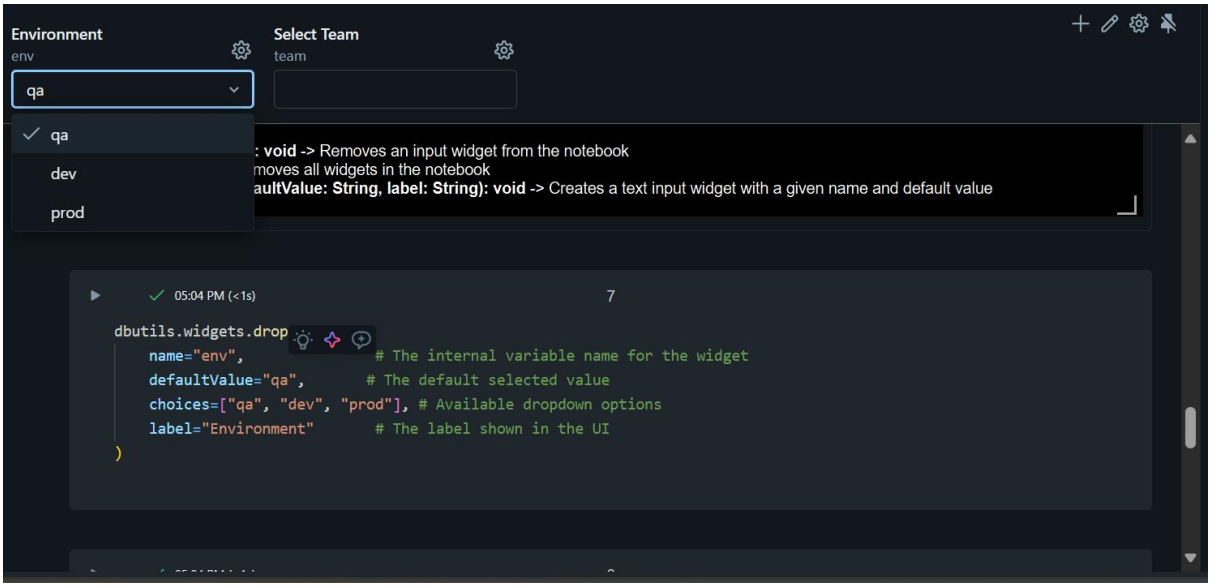
1 Dropdown, 2 combobox, 3 multiselect, 4 input

1. Create Input widgets - Shows a dropdown with fixed options.

```
2 minutes ago (<1s)                                                    7
dbutils.widgets.dropdown(
    name="env",               # The internal variable name for the widget
    defaultValue="qa",       # The default selected value
    choices=["qa", "dev", "prod"], # Available dropdown options
    label="Environment"       # The label shown in the UI
)
```

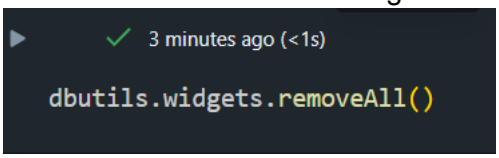
```
2 minutes ago (<1s)                                                    8
dbutils.widgets.get("env")
```

```
'qa'
```

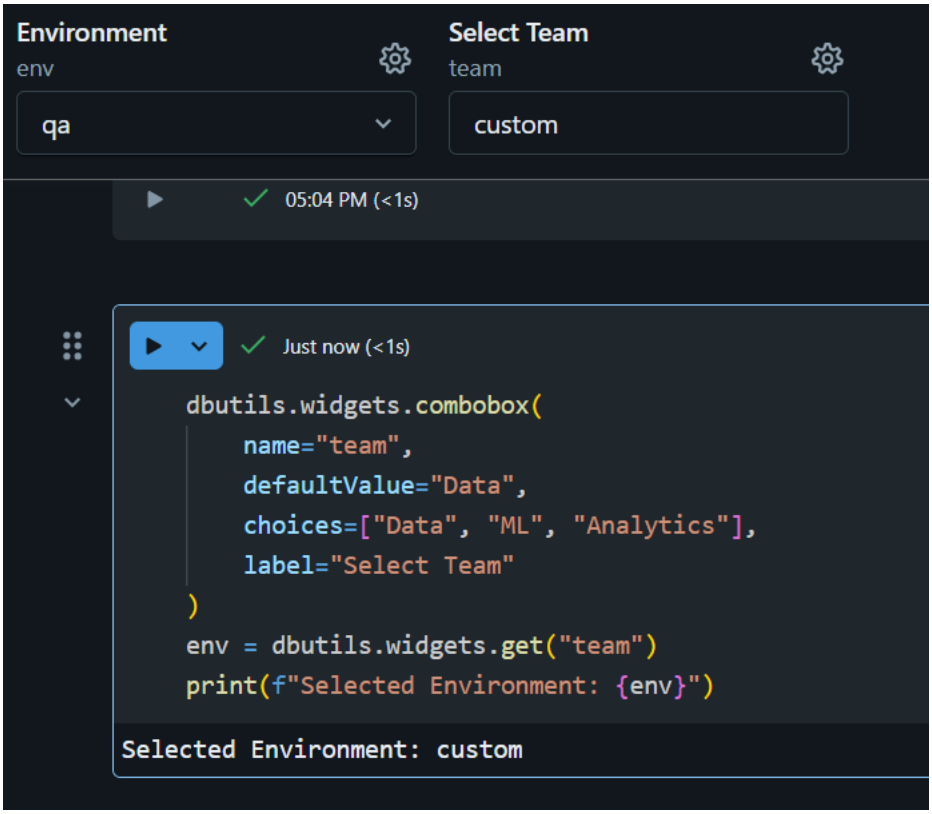


widgets in Databricks are **stateful**. Once created, their value is preserved until you:

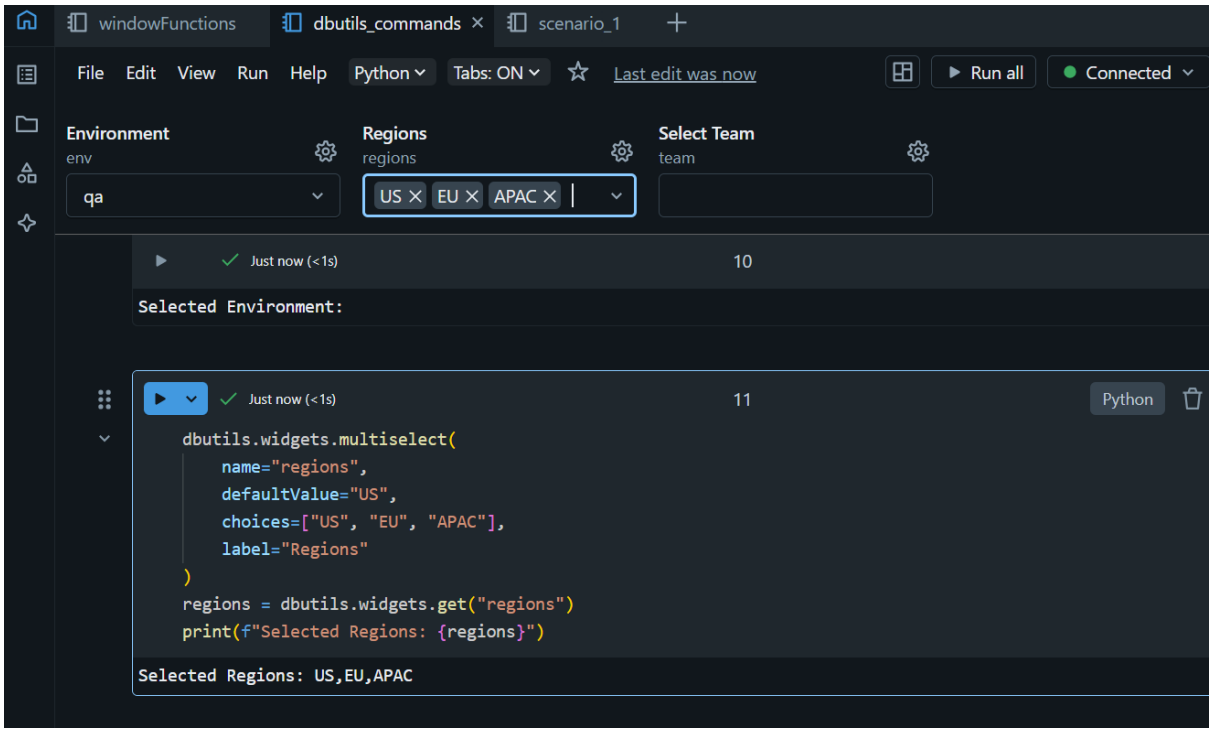
- Manually change the selection in the UI, or
- Remove and recreate the widget.



2. Combobox - Like a dropdown but allows typing a custom value.



3. Multiselect - Select multiple options.



4. Input

5. Get all widgets

6. Remove widgets
   - dbutils.widgets.remove("input_path") # Remove specific widget
   - dbutils.widgets.removeAll() # Remove all widgets

Note:

Widget values are always returned as **strings** — cast if needed (`int()`, `float()`).

Widgets persist in a notebook until removed or reset.

Dbutils.fsmount()

The dbutils.fs.mount() command in Databricks is used to mount a cloud storage location (like AWS S3, Azure Blob Storage, or ADLS) to the Databricks File System (DBFS). This allows you to interact with cloud storage using familiar file paths within Databricks.

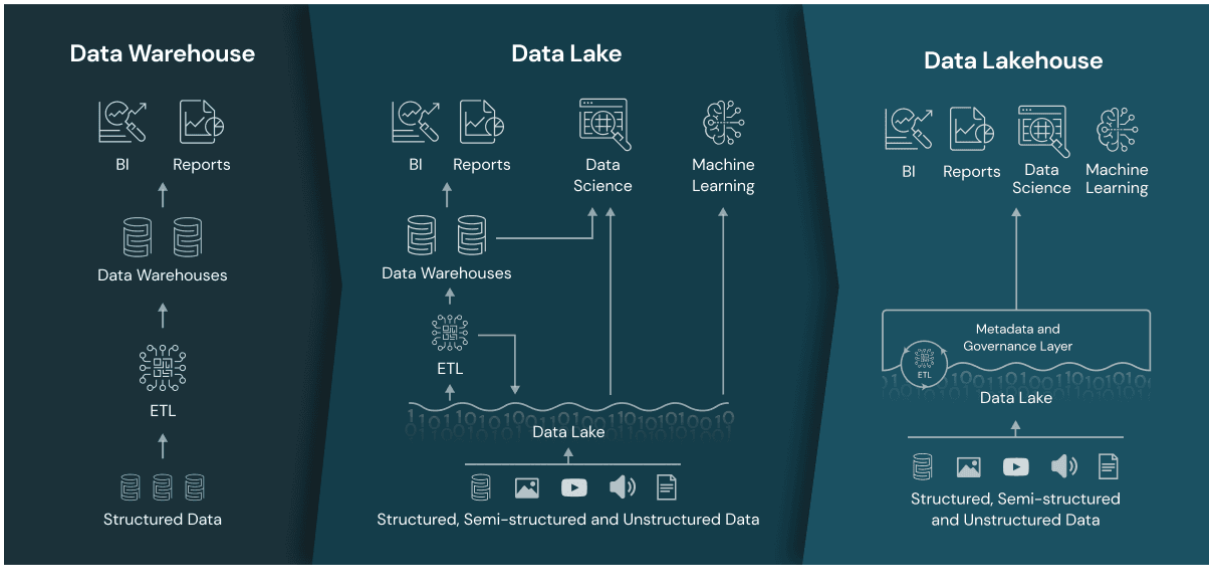
```
dbutils.fs.mount(
    source: str,
    mount_point: str,
    extra_configs: Optional[Dict[str, str]] = None
)
```

Parameters:

- **source**: The URI of the cloud storage location (e.g., S3 bucket or Azure Blob container).
- **mount_point**: The DBFS path where the storage will be mounted.
- **extra_configs**: A dictionary containing authentication or configuration details (e.g., keys, tokens).

```
dbutils.fs.mount(
   source="s3a://your-bucket-name",
   mount_point="/mnt/your-mount-point",
   extra_configs={
      "fs.s3a.access.key": "your-access-key",
      "fs.s3a.secret.key": "your-secret-key"
   }
)
```

Data warehouse vs Data lake vs Data Lakehouse



What Is the Different Between a Data Warehouse, a Data Lake, and a Data Lakehouse?

Data Structure:

- Data Warehouses hold structured data, whereas
- Data lakes and Data Lakehouse store both structured and unstructured data.

Processing:

- For data analysis, data warehouses employ SQL-based processing, whereas
- data lakes and data lakehouses use a variety of processing engines, such as Spark and Databricks.

Storage:

- Data warehouse store structured data, whilst

- Data lake and Lakehouse store raw data.

Schema:

- Data warehouses feature a predefined schema, whereas
- data lakes and lakehouses employ a schema-on-read method, where the schema is built during analysis.

Use case: Data warehouses are often used for business intelligence and reporting, whereas data lakes and lakehouses are used for advanced analytics such as machine learning and data science.

Data Warehouse – Structured & Optimized for Analytics

- Purpose: Store structured data for reporting, BI, and analytics. (DML operation is easy because structured).
- Data Type: Tables with fixed schema (rows & columns).
- Typical Sources: ERP, CRM, transactional systems (after ETL).
- Performance: Very fast queries due to indexing, pre-aggregation, and columnar storage.
- Examples: Snowflake, Amazon Redshift, Google BigQuery, Teradata.
- Limitations: Not great for unstructured or semi-structured data (images, JSON, logs).

Example Scenario:

- A bank storing monthly customer transactions for regulatory reporting.

Data Lake – Flexible, Raw Storage

- **Purpose:** Store **any type of data** — structured, semi-structured, unstructured — cheaply at scale. (DML operation is difficult because unstructured).
- **Data Type:** CSV, JSON, Parquet, images, videos, logs.
- **Typical Sources:** Direct ingestion from IoT, APIs, raw dumps from databases.
- **Performance:** Slow for analytics without extra processing, since data isn't organized for querying.
- **Examples:** AWS S3, Azure Data Lake Storage (ADLS), Google Cloud Storage (GCS), Hadoop HDFS.
- **Limitations: Without governance**, can turn into a **data swamp** (hard to find, low quality).

Example Scenario:

- An e-commerce company storing **clickstream logs, images, videos, sensor data** for future analytics.

Data Lakehouse – Best of Both Worlds

- **Purpose:** Combine **data lake flexibility** + **data warehouse reliability/performance**.
- **Data Type:** Any type (structured, semi-structured, unstructured).
- **Features:** ACID transactions, schema enforcement, indexing, high-performance queries **on top of raw files**.
- **Examples:** Databricks Lakehouse, AWS Athena with Iceberg, Snowflake's Unistore.

Example Scenario:

- A retail company storing **raw customer clickstream** + **structured sales data** in one place.

Delta Lake – A Technology to Build a data lake

- **Purpose:** Add **ACID transactions, schema enforcement, and time travel** to your data lake.
- **How it Works:** Stores data in Parquet files + keeps a **_delta_log** folder to track all changes.
- **Benefits:** Allows UPDATE, DELETE, MERGE on big data, supports time travel, unifies batch & streaming.
- **Examples:** Open-source Delta Lake (Databricks, Spark).

Data Warehouse	Data Lake	Delta Lake
Only Structured Data	Structured/Semi-structured/Unstructured	Structured/Semi-structured/Unstructured/Streaming
Schema-on-Write	Schema-on-Read	Schema-on-Read
Supports ACID Transaction	Minimal support to ACID Transactions	Supports ACID Transaction
Does not corrupt the system	Leaves system in corrupted state	Does not corrupt the system

While uploading file, if corrupted **data lake** will leave the process and the system in corrupted state.

What is delta lake?

Delta Lake is an **open-source storage layer** built on top of data lakes (like ADLS, S3, or Blob) that brings **reliability**, **consistency**, and **performance** to big data processing.

Delta lake is additional layer sitting on top of data lake which provides feature like ACID transaction, etc.

Delta Lake is an open-source storage layer that brings ACID transactions, scalable metadata handling, and schema enforcement to data lakes. It extends Parquet files with a file-based transaction log, enabling reliable, performant tables in a Lakehouse architecture.

If they ask specifically in a Databricks context, you can add:

In Databricks, Delta Lake is the default table format for the Lakehouse, providing features like time travel, upserts, deletes, and unified batch and streaming, all on top of cloud object storage.

Delta Lake is immutable

Delta Table:

What is delta table?

A **Delta table** is a data table stored in a **Delta Lake format**, which is an open-source storage layer built on top of Parquet.

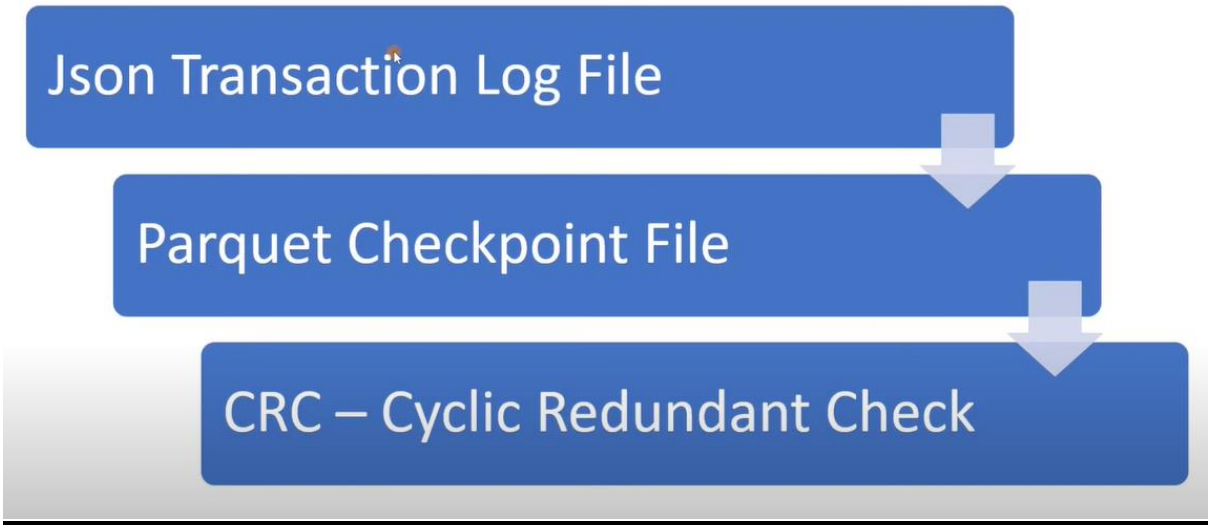
- It stores data as **Parquet files** with an additional **transaction log (_delta_log)** that provides **ACID transactions**, schema handling, and time travel.
- Parquet file always returns the latest version.

Physically, a Delta table is a folder in storage that contains:

1. **Parquet data files** – the actual data.
2. A **_delta_log** folder – the transaction log. This log stores:
    - **JSON files** – one per commit, describing changes like file additions or deletions.
    - **Checkpoint Parquet files** – periodic snapshots of the table state to speed up reads.

When you query a Delta table, Delta Lake reads the latest checkpoint and any newer JSON logs to determine the current snapshot and then reads only the necessary Parquet files.

This design lets Delta tables support time travel, incremental updates, and concurrent writes reliably.



Json transaction log file: every operation is logged as 0,1,2,3,…

Parquet checkpoint file: for every 10 log's one parquet checkpoint file is created.

Cyclic redundant check: prevent accidental damage to data

Features of delta table

Feature	Why It Matters
ACID Transactions	Ensures **Atomicity, Consistency, Isolation, Durability**, so concurrent reads/writes won't corrupt data.
Schema Enforcement	Validates incoming data against the defined schema, blocking bad or unexpected columns/types.
Schema Evolution	Allows controlled updates to schema (e.g., adding new columns) without breaking existing data.
Time Travel	Query or restore data **as of a version or timestamp** to recover from bad writes or compare historical data.
Upserts & Deletes (MERGE)	Support MERGE, UPDATE, and DELETE statements directly, enabling Slowly Changing Dimensions (SCD) and GDPR-style record deletions.
Data Versioning	Every write creates a new **transaction log entry**, so each dataset version is tracked.
Efficient Reads (Data Skipping)	Maintains min/max statistics in transaction logs so queries can skip irrelevant files and scan less data.
Unified Batch & Streaming	The same table supports **streaming reads and writes**—no separate pipelines.
Optimize & Z-Ordering	Commands like OPTIMIZE and ZORDER reduce small files and improve query performance.

Feature	Why It Matters
Vacuum (File Cleanup)	Safely removes obsolete files after a retention period to save storage space.
Integration with Unity Catalog	Provides fine-grained access control and governance across multiple workspaces.

Delta table structure

A Delta table has **two main parts** in its storage location:

1. **Data files**
    - Stored as **Parquet** files (.parquet) in the table's directory.
    - Contain the actual column data.
2. **Transaction log** (_delta_log folder)
    - Contains JSON files **and** checkpoint Parquet files that track changes over time.

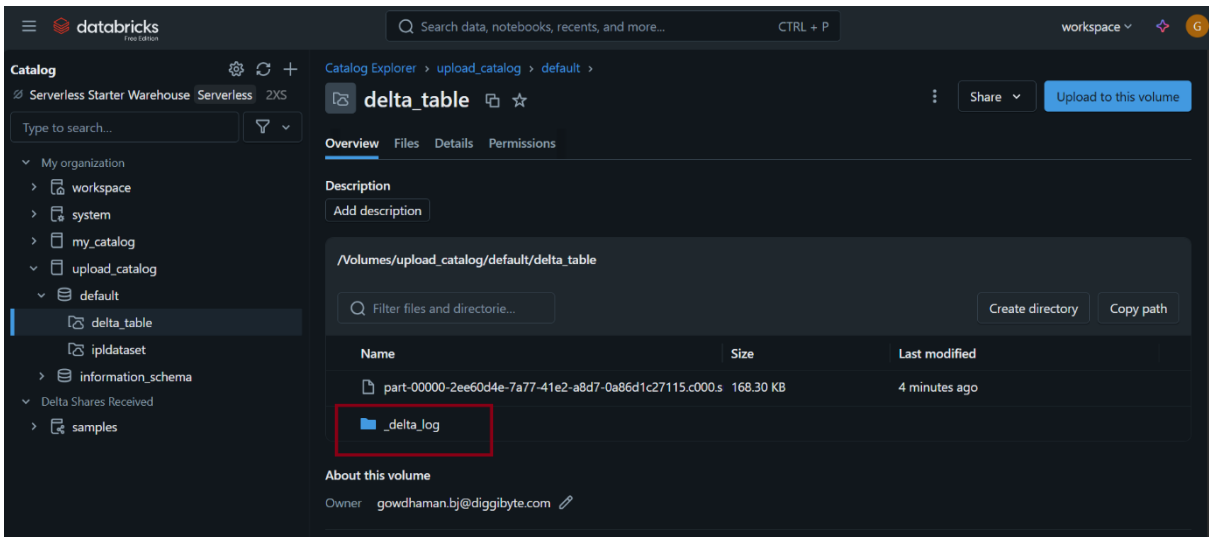
Inside _delta_log

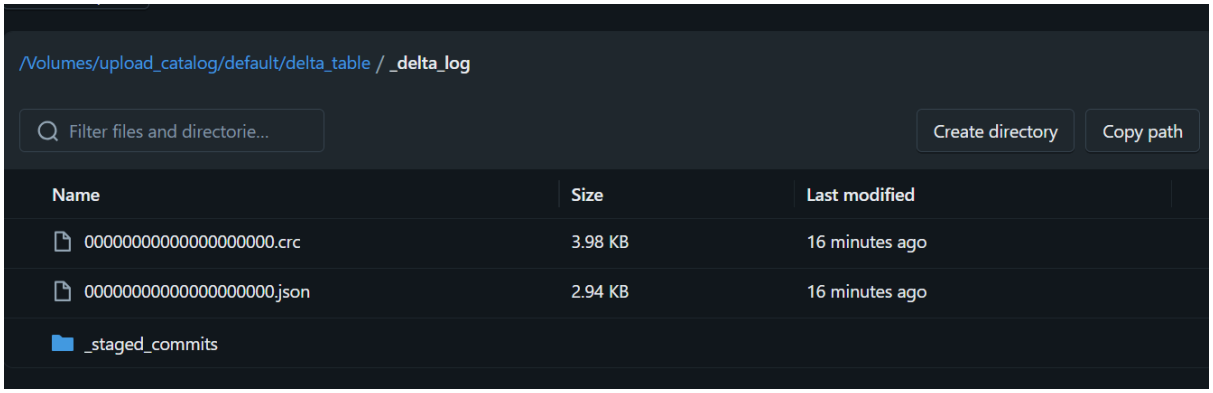
- **JSON log files** (00000000000000000010.json, etc.)
    - Each file represents a **single commit** to the table.
    - Contains metadata and actions (e.g., "added these data files", "removed these files").
    - Ordered by version number.
- **Checkpoint Parquet files** (00000000000000000010.checkpoint.parquet)
    - Store a **snapshot of the full table state** at a given version in an optimized format.
    - Speeds up reading the latest state so you don't need to read every JSON file from version 0.

We create the delta table by this way, but in free edition we are not having access. So, we created by read the csv file and created delta table.


read the csv
2 minutes ago (12s)
4
df_read_csv = spark.read.format("csv").option("header", "true").load("/Volumes/upload_catalog/default/ipldataset/sales_data_05092024.csv")
df_read_csv.display()
See performance (1)
Optimize
df_read_csv: pyspark.sql.connect.dataframe.DataFrame = [Order_ID: string, Customer_Name: string ... 9 more fields]
Table


4 minutes ago (6s)
5
df_read_csv.write.format("delta") \
.mode("overwrite") \
.save("/Volumes/upload_catalog/default/delta_table")
See performance (1)

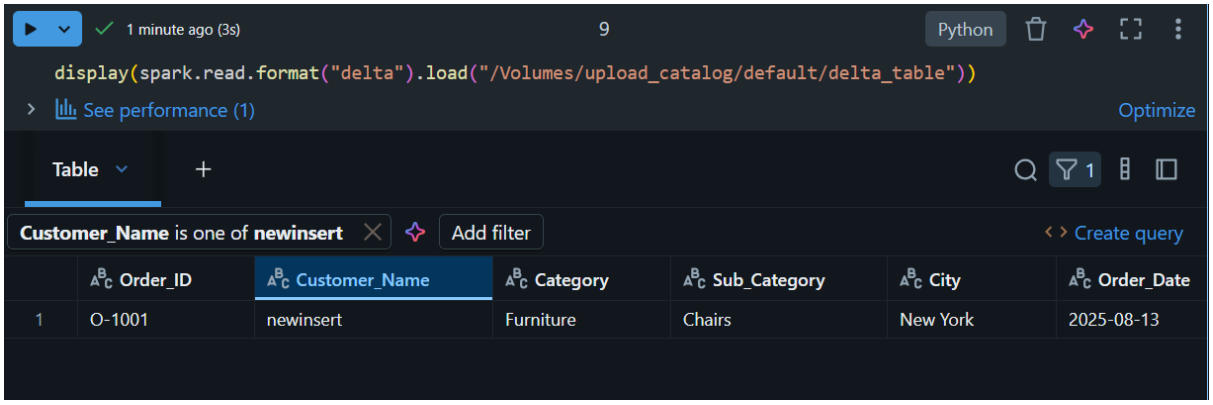

databricks
Search data, notebooks, recents, and more...
CTRL + P
workspace
Catalog
Serverless Starter Warehouse Serverless 2XS
Type to search...
My organization
workspace
system
my_catalog
upload_catalog
default
delta_table
ipldataset
information_schema
Delta Shares Received
samples
Catalog Explorer › upload_catalog › default ›
delta_table
Share
Upload to this volume
Overview
Files
Details
Permissions
Description
Add description
/Volumes/upload_catalog/default/delta_table
Filter files and directorie...
Create directory
Copy path
Name
Size
Last modified
part-00000-2ee60d4e-7a77-41e2-a8d7-0a86d1c27115.c000.s
168.30 KB
4 minutes ago
_delta_log
About this volume
Owner
gowdhaman.bj@diggibyte.com


/Volumes/upload_catalog/default/delta_table / _delta_log
Filter files and directorie...
Create directory
Copy path
Name
Size
Last modified
00000000000000000000.crc
3.98 KB
16 minutes ago
00000000000000000000.json
2.94 KB
16 minutes ago
_staged_commits

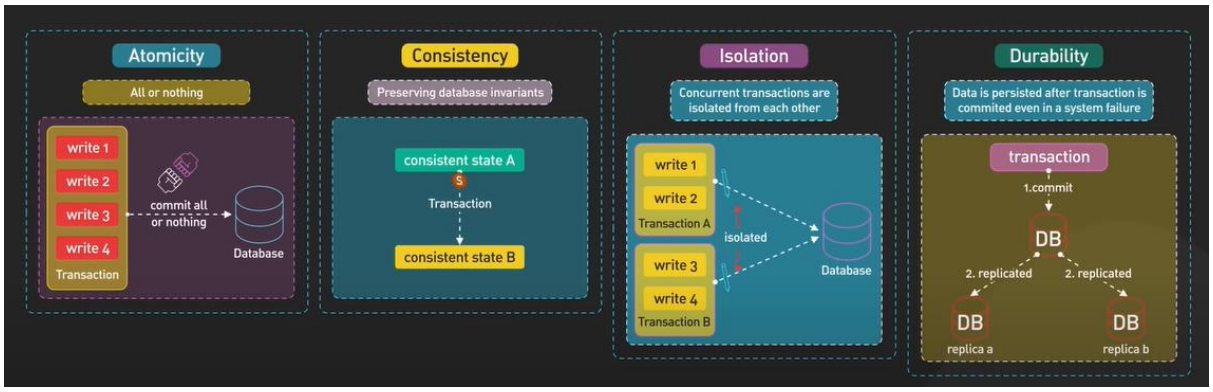
Insert operation - append

Log file updated

Read that file.



ACID



1. Atomicity

- **All or nothing** rule.
- A transaction either **completes fully** or **rolls back completely** if something fails.
- Example:
    - If you transfer money ₹100 from **Account A** → **Account B**, either both the **debit** and **credit** happen, or neither happens.
    - In Spark/Delta → If a write job fails midway, partial data won't be committed.

---

2. Consistency

- The database moves from **one valid state to another valid state**.
- Ensures rules, constraints, and schema are not violated.
- Example:
    - In a banking DB, balances can't go negative if rules prevent it.
    - In Spark/Delta → If schema requires an **integer column**, you can't insert a string value.

---

3. Isolation

- Multiple transactions running **concurrently** should not interfere with each other.
- Example:
    - If two people are booking the **last train ticket**, isolation ensures that only one booking is confirmed.
    - In Spark/Delta → Two jobs writing to the same Delta Table won't corrupt the data because of transaction logs.

---

4. Durability

- Once a transaction is committed, it is **permanently stored**, even in case of system crash/failure.
- Example:
    - Once money is transferred, it will remain recorded in the DB even if the server restarts.
    - In Spark/Delta → Changes are written to **Parquet + transaction log**; replay ensures durability.

Time Travel

- Purpose → To read historical versions of data.
- Does NOT modify the table, it's just for reading. (read-only)
- Useful for auditing, debugging, or comparing old vs new data

Time Travel in Delta Lake allows you to query or restore a table to a previous version or timestamp. It helps in auditing, recovering from accidental updates or deletes, and reproducing historical reports.

Time Travel means you can query a Delta table as it existed at a specific point in time — either by version number or by **timestamp**.

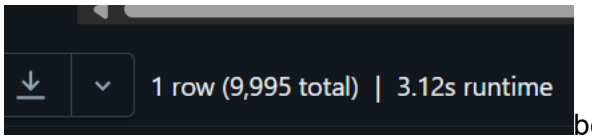
Hint: Snapshot Analysis, Debugging/troubleshooting, Data recovery

Time Travel helps you inspect or fetch old data.

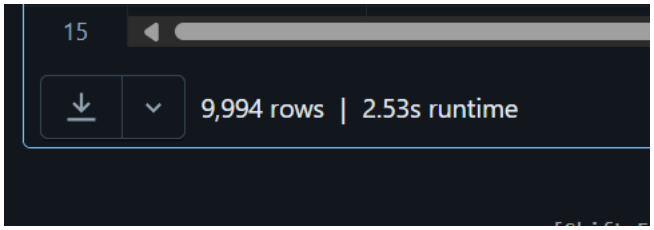
Data Versioning: Delta Lake automatically versions the data stored in your data lake. Each write operation is assigned a version number, and you can access different versions of the data using either a **timestamp or a version number**.

Audit Data Changes: Time travel simplifies auditing by allowing you to view the history of table changes using the DESCRIBE HISTORY command or through the UI.

Rollbacks: Time travel makes it easy to roll back to a previous state in case of bad writes or accidental deletions. **This can be done using either a timestamp or a version number**.



before rollback(total records: 9995)



Restore command:

How to restore the delta table to one of its previous versions?

We need use the restore command (don't confuse with time travel)

Restore actually rolls the table back to that past version, creating a new version in the log.

RESTORE is like a rollback — it physically brings the table back to a previous version.

While Time Travel is *read-only*, RESTORE actually changes the *current state* of the table.

-- Restore table to version 5

RESTORE TABLE sales TO VERSION AS OF 5;

-- Restore table to a timestamp

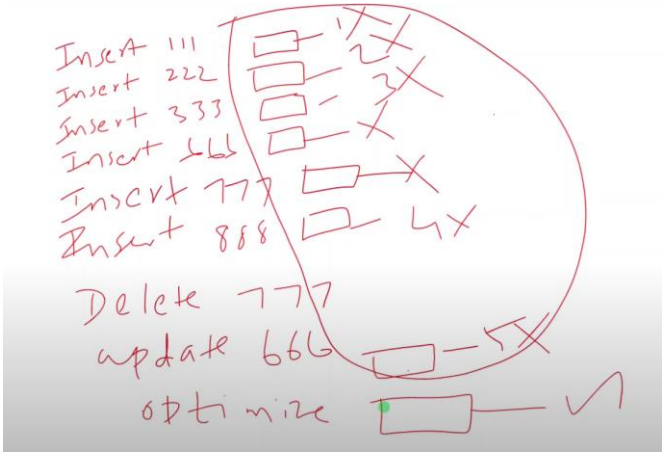
RESTORE TABLE sales TO TIMESTAMP AS OF '2025-09-01 12:00:00';

Restore command will permanently restoring to the one of the previous versions to the latest version. (but in time travel just we are going back in time and just we are doing the data analysis).

Vaccum commands

Vaccum commands are mainly used to clean up the absolute files.

- The VACUUM command in Databricks is used to remove unused data files from Delta tables.
- These files are typically created during operations like updates, deletes, or overwrites, and are no longer needed after a certain retention period.
- This helps in optimizing storage and maintaining performance.



Step1: insert 1,2,3,6,7,8 (total 6 files created)

Step2: delete 7 (file 7 is deleted) (physically not removed)

Step3: update 6(old number 6 file is made inactive[physically not removed] and created new number 6 file)

Now we have total of 5 files active files

Step4: optimize command then that 5 valid file is combined into a single file (5 → 1 file).

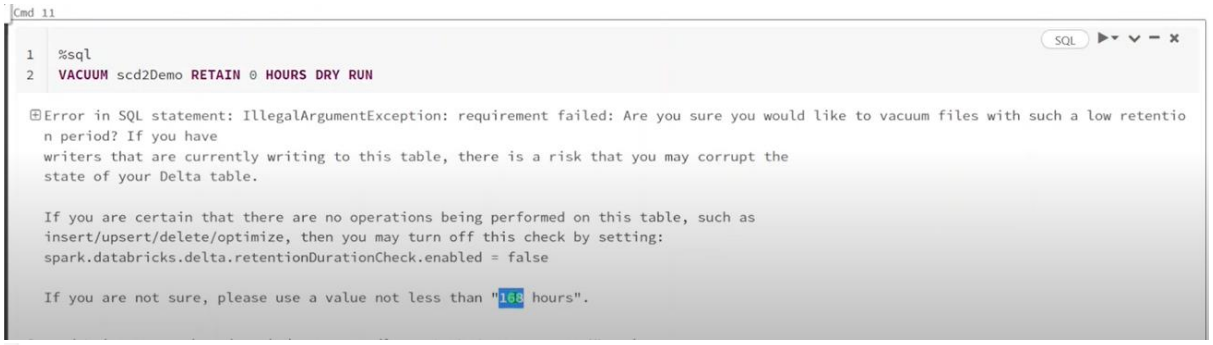
So, we are combined that 5 file are made as invalid (total of 7 invalid file are their)

Now there is 7 unwanted file we must delete it for this VACCUM command is used.


```
%sql
VACUUM scd2Demo DRY RUN
```

to check which are unwanted file (**important data is not deleted, it combine into single file by using optimize command**)

VACUUM table_name [RETAIN num_hours];




```
%sql
VACUUM scd2Demo RETAIN 0 HOURS
```

We can't delete by setting 0 hours

Hint:

- Cleanup inactive files
- Default 7 days (168 hrs)
- Configurable
- Dry run

What is Dry run?

Simulated vacuum command → by mistake if we remove the sensitive information, In order to avoid that by using DRY RUN command we can check what are the files are been deleted.

- Preview files to be deleted. (if we are ok with files then proceed with vacuum command)
- Avoid accidental delete.

Schema Enforcement (Schema Validation)

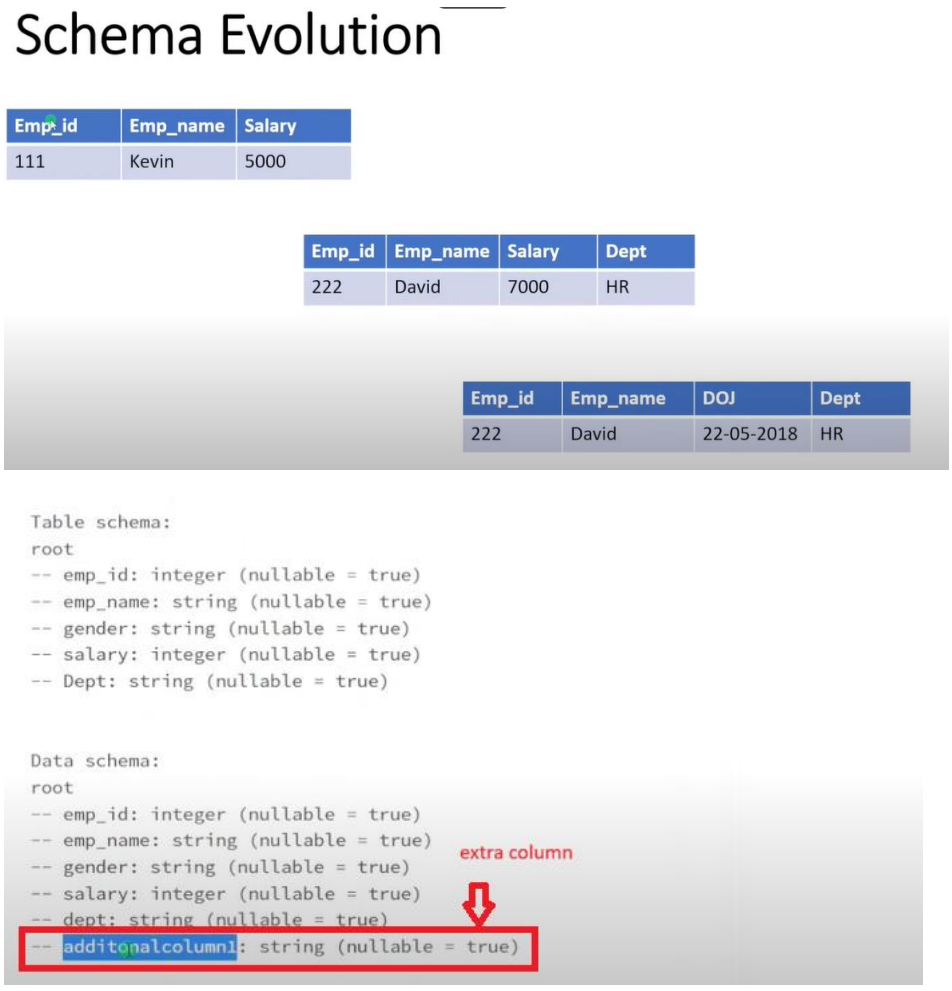
Schema enforcement means Spark checks that the incoming data matches the defined table schema (column names, types, and nullability) before writing it.
If the data doesn't match, the write is rejected, preventing bad or inconsistent data from entering your tables.

- Means Spark/Delta ensures incoming data matches the existing table schema.
- If the data type or column mismatch → Spark throws an error (it won't corrupt your data).

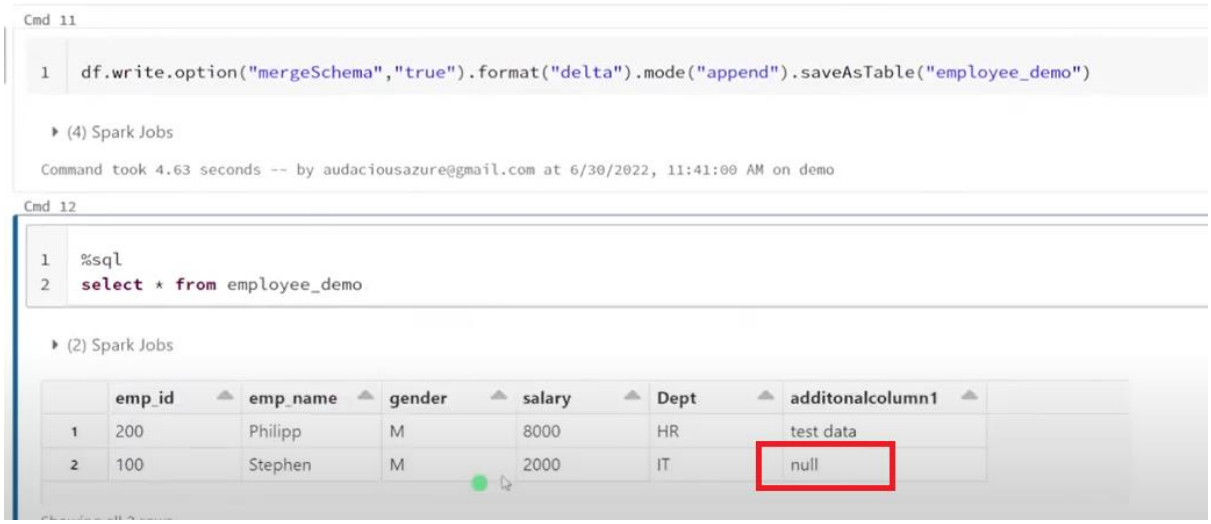
Example: id:int, name: string; (if insert id="a". this is not allowed)

Schema evaluation

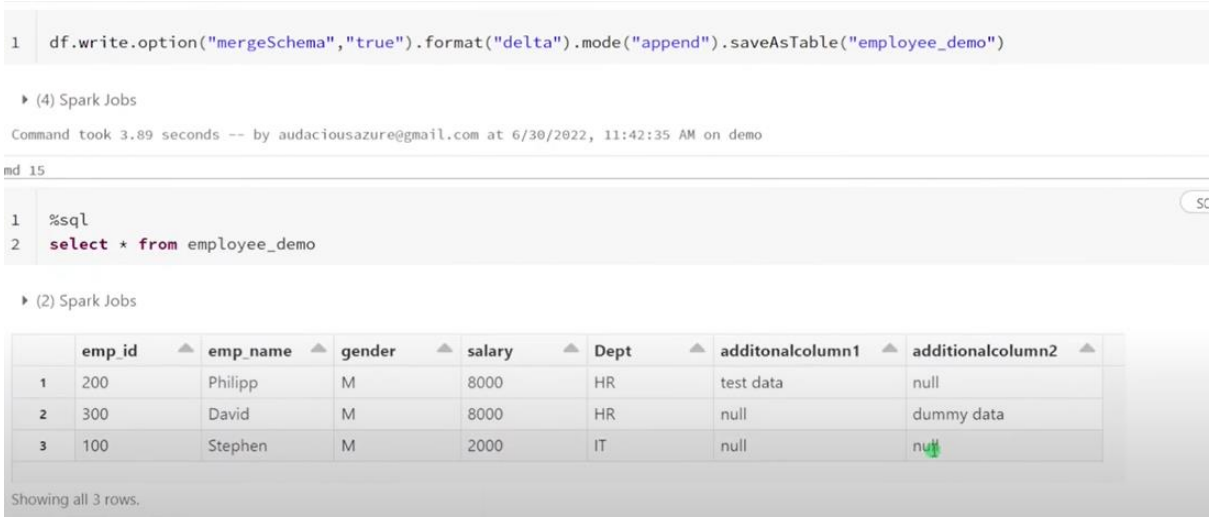
- Means Spark/Delta allows schema to change over time.
- You can add new columns or change existing columns.



We have to use **option("mergeSchema","true")** schema will be updated.



If the data is not their then **null** will added.



We can remove and add the column.

MergeSchema will just accept only adding new columns

OverwriteSchema

We have to drop some existing columns or rename certain columns then we can go with overwrite schema.

overwriteSchema basically completely removing the existing schema and recreating the new schema as per the input data.

Example:

- Existing table → id, name, age
- New DataFrame → id, name, city
- After **overwriteSchema** = true → table schema = id, name, city (✕ age column is removed).

Example:

- Existing table → id, name, age
- New DataFrame → id, name, city
- After **mergeSchema** = true → table schema = id, name, age, city

---

Optimize

Whenever we are working with the delta table over the time it used to create a huge number of files, small data files it is not good for performance (because engine must maintain huge number of metadata) the smaller file are combined into optimal size.

(100mb + 100mb + 100mb +100mb +100mb +100mb +100mb +100mb +100mb +100mb) combined into 1GB file.

Default size of optimize: 1GB

If any of the file is having lesser than 1gb then it will combine merge the files to 1gb that is the concept of optimize.

Note:

For every insert operation one parquet file is created and log is maintained.

- If delete record, the deleted row in parquet is not removed (physically not removed, in log file it maintained) (because it maintains for time travel).
- If update the record present in the table (old parquet file is also their) and also new parquet file is created and log maintained. (when whether we are updating or deleting older version of file is not removed immediately)

After optimizing the file are combined into single file (upto 1gb) as main file. But file are physically present. Points only to the single file.

it works on the data files (Parquet files), not the log files.

What OPTIMIZE Does

- Delta Lake writes many small Parquet files over time (from streaming jobs, frequent writes, etc.).
- OPTIMIZE **compacts** those **small Parquet files** into **fewer large Parquet files** (for example, from hundreds of 10-MB files to a handful of 1-GB files).
- This improves **read performance** because:
    - Fewer files → fewer metadata lookups.

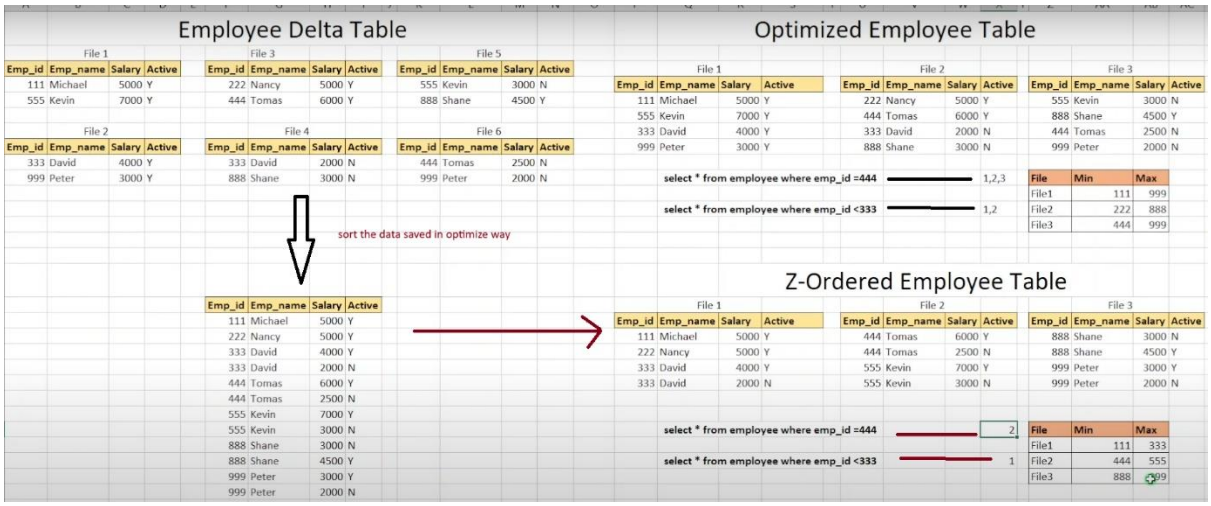
- Better data skipping and caching.

It does not merge or alter the _delta_log transaction files—those remain as-is.

Z-ordering – prevent full scan, efficient indexing

It is extension of optimize. It is used along with optimize. (optimize is does not care about data ordering, it would randomly combine the file data, and it will create optimal size of the file).

If you are using z-ordering with optimize, then it will combine small files into larger one but time it will also reorder the data which will be helpful in performance improvement.



Syntax:

Zorder (column_name)

We are going for liquid clustering because without optimization Z-ordering will not work.

What is Liquid Clustering?

Liquid Clustering is a **next-generation data clustering method in Delta Lake** that **automatically organizes data** to improve query performance — **without you having to manage partitions or static Z-Order indexing**.

Think of it as **"self-organizing data"**:

- Traditional partitioning needs you to pick a column (like country) → creates folder structure.
- Z-Ordering helps queries, but you must periodically run OPTIMIZE ZORDER.
- **Liquid Clustering removes these manual steps.** It adapts dynamically as data changes.

-- Create a table with liquid clustering on user_id

```
CREATE TABLE transactions (
  txn_id BIGINT,
  user_id STRING,
  amount DOUBLE,
  txn_date DATE
)
USING DELTA
CLUSTER BY (user_id, txn_date);
```

Now:

- You don't worry about folder partitions or z-order optimize jobs.
- As data grows, Delta keeps clustering user_id + txn_date values together.

Summary

Liquid Clustering is a new way of organizing Delta tables. Unlike static partitioning or Z-Ordering, it's **dynamic, automatic, and fine-grained**. You just specify clustering columns once, and Delta optimizes storage layout continuously in the background. This improves query performance while reducing maintenance overhead.

Difference between Managed vs unmanaged table?

Managed table	Unmanaged table
Data and metadata managed within the databricks or spark environment.	Only the metadata is managed with the databricks or spark environment. Actual data is stored in the external storage system it could be S3 bucket etc
To drop table - use drop command it drop metadata and data.	Using "drop" Removes only the metadata
	This table gives more control over the data to developers(gives security to data)

How to drop an unmanaged table?

Need to use dbutility command (rm command)

Data Skipping – reduced data reading

What is Data Skipping?

- Normally, when Spark queries a Delta table, it might scan **all the files** in that table (which is slow for large datasets).
- Instead of scanning *all files* in a table, Spark looks at metadata (like min/max values per column in each file) and **skips files that can't possibly match the query filter**.
- During queries, Spark looks at these stats → and skips files that definitely don't match the query filter.

What is medallion architecture in delta lake?

Bronze – data lake (kept)

Siliver/gold – delta lake (in most the case the delta lake is kept on top of the data lake)

How to overwrite only selective records in delta table??

ReplaceWhere – predicate selection

Dynamic partition overwrite – logical partition overwrite.

What is deletion vectors in delta lake?

Hint:

- Parquet → immutable
- Change in single record overwrite entire parquet
- Making modified record

If we are performing a certain DML operations like delete. We are having millions of records in a delta table we want to delete certain record (may be 10 records) in order to remove only those few records, it must overwrite (it must recreate that entire parquet file) so hit it the performance.

In order to avoid that we can enable deletion vectors as a result whenever we want to delete few records from a big file it is not going to overwrite (or recreate) the parquet file instead of that it is going to create a flag (like is_deleted: boolean) within in that big file.

Just marking the deleted records as a result, it improves the performance.

What is delta clone? Shallow clone vs deep clone?

Delta clone → is a copy a snapshot of delta table (a particular version of delta table can be taken as a snapshot that majorly used for archival need and also backfilling).

Shallow clone => copy only the metadata, not copies actual data

Deep clone => copy metadata and actual data.

Delta sharing?

With help of this feature, we can share the data within organization.

Hint:

- Data sharing within organization.
- No need to copy.
- Strong security.

Write operation

Read a file

What is Unity Catalog?

- **Unity Catalog (UC) is a unified governance solution for data on Databricks.**
- **Unity Catalog is a centralized data catalog that provides access control, auditing, lineage and quality monitoring across Databricks workspaces.**

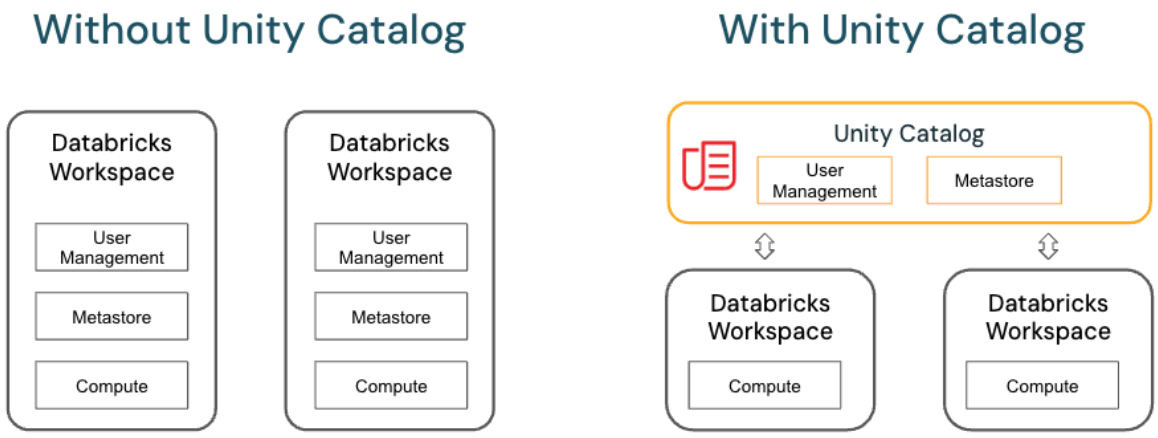
It provides **one place to manage access, security, lineage, and auditing** for all your data and AI assets across **workspaces, regions, and clouds**.

Think of it as the **"single source of truth"** for managing data in Databricks.

Why Unity Catalog?

- Before Unity Catalog, each workspace managed its own metadata and permissions.
- This caused fragmented access control, duplicated datasets, and security gaps.
- Unity Catalog standardizes governance across an organization, making compliance easier.

Feature	Description
Centralized Governance	Manage permissions for all data assets in one place.
Fine-Grained Access Control	Grant access at table, column, or row level, not just entire databases.
Cross-Workspace Sharing	Share data securely across multiple Databricks workspaces or teams.
Audit & Lineage	Track who accessed what data and how it changed over time.
Support for Delta Lake	Works seamlessly with Delta tables for ACID transactions and time travel.



Key Features of Unity Catalog

Centralized Governance

- Manage permissions (tables, views, ML models, files, functions) in one place.

- Unity Catalog uses a **3-level hierarchy**: Catalog → Schema → Table/View

Data Lineage

- Automatically tracks **how data flows** across queries, tables, and jobs.
- Helps with debugging, impact analysis, and compliance.

Cross-Workspace Sharing

- Share data across multiple Databricks workspaces.
- Integrates with **Delta Sharing** for secure external sharing.

Supports Multiple Data Sources

- Works with Delta tables, external data (ADLS, S3, GCS), ML models, files, BI dashboards.

Benefits of Unity Catalog

- One place to **control access** across all data assets.
- Simplifies **multi-cloud & multi-region governance**.
- Provides **lineage + auditing** for compliance.
- Works seamlessly with **Delta Lake**.

Summary

Unity Catalog is Databricks' centralized governance layer that unifies security, access control, lineage, and auditing across all data and AI assets. It introduces a 3-level namespace (catalog.schema.table), supports fine-grained permissions like row-level security and column masking, and integrates with Delta Sharing for secure data sharing. It helps enterprises meet compliance while simplifying data governance.

What is DBFS?

The term *DBFS* is used to describe two parts of the platform:

- DBFS root
- DBFS mounts

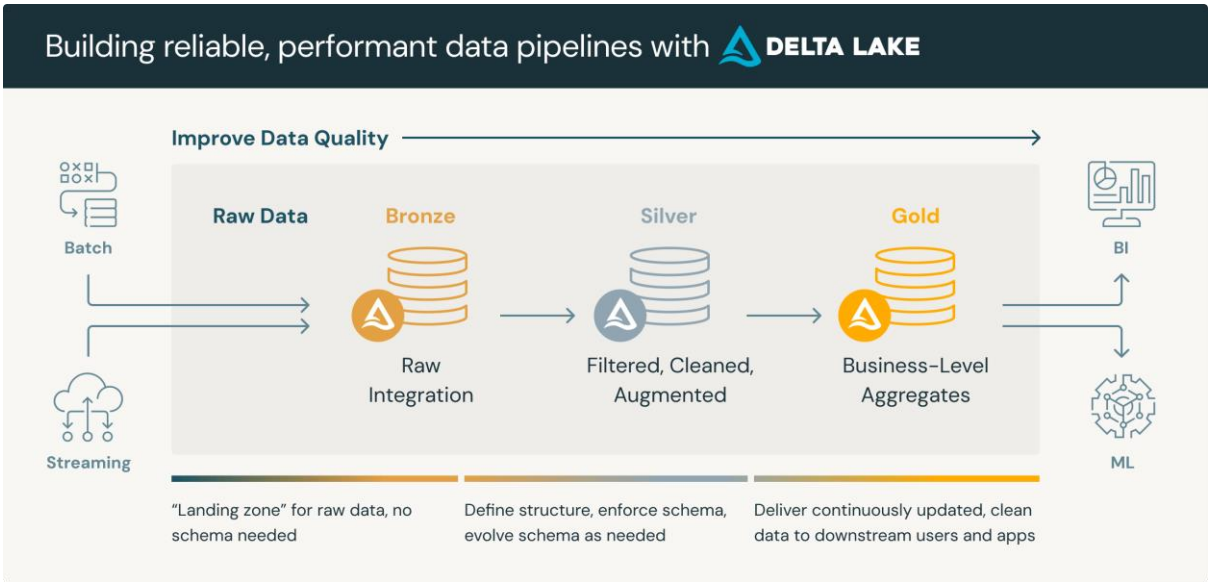
Storing and accessing data using DBFS root or DBFS mounts is a deprecated pattern and not recommended by Databricks. For recommendations for working with files.

What is the Databricks File System?

The term *DBFS* comes from Databricks File System, which describes the distributed file system used by Databricks to interact with cloud-based storage.

The underlying technology associated with DBFS is still part of the Databricks platform. For example, dbfs:/ is an optional scheme when interacting with Unity Catalog volumes.

Medallion architecture - Medallion Architecture is a layered approach to organization data in the Databricks



Batch – data will be update for the particular time difference.

Streaming – is live streaming of data.

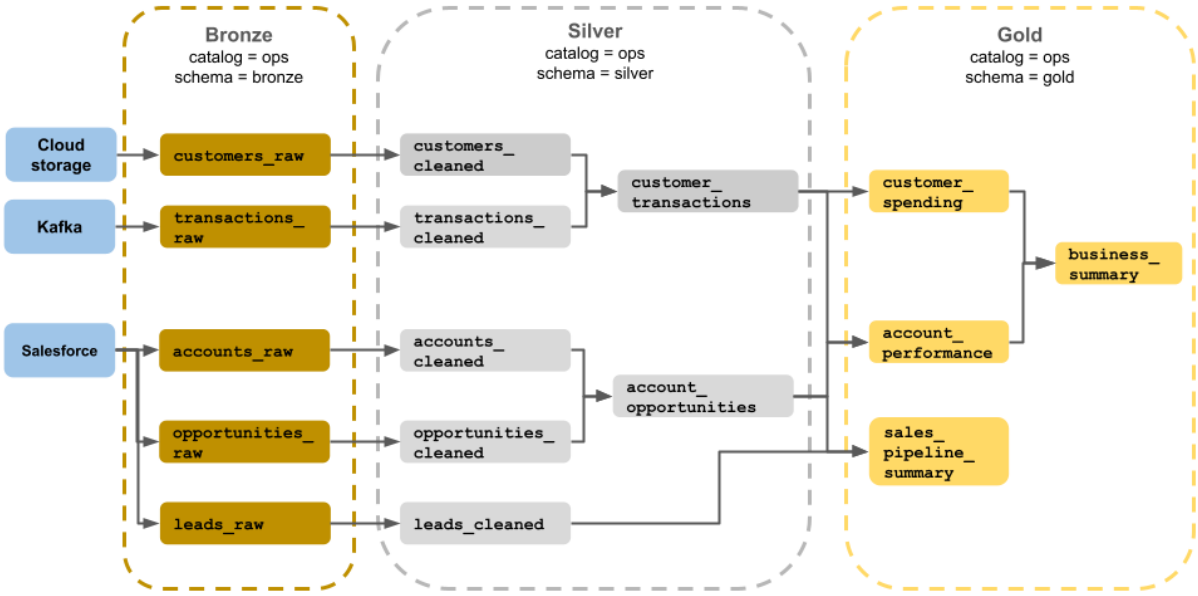
A **medallion architecture** is a data design pattern used to logically organize data in a lakehouse, with the goal of incrementally and progressively improving the structure and quality of data as it flows through each layer of the architecture (from Bronze ⇒ Silver ⇒ Gold layer tables). Medallion architectures are sometimes also referred to as "multi-hop" architectures.

Layer Breakdown

1. **Bronze Layer (Raw Zone)**
    - This is where raw, unaltered data lands from diverse sources like logs, APIs, or relational databases.
    - It retains full fidelity and historical context, often stored in immutable, append-only formats.
    - Ideal for auditing, reprocessing, or fallback scenarios.

2. **Silver Layer (Cleaned / Validated Zone)**
    - The data undergoes transformation: cleansing, validation, deduplication, formatting, and schema enforcement.
    - It becomes more consistent and reliable filtered enough for analytical tasks but still flexible.
    - Often used for preparing datasets for analytics or machine learning.

3. **Gold Layer (Enriched Zone)**
    - Data here is tailored for specific analytics needs—aggregated, enriched, and optimized (e.g., star schemas).
    - It serves business intelligence, reporting, executive dashboards, and advanced ML models.
    - Delivers high usability and performance.

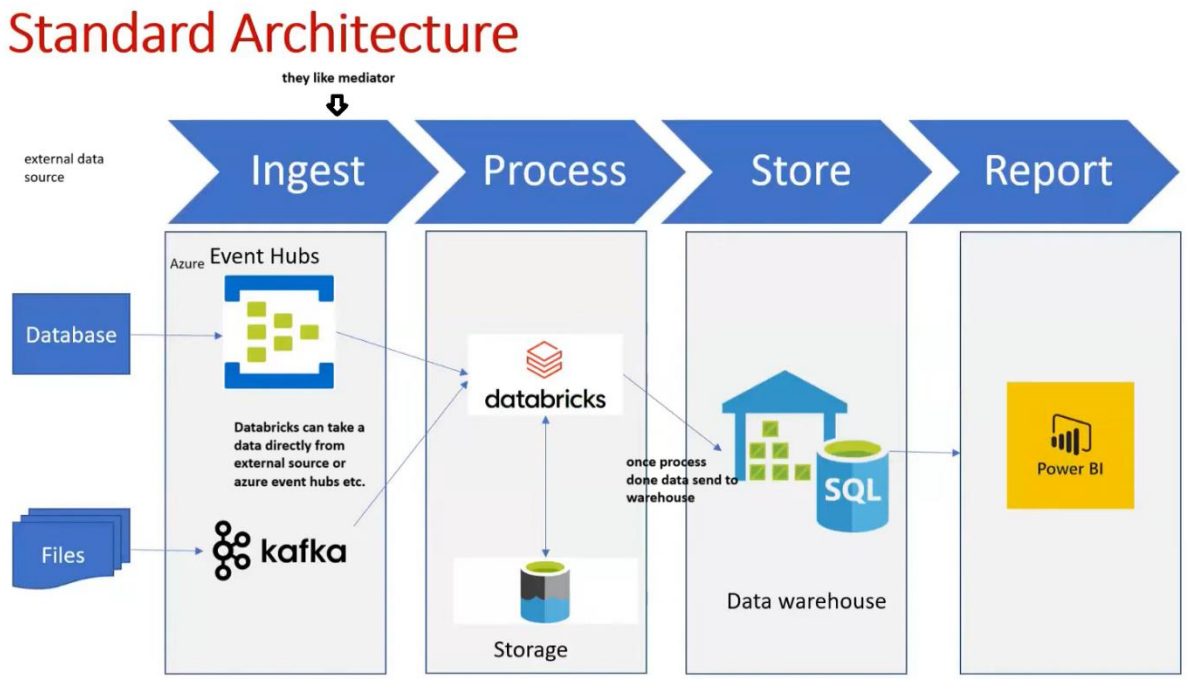


Structured Streaming

Streaming – deal with real time data.

Terminologies

- **ReadStream**:- to read the streaming data (entry point, using this consume the data from external sources (sources can be database, file system or azure event hubs).
- **WriteStream**:- once processed (filtered or other operation done) write the data into the target area.
- **Checkpoint**:- checkpoint plays key role in fault-tolerant and incremental stream processing pipelines. It maintains intermediate state on HDFS compatible file system to recover from failures.
- **Trigger**:- data continuously flows into a streaming system. The special event trigger initiates the streaming. **Default, Fixed internal, one-time**.
- **Output mode**: Append, Complete, Update.



Auto-scaling

Auto-Scaling in Spark (Databricks) is the feature that dynamically increases or decreases the number of worker nodes in a cluster based on workload, so you get performance when needed and save costs when idle