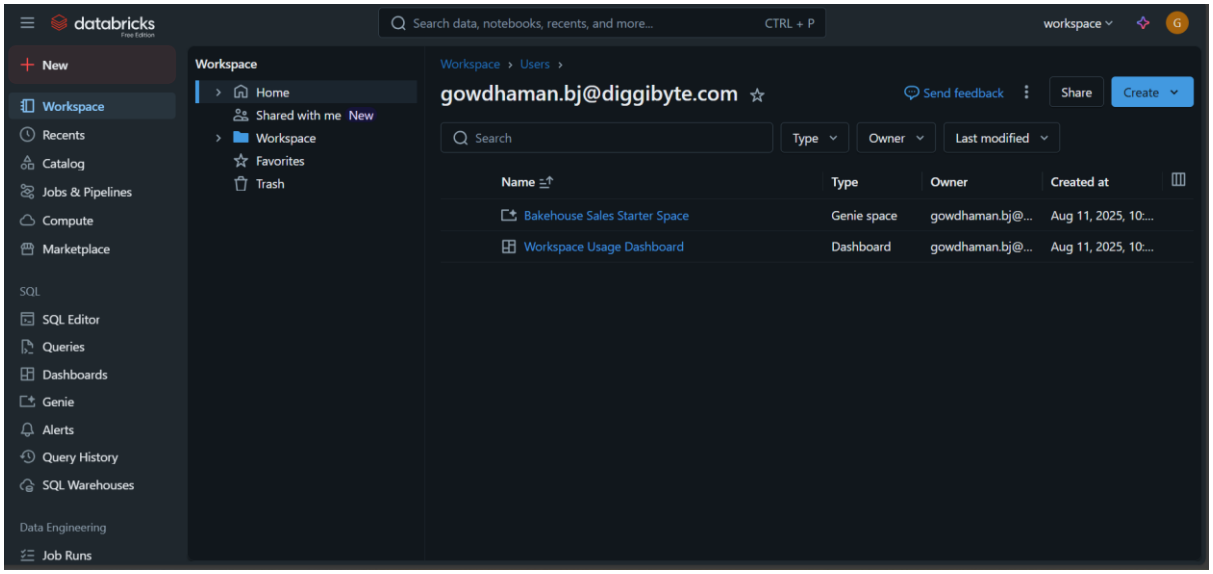
Databricks

Databricks is a cloud-based platform for big data analytics, machine learning, and collaborative data engineering. Its web-based UI provides tools for writing code, managing data, and orchestrating workflows.

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.

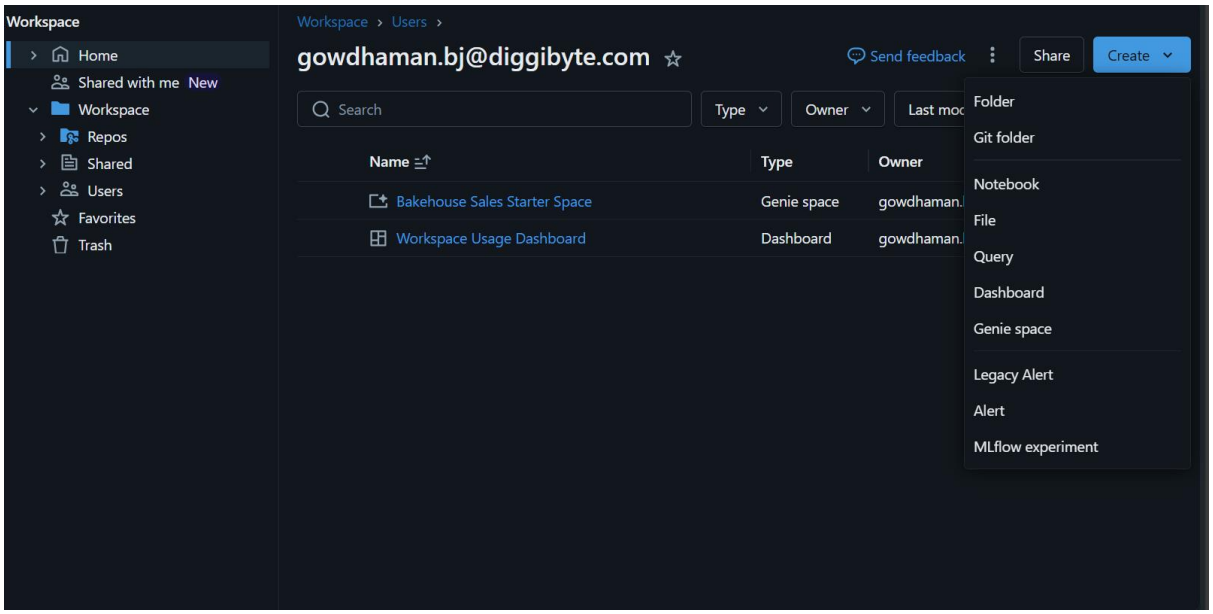
Filters (Above file list)

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

1.3 Workspaces

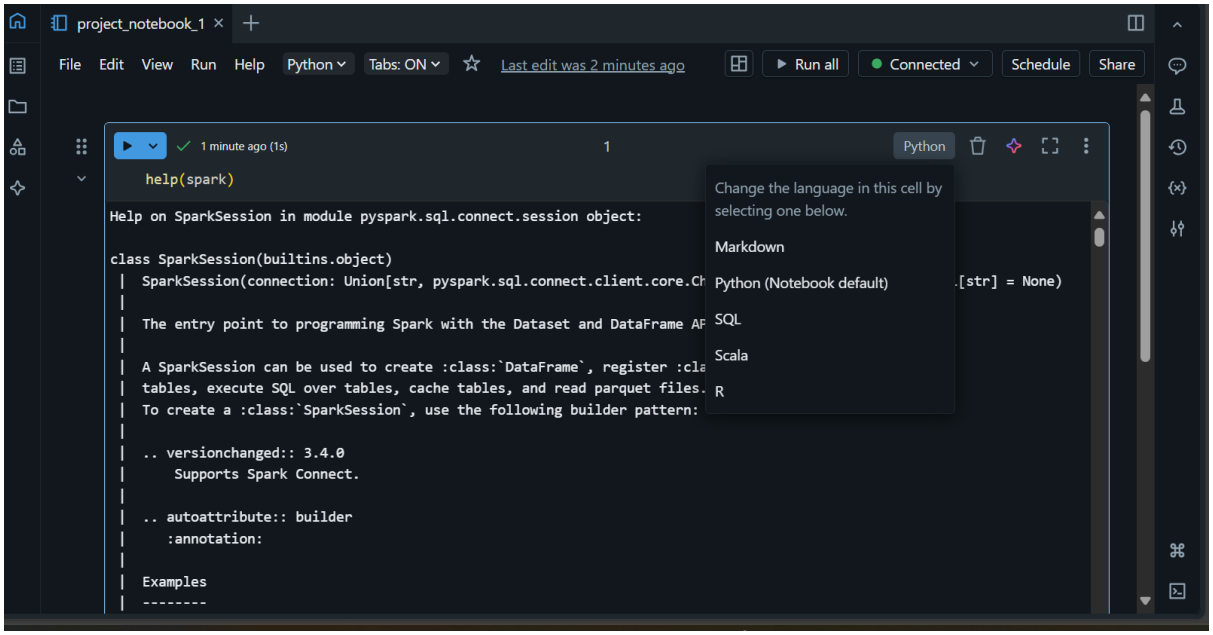
A **Workspace** is the environment in Databricks where users store and organize their projects.

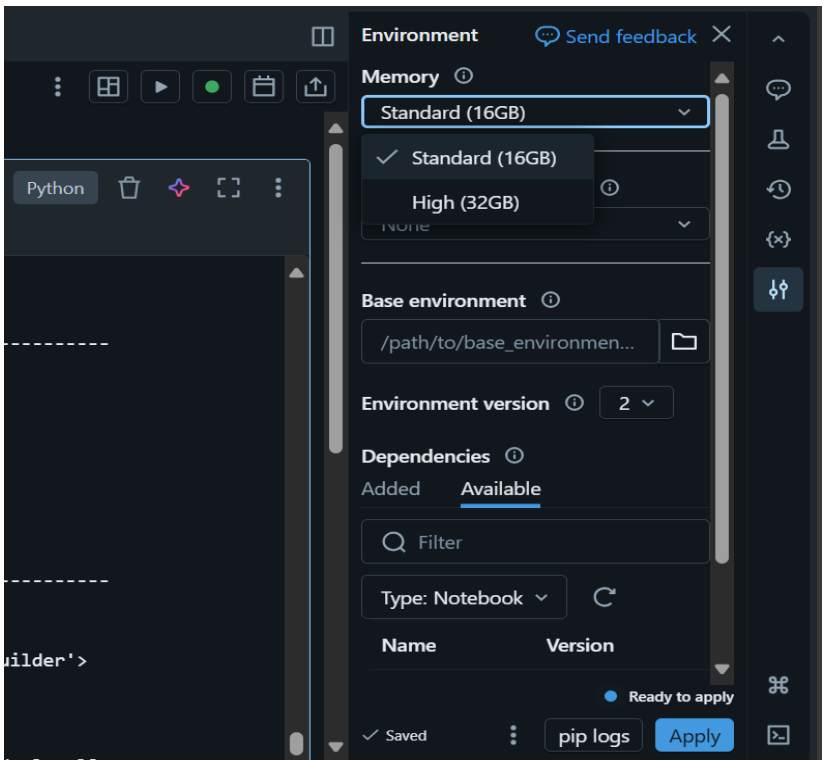
- Contains folders and items such as notebooks, libraries, dashboards, and experiments.
- Supports collaborative development — multiple users can work together.
- Can be personal (only visible to you) or shared (visible to the team).
- Can have access controls for fine-grained permissions.



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)

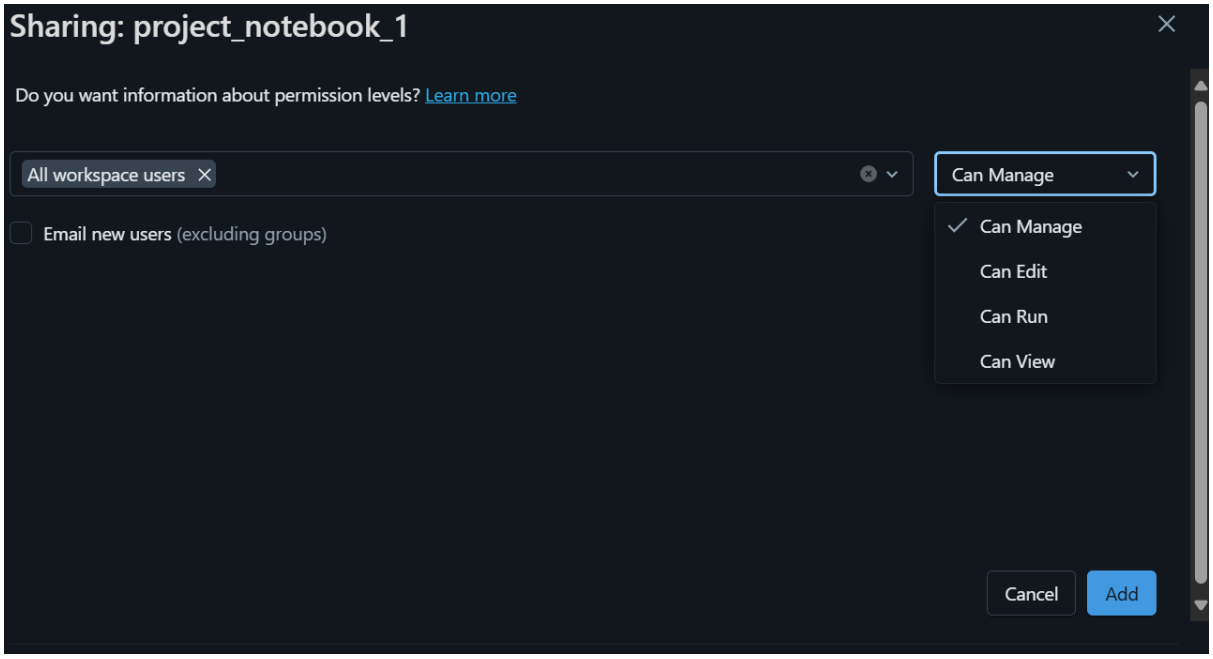
    - **Job Clusters** (spawned automatically for scheduled jobs)
- Configurable settings:
    - Node type & size
    - Autoscaling
    - Termination after inactivity
- Attached to:
    - Notebooks
    - Jobs
    - Workflows
- Supports installing libraries directly.

Managing Clusters

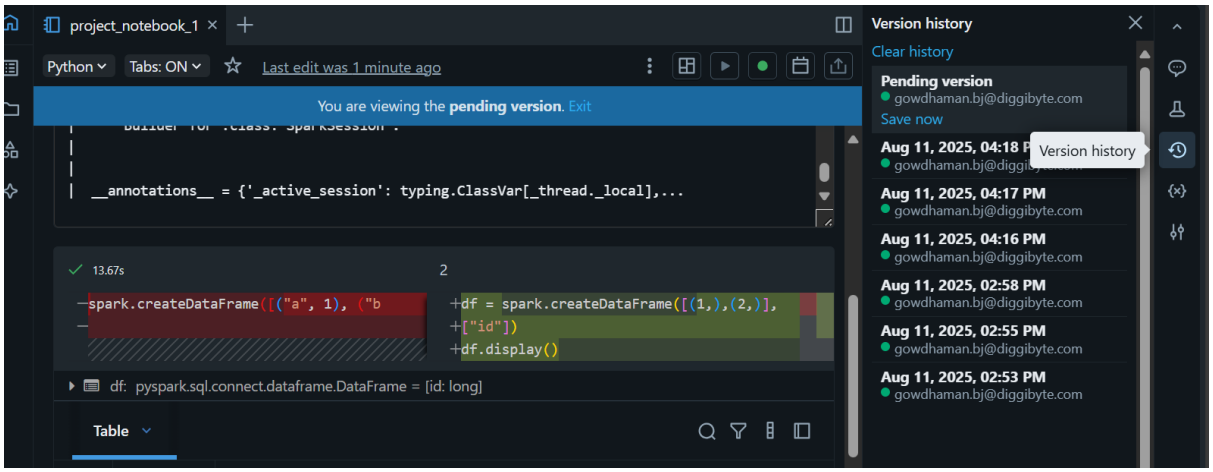
- **Clusters** are compute resources that execute notebooks and jobs.
- **Key management tasks:**
    - Create clusters with specific node types, sizes, and configurations.
    - Set **auto-scaling** and **auto-termination** to control costs.
    - Restart or terminate when needed.
    - Attach/detach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

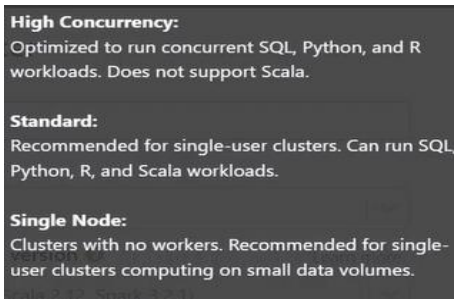
1. A **cluster** in Databricks is a set of virtual machines (driver + worker nodes) used to run notebooks, jobs, and queries.

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**

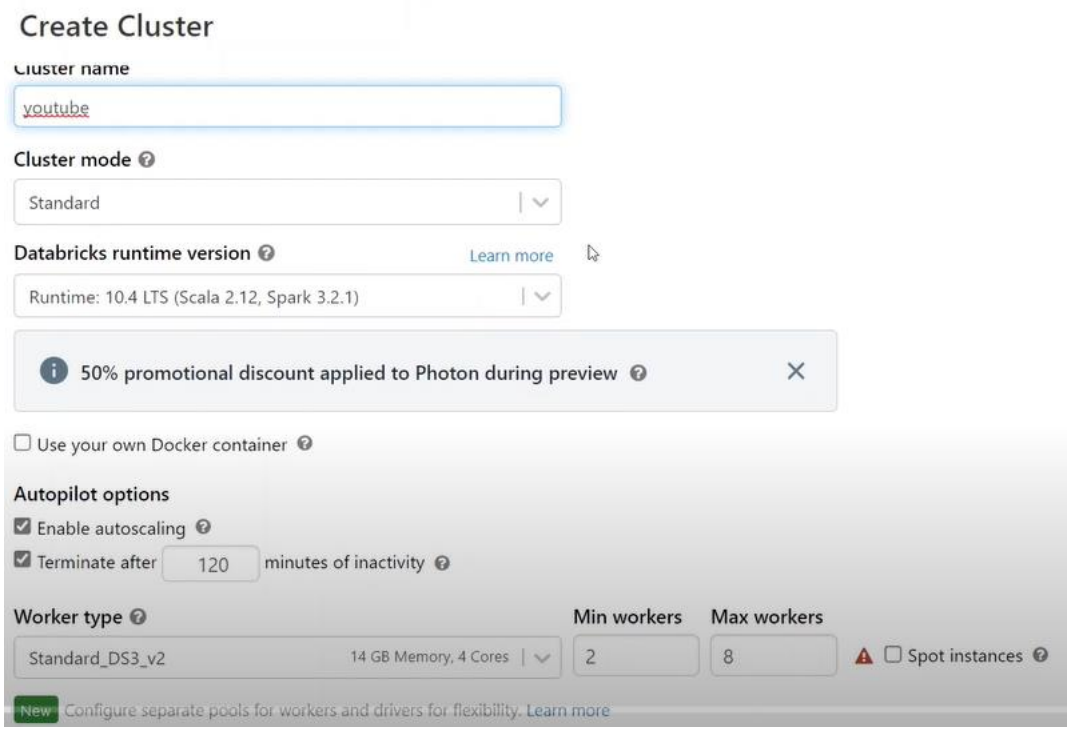
- Optimized for serving multiple users simultaneously.
- Better for BI tools, dashboards, and SQL analytics.
- Uses **fine-grained resource sharing** for better concurrency.

3.3 Worker Node Types

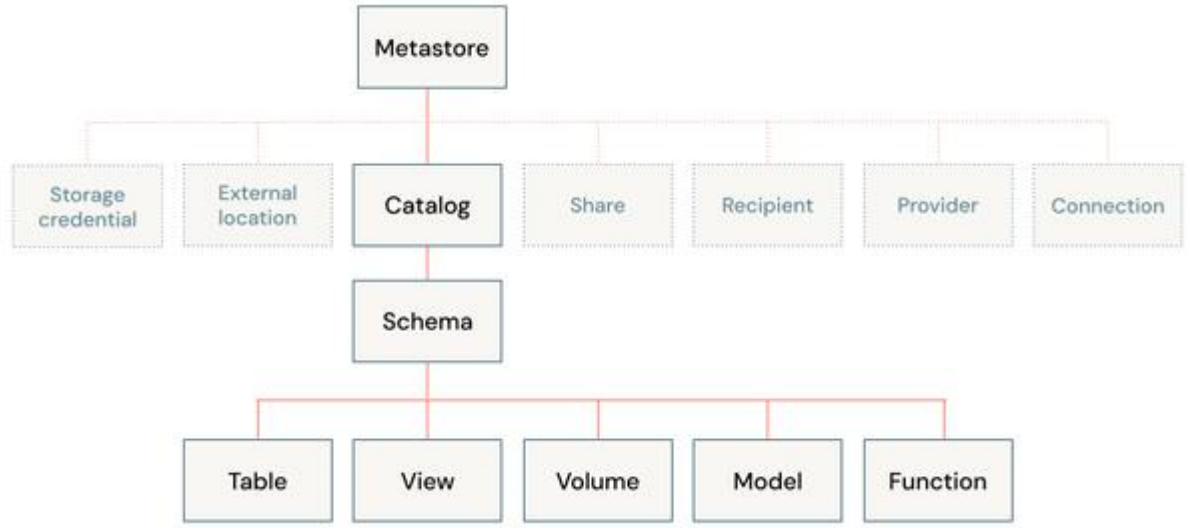
- Worker nodes perform the actual data processing.
- Choose based on workload:
  - **General Purpose** – Balanced CPU & memory (ETL, general analytics).
  - **Memory Optimized** – Large memory for heavy joins, caching, ML.
  - **Compute Optimized** – Higher CPU for processing-heavy workloads.
  - **GPU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

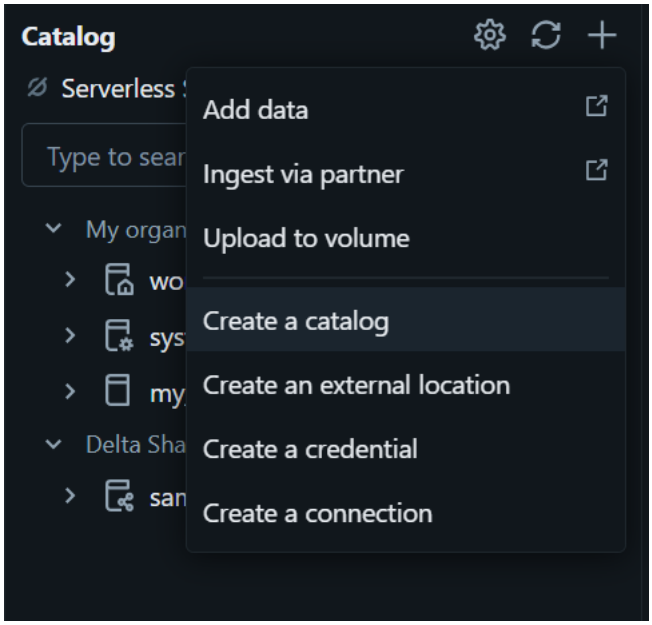


Catalog



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

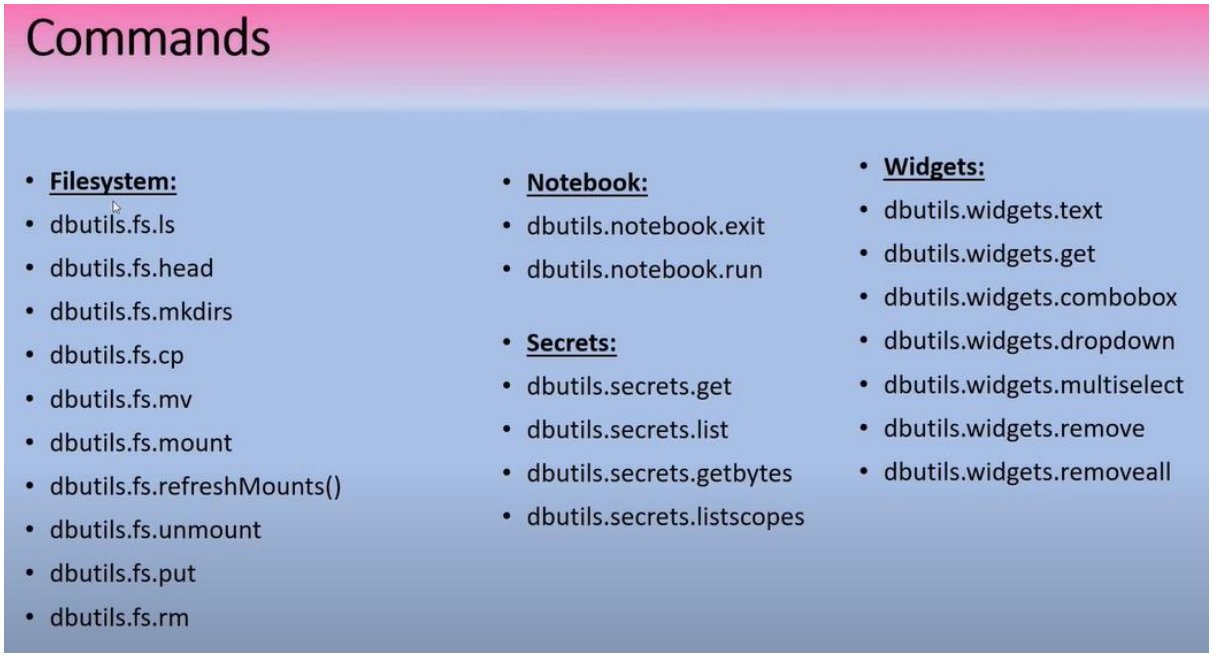
- Stores **machine learning models** registered in the **Databricks Model Registry**.
- Contains:
    - Model artifacts (trained weights, serialized files like .pkl or MLflow model format)
    - Versioning metadata
    - Associated training parameters and metrics
- Example:
    - A trained XGBoost fraud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



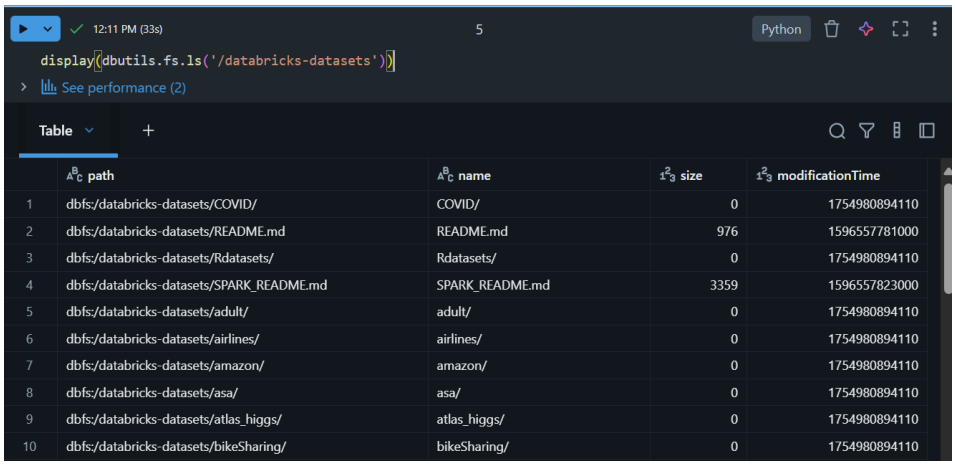
File System Utilities (dbutils.fs)

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"}
)
```



```
display(dbutils.fs.ls('/databricks-datasets'))
```

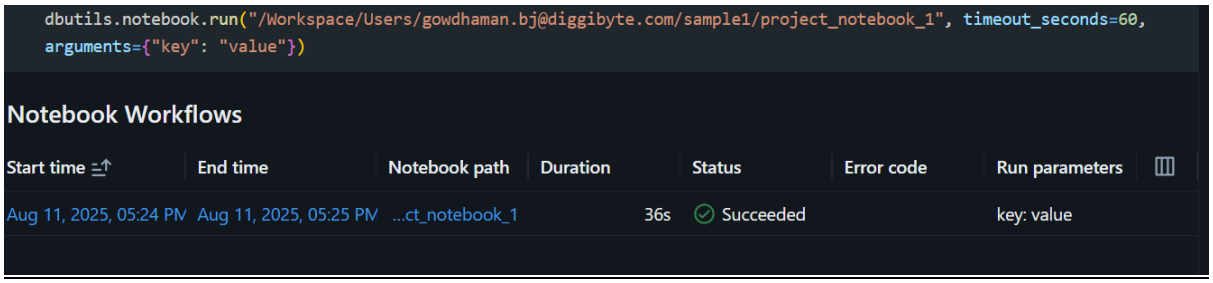
See performance (2)

Table

	path	name	size	modificationTime
1	dbfs:/databricks-datasets/COVID/	COVID/	0	1754980894110
2	dbfs:/databricks-datasets/README.md	README.md	976	1596557781000
3	dbfs:/databricks-datasets/Rdatasets/	Rdatasets/	0	1754980894110
4	dbfs:/databricks-datasets/SPARK_README.md	SPARK_README.md	3359	1596557823000
5	dbfs:/databricks-datasets/adult/	adult/	0	1754980894110
6	dbfs:/databricks-datasets/airlines/	airlines/	0	1754980894110
7	dbfs:/databricks-datasets/amazon/	amazon/	0	1754980894110
8	dbfs:/databricks-datasets/asa/	asa/	0	1754980894110
9	dbfs:/databricks-datasets/atlas_higgs/	atlas_higgs/	0	1754980894110
10	dbfs:/databricks-datasets/bikeSharing/	bikeSharing/	0	1754980894110

Accessing dbutils in Notebooks:

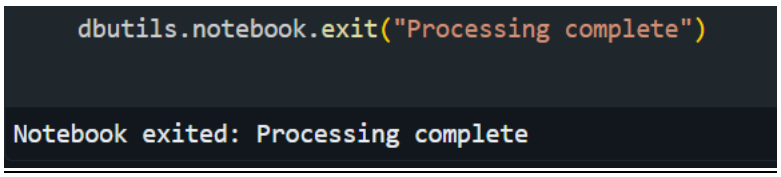
- Open a Databricks notebook cell (Python or Scala).
- Type dbutils and call functions directly.
- Works only when the notebook is attached to a **running cluster** or **SQL warehouse**



```
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")
```

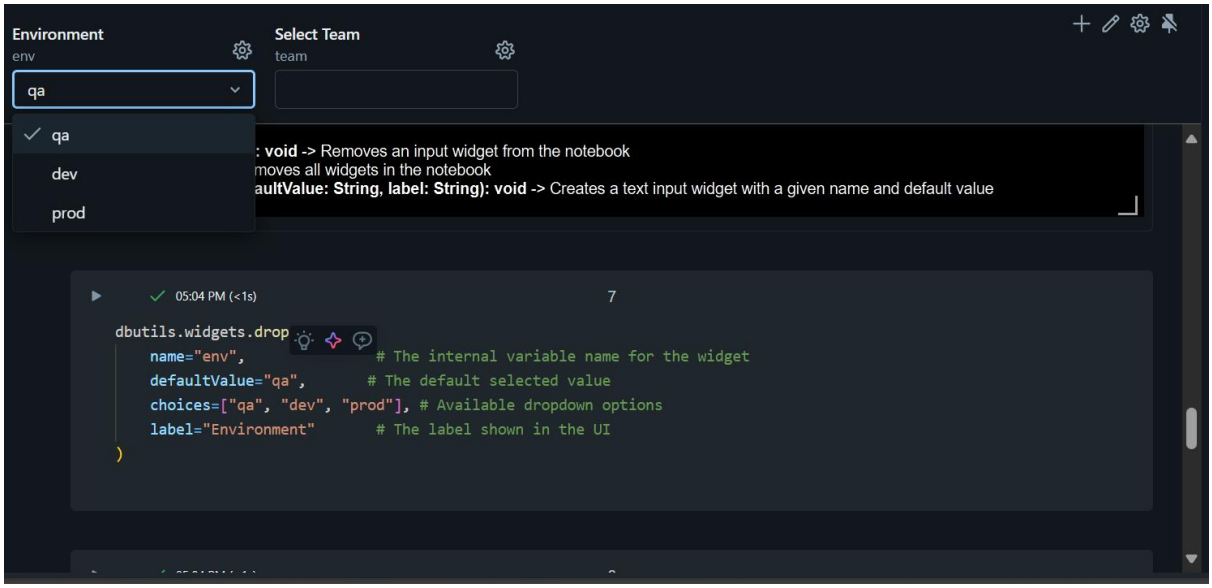
```
Notebook exited: Processing complete
```

dbutils.widgets

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

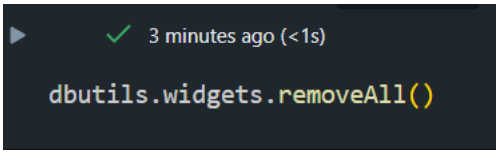
1 Dropdown, 2 combobox, 3 multiselect, 4 input

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

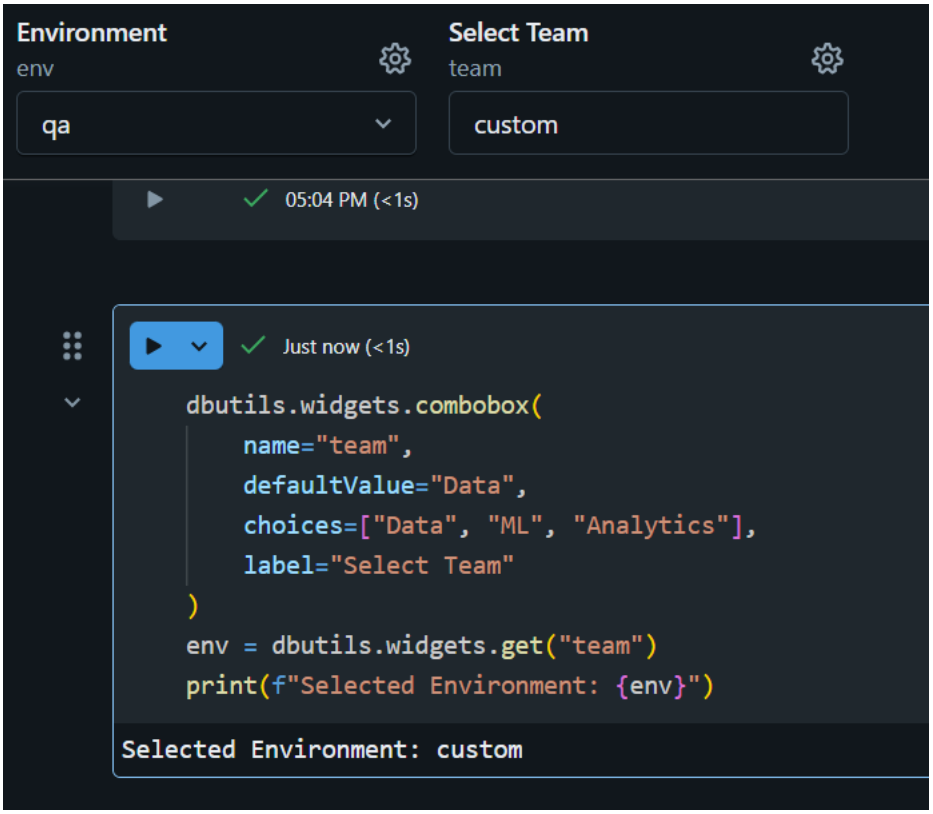


widgets in Databricks are **stateful**. Once created, their value is preserved until you:

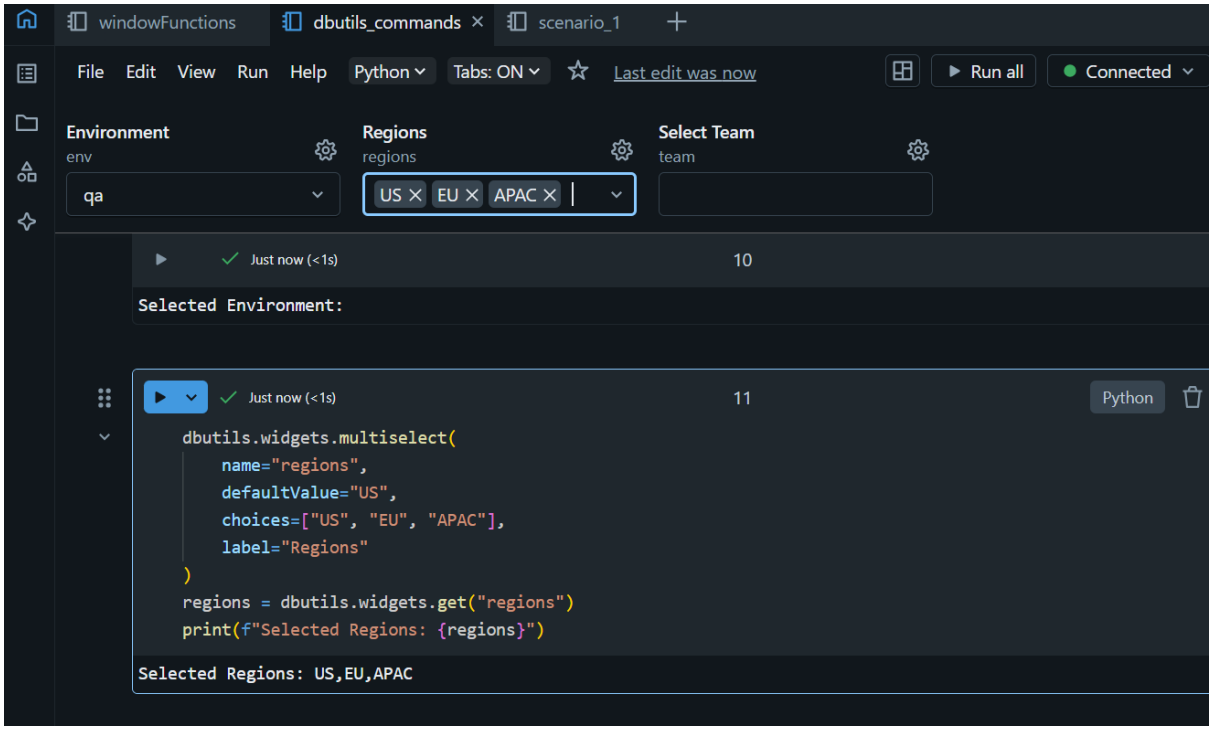
- Manually change the selection in the UI, or
- Remove and recreate the widget.



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
)
```

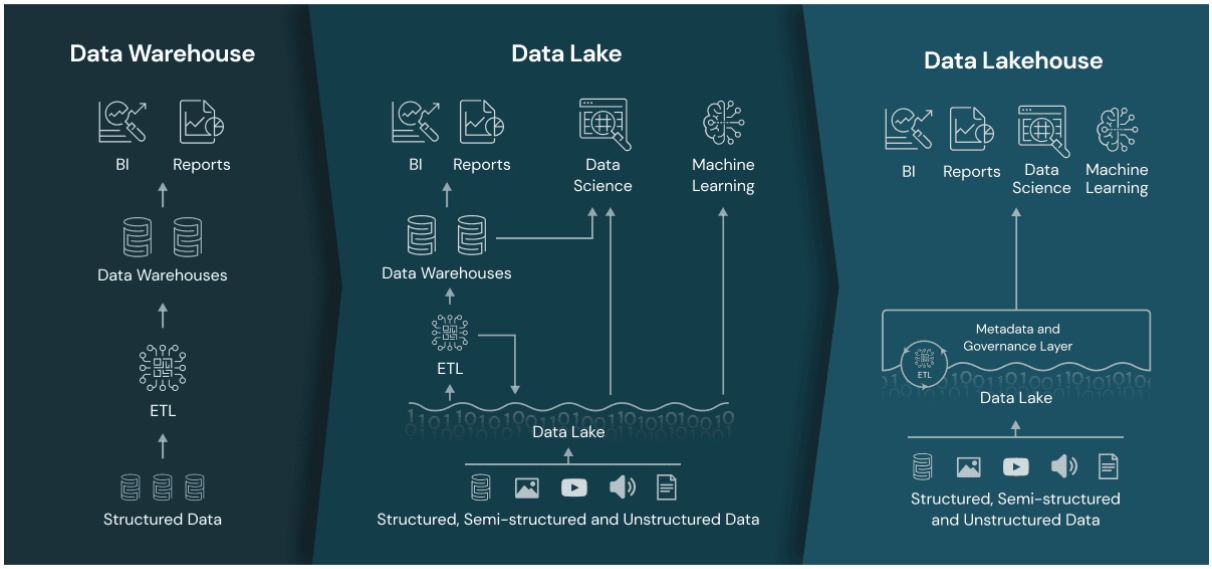
Parameters:

- **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 lakehouses 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 warehouses store structured data, whilst
- data lakes and lakehouses 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.
- 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).
- Analogy: A clean, organized library where every book is labeled and cataloged.

2️⃣Data Lake – *Flexible, Raw Storage*

- **Purpose:** Store **any type of data** — structured, semi-structured, unstructured — cheaply at scale.
- **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).
- **Analogy:** A huge warehouse where you dump everything without much organization.

3️⃣ 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.
- **Analogy:** A warehouse where you can dump everything, but there are shelves, labels, and a librarian who ensures order.

4️⃣ Delta Lake – *A Technology to Build a Lakehouse*

- **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).
- **Analogy:** A smart upgrade kit for your data lake, turning it into a reliable, query-friendly system.

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 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 combination of data files in the form of parquet + transaction log files into form of json

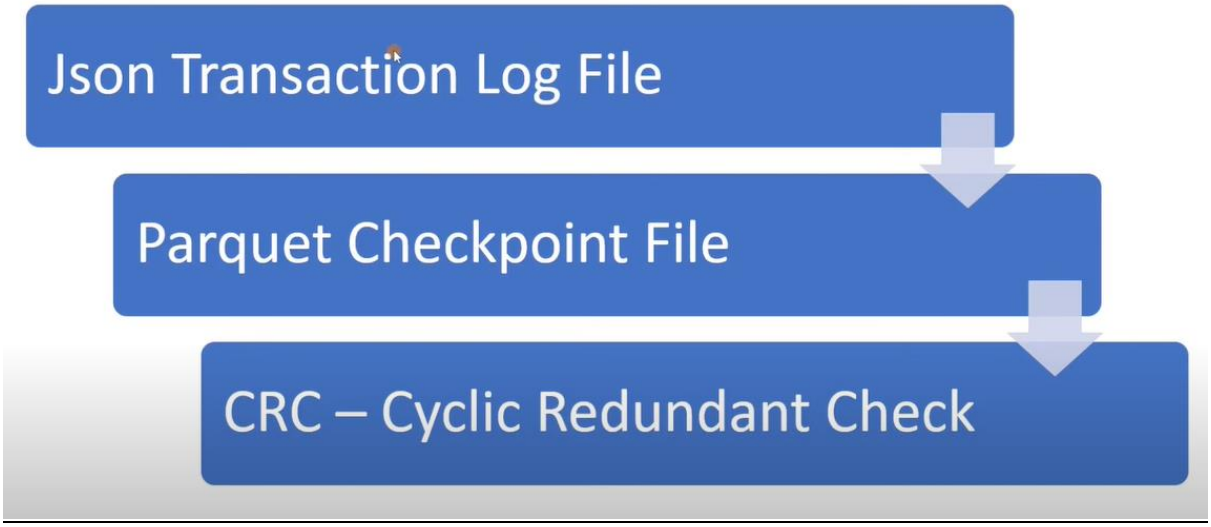
A Delta table is a data storage format built on top of Parquet that adds a transaction log to enable ACID transactions, schema enforcement, time travel, and efficient updates.

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 is created.

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.

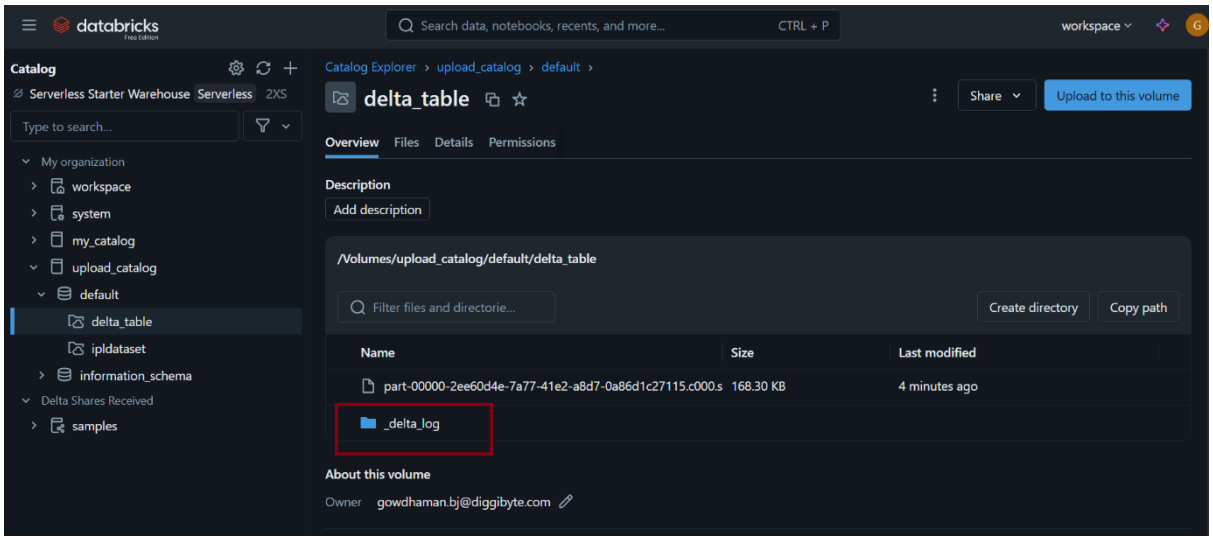
Inside _delta_log

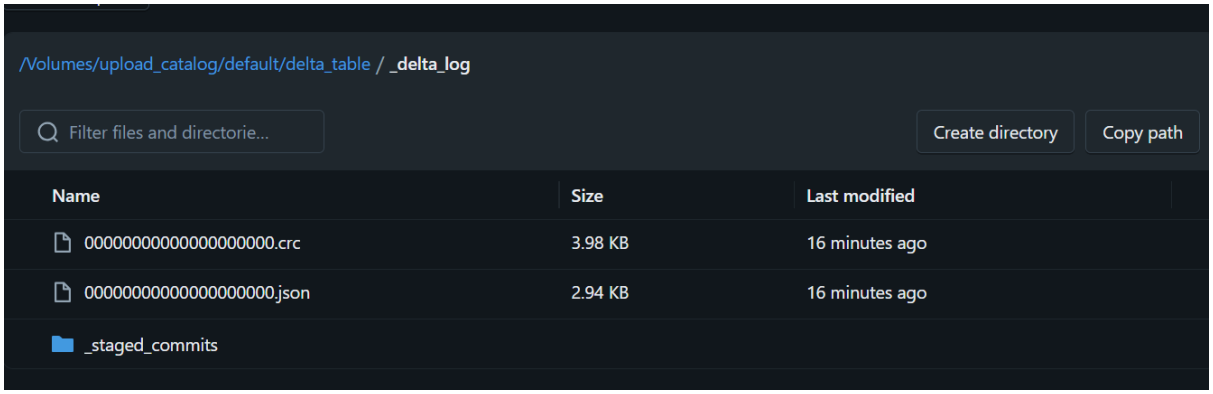
- **JSON log files** (00000000000000000010.json, etc.)

- 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.

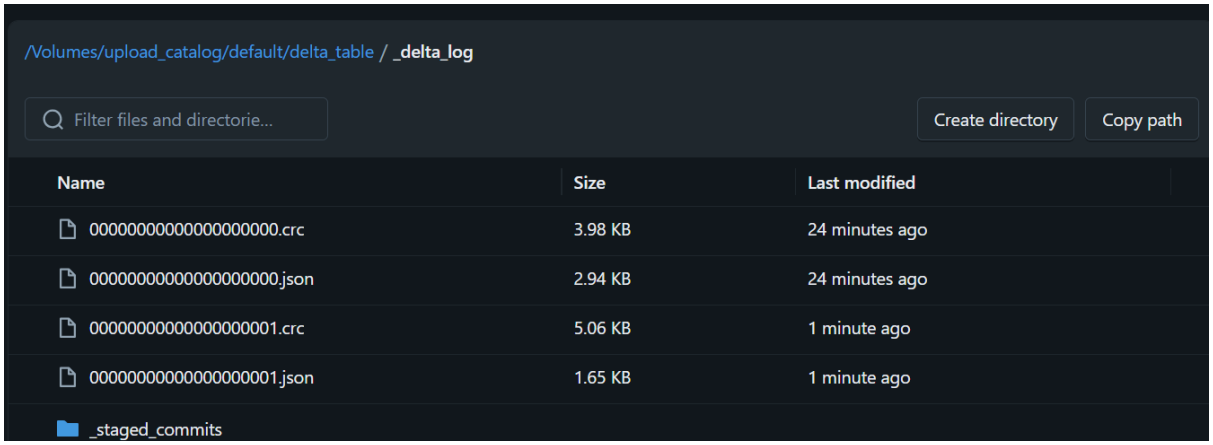
- **Checkpoint Parquet files** (*00000000000000000010.checkpoint.parquet*)
  - Store a **snapshot of the full table state** at a given version in an optimized format.
  - Speeds up reading the latest state so you don't need to read every JSON file from version 0.

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.



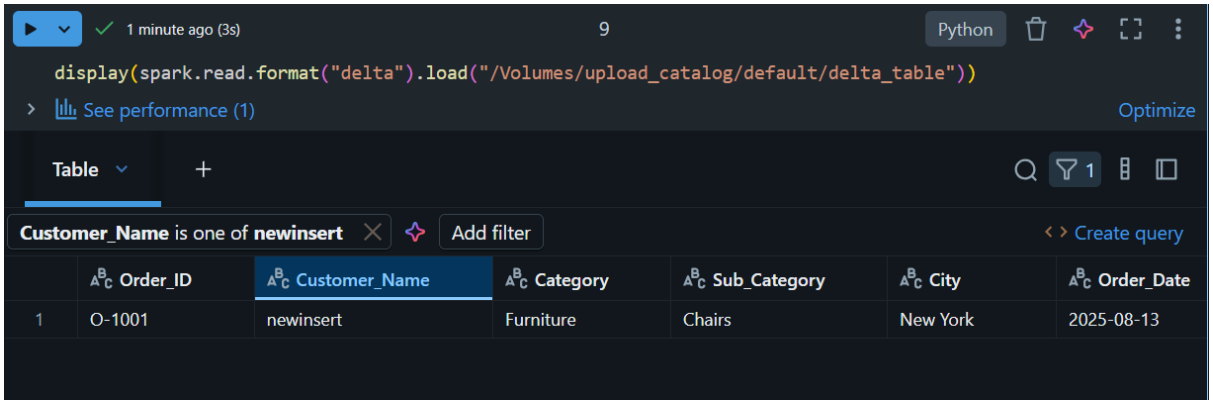


Insert operation - append

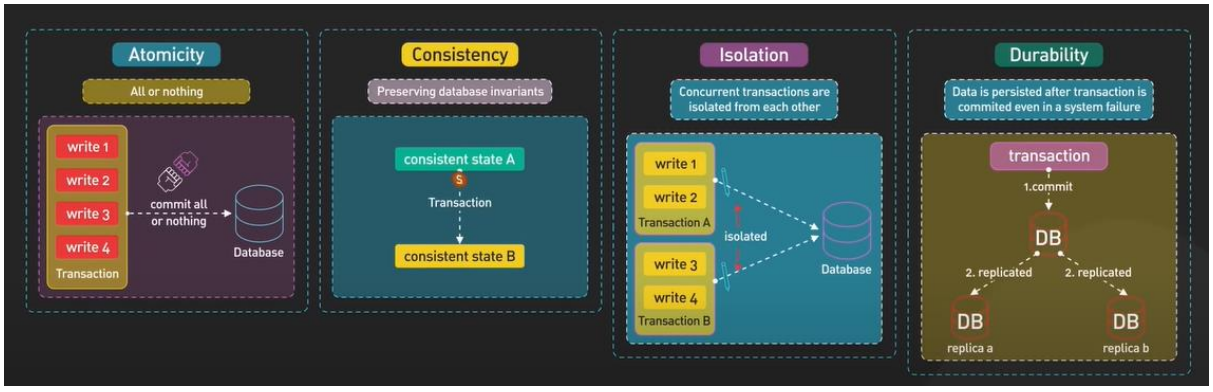


Log file updated

Read that file.



ACID



Atomicity -> Transaction Logs | Consistency -> Schema Enforcement | Concurrent Writes Isolated | Durable Storage via Write

Atomicity

Atomicity ensures that all operations within a transaction are completed successfully. If any part of the transaction fails, the entire transaction is rolled back, leaving the database in its previous state. This prevents partial updates that could lead to data corruption.

Consistency

Consistency ensures that a transaction brings the database from one valid state to another. It guarantees that any data written to the database must be valid according to all defined rules, including constraints, cascades, and triggers.

Isolation

Isolation ensures that concurrent transactions do not interfere with each other. Each transaction is executed in isolation, meaning that intermediate states are not visible to other transactions. This prevents issues such as dirty reads, non-repeatable reads, and phantom reads.

Durability

Durability guarantees that once a transaction has been committed, it will remain so, even in the event of a system failure. This ensures that the results of the transaction are permanently recorded in the database

---

Time Travel

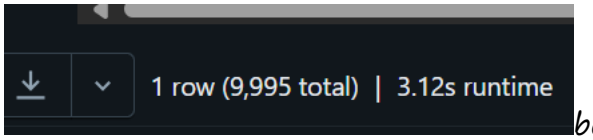
This feature allows users to access historical versions of their data, making it easier to audit changes, reproduce experiments, and roll back to previous states in case of errors.

Hint: Snapshot Analsis, Debugging/troubleshooting, Data recovery

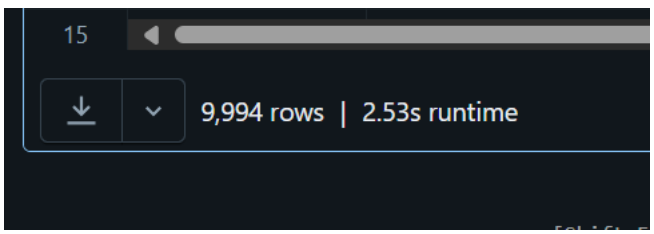
Data Versioning: Delta Lake automatically versions the data stored in your data lake. Each write operation is assigned a version number, and you can access different versions of the data using either a timestamp or a version number.

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 version?

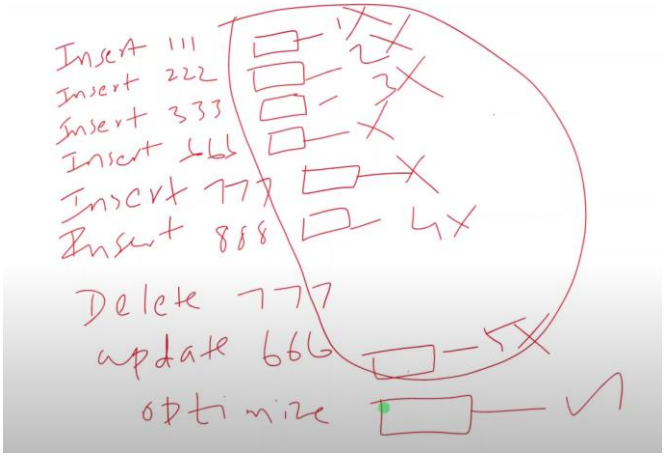
We need use the restore command (don't confuse with time travel)

Restore command will permanently restoring to the one of the previous version 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)

Step3: update 6(old number 6 file is deleted and created new number 6 file)

Now we have total of 5 files

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

Now there is 5 unwanted file we must delete it for this VACCUM command is used.

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

to check which are unwanted file

VACUUM table_name [RETAIN num_hours];

```
%sql
VACUUM scd2Demo RETAIN 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 evaluation

Schema Evolution

Emp_id	Emp_name	Salary
111	Kevin	5000

Emp_id	Emp_name	Salary	Dept
222	David	7000	HR

Emp_id	Emp_name	DOJ	Dept
222	David	22-05-2018	HR

```
Table schema:
root
-- emp_id: integer (nullable = true)
-- emp_name: string (nullable = true)
-- gender: string (nullable = true)
-- salary: integer (nullable = true)
-- Dept: string (nullable = true)
```

```
Data schema:
root
-- emp_id: integer (nullable = true)
-- emp_name: string (nullable = true)
-- gender: string (nullable = true)
-- salary: integer (nullable = true)
-- dept: string (nullable = true)
-- additonalcolumn1: string (nullable = true)
```

extra column

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

```
1  df.write.option("mergeSchema","true").format("delta").mode("append").saveAsTable("employee_demo")
```

▸ (4) Spark Jobs

Command took 4.63 seconds -- by audaciousazure@gmail.com at 6/30/2022, 11:41:00 AM on demo

md 12

```
1  %sql
2  select * from employee_demo
```

▸ (2) Spark Jobs

	emp_id	emp_name	gender	salary	Dept	additonalcolumn1
1	200	Philipp	M	8000	HR	test data
2	100	Stephen	M	2000	IT	null

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 gone).

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 has to 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 also the deleted record parquet is not removed (physically not removed) (because it maintains for time travel).

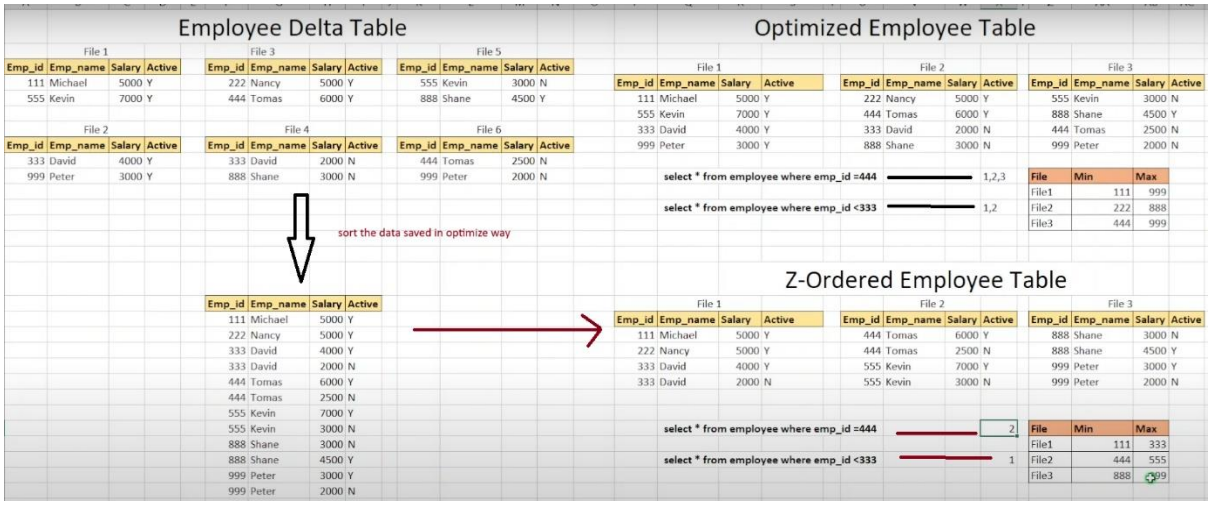
If update also new parquet file is created and log maintained. (when whether we are updating or deleting older version of file is not removed immediately)

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.

Z-orderding – 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 use 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)

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

Data skipping is the internal property of the delta engine using which it is efficiently performing the data operations.

Whenever we are creating delta lake and we are performing various operation on delta lake statistics has been collected (whenever we are performing certain

operation or analysis delta engine will consider those statistic based on help of that) skip the data file which are not required for the scanning.

With help of this we can avoid the full table scan, which improves the performance.

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.

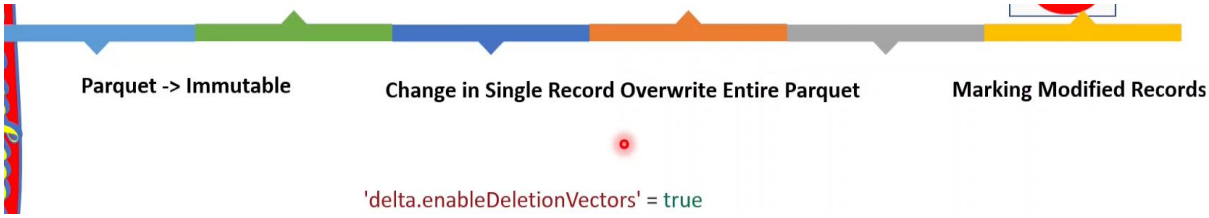
What is deletion vectors in delta lake?

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) inorder to remove only those few records it must overwrite (it has to 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

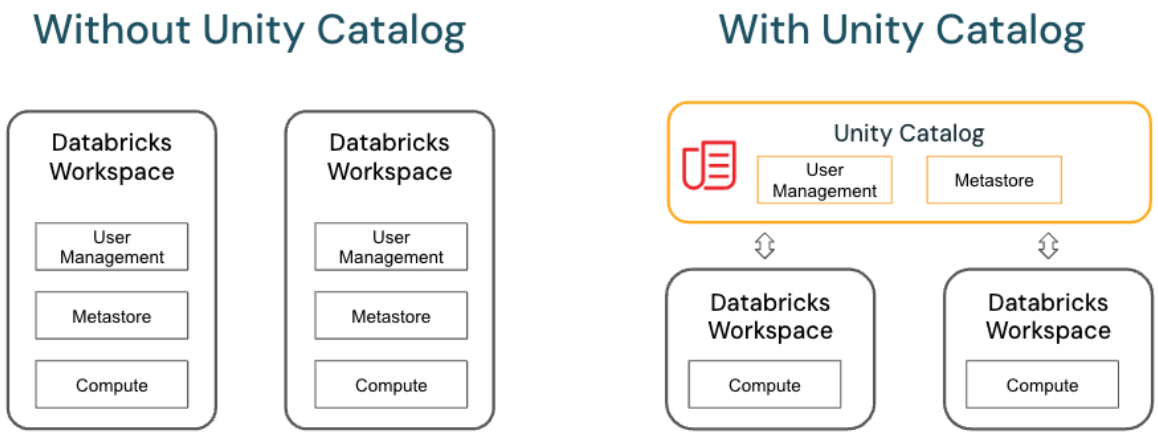
```
06:33 PM (3s)                                    12
df1.write.mode("overwrite").option("header", True).csv("/Volumes/my_catalog/source/
folder_managed_files/people_basic_csv")
> See performance (1)                                                    Optimize

06:37 PM (2s)                                    13
df2.write.mode("overwrite").option("header",True).csv("/Volumes/my_catalog/default/
sample")
> See performance (1)                                                    Optimize
```

Read a file

```
06:42 PM (7s)                                    14
df_read1 = spark.read.format("csv").option("header", True).load("/Volumes/
my_catalog/source/folder_managed_files/people_basic_csv")
df_read1.display()
df_read2 = spark.read.format("csv").option("header", True).load("/Volumes/
my_catalog/default/sample")
df_read2.display()
```

What is Unity Catalog?

Unity Catalog is a centralized data catalog that provides access control, auditing, lineage, quality monitoring, and data discovery capabilities across Databricks workspaces.



What is DBFS?

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

- DBFS root
- DBFS mounts

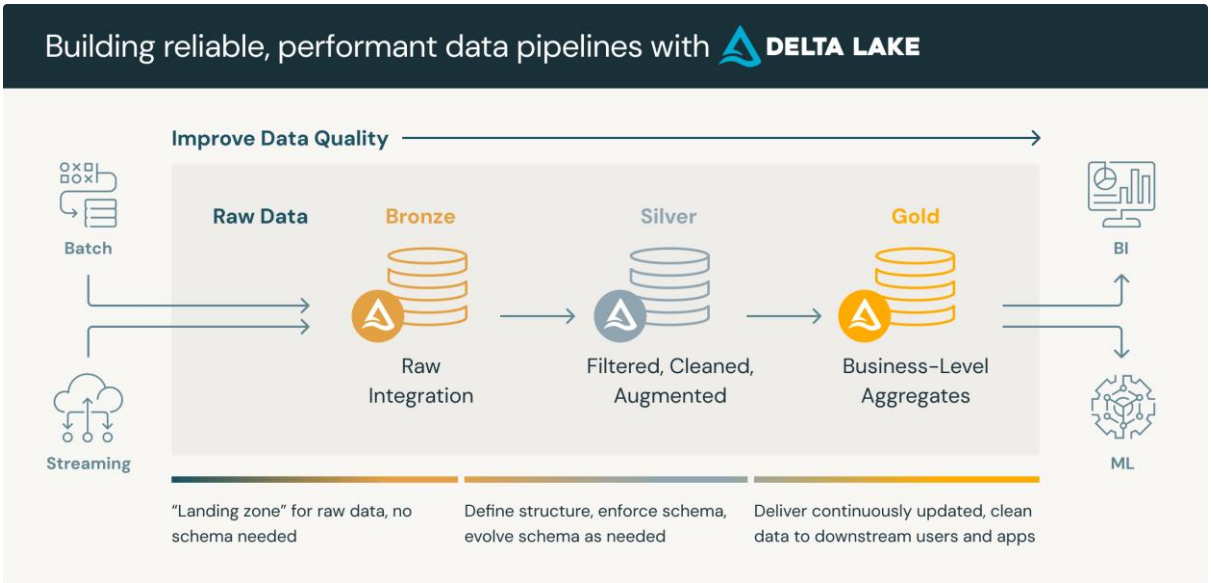
Storing and accessing data using DBFS root or DBFS mounts is a deprecated pattern and not recommended by Databricks. For recommendations for working with files.

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



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

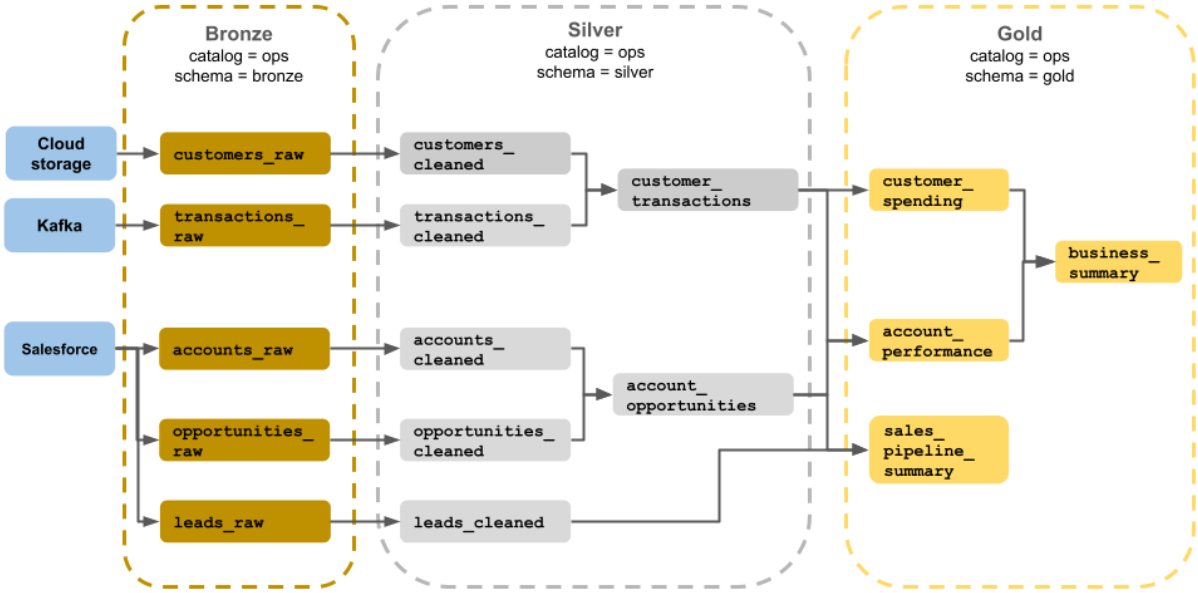
structure and quality of data as it flows through each layer of the architecture (from Bronze ⇒ Silver ⇒ Gold layer tables). Medallion architectures are sometimes also referred to as "multi-hop" architectures.

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.

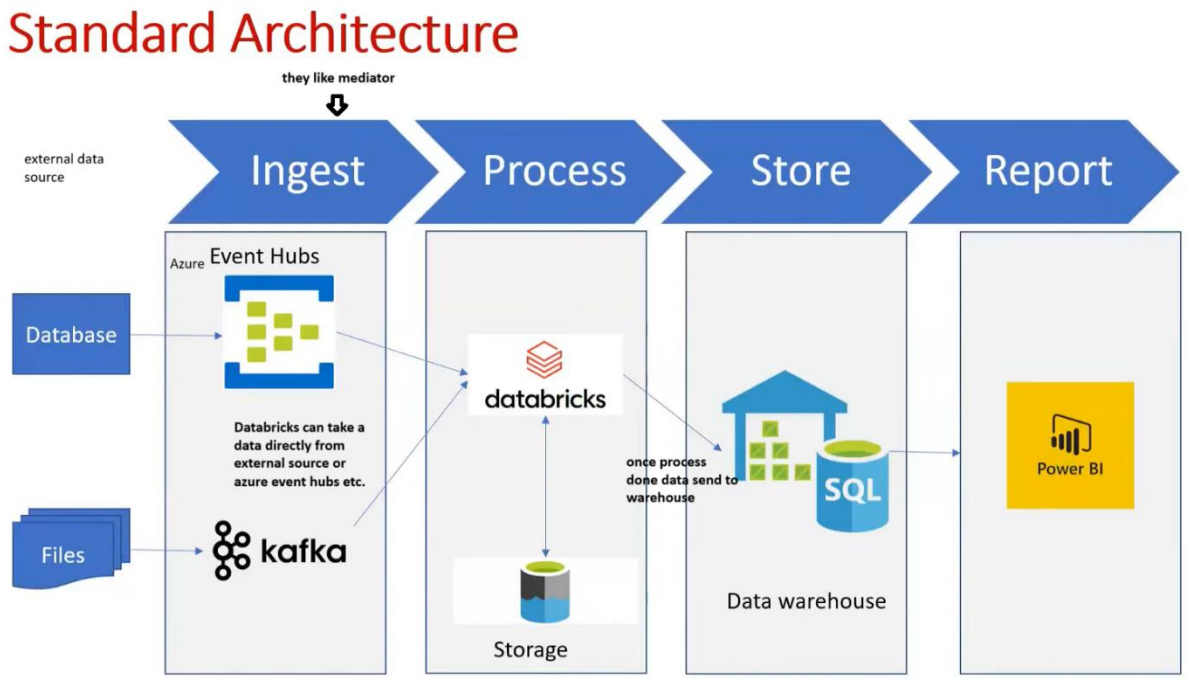


Structured Streaming

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