Data Formats Deep Dive

Databricks

This is the overview page for **Databricks**.

Azure Databricks Uc Creation

How to configure and create Unity Catalog in Azure

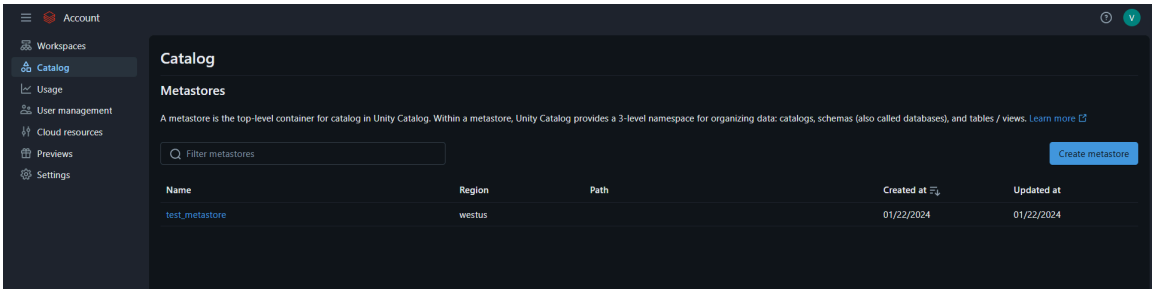
We need to setup Storage Account under the same RG where Dsatabricks Workspace is deployed.

Also we need to create a UC Databricks Connector to connect to the resource group.

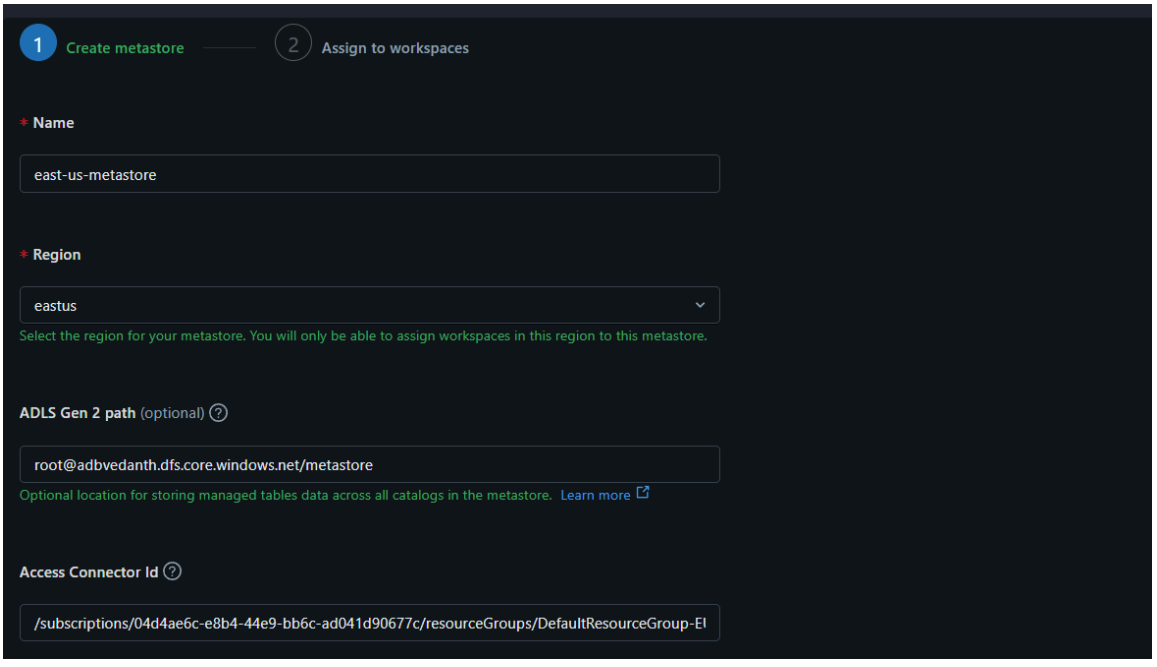
Then give Storage Data Contributor access to the UC Databrticks Connector Managed Identity.

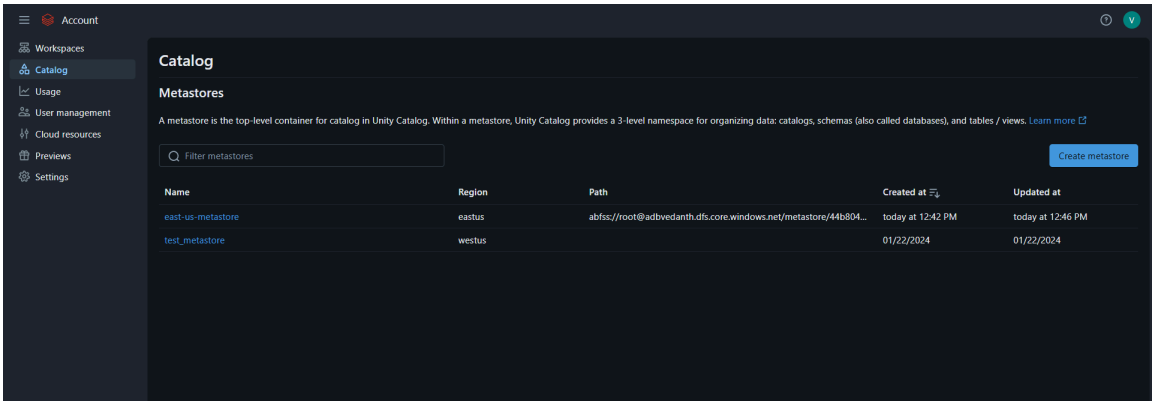
Go to Databricks Login Page and select the account that is the admin.

Go to catalog and create a new metastore.



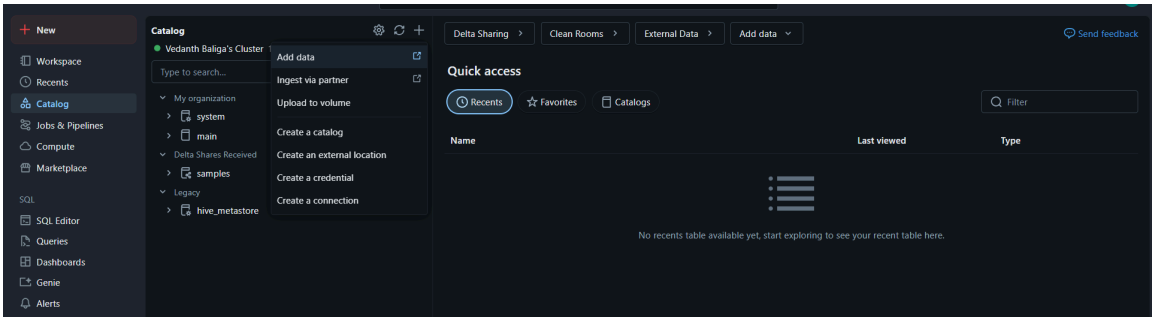
Fill in the details





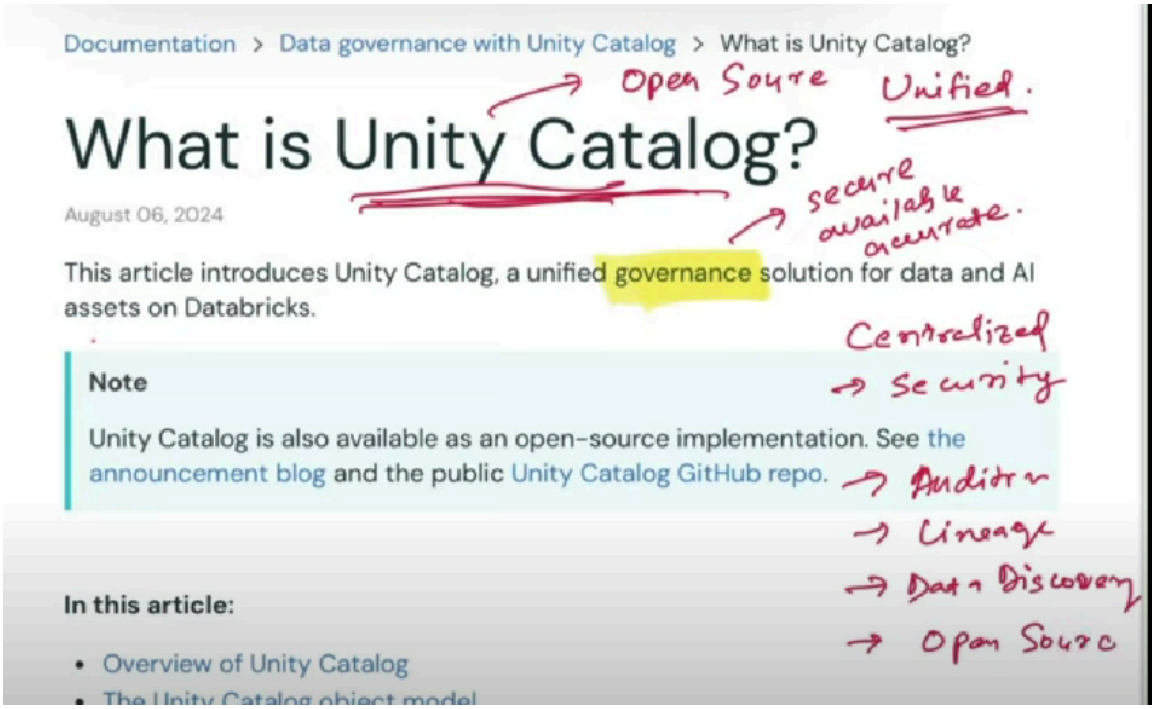
Imp : Login from your admin account

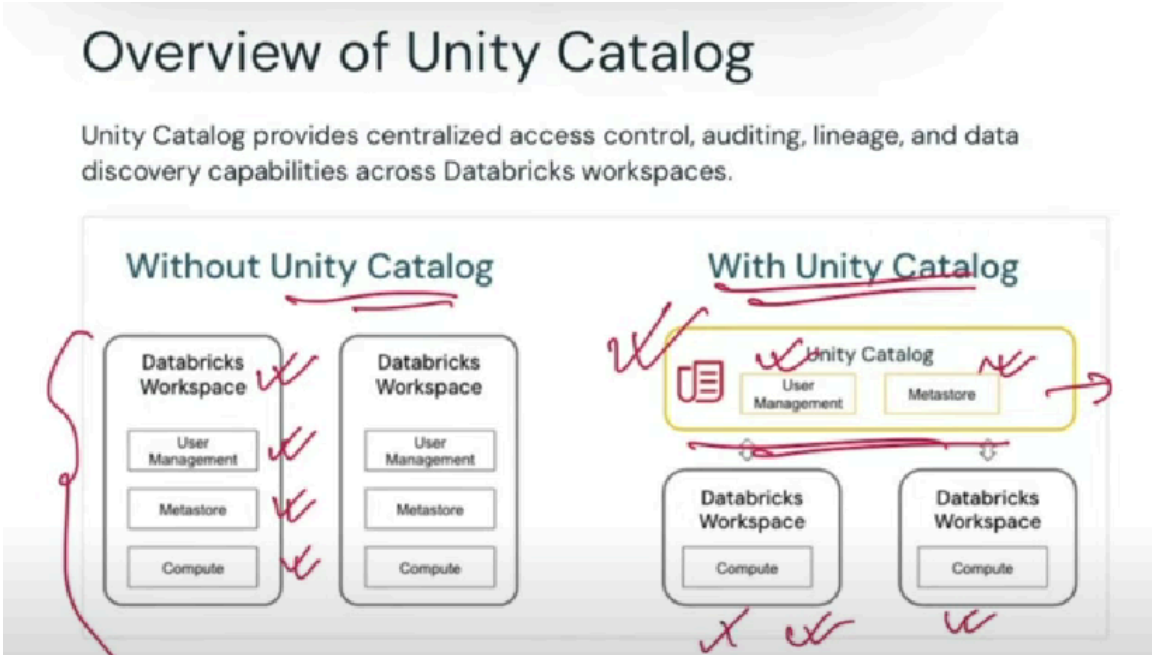
We can now see more options like Add Catalog, Add Credentials, Create Volume after UC is enabled.

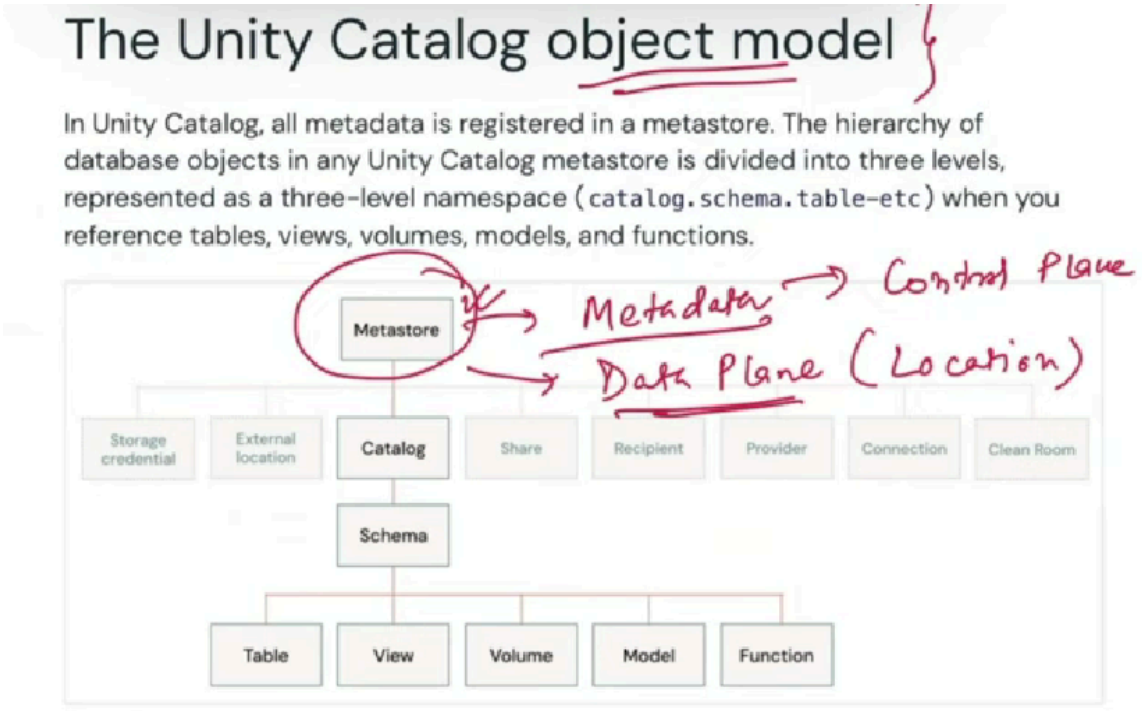


Databricks Uc Introduction

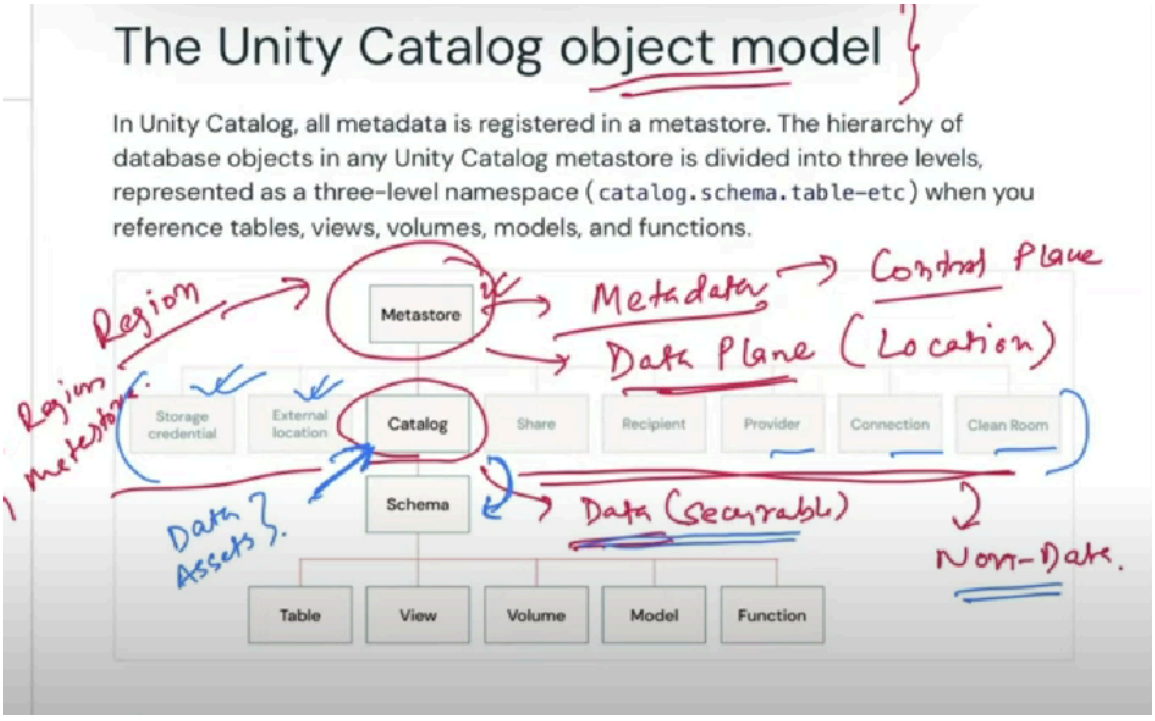
Databricks Unity Catalog Introduction







⚠ Metadata in Metastore is stored on control plane and actual data is on data plane.



Catalog is called data securable object because without having access to `USE CATALOG` we cannot query any data objects.

Databricks Managed External Tables Hive

Managed and External Tables in Hive Metastore

1️⃣ Your command

```
CREATE TABLE IF NOT EXISTS dev.bronze.table_1
LOCATION 'dbfs:/tmp/tmp_ext'
```

- You are explicitly setting a **LOCATION**.
- You are creating the table in the **Hive Metastore** (because you didn't specify Unity Catalog).
- That means: **this is an *external table***, *not* a managed one.

2️⃣ Where does the data go?

- If you **don't specify a LOCATION** and use Hive Metastore → the data is stored in the **Databricks-managed DBFS root (a hidden storage account in the Databricks Managed Resource Group)**. That's the "managed table" case.
- Since you **did specify** `LOCATION dbfs:/tmp/tmp_ext` → the data is stored in **DBFS**, which itself is backed by **the same hidden storage account** unless you mounted another storage.
- So in your case, yes — it is still in the **Databricks-managed storage account**, but specifically under the **DBFS** `/tmp` **folder**, not the default warehouse root (`/user/hive/warehouse`).

3️⃣ Important distinction: Hive Metastore vs Unity Catalog

- **Hive Metastore tables**:
- Data goes to DBFS (Databricks-managed storage account).
- Security is weaker (workspace-scoped, not account-scoped).
- **Unity Catalog tables**:

- Data must go to a customer-managed **external storage account** (ADLS Gen2 / S3 / GCS).
- Databricks does **not** put UC data into its hidden resource group storage.

---

4 Managed vs External recap

- **Managed table (no LOCATION)** → Hive Metastore puts data in Databricks-managed DBFS root.
- **External table (with LOCATION)** → Data goes exactly where you point it (`dbfs:/...` → Databricks storage, `abfss:/...` → your ADLS).

 So, in your example: Yes — the data **is still stored in the Databricks managed resource group storage account**, because `dbfs:/tmp/...` points to DBFS root, which is Databricks-managed storage.

Data Retention?

- In Hive Metastore once a managed table is deleted its whole data is also purged forever.
- This is not the case with UC, unless we do VACUUM even if we drop the table we still can see data in storage account for 7 days by default.

Uc External Location Storage Credentials

Managed Table Locations in Unity Catalog

1. External Metastore Location is provided.

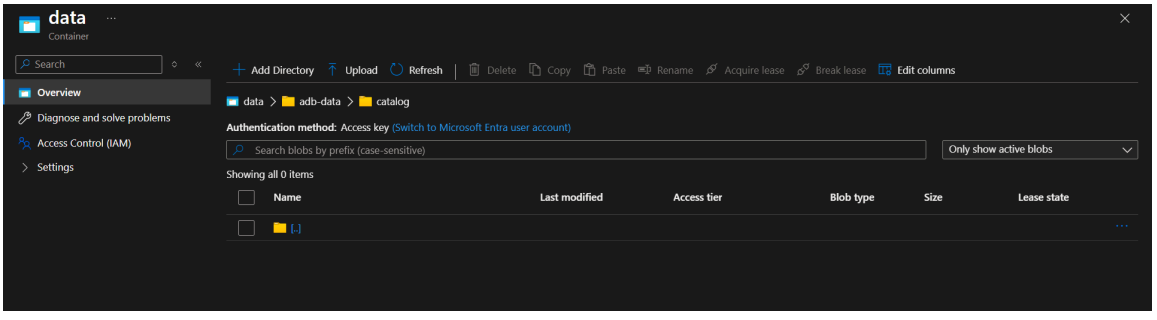
If the metastore storage account is provided by us, then all the catalog, schema and tables is created under this location. At catalog level the location doesnt matter its optional.

1. If Metastore location is not provided.

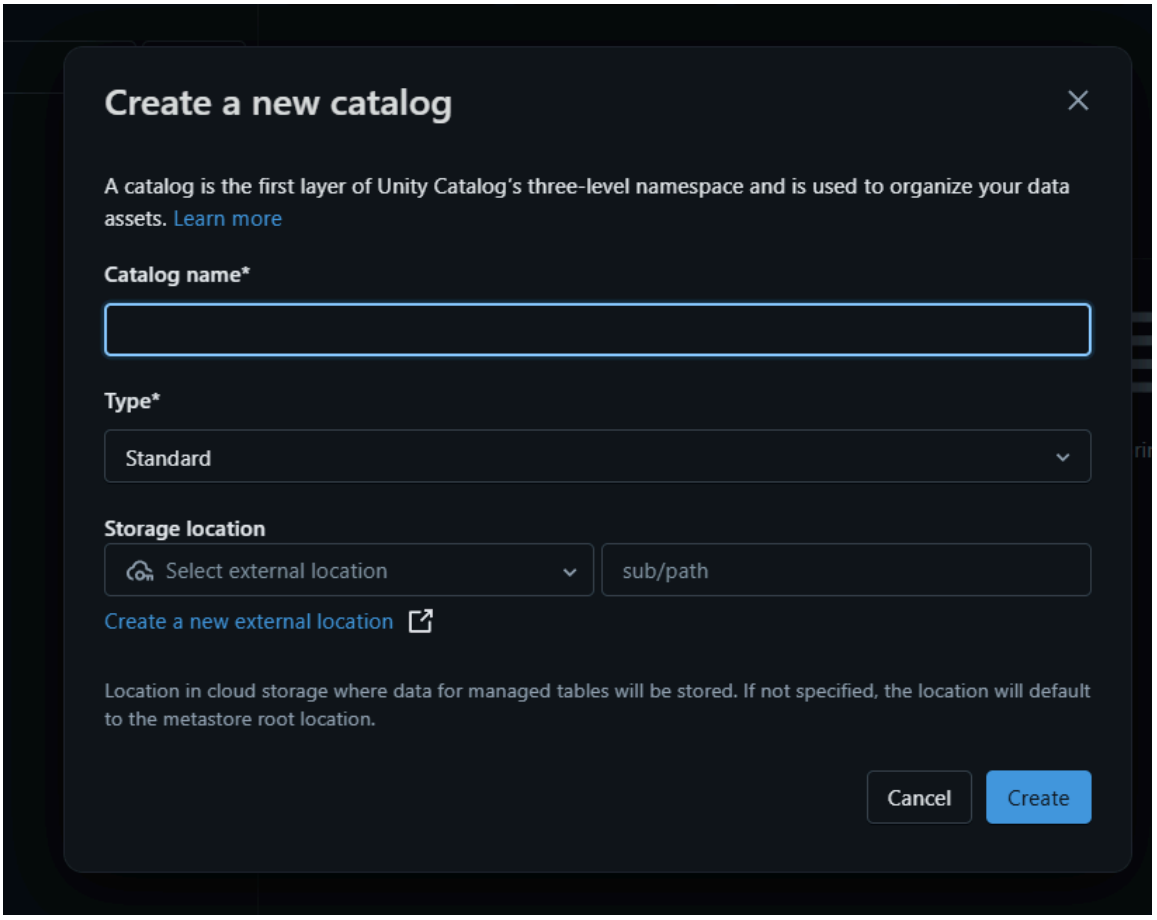
In this case we need to provide the location at catalog level to create tables. This is mandatory.

```
-- Managed Table locations

METASTORE

CATALOG (LOC)

SCHEMA

TABLES
```

Creating a container for our catalog



All data for the managed tables will be stored under this external location if we dont specify it at the schema level.



How things get affected when providing location at different levels in UC?

1️⃣ External location at Metastore level

When you create a Metastore, you must give it a storage root. Example:

```
CREATE METASTORE my_metastore
  LOCATION 'abfss://uc-metastore@<storageaccount>.dfs.core.windows.net/'
```

This location is the default root for managed tables if no other path is specified.

Any catalog/schema/table created as managed without its own external location will fall back here.

✅ So think of it like a global default storage for all managed tables across catalogs.

2️⃣ External location at Catalog level

When you create a catalog, you can optionally give it its own external location:

```
CREATE CATALOG raw_data
  USING EXTERNAL LOCATION my_external_location;
```

Now, managed tables inside this catalog will go under this catalog-specific external location (instead of the metastore root).

This is useful for separating zones (raw, curated, gold) into their own ADLS containers/folders.

So catalogs can “override” the metastore root location.

3 External Tables vs Managed Tables

Managed Table: Unity Catalog controls the data lifecycle. Dropping the table deletes data.

Data lives in either: The Metastore storage root, if no catalog location is set. The Catalog’s external location, if provided.

External Table: You explicitly provide a LOCATION when creating the table.

Example

```
-- Metastore root
CREATE METASTORE main_metastore
  LOCATION 'abfss://uc-metastore@stacc1.dfs.core.windows.net/';

-- Catalog with its own location
CREATE CATALOG raw_data
  USING EXTERNAL LOCATION raw_external;

-- Managed table in raw_data (goes to raw_external)
USE CATALOG raw_data;
CREATE TABLE users_managed (id INT, name STRING) USING DELTA;

-- External table (explicit path overrides everything)
CREATE TABLE users_external (id INT, name STRING)
USING DELTA
LOCATION 'abfss://special@stacc2.dfs.core.windows.net/custom_path/';
```

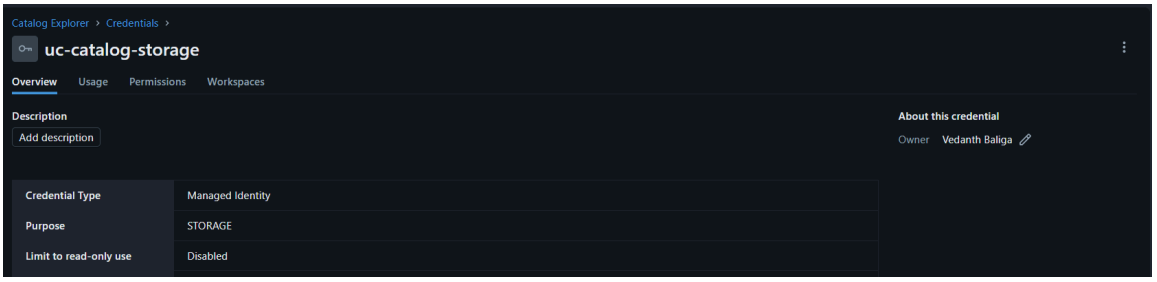
Behavior in Different Scenarios

Scenario	What Happens
Metastore with root storage	Managed tables go to metastore root (or catalog location, if defined).
Metastore with catalog-level external location	Managed tables inside that catalog go to catalog’s external location, not metastore root.
Metastore without root storage (not allowed)	UC creation fails — because even if all catalogs were external, Unity Catalog still enforces root storage for safety.
External table with explicit LOCATION	Works fine — UC just manages metadata, regardless of metastore root.

Creating External Location For Catalog

1. Creating the Storage Credential

This is basically using previous UC Connector to connect to our new storage container for the catalog data.



1. Create the external location

```
CREATE EXTERNAL LOCATION ext_catalog_dev
URL 'abfss://data@adbvedanthnew.dfs.core.windows.net/data/catalog'
WITH (STORAGE CREDENTIAL `uc-data-storage`);
```

```
CREATE CATALOG dev_ext MANAGED LOCATION
'abfss://data@adbvedanthnew.dfs.core.windows.net/data/catalog' COMMENT 'This
is external storage catalog'
```

Databricks Managed Location Catalog Schema Level

□ Unity Catalog Setup from Scratch (Azure + Databricks)

Using UC Connector as Managed Identity

□ Part 1: Azure Setup

1. Create a Storage Account

In Azure Portal, search Storage Accounts → Create.

Important settings:

Performance: Standard Redundancy: LRS (for testing) Enable Hierarchical Namespace (HNS) □ (must be ON for Unity Catalog). Create a container inside it (e.g. uc-data).

2. Assign Permissions to UC Managed Identity

In Azure Portal → Storage Account → Access Control (IAM) → Add Role Assignment. Assign these roles to the UC Managed Identity (the connector): - Storage Blob Data Owner (read/write access). - Storage Blob Delegator (needed for ABFS driver).

Scope: Storage Account level (recommended).

□ Tip: You can find the UC managed identity name in Databricks → Admin Console → Identity Federation.

□ Part 2: Databricks Setup (UI + SQL)

3. Verify Metastore

Go to Databricks Admin Console → Unity Catalog → Metastores.

If you don't have one, click Create Metastore. Region = same as Storage Account. Assign this metastore to your workspace.

□ Do not skip this. Unity Catalog won't work without a metastore.

4. Create Storage Credential (UI or SQL)

This ties the Azure Managed Identity to Unity Catalog.

UI: Go to Catalog → Storage Credentials → Create. Select Azure Managed Identity. Enter a name (e.g., uc-cred). Paste the UC Connector Managed Identity ID.

```
CREATE STORAGE CREDENTIAL uc_cred
WITH AZURE_MANAGED_IDENTITY 'your-managed-identity-client-id'
COMMENT 'UC Connector credential for ADLS';
```

5. Register an External Location

This points Unity Catalog to your ADLS container/folder.

UI: Go to Catalog → External Locations → Create. Name: ext_loc_dev Path: abfss://uc-data@.dfs.core.windows.net/ Storage credential: uc-cred (Storage Credential is at Storage Account level so for one cred is enough)

```
CREATE EXTERNAL LOCATION ext_loc_dev
URL 'abfss://uc-data@<storageaccount>.dfs.core.windows.net/'
WITH (STORAGE CREDENTIAL uc_cred)
COMMENT 'External location for dev data';
```

6. Create Catalog

Decide whether it will use:

Metastore root storage (if defined), OR Catalog-level managed location (recommended).

UI:

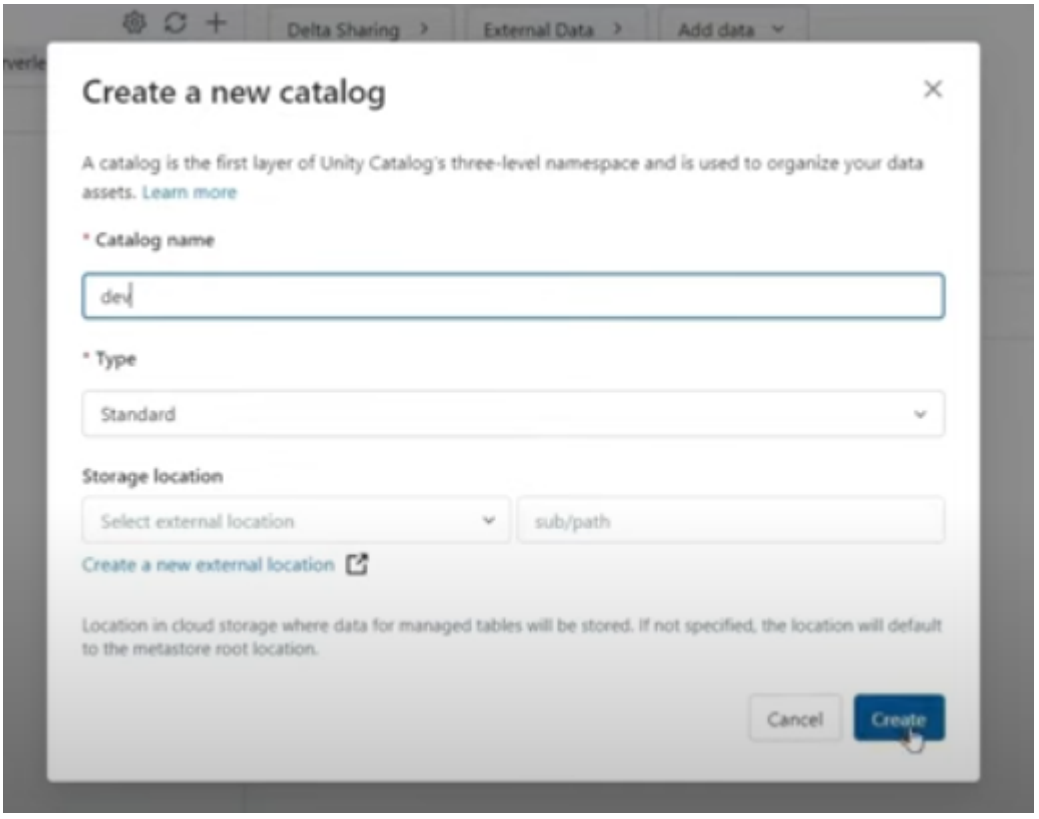
Go to Catalog Explorer → Create Catalog. Name: dev Managed Location:

```
abfss://uc-data@<storageaccount>.dfs.core.windows.net/dev
```

Storage credential: uc-cred.

SQL equivalent:

```
CREATE CATALOG dev
MANAGED LOCATION 'abfss://uc-data@<storageaccount>.dfs.core.windows.net/dev';
```



7. Create Schema

Schemas can also have their own managed locations (if needed).

```
CREATE SCHEMA dev.bronze
MANAGED LOCATION 'abfss://uc-
data@<storageaccount>.dfs.core.windows.net/dev/bronze';
```

9. Grant Permissions

For yourself or a group (like account users):

```
-- Catalog usage
GRANT USAGE ON CATALOG dev TO `account users`;

-- Schema usage
GRANT USAGE ON SCHEMA dev.bronze TO `account users`;

-- External location permissions
GRANT READ FILES, WRITE FILES ON EXTERNAL LOCATION ext_loc_dev TO `account
users`;

-- Table level
GRANT SELECT, MODIFY ON TABLE dev.bronze.trades TO `account users`;
```

10. Create Tables at Different Levels

```
CREATE SCHEMA dev.bronze
COMMENT 'This is schema in dev catalog without external location'
```

This gets created in metastore level container because its managed and we havent specified external location at catalog level.

```
-- CREATE A TABLE UNDER ALL THREE SCHEMA
CREATE TABLE IF NOT EXISTS dev.bronze.raw_sales (
  id INT,
  name STRING,
  invoice_no INT,
  price double
);

INSERT INTO dev_ext.bronze.raw_sales VALUES (1,'Cookies',1,200.50);
```

This gets created in catalog level container because the catalog associated to the schema is external.

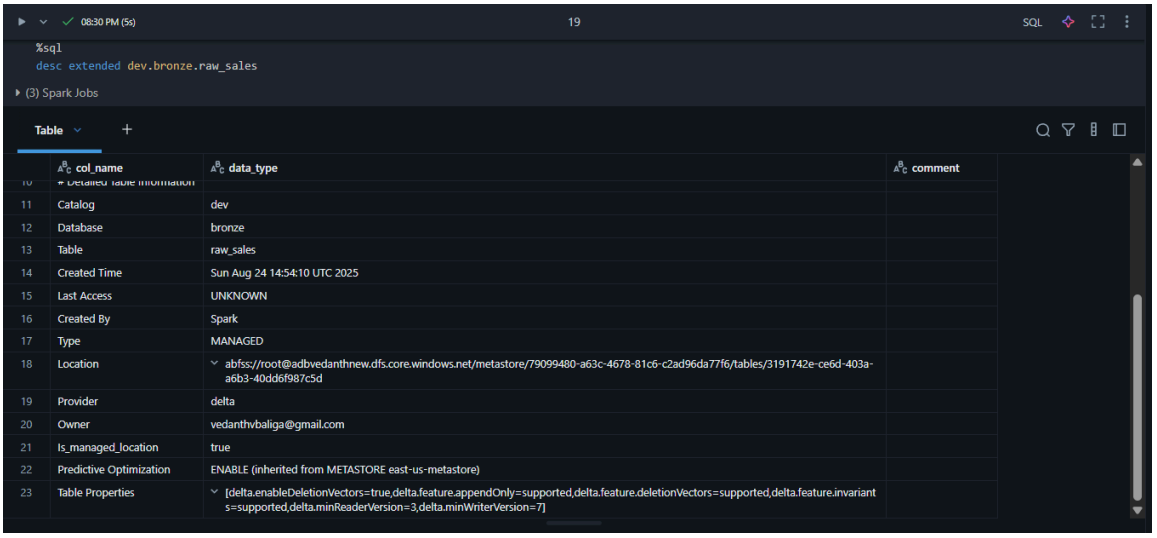
```
-- CREATE A TABLE UNDER ALL THREE SCHEMA
CREATE TABLE IF NOT EXISTS dev_ext.bronze.raw_sales (
  id INT,
  name STRING,
  invoice_no INT,
  price double
);

INSERT INTO dev_ext.bronze.raw_sales VALUES (1,'Cookies',1,200.50);
```

Creating schema in external location.

```
CREATE EXTERNAL LOCATION 'ext_schema'
MANAGED LOCATION 'https://adbvedanthnew.databricks.net/adb/schema/bronze_ext'
```

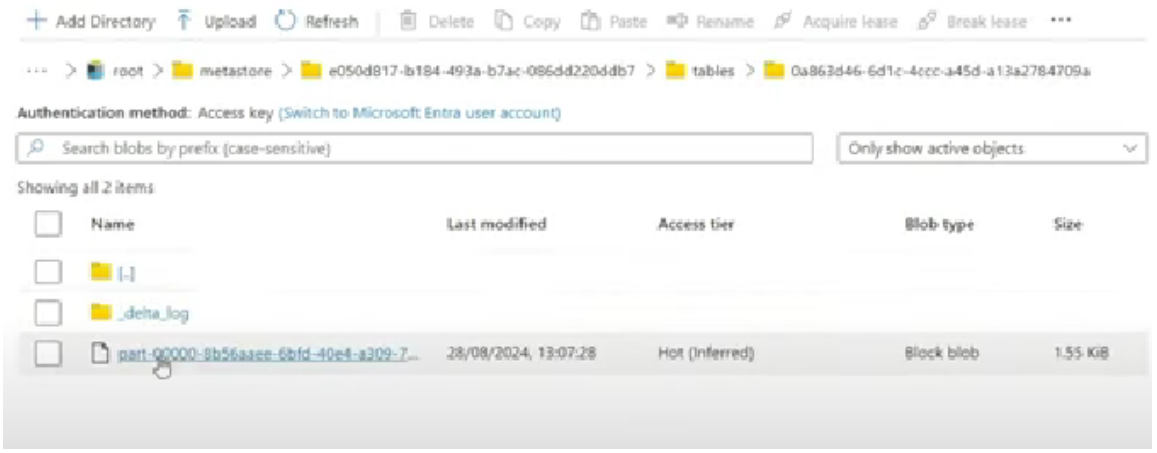
This gets created in schema level since we specified external location at schema level.

```
-- CREATE A TABLE UNDER ALL THREE SCHEMA
CREATE TABLE IF NOT EXISTS dev_ext.bronze_ext.raw_sales (
  id INT,
  name STRING,
  invoice_no INT,
  price double
);

INSERT INTO dev_ext.bronze_Ext.raw_sales VALUES (1,'Cookies',1,200.50);
```



Where data is stored?

1. Stored in metastore root location



1. Store in Catalog Level Ext Location

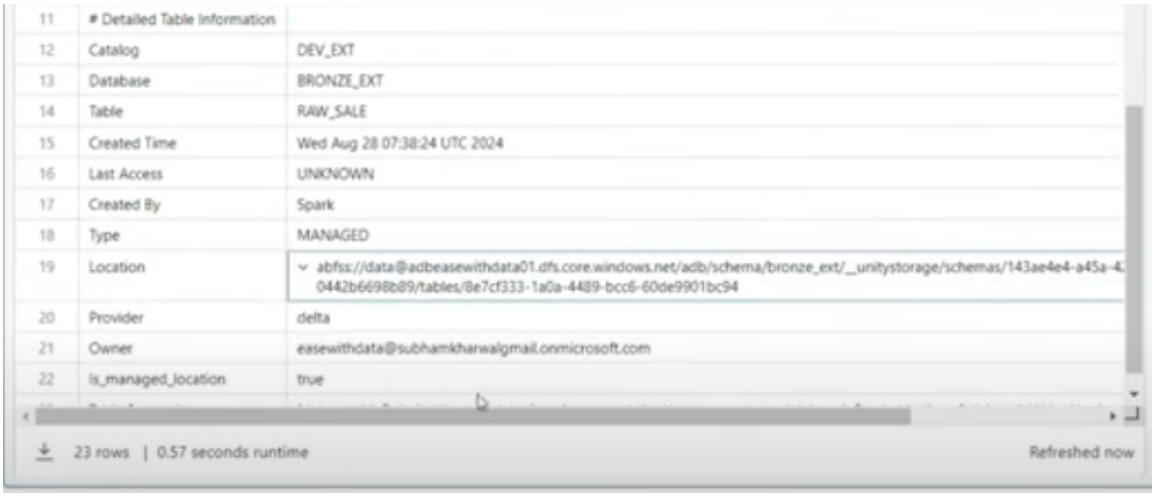
```
DESC EXTENDED DEV_EXT.BRONZE.RAW_SALE;
```

11	# Detailed Table Information	
12	Catalog	DEV_EXT
13	Database	BRONZE
14	Table	RAW_SALE
15	Created Time	Wed Aug 28 07:38:02 UTC 2024
16	Last Access	UNKNOWN
17	Created By	Spark
18	Type	MANAGED
19	Location	abfss://data@adbeasewithdata01.dfs.core.windows.net/adb/catalog/__unitystorage/catalogs/5a84b28b-da4a-4979-9898-e1
20	Provider	delta
21	Owner	easewithdata@subhamkharwalgmail.onmicrosoft.com
22	is_managed_location	true
23	Table Properties	[delta.enableDeletionVectors=true,delta.feature.deletionVectors=supported,delta.minReaderVersion=3,delta.minWriterVersion=

23 rows | 0.68 seconds runtime — Refreshed now

1. Store in Schema Level External Location

```
DESC EXTENDED DEV_EXT.BRONZE_EXT.RAW_SALE
```



11	# Detailed Table Information	
12	Catalog	DEV_EXT
13	Database	BRONZE_EXT
14	Table	RAW_SALE
15	Created Time	Wed Aug 28 07:38:24 UTC 2024
16	Last Access	UNKNOWN
17	Created By	Spark
18	Type	MANAGED
19	Location	abfss://data@adbeasewithdata01.dfs.core.windows.net/adb/schema/bronze_ext/__unitystorage/schemas/143ae4e4-a45a-4... 0442b6698b89/tables/8e7cf333-1a0a-4489-bcc6-60de9901bc94
20	Provider	delta
21	Owner	easewithdata@subhamkharwalgmail.onmicrosoft.com
22	is_managed_location	true

23 rows | 0.57 seconds runtime — Refreshed now

Ctas Deep Clone Shallow Clone Databricks

CTAS | Deep Clone | Shallow Clone

Create Table As Select (CTAS)

```
CREATE TABLE dev.bronze.sales_ctas
AS
SELECT * FROM dev.bronze.sales_managed
```

A new physical location is created and data is now stored in that for the new table.

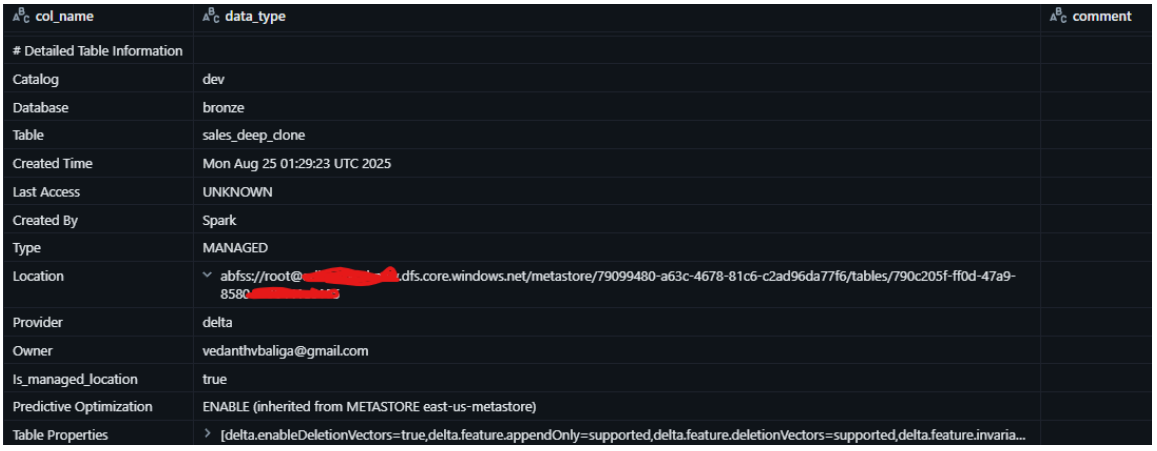
```
-- location is different from original table, the physical location is
changed.
desc extended dev.bronze.sales_ctas
```

Deep Clone

```
CREATE TABLE dev.bronze.sales_deep_clone DEEP CLONE dev.bronze.sales_managed;
select * from dev.bronze.sales_deep_clone;
desc extended dev.bronze.sales_deep_clone;
```

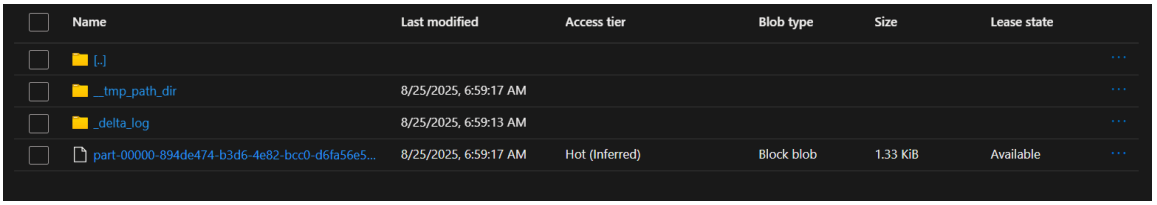
This copies both table data and metadata.

The data is created in new locaiton.

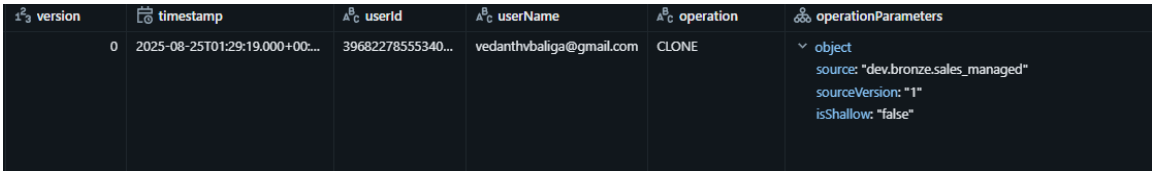


col_name	data_type	comment
# Detailed Table Information		
Catalog	dev	
Database	bronze	
Table	sales_deep_clone	
Created Time	Mon Aug 25 01:29:23 UTC 2025	
Last Access	UNKNOWN	
Created By	Spark	
Type	MANAGED	
Location	abfss://root@[redacted].dfs.core.windows.net/metastore/79099480-a63c-4678-81c6-c2ad96da77f6/tables/790c205f-ff0d-47a9-8580[redacted]	
Provider	delta	
Owner	vedanthvbaliga@gmail.com	
Is_managed_location	true	
Predictive Optimization	ENABLE (inherited from METASTORE east-us-metastore)	
Table Properties	[delta.enableDeletionVectors=true,delta.feature.appendOnly=supported,delta.feature.deletionVectors=supported,delta.feature.invaria...	

Both delta logs and data copied over

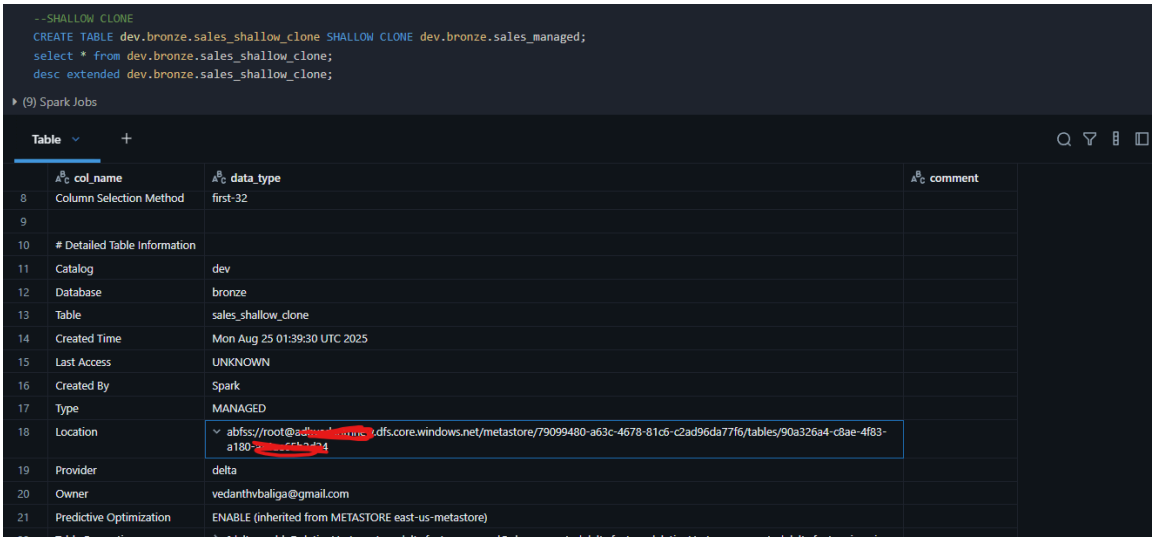


We can see latest version of source from where data from table is copied.

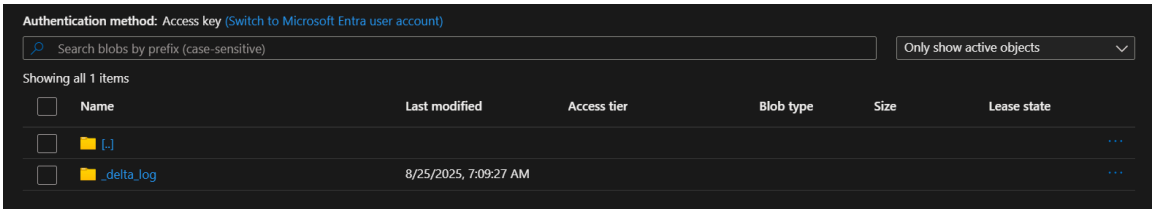


Shallow Clone

The table is created at a new location

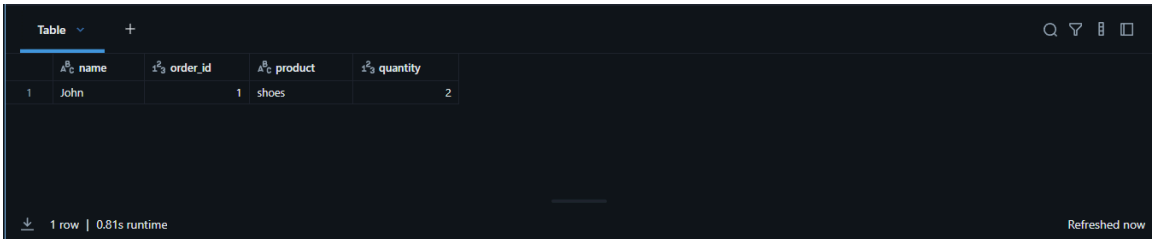


Only Delta Log copied over the physical data stored at original location.

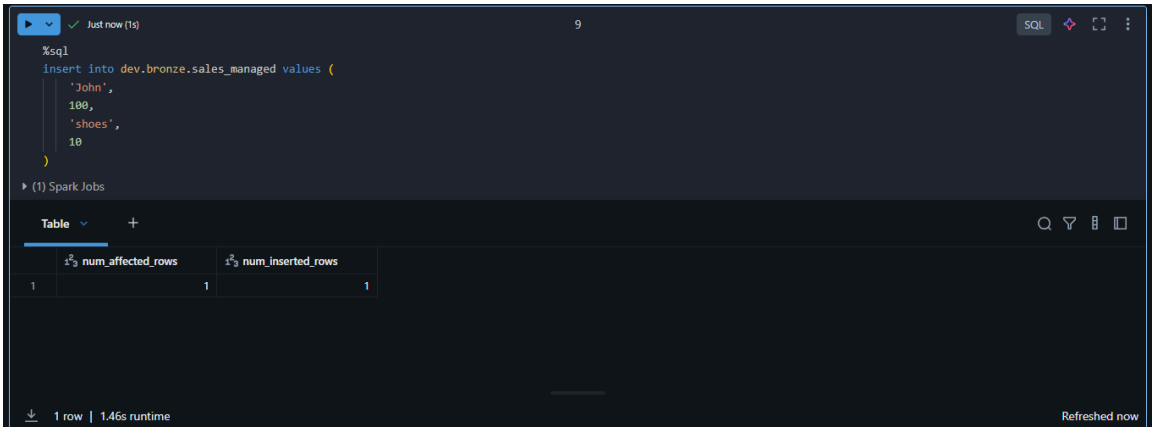


Inserting Data into original table and see if it reflects in Shallow Clone

```
select * from dev.bronze.sales_shallow_clone
```



Insert data into main table



Requery shallow clone

We can see only one row the 2nd row inserted is not visible.

Shallow Clone points to the source version of original table when created and does not update the data.

When we do the vice versa and insert new record in shallow table, it does not impact the original table. The new data is stored in the shallow table's own location.

Only when we VACUUM the main table, all records from shallow table also gets deleted.

Rbac Custom Roles Serviceprincipals

Role Based Access Control | Custom Role Definitions | Service Principals

1 Azure RBAC Roles (Role-Based Access Control)

Purpose: RBAC in Azure controls who can do what on which resource.

Key Concepts:

Scope: Defines where the role applies (Subscription → Resource Group → Resource).

Role: Defines what actions can be performed.

Principal: The identity (user, group, or application) assigned the role.

Role	Description	Example Use Case
Owner	Full access to everything, including assigning roles	Admin who manages the subscription
Contributor	Can create/update/delete resources, but cannot assign roles	DevOps engineer creating storage, VMs
Reader	Can view resources but cannot make changes	Auditor reviewing configurations
Storage Blob Data Owner/Contributor/Reader	Specific to storage accounts	Grant a user ability to read/write blobs in a container

2 Custom Roles

Purpose: When built-in roles are too broad or restrictive, you can create custom roles with exactly the permissions you need.

How it works:

- Define a JSON file with allowed actions (Microsoft.Storage/storageAccounts/blobServices/containers/read, etc.)
- Assign it to users/groups/service principals.

Example JSON for a custom role (ADLS read-only access):

```
{
  "Name": "ADLS ReadOnly",
  "IsCustom": true,
```

```
  "Description": "Read-only access to ADLS Gen2 containers",
  "Actions": [
    "Microsoft.Storage/storageAccounts/blobServices/containers/read",
    "Microsoft.Storage/storageAccounts/blobServices/containers/blobs/read"
  ],
  "NotActions": [],
  "AssignableScopes": ["/subscriptions/<subscription-id>"]
}
```

When to use:

You want a minimal-privilege principle, e.g., a BI service can only read blobs, not delete them.

3 Service Principals

Purpose: A service principal is like a “user identity” for applications, scripts, or automated services.

Why needed:

- Azure RBAC requires an identity for access.
- You don’t want to use your personal account for automated tasks.

Example: Databricks accessing ADLS via a service principal.

Types of Service Principals Authentication:

- Client Secret: Password-like string.
- Certificate: Secure certificate authentication.
- Managed Identity (Recommended for Databricks UC Connector): Azure handles the credentials for you.

How it works in practice (Databricks + ADLS)

- Create a service principal in Azure AD.
- Assign RBAC (e.g., Storage Blob Data Contributor) on the storage account/container.
- Use this SP to create a Databricks Storage Credential (or UC connector).
- Unity Catalog or your clusters use the SP to access storage without exposing your personal account.

Azure AD Interview Questions (for Data Engineers)

Q1. What is Azure Active Directory (Azure AD)?

Answer: Azure AD is Microsoft’s cloud-based identity and access management service. It authenticates users, applications, and services, and authorizes them to access Azure resources. Unlike on-prem AD, Azure AD is designed for cloud-first apps, RBAC, and SSO.

Q2. What is the difference between Azure AD Users, Groups, Service Principals, and Managed Identities?

Answer:

Users → Human identities (employees, admins).

Groups → Collection of users/SPs for easier role assignment.

Service Principal (SP) → Non-human identity for applications to access resources.

Managed Identity → A special type of SP managed automatically by Azure, used by Azure services (like Databricks, ADF) to access resources without credentials.

Q3. What's the difference between Service Principal and Managed Identity?

Answer:

Service Principal → You create it manually, assign roles, and manage secrets/certs.

Managed Identity → Azure creates/rotates credentials automatically, no secrets to manage. Example: Databricks UC Connector → Managed Identity (no secrets). A legacy pipeline using Python SDK → Service Principal (with client secret).

Q4. How does RBAC work in Azure?

Answer:

RBAC (Role-Based Access Control) grants permissions at scope levels: Subscription → Resource Group → Resource.

Roles are assigned to principals (user, group, SP, or MI).

Built-in roles include: Owner, Contributor, Reader, Storage Blob Data Reader/Contributor. Example: Assign Storage Blob Data Contributor to a Databricks SP so it can read/write to ADLS.

Q5. How is Azure AD different from On-prem Active Directory?

Answer:

AD (On-prem) → Kerberos/NTLM, domain-joined machines, Windows environments.

Azure AD → OAuth2, SAML, OpenID Connect, cloud-first, SSO, SaaS app integration.

Azure AD cannot join servers to a domain but can integrate with ADDS (hybrid).

Q6. What is Conditional Access in Azure AD?

Answer: It enforces policies like MFA, location restrictions, or device compliance before granting access. Example: Require MFA for accessing Databricks workspace from outside corporate network.

Q7. What is a Custom Role in Azure AD?

Answer:

Built-in roles may not cover all needs.

Custom roles let you define granular actions (e.g., “read blobs, but not delete”). Example: A custom role for analysts → can read raw/curated ADLS folders but not write/delete.

Q8. What are the authentication protocols supported by Azure AD?

Answer:

OAuth 2.0 → App-to-app access (SPs, APIs)

OpenID Connect (OIDC) → User authentication + SSO

SAML 2.0 → Enterprise SSO with third-party apps

SCIM → User/group provisioning

Example: Databricks notebooks → ADLS (OAuth 2.0 via SP/MI).

Q9. Explain a real-world flow of Databricks accessing ADLS with Unity Catalog and Azure AD.

Answer:

User runs a query in Databricks notebook.

Unity Catalog enforces permissions (does user have SELECT?).

Databricks uses storage credential (SP or Managed Identity) registered in UC.

Azure AD authenticates the SP/MI.

ADLS authorizes via RBAC role (Storage Blob Data Contributor).

Data is read/written securely, no secrets exposed.

Q10. What is the difference between Directory Roles vs Azure RBAC Roles?

Answer:

Directory Roles → Control Azure AD objects (users, groups, SPs). Example: Global Admin, User Administrator.

RBAC Roles → Control access to Azure resources (storage, VMs, databases). Example: Storage Blob Data Contributor.

Q11. How would you give different levels of access to Finance vs Data Science teams on the same ADLS account?

Answer:

Create two groups in Azure AD → finance-users, ds-users.

Assign RBAC roles:

Finance → Read access only (custom role or Blob Data Reader).

DS → Read/Write on curated container (Blob Data Contributor).

In Unity Catalog, assign table permissions → Finance: SELECT, DS: SELECT/INSERT/UPDATE.

Q12. What are some common security best practices with Azure AD in Data Engineering?

Answer:

Use Managed Identities instead of secrets.

Use Groups for access, not direct user assignments.

Use Conditional Access (MFA, network restrictions).

Follow least privilege principle with custom roles.

Enable logging (Azure AD logs, storage logs, Databricks audit logs) for compliance.

OpenID Connect (OIDC)

OpenID Connect is an identity layer built on top of OAuth 2.0.

OAuth 2.0 → Handles authorization (what an app can do on your behalf).

OIDC → Adds authentication (who you are, your identity).

How it works (simple flow):

A user tries to log in to an app (e.g., Databricks). The app redirects them to Azure AD (the identity provider) using OIDC. Azure AD authenticates the user (password, MFA, etc.).

Azure AD returns tokens:

ID Token (JWT) → contains identity info (username, email, groups). Access Token → lets the app call APIs on user's behalf. The app trusts the ID token and logs the user in.

Example in Azure AD + Databricks

You open Databricks workspace in browser. Databricks uses OIDC with Azure AD to authenticate you. Azure AD issues an ID token with your email + groups. Databricks checks your group → grants access based on Unity Catalog permissions.

Why OIDC is important?

It enables SSO (Single Sign-On) across cloud apps. Works with MFA and Conditional Access. Uses JWT tokens that are stateless and easy to validate.

 Summary for interview: OpenID Connect is an authentication protocol built on OAuth 2.0. It issues ID tokens (JWTs) that allow apps to verify a user's identity and support SSO. In Azure AD, OIDC is used when logging into cloud apps like Databricks, Power BI, or ADF.

Imagine this:

You want to enter a party . The party organizer is the app (like Databricks). At the door, they don't know you… so they send you to the government office (Azure AD).

What happens:

You go to the government office (Azure AD). You show your ID card, fingerprint, maybe even OTP → they confirm you are really you . They give you a badge (the ID token). You take that badge back to the party .

The party organizer looks at the badge → "Okay, you are Vedanth, you're allowed in."

If you want to get food or drinks inside, the badge can also have permissions (access token) telling the staff what you're allowed to do.

Difference:

- OAuth 2.0 → Badge only says what you can do inside the party.
- OIDC → Badge also says who you are.

Deletion Vectors Delta Lake

Deletion Vectors in Delta Lake

What are Deletion Vectors?

Normally, when you delete rows in a Delta table, Delta rewrites entire Parquet files without those rows.

This is called copy-on-write → expensive for big tables.

Deletion Vectors (DVs) are a new optimization:

Instead of rewriting files, Delta just marks the deleted rows with a bitmap (a lightweight "mask"). The data is still physically there, but readers skip the "deleted" rows.

Think of it like putting a red X mark on rows instead of erasing them immediately.

Why are they useful?

 Much faster deletes/updates/merges (because files aren't rewritten).

⚡ Less I/O → good for big data tables.

 Efficient for streaming + time travel.

Example Without deletion vectors

1. Create a sales table

```
CREATE TABLE dev.bronze.sales as
select * from
read_files(
  'dbfs:/databricks-datasets/online_retail/data-001/data.csv',
  header => true,
  format => 'csv'
)
```

1. Set Deletion Vectors false

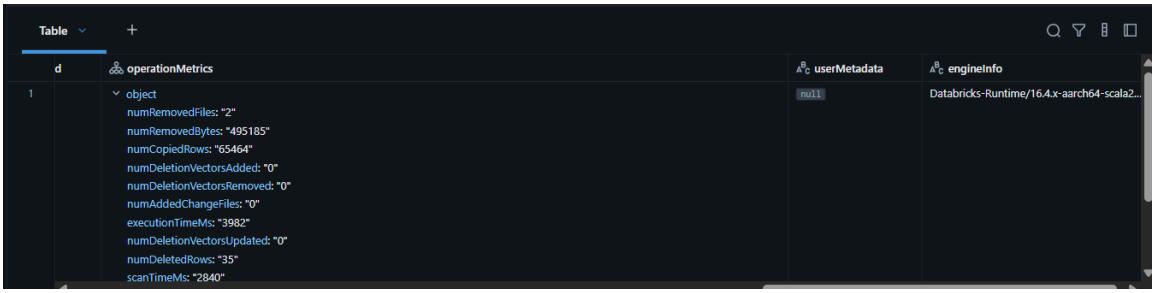
```
ALTER TABLE dev.bronze.sales SET TBLPROPERTIES (delta.enableDeletionVectors =
false);
```

Owner	vedanthvbaliga@gmail.com
Is_managed_location	true
Predictive Optimization	ENABLE (inherited from METASTORE east-us-metastore)
Table Properties	[delta.enableDeletionVectors=false,delta.feature.appendOnly=supported,delta.feature.deletionVectors=supported,delta.feature.invaria...

1. Delete some rows

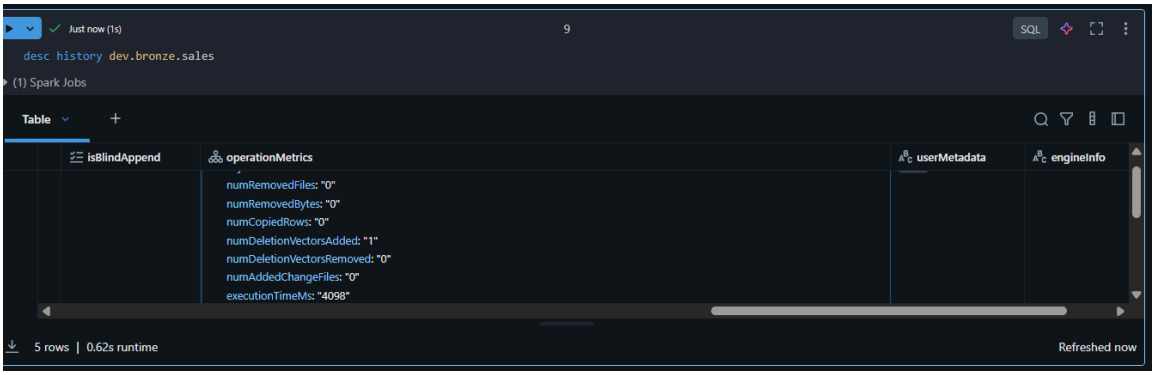
```
-- delete InvoiceNo = '540644'
delete from dev.bronze.sales
where InvoiceNo = '540644'
```

1. Describe history



Observe that all rows (65000+) are removed and rewritten.

Example with deletion vectors



We can see that one deletion vector is added no files are rewritten.

Running optimize would remove those files / records.

Liquid Clustering Delta Lake

Liquid Clustering in Delta Lake and Databricks

Traditional Partitioning (the old way)

When you create a Delta table, you pick a partition column (e.g., date).

Data is physically stored in folders like:

```
/table/date=2025-08-25/
/table/date=2025-08-26/
```

Queries on date are very fast (partition pruning).

But… problems:

You must choose the partition column upfront (hard to change later). Skew → some partitions get huge, others tiny. If you query on a different column (say country), partitioning doesn't help.

What is Liquid Clustering?

Liquid Clustering is next-gen partitioning without rigid partitions.

Instead of fixed folder partitions, Delta uses clustering columns.

Data is automatically organized into files that are co-located based on clustering keys.

No fixed directories — clustering boundaries are "liquid," meaning they can shift over time.

Think of it like:

Partitioning = chopping the cake into fixed slices . Liquid Clustering = marbling the cake so flavors are naturally grouped but flexible .

Feature	Partitioning	Liquid Clustering
Physical layout	Fixed folders	Flexible file clustering
Schema rigidity	Hard to change	Easy to evolve
Multiple dimensions	Poor support	Natively supported
Skew handling	Manual	Auto-rebalanced
Query optimization	Partition pruning	Clustering pruning + skipping

The root problem: concurrent writes

In a traditional partitioned Delta table:

If two jobs write to the same partition folder (say date=2025-08-25), they may overwrite each other's files, create tons of small files, or cause conflicts.

Delta's transaction log (_delta_log) prevents corruption, but still you can get:

- Write conflicts
- Compaction/reorg problems
- Skewed partitions

What Liquid Clustering does differently

Liquid Clustering removes the dependency on static partition folders.

There is no date=2025-08-25/ folder.

Instead, data is stored in files spread across the table storage, tagged internally with clustering metadata.

When multiple jobs write:

- The Delta transaction log coordinates atomic commits.
- Writers don't fight for the same fixed folder (no "hotspot").
- Databricks automatically distributes new rows into the right clustering ranges.

⚡ **How concurrent writes are prevented**

Transaction log serialization

Every write creates a new JSON transaction in _delta_log. If two jobs conflict, Delta retries or errors out gracefully — no corruption.

No rigid partitions

Since clustering is "liquid", two writers can both insert data with the same date or country values. Delta decides file placement dynamically (not tied to a single folder).

Background clustering

Databricks runs auto-optimization jobs to maintain clustering quality. Even if concurrent writes scatter data, the optimizer later reorganizes files.

Reduced small files problem

With partitions, concurrent writers often create many tiny files in the same folder. With Liquid Clustering, writers spread load across cluster ranges → fewer hotspots.

Pizza Shop Analogy

Imagine you and your friends are delivering pizzas to an office building.

Old Way (Partitions)

The building has one mailbox per floor. If two delivery guys (writers) come to the same floor mailbox at the same time, they fight for space. The mailbox gets messy, pizzas overlap, and sometimes one delivery overwrites the other. This is like partitioned Delta tables → if two jobs write to the same partition folder, conflicts happen.

New Way (Liquid Clustering)

Now the building switches to smart lockers (clustering ranges).

When a delivery comes in, the system automatically assigns any free locker on that floor.

Two delivery guys can deliver pizzas for the same floor at the same time, but the system spreads them across different lockers.

Later, the building staff reorganizes lockers (background clustering) so pizzas for the same person are grouped together neatly.

This is like Liquid Clustering → no fixed folders, data is dynamically placed, and reorganized in the background.

The Delta log is like the building's register that records every pizza delivered → so no one loses track.

Are there Trade Offs?

With partitions (mailboxes), if you know the floor (partition key), you go directly to that mailbox — super fast for point lookups.

With liquid clustering (smart lockers), pizzas for the same floor (or customer) might be spread across multiple lockers. To find all pizzas for “floor 5,” you may have to open several lockers instead of one → sounds slower, right?

Why it’s not actually that slow in practice:

Clustering index in metadata

Delta keeps track of where rows are stored (think: a digital map of which lockers hold floor 5 pizzas). Readers don’t randomly scan every file; they check the index and skip irrelevant files.

File skipping + statistics

Each data file stores min/max values of the clustering column. So if you query “customer_id = 123,” Delta can skip 90% of files if their min/max range doesn’t cover 123.

Background reclustering

Liquid clustering reorganizes lockers in the background, so “similar pizzas” get grouped closer over time. This means queries get faster the more the system reclusters.

Trade-off (balanced)

Old partitions → fast for single key lookups, but slow for big aggregations (because partitions may be uneven/skewed). Liquid clustering → slightly slower for tiny point lookups, but much faster and balanced for mixed workloads (point lookups + large scans).

Concurrency Liquid Clustering

Multi Version Concurrency Control in Liquid Clustering

Why concurrent write failures still happen

Delta uses Optimistic Concurrency Control (OCC) for all writes, even with Liquid Clustering:

Each writer reads a snapshot of the table (say version 5).

Both writers prepare their changes.

When committing:

Delta checks if the table's latest version is still 5.

If yes → commit succeeds (new version 6).

If another writer already committed version 6 → your commit fails with a ConcurrentWriteException.

This ensures consistency. Without this, two writers could overwrite each other's updates silently.

Liquid Clustering helps with performance, not concurrency

Normally, clustering/partitioning means two writers updating the same partition can easily conflict (both touching the same small set of files). With Liquid Clustering, rows are dynamically redistributed across files, so writers are less likely to clash on the exact same files.

But if two jobs still update overlapping rows (or even metadata) → OCC detects the conflict and one fails.

So: Liquid reduces probability of collisions but does not eliminate them.

Multi Version Concurrency Control

Two delivery guys arrive at the same time

Job A delivers pizzas for floor 5

Job B delivers pizzas for floor 5

With old mailboxes → they'd fight for the same mailbox. Chaos 😵

With liquid clustering + MVCC →

Each delivery guy puts pizzas into their own lockers (new files).

No overwrites. No conflicts.

The building's delivery register (Delta Log)

Every delivery is recorded in the logbook at reception (Delta transaction log).

The log has versions:

Version 1: Deliveries from Job A

Version 2: Deliveries from Job B

So if you "replay the log," you see all deliveries, in order.

Readers never see half-finished deliveries

If someone checks the log while Job A is writing, they still only see the previous version (Version 0).

Once Job A finishes, the log moves to Version 1.

Then readers see all of Job A's pizzas atomically.

→ This guarantees snapshot isolation = you only ever see a consistent view.

Concurrent jobs don't lose pizzas

Even if Job A and Job B write at the same time, MVCC ensures:

Job A's new files → recorded in Version 1

Job B's new files → recorded in Version 2

Both sets of deliveries are preserved.

Real Delta Lake terms:

Log = _delta_log JSON + Parquet files (transaction history).

New version = commit when a write finishes.

Readers always query a stable snapshot version, not files mid-write.

Concurrent writers: no overwrite, because each write creates new files, and old files are marked as removed in the log.

Takeaway:

MVCC in Delta is like a time machine + logbook — every write creates a new version of the table, so no data is lost, no half-baked updates are visible, and readers/writers can happily work in parallel.

New files are created everytime we write to clustered delta table.

Delta Lake never updates files in-place (because they're immutable in cloud storage).

Instead, on every write (insert, update, merge, etc.):

Delta writes new Parquet files with the updated data.

The Delta log (_delta_log) is updated with JSON/Checkpoint metadata pointing to the new set of files.

Old files are marked as removed, but not physically deleted until `VACUUM`

 With Liquid Clustering

Liquid Clustering's job is to keep files balanced by row count (not by fixed partition values).

When you insert → Delta writes new files sized according to Liquid's clustering strategy (e.g., ~1M rows per file).

When you update/merge/delete → Delta rewrites the affected rows into new files, distributed across existing clustering ranges.

The old files are marked as deleted in the log.

 Every commit adds new files and retires old ones.

Demo

 Setup

We'll assume:

A Delta table with Liquid clustering enabled.

Target file size ~1M rows per file (for simplicity).

Step 1 : Create Empty Table

```
CREATE TABLE sales_liquid (
  order_id STRING,
  customer_id STRING,
  amount DECIMAL(10,2),
  date DATE
)
```

```
USING DELTA
CLUSTER BY (date);  -- Liquid Clustering on "date"
```

 At this point:

_delta_log/ has version 000000.json (empty schema).

No data files yet.

Step 2 : Insert First Batch

```
INSERT INTO sales_liquid SELECT ... 2M rows ...
```

 What happens:

Liquid clustering creates ~2 files of ~1M rows each.

_delta_log/000001.json records addFile for these.

File status:

```
file_0001.parquet (~1M)
file_0002.parquet (~1M)
```

Step 3 : Insert Second Batch

```
INSERT INTO sales_liquid SELECT ... 0.5M rows ...
```

 What happens:

New data → 1 new file of ~500k rows.

No rewrite of old files.

_delta_log/000002.json adds metadata.

```
file_0001.parquet (~1M)
file_0002.parquet (~1M)
file_0003.parquet (~0.5M)
```

Step 4 : Update 800k rows spread over files 1 and 2

```
UPDATE sales_liquid SET amount = amount * 1.05 WHERE date BETWEEN '2023-01-01'
AND '2023-03-01';
```

 What happens:

Delta does not edit file_0001/0002 → marks them as removed.

Writes new replacement files (~800k rows redistributed).

```
file_0001  removed
file_0002  removed
file_0003.parquet (~0.5M)
file_0004.parquet (~0.4M)
file_0005.parquet (~0.4M)
```

Step 5 : Delete 100k rows

```
DELETE FROM sales_liquid WHERE customer_id = 'C123';
```

 What happens:

Affected rows come from file_0005.

File_0005 is removed, replaced with smaller rewritten file.

```
file_0003.parquet (~0.5M)
file_0004.parquet (~0.4M)
file_0006.parquet (~0.3M)
```

Copy Into Databricks

`COPY INTO` in Databricks

COPY INTO feature is used to load files from volumes to Databricks tables and has feature of idempotency ie the data does not get duplicated in the table.

Steps

1. Create Volume

```
CREATE VOLUME dev.bronze.landing
```

1. Create folder inside volume

```
dbutils.fs.mkdirs("/Volumes/dev/bronze/landing/input")
```

1. Copy sample dataset into volume

```
dbutils.fs.cp("/databricks-datasets/definitive-guide/data/retail-data/by-day/2010-12-01.csv","/Volumes/dev/bronze/landing/input")

dbutils.fs.cp("/databricks-datasets/definitive-guide/data/retail-data/by-day/2010-12-02.csv","/Volumes/dev/bronze/landing/input")
```

1. Create bronze table

```
COPY INTO dev.bronze.invoice_cp
FROM '/Volumes/dev/bronze/landing/input'
FILEFORMAT = CSV
PATTERN = '*.csv'
FORMAT_OPTIONS ( -- for the files, if they are different format one has 3 cols other has 5 then merge them
  'mergeSchema' = 'true',
  'header' = 'true'
)
COPY_OPTIONS ( -- at table level meerge Schema
  'mergeSchema' = 'true'
)
```

1. Select from table

We can see 5217 rows.

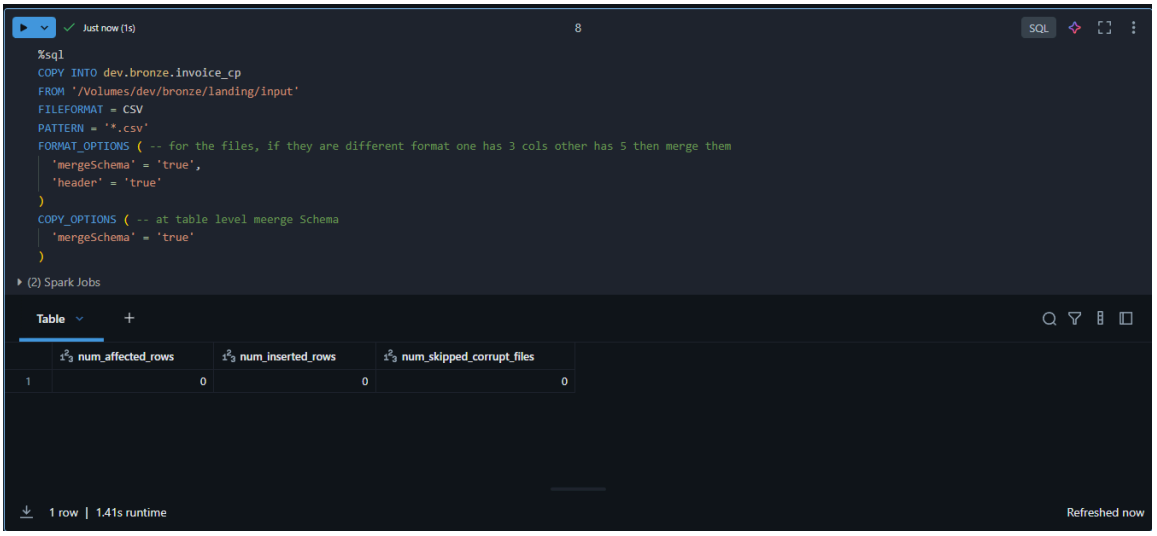
```
SELECT * FROM dev.bronze.invoice_cp
```

(2) Spark Jobs

Table +

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
3	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
4	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
5	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
6	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	2010-12-01 08:26:00	7.65	17850.0	United Kingdom
7	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	2010-12-01 08:26:00	4.25	17850.0	United Kingdom
8	536366	22633	HAND WARMER UNION JACK	6	2010-12-01 08:28:00	1.85	17850.0	United Kingdom
9	536366	22632	HAND WARMER RED POLKA DOT	6	2010-12-01 08:28:00	1.85	17850.0	United Kingdom
10	536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	2010-12-01 08:34:00	1.69	13047.0	United Kingdom
11	536367	22745	POPPY'S PLAYHOUSE BEDROOM	6	2010-12-01 08:34:00	2.1	13047.0	United Kingdom
12	536367	22748	POPPY'S PLAYHOUSE KITCHEN	6	2010-12-01 08:34:00	2.1	13047.0	United Kingdom
13	536367	22749	FELTCRAFT PRINCESS CHARLOTTE DOLL	8	2010-12-01 08:34:00	3.75	13047.0	United Kingdom
14	536367	22310	IVORY KNITTED MUG COSY	6	2010-12-01 08:34:00	1.65	13047.0	United Kingdom
15	536367	84969	BOX OF 6 ASSORTED COLOUR TEASPOONS	6	2010-12-01 08:34:00	4.25	13047.0	United Kingdom

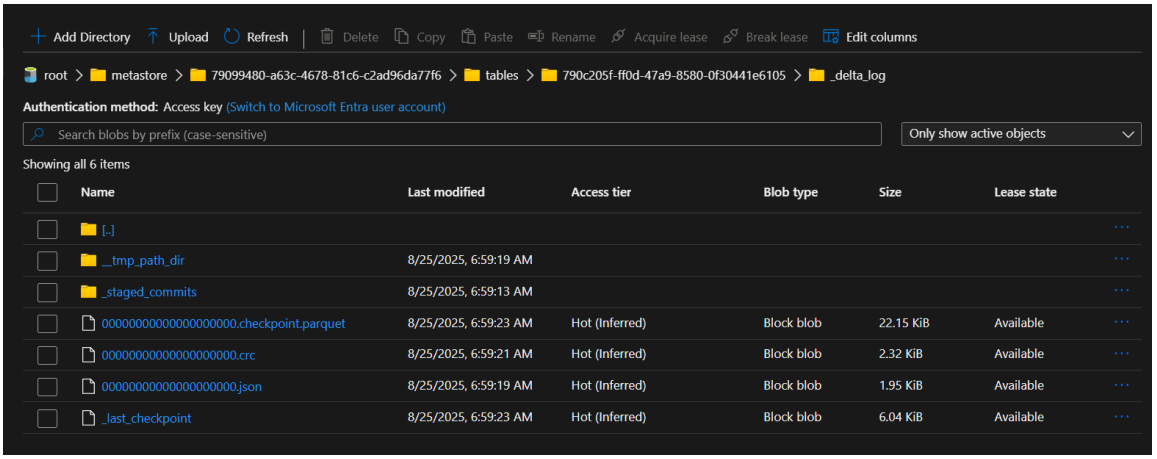
1. Run COPY INTO again



No affected rows so copy into does not duplicate. Its idempotent.

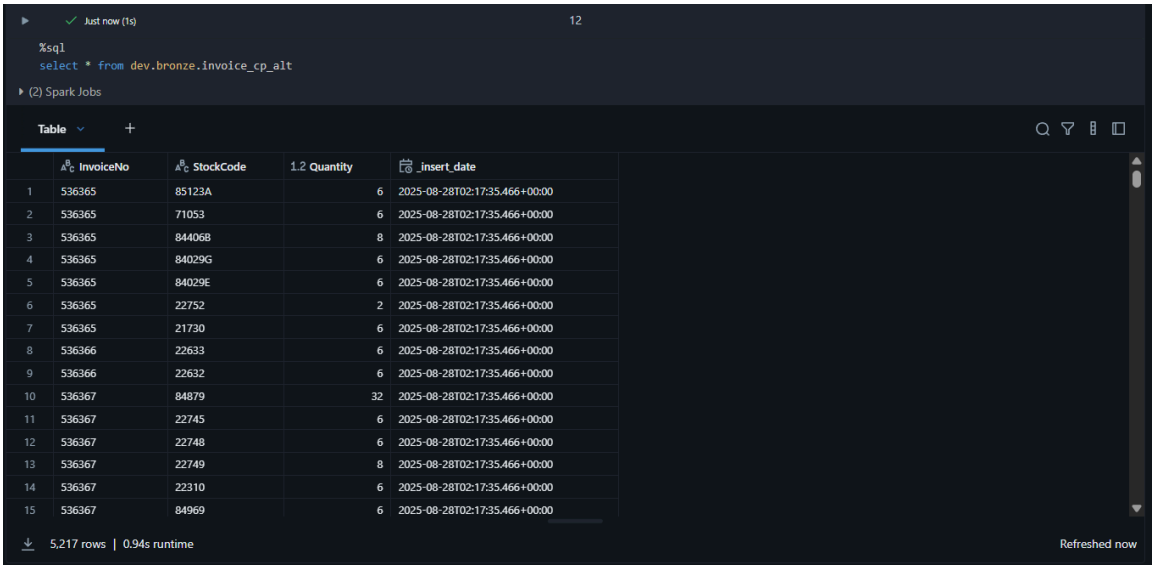
How does copy into maintain the log of data files ingested?

The delta log maintains json version tracking that has information and path of files processed.



Custom Transformations while loading

```
COPY INTO dev.bronze.invoice_cp_alt
FROM
(
  SELECT InvoiceNo,StockCode,cast(Quantity as DOUBLE),current_timestamp() as
_insert_date
  FROM
  '/Volumes/dev/bronze/landing/input'
)

FILEFORMAT = CSV
PATTERN = '*.csv'
FORMAT_OPTIONS ( -- for the files, if they are different format one has 3 cols
other has 5 then merge them
  'mergeSchema' = 'true',
  'header' = 'true'
)
COPY_OPTIONS ( -- at table level meerge Schema
  'mergeSchema' = 'true'
)
```



Autoloader Databricks

Auto Loader Concept in Databricks

Auto Loader is a Databricks feature for incrementally and efficiently ingesting new data files from cloud storage (S3, ADLS, GCS) into Delta Lake tables.

It solves the problem of:

“How do I continuously load only the new files that arrive in my data lake, without reprocessing old files every time?”

⚙ How it works

New files detection

Auto Loader uses file notification or directory listing to detect new files in cloud storage.

Each file is processed exactly once.

Schema handling

Auto Loader can infer schemas automatically and evolve them as new fields appear.

Supports schema evolution modes like:

addNewColumns → automatically adds new columns.

rescue → unexpected fields are captured in _rescued_data column instead of failing.

Incremental state tracking

Auto Loader stores state in a schema location checkpoint directory, so it knows which files are already ingested.

Streaming or batch

Auto Loader works as a Structured Streaming source but can also be triggered in a batch-like mode.

Key Features

Scalable ingestion: Handles billions of files.

Efficient: Processes only new/changed files, no need for full scans.

Schema evolution: Adapts to changing data over time.

Rescue data: Keeps unrecognized/mismatched fields safe for later analysis.

Integration: Works seamlessly with Delta Lake, Structured Streaming, and Databricks Workflows.

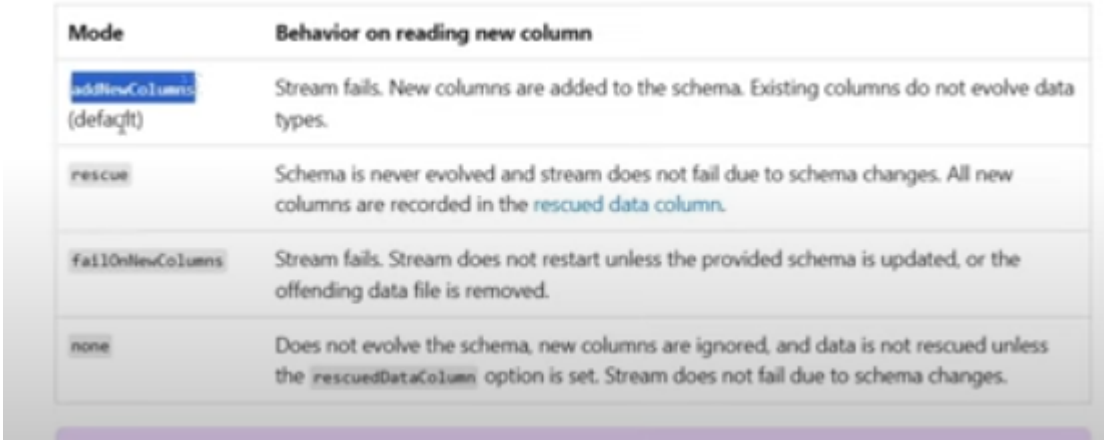
Modes of schema evolution

none → no schema changes allowed.

addNewColumns → automatically add new columns to the table.

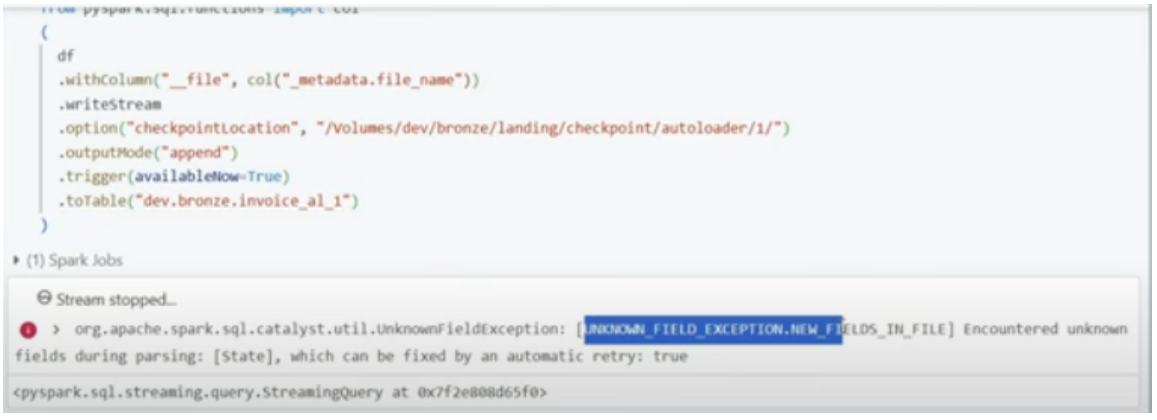
rescue → unexpected fields go into _rescued_data.

Manual → you evolve schema explicitly using ALTER TABLE.



Mode	Behavior on reading new column
addNewColumns (default)	Stream fails. New columns are added to the schema. Existing columns do not evolve data types.
rescue	Schema is never evolved and stream does not fail due to schema changes. All new columns are recorded in the rescued data column.
failOnNewColumns	Stream fails. Stream does not restart unless the provided schema is updated, or the offending data file is removed.
none	Does not evolve the schema, new columns are ignored, and data is not rescued unless the rescuedDataColumn option is set. Stream does not fail due to schema changes.

When we run readStream cell for first time, then run writeStream it fails



```
(
  df
  .withColumn("__file", col("_metadata.file_name"))
  .writeStream
  .option("checkpointLocation", "/Volumes/dev/bronze/landing/checkpoint/autoloader/1/")
  .outputMode("append")
  .trigger(availableNow=True)
  .toTable("dev.bronze.invoice_al_1")
)
```

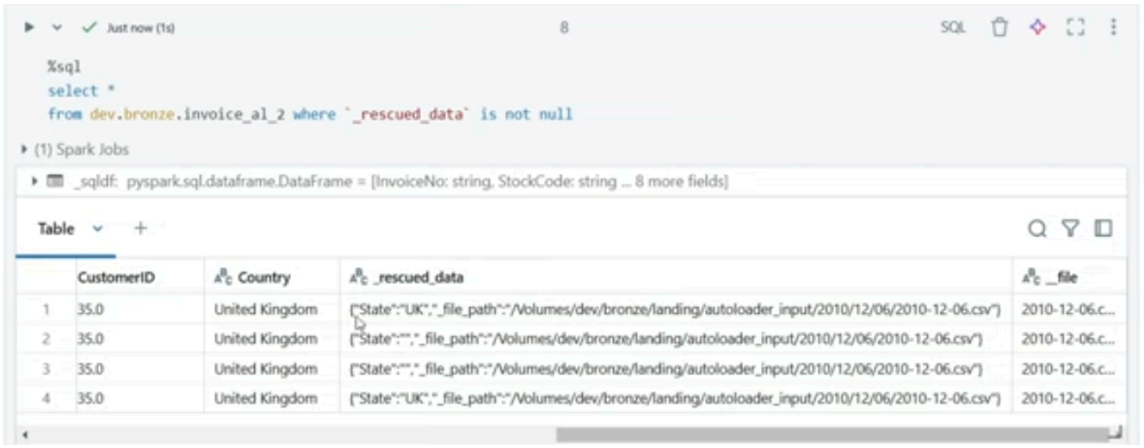
```
▸ (1) Spark Jobs
Stream stopped...
org.apache.spark.sql.catalyst.util.UnknownFieldException: [UNKNOWN_FIELD_EXCEPTION.NEW_FIELDS_IN_FILE] Encountered unknown fields during parsing: [State], which can be fixed by an automatic retry: true
<pyspark.sql.streaming.query.StreamingQuery at 0x7f2e808d65f0>
```

But when it fails, the new column State is added to the schemaLocation folder.

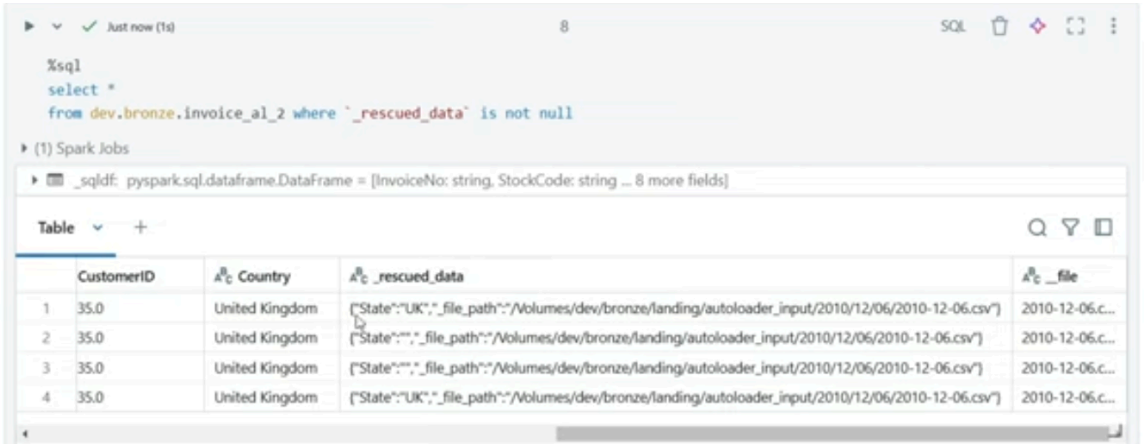
The next time we run readStream it reads using the new schema and now after running writeStream it runs.

```
InvoiceNo: string
StockCode: string
Description: string
Quantity: integer
InvoiceDate: string
UnitPrice: double
CustomerID: string
Country: string
State: string
_rescued_data: string
```

In rescue mode we have a new column added called _rescue_data that holds the records that dont match the schema.



In None mode, we dont see a rescue_column option,



There is no failure but the new column data does not get added. There is no schema evolution.

Why use Auto Loader instead of plain Structured Streaming?

Without Auto Loader: you'd have to rescan directories and manually deduplicate files.

With Auto Loader: file discovery and state management are built-in → scalable & cost-efficient.

---

1. What checkpointing is in Autoloader

When you use **Autoloader** (`cloudFiles`), it's powered by **Structured Streaming**.

- Spark Structured Streaming needs to **remember progress** (which files have been processed, offsets, watermark, etc.).
- That "memory" is kept in the **checkpoint location** (usually in cloud storage like `dbfs:/checkpoints/...`).
- Without checkpoints, a restart would re-read the same files.

---

2. State storage in Structured Streaming

When you do **stateful operations** (e.g., `dropDuplicates` , aggregations, joins with watermarks, etc.), Spark must maintain a **state store**.

- By default, Spark stores this in **HDFS-compatible storage** under your checkpoint directory.
- However, Spark also supports **RocksDB as the backend for state storage**. RocksDB is faster and memory-efficient because it keeps state on **local disk** (per executor) rather than only on JVM heap.

---

3. Using RocksDB with Autoloader

In Databricks, you can enable RocksDB for Autoloader pipelines with stateful ops:

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .load("/mnt/raw/crypto")  # Autoloader source
  .withWatermark("event_time", "10 minutes")
  .groupBy("symbol", window("event_time", "5 minutes"))
  .agg(F.avg("price").alias("avg_price"))
  .writeStream
  .format("delta")
  .option("checkpointLocation", "dbfs:/checkpoints/crypto_autoloader")
  .outputMode("append")
  .option("stateStore.rocksdb.enabled", "true")  # enable RocksDB
  .start("/mnt/bronze/crypto"))
```

4. How it works

- **Checkpoint location**: still needed (to store metadata, offsets, etc.).

- **State store**: if RocksDB enabled → state is stored on **local disk of each executor** (backed by RocksDB), instead of pure JVM memory.
- Each executor writes RocksDB files under its working directory.
- Spark will still write **state snapshots** to the checkpoint location for recovery.

---

5. Why RocksDB helps

- Without RocksDB: state is stored in **hash maps in JVM heap** → can cause OOM for large joins/aggregations.
- With RocksDB: state spills efficiently to disk, uses compression, and avoids large JVM GC pressure.
- Works especially well in **Autoloader pipelines with deduplication or watermark joins**.

---

6. Things to remember

- You must use **Databricks Runtime 10.4+** for RocksDB state store.
- The option is:

```
spark.sql.streaming.stateStore.providerClass=org.apache.spark.sql.execution.st
reaming.state.RocksDBStateStoreProvider
```

(Databricks simplifies this with `stateStore.rocksdb.enabled`). * Still need cloud **checkpointLocation** → otherwise recovery after restart won't work.

So in summary:

- Autoloader always needs a **checkpoint location**.
- If you're doing stateful ops, you can choose between **default HDFS state store** vs **RocksDB**.
- With RocksDB, state is on executor local disk, but still backed up into the checkpoint folder for recovery.

For file notification mode use `option(cloudFiles.useNotifications,True)`

Intro Databricks Lakeflow Declarative Pipelines

Lakeflow Declarative Pipelines Introduction

Here's a clear introduction to **Lakeflow declarative pipelines** in Databricks:

---

1. What is Lakeflow?

Lakeflow is a **declarative pipeline framework** by Databricks designed to simplify building and managing data pipelines over Delta Lake. Unlike traditional ETL pipelines where you imperatively write every transformation step, Lakeflow allows you to **declare the desired end state of your data** and lets Databricks handle the execution, orchestration, and dependency management.

Think of it like **"telling Databricks what you want, not how to do it"**.

---

2. Key Concepts

Concept	Description
Pipeline	A logical collection of transformations (like ETL jobs) defined declaratively.
Lakeflow Table	A Delta table managed by Lakeflow, which tracks lineage, schema, and dependencies.
Declarative Config	JSON/YAML-like specification describing sources, transformations, and targets.
State Management	Lakeflow keeps track of which data has been processed and ensures **idempotent updates**.
Incremental Processing	Automatically detects new/changed data and applies transformations incrementally.

---

3. How It Differs from Normal Pipelines

Feature	Traditional ETL	Lakeflow Declarative Pipeline
Definition	Imperative: step-by-step code	Declarative: specify inputs, outputs, transformations
Orchestration	Manual or Airflow/Scheduler	Built-in, dependency-aware orchestration
Data Lineage	Requires extra tooling	Automatic tracking of lineage between tables
Error Handling	Manual retries	Automatic state management & retries
Incremental Loads	Developer writes logic	Lakeflow detects changes & processes incrementally

---

4. Basic Pipeline Flow

1. **Define sources**: Raw Delta tables, cloud storage, or external systems.
2. **Declare transformations**: For example, aggregations, joins, or enrichments.
3. **Specify targets**: Delta tables managed by Lakeflow.
4. **Run pipeline**: Databricks ensures only necessary transformations run, handles dependencies, and maintains consistency.

---

5. Advantages

- **Less boilerplate code** → focus on business logic, not orchestration.
- **Automatic incremental updates** → avoids reprocessing all data.
- **Built-in lineage and auditing** → helps with compliance and debugging.
- **Easier pipeline management** → declarative config files version-controlled like code.

---

6. Example (Conceptual)

```yaml
# Sample declarative pipeline config
pipeline_name: sales_pipeline
tables:
  - name: raw_sales
    source: s3://data/raw_sales
  - name: sales_summary
    depends_on: raw_sales
    transformations:
      - type: aggregate
        group_by: ["region", "category"]
        metrics:
          - name: total_sales
            operation: sum(amount)
```

Here, you **declare** the desired summary table, and Lakeflow automatically handles reading `raw_sales`, performing aggregation, and writing `sales_summary` incrementally.

Here's a detailed breakdown of **Streaming Tables, Materialized Views, and Normal Views in Databricks DLT (Delta Live Tables)** and their typical usage:

---

1. Streaming Tables (aka Live Tables)

Definition: A **streaming table** in DLT is a Delta table that is continuously updated as new data arrives. It's typically used for **real-time pipelines**.

Key Characteristics:

Feature	Description
Data Ingestion	Continuous from streaming sources (Kafka, Event Hubs, cloud storage).
Storage	Delta table on Databricks (supports ACID).
Processing Mode	`Append` or `Upsert` (via `MERGE`).
Latency	Near real-time; updates as soon as micro-batches are processed.
Schema Evolution	Supported, can evolve automatically or manually.

Use Case:

- Real-time dashboards (e.g., sales, stock prices, payments).
- Event-driven pipelines (e.g., fraud detection, monitoring trades).

Example in DLT (Python):

```
import dlt
from pyspark.sql.functions import *

@dlt.table
def streaming_sales():
    return (
        spark.readStream.format("cloudFiles")
             .option("cloudFiles.format", "json")
             .load("/mnt/sales/json")
             .withColumn("processed_time", current_timestamp())
    )
```

---

2. Materialized Views

Definition: A **materialized view** is a Delta table that stores the **precomputed results** of a query. In DLT, it is a table whose content is automatically refreshed based on its dependencies.

Key Characteristics:

Feature	Description
Data Storage	Delta table with physical storage (unlike normal views).
Refresh	Incremental refresh based on upstream table changes.
Performance	Faster query since data is precomputed.
Up-to-date	Always consistent with underlying tables (incrementally).

Use Case:

- Aggregations (e.g., daily sales per region).
- Joins and transformations that are expensive to compute on demand.
- Serving layer for dashboards or ML pipelines.

Example in DLT (Python):

```
import dlt
from pyspark.sql.functions import sum

@dlt.table
```

```python
def sales_summary():
    sales = dlt.read("LIVE.streaming_sales")
    return sales.groupBy("region").agg(sum("amount").alias("total_sales"))
```

> Here `sales_summary` is materialized: it stores the aggregated totals physically and refreshes as new streaming data arrives.

---

3. Normal Views

Definition: A **normal view** in DLT is like a **virtual table**: it does **not store data physically**. Instead, it executes the underlying query each time you read it.

Key Characteristics:

Feature	Description
Data Storage	None; only query definition stored.
Performance	Slower for large or complex queries (computed on-demand).
Up-to-date	Always reflects the latest upstream data.
Incremental Processing	Not directly supported (since it's virtual).

Use Case:

- Lightweight transformations that do not need storage.
- Ad-hoc analytics.
- Debugging or temporary transformations in pipelines.

Example in DLT (Python):

```python
import dlt

@dlt.view
def high_value_sales():
    sales = dlt.read("LIVE.streaming_sales")
    return sales.filter(sales.amount > 10000)
```

> `high_value_sales` is a **normal view**: it does not store any data and computes results on demand.

---

4. Quick Comparison

Feature	Streaming Table	Materialized View	Normal View
Storage	Delta (persistent)	Delta (precomputed)	None
Latency	Near real-time	Incrementally refreshed	On-demand
Performance	High for reads	High (precomputed)	Low for complex queries
Use Case	Real-time ingestion	Aggregations, dashboards	Lightweight transformations, ad-hoc analysis

---

 Summary / Guidance:

- **Use streaming tables** for continuous ingestion from sources.
- **Use materialized views** when you want precomputed results for fast reads and dashboards.
- **Use normal views** for on-the-fly filtering or temporary logical tables.

Dlt Batch Vs Streaming Workloads

Delta Live Tables : Batch vs Streaming Workloads

Here's a detailed breakdown of **batch vs streaming workloads for Delta Live Tables (DLT) streaming table sources** and when to use each:

---

1. Conceptual Difference

Aspect	Batch Workload	Streaming Workload
Data Processing	Processes a finite dataset at once.	Continuously processes new data as it arrives.
Latency	High (runs periodically).	Low (near real-time).
Trigger	Manual or scheduled (e.g., daily, hourly).	Continuous / micro-batches.
State Management	Not required; table is fully recomputed or append-only.	Required; DLT tracks which data has already been processed.
Use Cases	Historical analytics, daily aggregates, backfills.	Real-time dashboards, alerts, event processing, fraud detection.

---

2. How DLT Handles Sources

DLT streaming table sources can come from:

1. **Streaming sources**
2. Kafka, Event Hubs, cloud storage (`cloudFiles`), Delta tables.
3. Automatically **incremental**, supports schema evolution.
4. Example (Python):

```python
@dlt.table
def streaming_sales():
    return (
        spark.readStream.format("cloudFiles")
             .option("cloudFiles.format", "json")
             .load("/mnt/sales/json")
    )
```

1. **Batch sources**
2. Existing Delta tables, Parquet, CSV, or JDBC.
3. DLT can treat these as **batch reads** if no streaming input is detected.
4. Example (Python):

```python
@dlt.table
def daily_sales():
    return spark.read.format("delta").load("/mnt/sales/delta")
```

> DLT automatically detects whether the source is streaming or batch, but you can explicitly control behavior with `spark.readStream` vs `spark.read`.

---

3. Streaming Table Behavior in DLT

Feature	Behavior for Batch Sources	Behavior for Streaming Sources
Incremental Processing	Only new runs; full dataset read each time unless optimized.	DLT tracks offsets/checkpoints to process only new data.
State Management	Not needed.	DLT maintains state to support upserts, aggregations, joins.
Dependencies	Upstream changes processed on next pipeline run.	Changes propagate continuously downstream (materialized views get updated incrementally).
Latency	Minutes to hours (depends on schedule).	Seconds to minutes (micro-batch).
Use Case in DLT	Backfill historical data, batch pipelines.	Real-time dashboards, streaming aggregations, event-driven analytics.

4. Choosing Between Batch and Streaming

Scenario	Preferred Workload
Daily sales report from last month	Batch
Real-time fraud detection on transactions	Streaming
Hourly ETL from a static CSV dump	Batch
Continuous clickstream analytics for dashboards	Streaming
Incremental aggregation from an append-only Delta table	Can be either, depending on freshness requirements

---

5. Practical Notes for DLT

1. **Streaming tables can consume batch data**
2. You can define a streaming table that reads from a Delta table (`spark.readStream.format("delta")`), so it behaves like streaming even if the upstream table is batch.
3. **Materialized views downstream**
4. Always incremental; DLT ensures updates propagate automatically.
5. **Checkpointing**
6. DLT automatically handles checkpoints for streaming sources. You don't need to manage offsets manually.

 Summary:

- **Batch workloads** → finite, scheduled processing.
- **Streaming workloads** → continuous, low-latency processing with state management.
- **DLT streaming tables** adapt to both but are most powerful when the source is streaming.

Do you want me to make that diagram?

Dlt Data Storage Checkpoints

Data Storage Internals, Checkpoints and Renaming Streaming Tables and Views

Here's a detailed explanation of **how Delta Live Tables (DLT) handles storage for streaming tables, pipeline dependency, and renaming behavior**:

---

1. Where is data for streaming tables stored?

- **Streaming tables in DLT are stored as Delta tables on the Databricks File System (DBFS) or your cloud storage configured for the pipeline**, e.g., **S3, ADLS Gen2, or GCS**.
- Every streaming table has a **physical Delta table location**, even though you define it declaratively in DLT.
- The storage location is typically **managed by the DLT pipeline**, but you can explicitly configure it in advanced pipeline settings.

Key points:

- **Incremental state** (processed offsets, checkpoints) is stored in **_system-managed checkpoints** within the pipeline's storage path.
- **Upserts and merges** are persisted in the Delta table itself.
- **Data retention** and compaction follow normal Delta table rules.

---

2. Is storage dependent on the pipeline?

Yes, partially:

- **Pipeline-specific storage**: Each DLT pipeline manages its own metadata and checkpoints for the streaming tables it owns.
- **Shared tables**: If multiple pipelines reference the same Delta table (e.g., using `LIVE.<table_name>`), the physical Delta table is **shared**, but each pipeline maintains its own lineage and state metadata.

Implication:

- Deleting a pipeline **does not delete the underlying Delta table** automatically, unless you explicitly choose managed tables.
- Changing pipelines (like moving a table to a different pipeline) requires careful handling to avoid breaking downstream dependencies.

---

3. What happens when we rename streaming tables?

- **DLT does not support a "rename" operation in place** for streaming tables.
- If you rename a table in DLT:
- The new table name points to a **new managed object** in the pipeline.
- The **underlying Delta data is copied or remapped** depending on configuration.
- Any downstream references (`LIVE.<old_name>`) break unless you **update them to the new name**.
- **Best practice:**
- Avoid renaming streaming tables in active pipelines.
- If renaming is needed, **create a new table with the desired name** and **point downstream materialized views or pipelines** to it.

---

4. Practical Notes / Recommendations

Aspect	Recommendation
Storage	Let DLT manage default Delta table locations unless you need a custom path.
Pipeline dependency	Be aware that streaming tables are tied to pipeline metadata (checkpoints, lineage).
Renaming	Prefer creating a new table and updating downstream references; avoid in-place renames for live pipelines.
Backup	If renaming or moving, snapshot or backup Delta tables to avoid data loss.

 Summary:

1. Streaming tables **always persist data as Delta tables** in the pipeline's storage.
2. Storage and checkpoints are **pipeline-dependent**, but the data itself can be shared.
3. **Renaming a streaming table breaks dependencies**; best approach is to create a new table instead of renaming.

Databricks Secret Scopes Using Azure Key Vault

---

1. Prerequisites

- An **Azure Databricks workspace** (Premium or above recommended).
- An **Azure Key Vault** created.
- Appropriate RBAC permissions (Owner/Contributor + Key Vault Administrator).
- Networking planned: VNet injection or secure workspace.

---

2. Create an Azure Key Vault

1. Go to **Azure Portal → Create a resource → Key Vault**.
2. Set:
3. **Resource Group**: choose existing/new.
4. **Region**: same as your Databricks workspace (recommended for latency & compliance).
5. **Pricing tier**: Standard.
6. In **Access configuration**:
7. Choose **RBAC** (recommended) OR **Vault access policy**.
8. For Databricks, RBAC is simpler to manage at scale.
9. Finish creation.

---

3. Configure Networking on Key Vault

- In **Key Vault → Networking**:
- Set **Public access**:
  - Choose *Disabled* if you want **private access only**.
  - Or choose *Selected networks* and allow only your Databricks subnets.

- If you're using **Private Endpoints**:
- Click *+ Private Endpoint*.
- Link it to your Databricks VNet subnet.
- Approve the private endpoint connection in Key Vault.

---

4. Store Secrets in Key Vault

1. In Key Vault → **Objects → Secrets** → *+ Generate/Import*.
2. Add secrets, e.g.:
3. `db-password`
4. `api-key`

---

5. Create a Databricks Secret Scope Backed by Key Vault

In Databricks workspace:

1. Open **Azure Databricks → Manage Account → User Settings → Access Tokens**.
2. Generate a **personal access token** if you'll use the CLI.
3. Run Databricks CLI (or REST API) to create the scope:

```
databricks secrets create-scope \
  --scope my-keyvault-scope \
  --scope-backend-type AZURE_KEYVAULT \
  --resource-id "/subscriptions/<sub-id>/resourceGroups/<rg-
name>/providers/Microsoft.KeyVault/vaults/<kv-name>" \
  --dns-name "https://<kv-name>.vault.azure.net/"
```

 Now, secrets in Key Vault are accessible from Databricks under the scope `my-keyvault-scope`.

---

6. Access Secrets in Databricks Notebook

```
# Example: get db-password stored in Key Vault
db_password = dbutils.secrets.get(scope="my-keyvault-scope", key="db-
```

```
password")

print("Fetched secret length:", len(db_password))  # don't print actual
secret!
```

---

7. Networking Configurations (Important)

You have two secure options for Key Vault ↔ Databricks:

Option A: VNet Injection (most common for secure workspaces)

- Deploy Databricks workspace in **your own VNet**.
- Add **service endpoints** for Key Vault:
- Go to **Databricks VNet → Subnets → Service endpoints → Add KeyVault**.
- In Key Vault **Networking**, allow that subnet.

Option B: Private Endpoint

- Create a **private endpoint** for Key Vault in the Databricks VNet.
- Approve the connection in Key Vault.
- Disable public network access for maximum security.

---

8. Permissions

- In **Key Vault → Access Control (IAM)**:
- Assign your Databricks workspace managed identity:
  - **Key Vault Secrets User** role.
  - Or **Key Vault Reader** role (depending on setup).

Summary

1. Create Key Vault → Add secrets.
2. Configure networking (service endpoint or private endpoint).

3. Assign RBAC so Databricks can access Key Vault.
4. Create secret scope in Databricks (backed by Key Vault).
5. Access secrets inside notebooks with `dbutils.secrets.get`.

Would you like me to also draw you a **network diagram** (Databricks → VNet → Key Vault with private endpoint) so you can visualize the traffic flow?

Databricks Control Plane and Data Plane

What is Databricks Lakehouse Architecture?

Traditionally, companies had **two separate systems**:

- **Data Lake** (cheap storage, e.g., Azure Data Lake, S3, Blob): stores raw structured, semi-structured, unstructured data → flexible but lacks strong data management (ACID, governance, BI).
- **Data Warehouse** (expensive but fast): optimized for SQL queries, BI, and analytics → great schema enforcement and governance but limited flexibility and costly.

The **Lakehouse** combines both in one system:

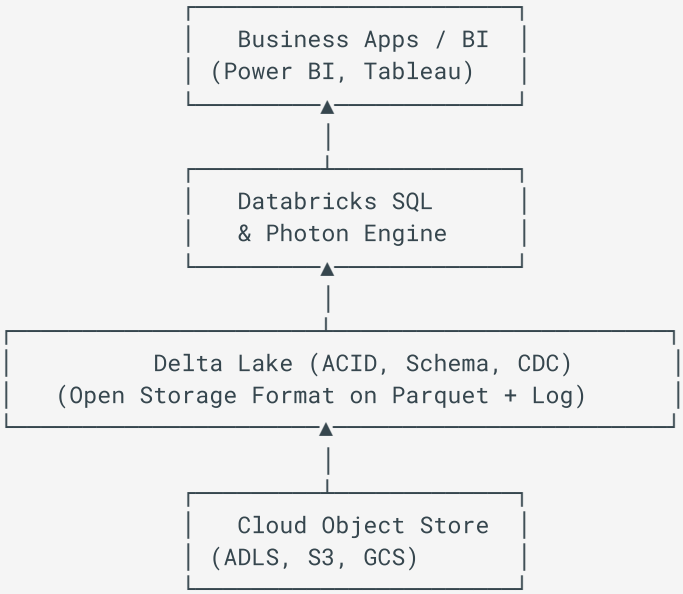
- The **low-cost, flexible storage** of a **data lake**
- The **governance, ACID transactions, performance** of a **warehouse**

Core Components of Databricks Lakehouse

1. **Storage Layer (Data Lake foundation)**
2. Data stored in **open formats** like Parquet, ORC, Avro, Delta.
3. Uses **cloud object storage** (e.g., Azure Data Lake Storage Gen2, AWS S3, GCS).
4. **Delta Lake (the secret sauce)**
5. Adds **ACID transactions** on top of data lake storage.
6. Provides **schema enforcement, schema evolution, time travel, data versioning**.
7. Solves problems like “eventual consistency” and corrupted files in raw data lakes.
8. **Unified Governance (Unity Catalog)**
9. Centralized **metadata & permissions** for files, tables, ML models, dashboards.
10. Manages security, lineage, and data discovery across the Lakehouse.
11. **Compute Layer (Databricks Runtime / Spark + Photon)**
12. Uses Apache Spark + Photon execution engine for **batch, streaming, ML, BI**.
13. Same engine for **ETL, streaming, AI, SQL queries** → no silos.

14. **Data Management Features**
15. **Streaming + Batch = One Pipeline** (via Delta Live Tables).
16. **Materialized Views, Incremental Processing, Change Data Capture (CDC)**.
17. MLflow integration for **machine learning lifecycle management**.

Architecture Diagram (Conceptual Flow)



⚡ Benefits of Lakehouse

- **One platform** → no need for separate warehouse + lake.
- **Cost efficient** → cheap storage, scalable compute.
- **Flexibility** → structured + semi-structured + unstructured.
- **ACID reliability** → transactions, schema enforcement.
- **End-to-end** → supports ETL, real-time streaming, ML/AI, BI in the same system.

In Databricks Azure Context

- Storage → **Azure Data Lake Storage (ADLS Gen2)**
- Security/Governance → **Azure Key Vault + Unity Catalog**

- Compute → **Databricks Clusters with Photon**
- Serving → **Power BI (Direct Lake Mode)**

Data Plane vs Control Plane

---

1. Simple Analogy

Think of Databricks like **Uber**:

- **Control Plane = Uber App** → handles **where you go, who drives, billing, monitoring**.
- **Data Plane = The Car** → where the **actual ride happens** (your data processing).

So, Databricks separates **management functions (control)** from **execution functions (data)**.

---

2. Databricks Architecture

Control Plane

- Managed by **Databricks itself** (runs in Databricks' own AWS/Azure/GCP accounts).
- Contains:
- **Web UI / REST API** → you log in here, create clusters, manage jobs.
- **Cluster Manager** → decides how to spin up VMs/compute.
- **Job Scheduler** → triggers pipelines, notebooks, workflows.
- **Metadata Storage** → notebooks, workspace configs, Unity Catalog metadata.
- **Monitoring / Logging** → cluster health, job logs, error reporting.

⚠ **Important**: Your **raw data does not go here**. This plane is about orchestration, configs, and metadata.

Data Plane

- Runs inside **your cloud account** (your subscription/project).
- Contains:

- **Clusters/Compute** (Spark Executors, Driver, Photon) → where the data is processed.
- **Your Data** → stored in ADLS, S3, or GCS.
- **Networking** → VNETs, Private Endpoints, Peering.
- **Libraries / Runtime** → Spark, Delta Lake, MLflow, etc.

⚠ **Key Point**: The **actual data never leaves your cloud account**. Processing happens within your boundary.

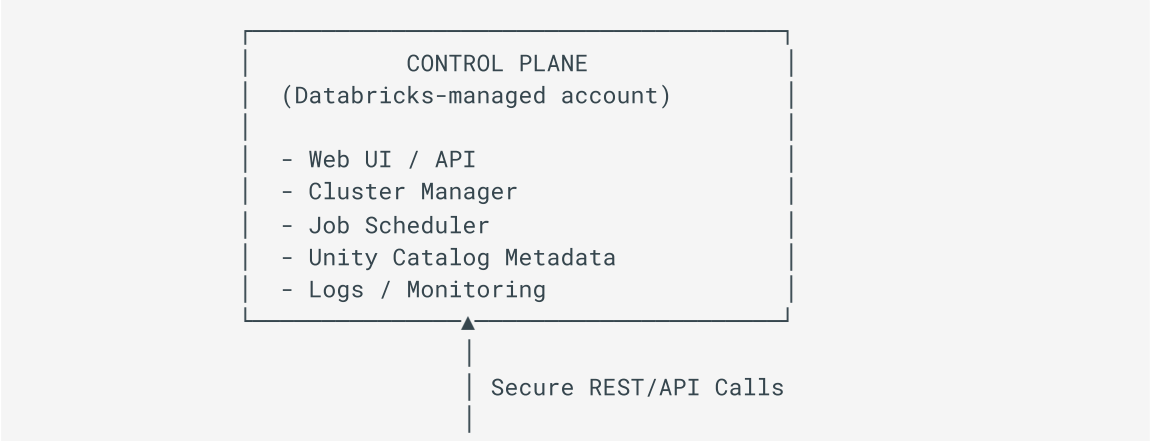
---

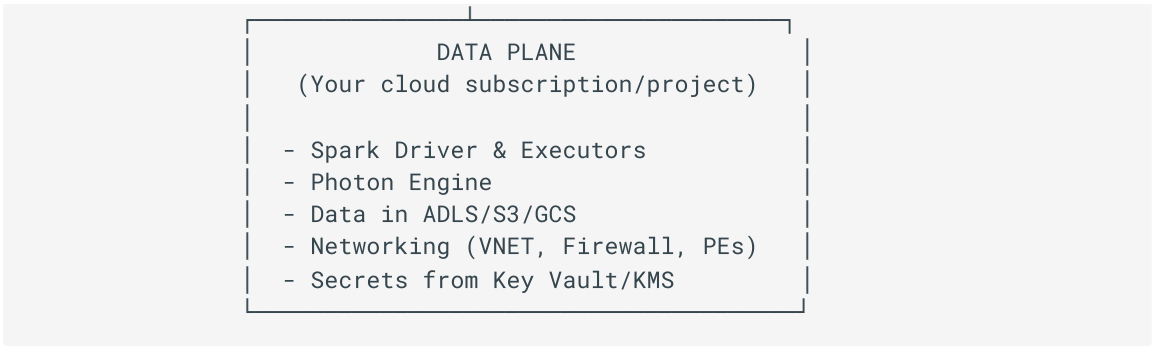
3. Security Perspective

- **Control Plane**:
- Managed by Databricks.
- Contains **metadata, credentials, configs**, but not raw data.
- Can be hardened with **SCIM, SSO, RBAC, IP Access Lists**.
- **Data Plane**:
- Fully inside **your cloud subscription**.
- Your **sensitive data** (PII, transactions, crypto, etc.) never touches Databricks' account.
- You control networking:
  - Private Link / VNET Injection → ensures traffic never goes over the public internet.
  - Key Vault / KMS for secrets.
  - Storage firewalls.

---

4. Architecture Diagram





⚡ 5. Why This Separation?

 Security → Your data never leaves your account. **Scalability** → Databricks manages orchestration, you manage compute. **Multi-cloud** → Same control plane works across AWS, Azure, GCP. **Compliance** → Helps with HIPAA, GDPR, financial regulations.

---

6. Special Feature: Databricks Serverless SQL

- Here, the **data plane compute is managed by Databricks** too (not your account).
- Good for quick BI queries (like Power BI), but some enterprises avoid it for sensitive data.

Databricks Dlt Code Walkthrough

Delta Live Tables Code Walkthrough

1. Create Streaming Table for Orders

```
@dlt.table(
  table_properties = {"quality":"bronze"},
  comment = "Orders Bronze Table"
)
def orders_bronze():
  df = spark.readStream.table("dev.bronze.orders_raw")
  return df
```

1. Create Materialized View for Customers

```
@dlt.table(
  table_properties = {"quality":"bronze"},
  comment = "Customers Materialized View"
)
def customers_bronze():
  df = spark.read.table("dev.bronze.customers_raw")
  return df
```

1. Create a view that joins above streaming table and materialized view

```
@dlt.view(
  comment = 'Joined View'
)
def joined_vw():
  df_c = spark.read.table("LIVE.customers_bronze")
  df_o = spark.read.table("LIVE.orders_bronze")

  df_join = df_o.join(df_c,how = "left_outer",on =
df_c.c_custkey==df_o.o_custkey)

  return df_join
```

1. Add a new column to the view

```
@dlt.table(
  table_properties = {"quality":"silver"},
  comment = "joined table",
  name = 'joined_silver'
)
def joined_silver():
  df =
```

```
spark.read.table("LIVE.joined_vw").withColumn("_insertdate",current_timestamp(
))
  return df
```

1. Create gold level aggregation

```
@dlt.table(
  table_properties = {"quality":"gold"},
  comment = "orders aggregated table",
)
def joined_silver():
  df = spark.read.table("LIVE.joined_silver")

  df_final =
df.groupBy('c_mktsegment').agg(count('o_orderkey').alias('sum_orders').withCol
umn('_insertdate',current_timestamp()))
  return df_final
```

Deleting DLT Pipeline

The tables / datasets in DLT are managed and linked to DLT pipelines. So if we delete a pipleine all fo them get dropped.

Incremental Load in DLT

When we inserted 10k records into orders_bronze, only those got ingested not the entire table.

Adding New Column

```
@dlt.table(
  table_properties = {"quality":"gold"},
```

```
  comment = "orders aggregated table",
)
def joined_silver():
  df = spark.read.table("LIVE.joined_silver")

  df_final =
df.groupBy('c_mktsegment').agg(count('o_orderkey').alias('sum_orders').agg(sum
('o_totalprice').alias('sum_price').withColumn('_insertdate',current_timestamp
()))
  return df_final
```

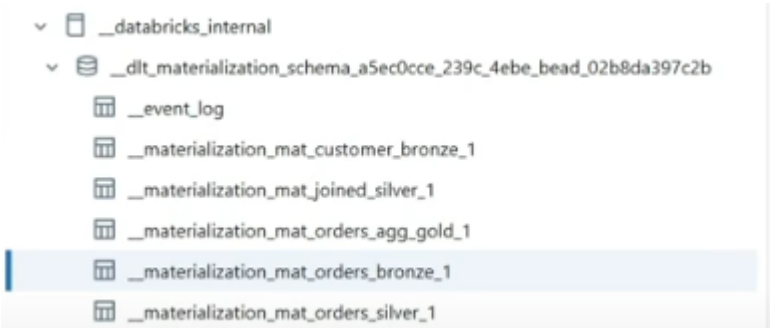
We dont have to manipulate ddl, the dlt pipeline will auto detect addition of new column.

Renaming Tables

We just change the name of the function in the table declaration and the table name will be renamed. The catalog will also reflect this.

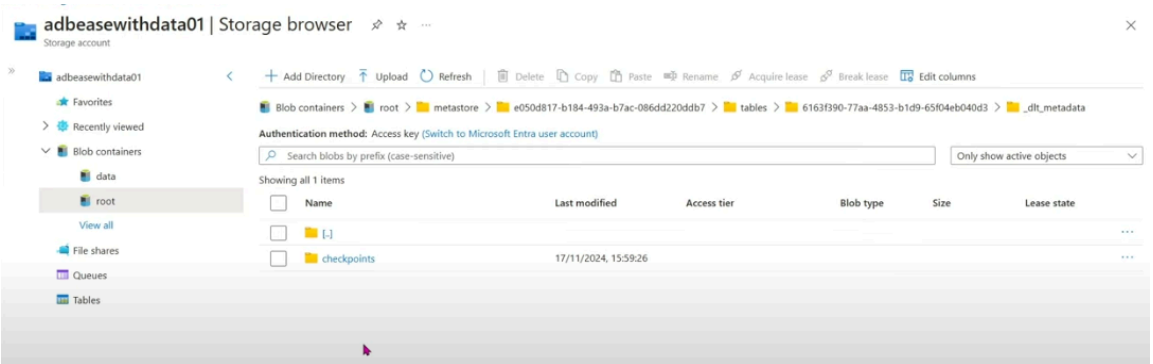
DLT Internals

Every streaming table, MV is supported by underlying tables in `_databricks_internal` schema.

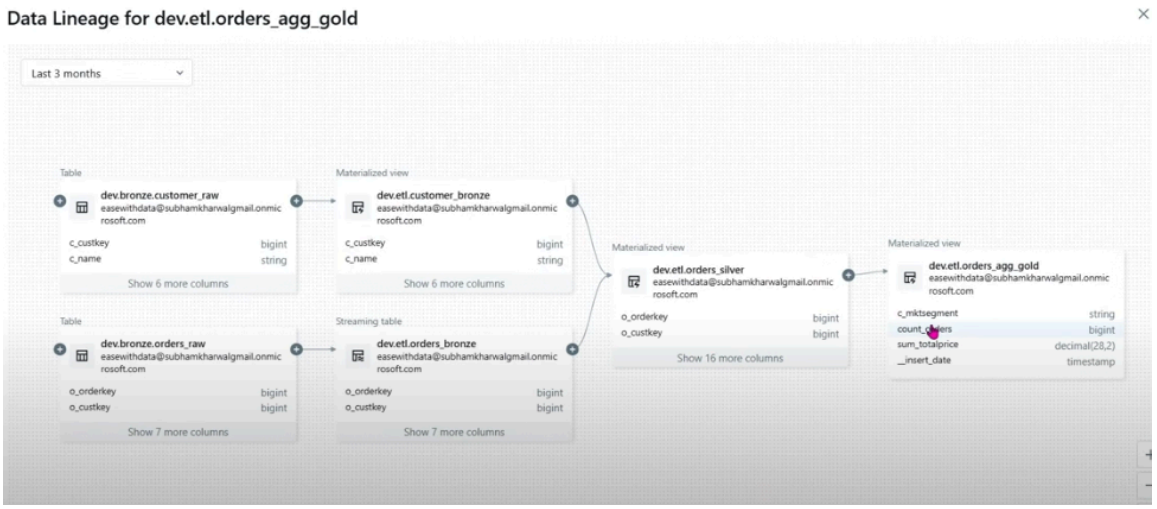


and they have a table_id associated with it.

If we go to these tables in storage account, we can see checkpoints that keep track of incremental data changes.



Data Lineage



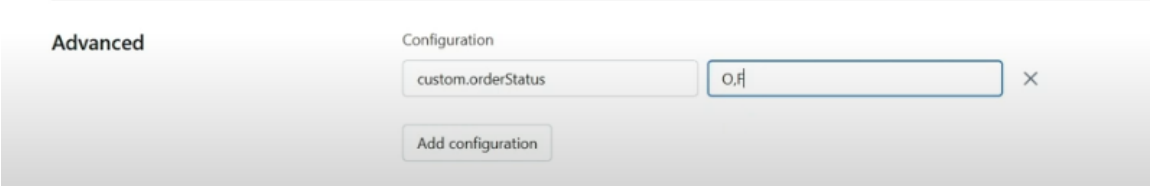
DLT Append Flow and Autoloader

```
@dlt.table(
  table_properties = {"quality":"bronze"},
  comment = "orders autoloader",
  name = "orders_autoloader_bronze"
)
def func():
  df = (
      spark.readStream
      .format("cloudFiles")
      .option("cloudFilesFormat","CSV")
      .option("cloudFiles.schemaLocation","...")
      .option("pathGlobFilter","*.csv")
      .option("cloudFiles.schemaEvolutionMode","none")
      .load("/Volumes/etl/landing/files"
)
return df
```

```
dlt.createStreamingTable("order_union_bronze")

@dlt.append_flow(
  target = "orders_union_bronze"
)
def order_delta_append():
  df = spark.readStream.table("LIVE.orders_bronze")
  return df

@dlt.append_flow(
  target = "orders_union_bronze"
)
def order_autoloader_append():
  df = spark.readStream.table("LIVE.orders_autoloader_bronze")
  return df
```

```
@dlt.view(
  comment = 'Joined View'
)
def joined_vw():
  df_c = spark.read.table("LIVE.customers_bronze")
  df_o = spark.read.table("LIVE.orders_union_bronze")

  df_join = df_o.join(df_c,how = "left_outer",on =
df_c.c_custkey==df_o.o_custkey)

  return df_join
```

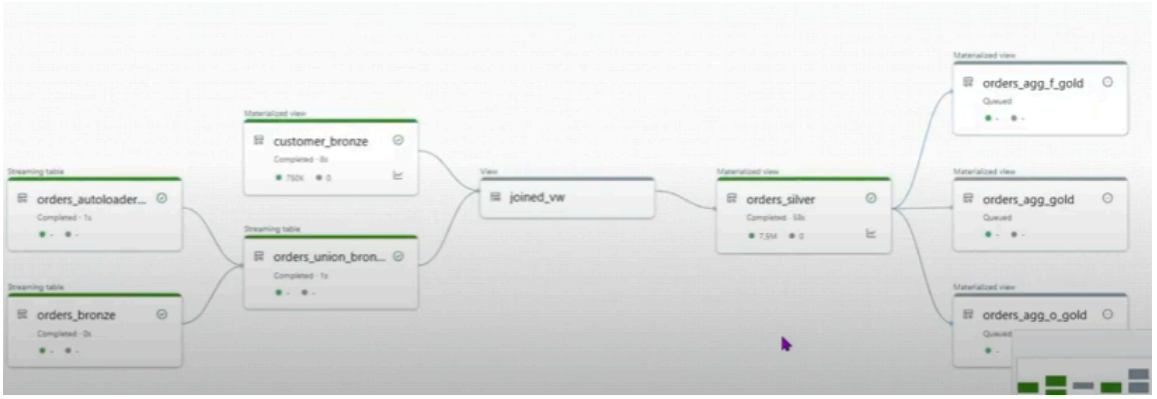
Custom Configuration



Use this param in code

```
_order_status = spark.conf.get("custom.orderStatus","_NA")
```

```
for _status in _order_status.split(","):
    # create gold table
    @dlt.table(
        table_properties = {"quality":"gold"},
        comment = "order aggregated table",
        name = f"orders_agg_{_status}_gold"
    )
    def orders_aggregated_gold():
        df = spark.read.table("LIVE.joined_silver")
        df_final = df.where(f"o_orderstatus =
'{_status}'").groupBy("c_mktsegment").agg(count('o_orderkey').alias("count_of_
orders"),sum("o_totalprice").alias('sum_totalprice')).withColumn("_insert_date
", current_timestamp())

        return df_final
```



DLT SCD1 and SCD2

Pre Requisites

```
-- Add columns in Source customer raw table
ALTER TABLE dev.bronze.customer_raw
ADD COLUMNS (
  __src_action STRING, -- Possible Values I, D, T
  __src_insert_dt TIMESTAMP
)
;
```

```
-- Update the new columns __src_insert_dt and __src_action
UPDATE dev.bronze.customer_raw
SET __src_action = 'I',
    __src_insert_dt = current_timestamp() - interval '3 days'
;
```

Input Source Table

```
@dlt.view(
  comment = "Customer Bronze streaming view"
)
def customer_bronze():
  df = spark.readStream.table("dev.bronze.customers_raw")
  return df
```

SCD Type1 Table

```
dlt.create_streaming_table('customer_sdc1_bronze')

dlt.apply_changes(
  target = "customer_scd1_bronze",
  source = "customer_bronze_vw",
  keys = ['c_custkey'],
  stored_as_scd_type = 1,
  apply_as_deletes = expr("__src_action = 'D'"),
  apply_as_truncates = expr("__src_action = 'T'"),
```

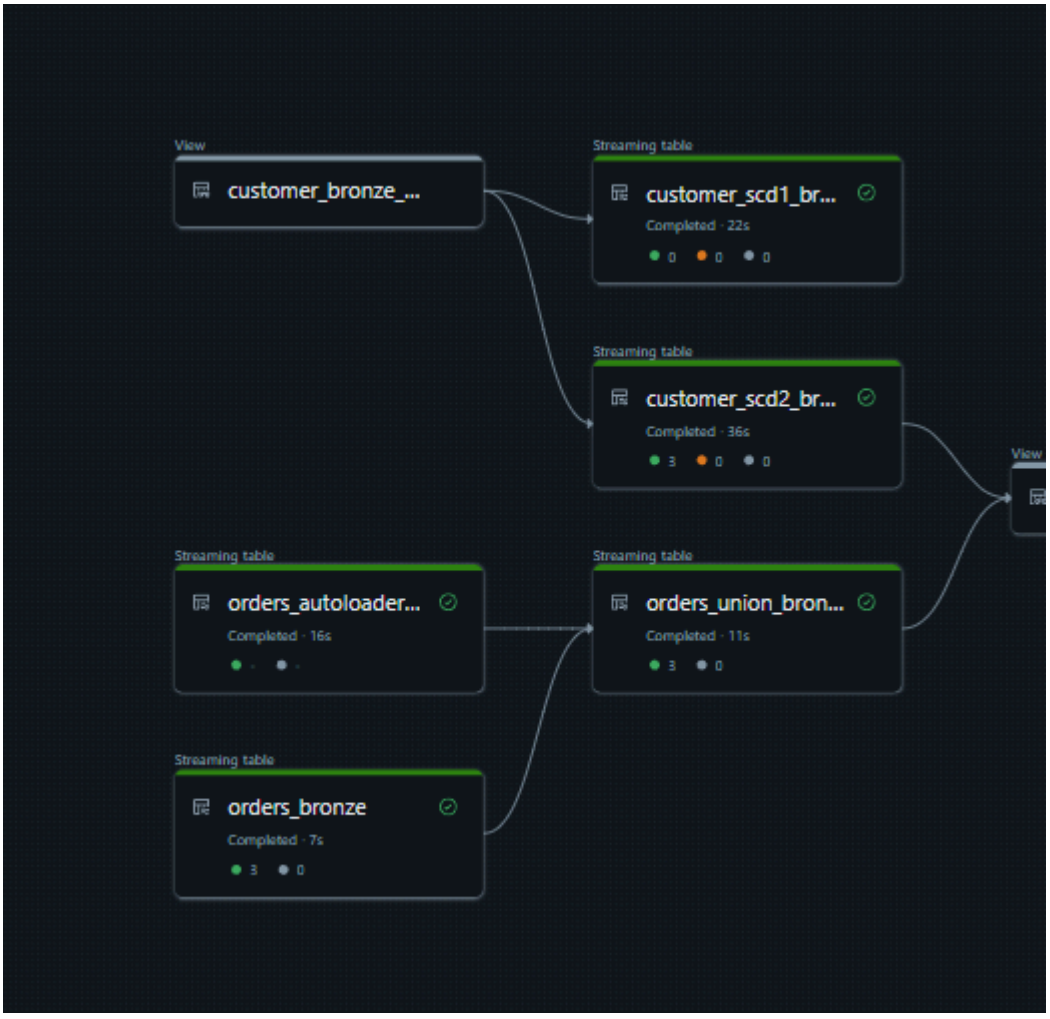
```
  sequence_by = "__src_insert_dt"
)
```

SCD Type 2 Table

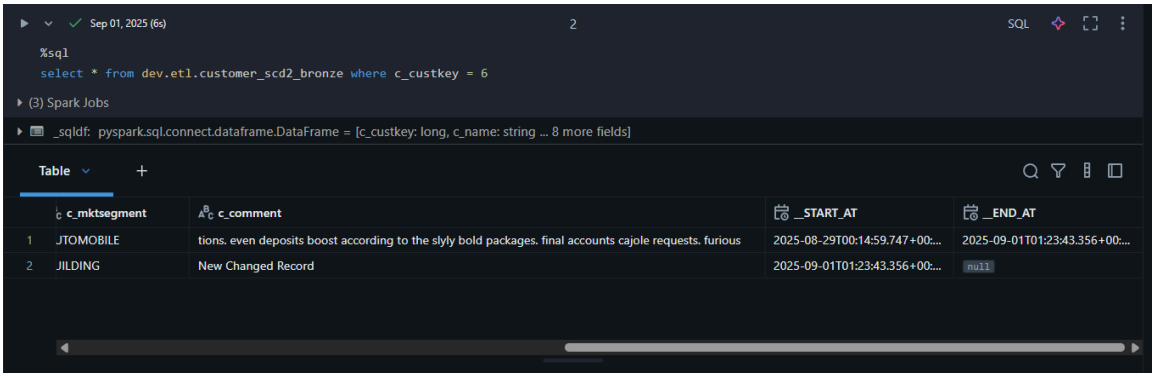
```
dlt.create_streaming_table('customer_sdc2_bronze')

dlt.apply_changes(
  target = "customer_scd1_bronze",
  source = "customer_bronze_vw",
  keys = ['c_custkey'],
  stored_as_scd_type = 2,
  except_column_list = ['__src_action','__src_insert_dt']
  sequence_by = "__src_insert_dt"
)
```

Changes in view to make SCD2 applicable

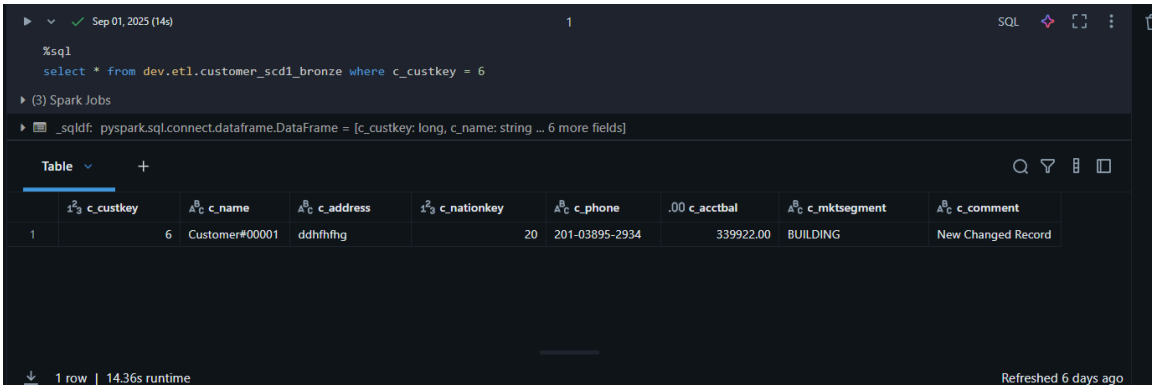
```
@dlt.view(
  comment = 'Joined View'
)
def joined_vw():
  df_c = spark.read.table("LIVE.customers_scd2_bronze").where("__END_AT is
null")
  df_o = spark.read.table("LIVE.orders_union_bronze")

  df_join = df_o.join(df_c,how = "left_outer",on =
df_c.c_custkey==df_o.o_custkey)

  return df_join
```



After inserting record with update the `__END_AT` for the new update is null signifying its the latest update

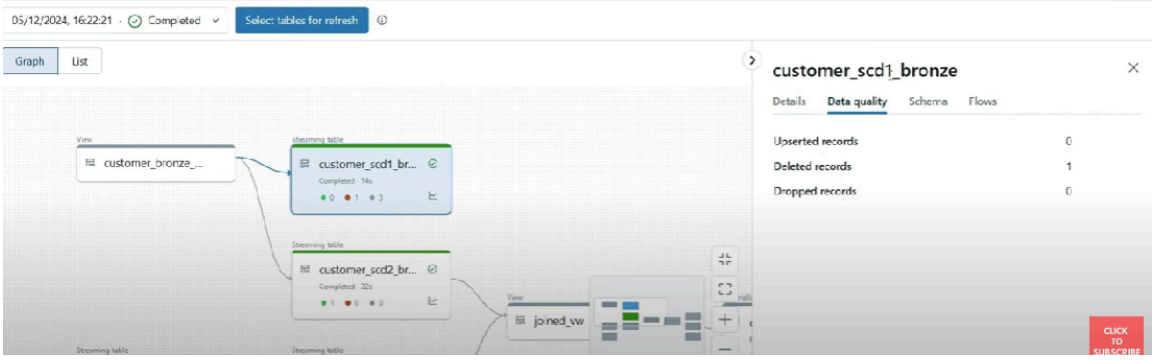


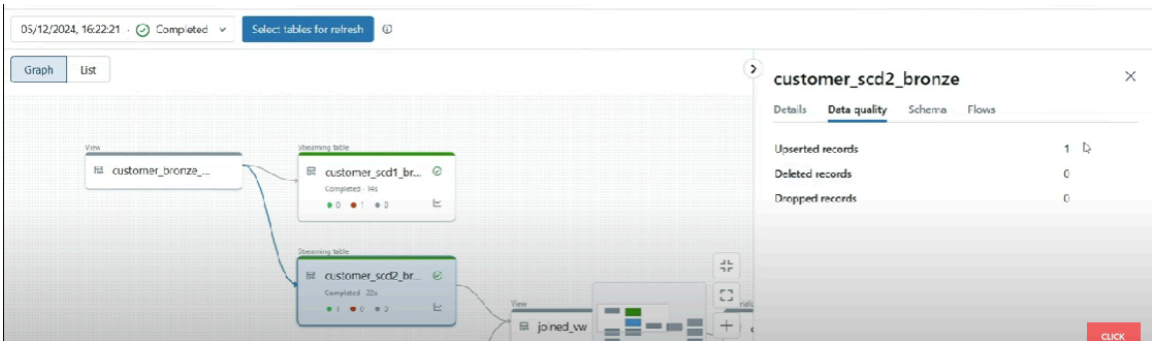
In SCD Type1 just the update is captured.



Insert Old Timestamp record

SCD Type1 vs SCD Type2 Delete Records





Rules for Data Quality : Warn, Drop and Fail

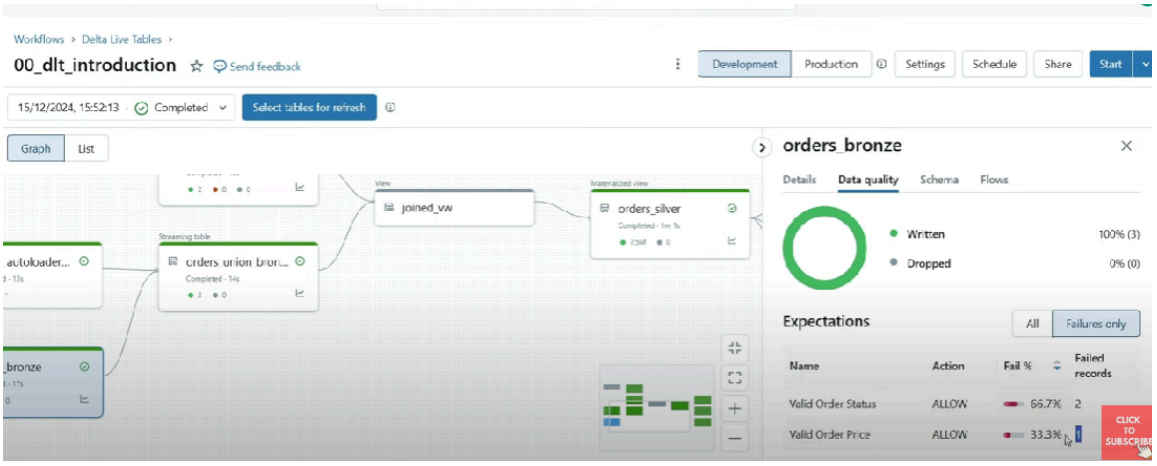
Defining the Rules

```
__order_rules = {
  "Valid Order Status" : "o_order_status in ('O','F','P')",
  "Valid Order Price" : "o_orderprice > 0"
}

__customer_rules = {
  "valid market segment" : "c_mktsegment is not null"
}
```

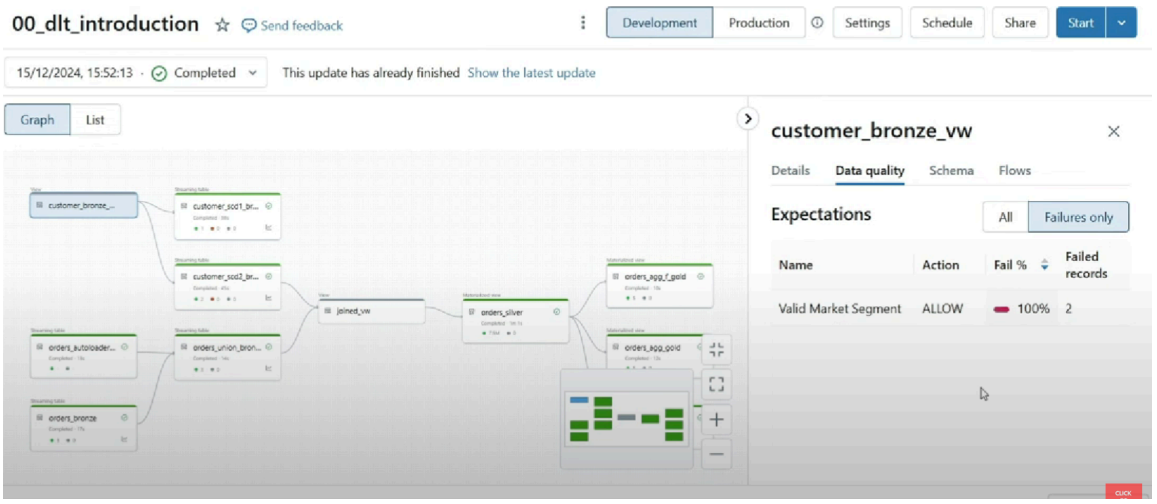
Adding the rules

```
@dlt.table(
  table_properties = {"quality":"bronze"},
  comment = "Orders Bronze Table"
)
@dlt.expect_all(__order_rules) # warn
def orders_bronze():
  df = spark.readStream.table("dev.bronze.orders_raw")
  return df
```

```
@dlt.table(
  table_properties = {"quality":"bronze"},
  comment = "Customers Materialized View"
)
@dlt.expect_all(__customer_rules) # warn
def customers_bronze():
  df = spark.read.table("dev.bronze.customers_raw")
  return df
```



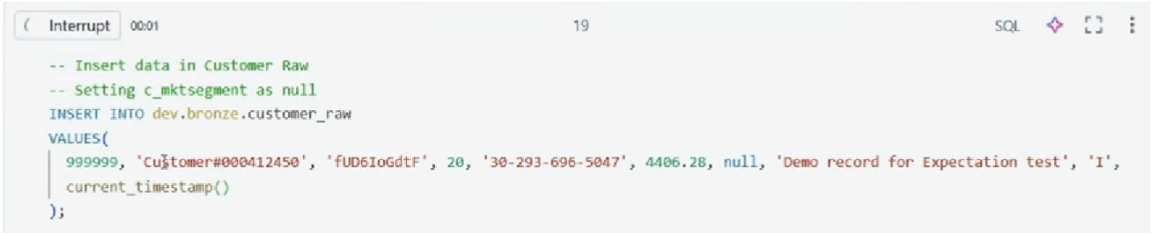
Edge Case



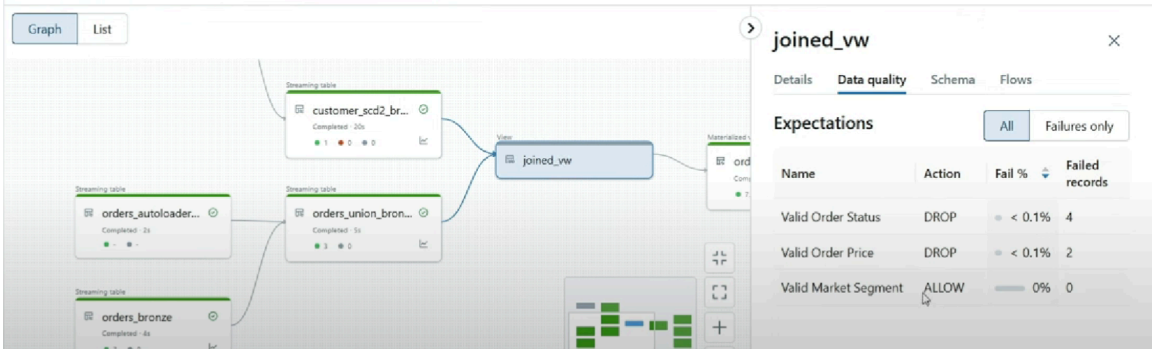
Number of failed records here is 2, but in source table only one record was flawed, but since there are two consumers it shows 2 records failed.

Using Expectations on the view

```
-- Insert data in Orders Raw
-- Setting o_orderstatus other than O, F or P
-- Setting o_orderprice > 0
INSERT INTO dev.bronze.orders_raw
VALUES
(999999, 227285, 'NA', 162169.66, '1995-10-11', '1-URGENT', 'Clerk#000000432', 0, 'Demo record 1 for Expectations test'),
(999999, 227285, 'O', -100, '1995-10-11', '1-URGENT', 'Clerk#000000432', 0, 'Demo record 2 for Expectations test'),
(999999, 227285, null, 999, '1995-10-11', '1-URGENT', 'Clerk#000000432', 0, 'Demo record 3 for Expectations test')
```



Even though on top we can see market segment is null, since we are doing a left join and the joined view does not have details for the customer 99999,(because it failed expectation and record was dropped), so there were no failed records at all.



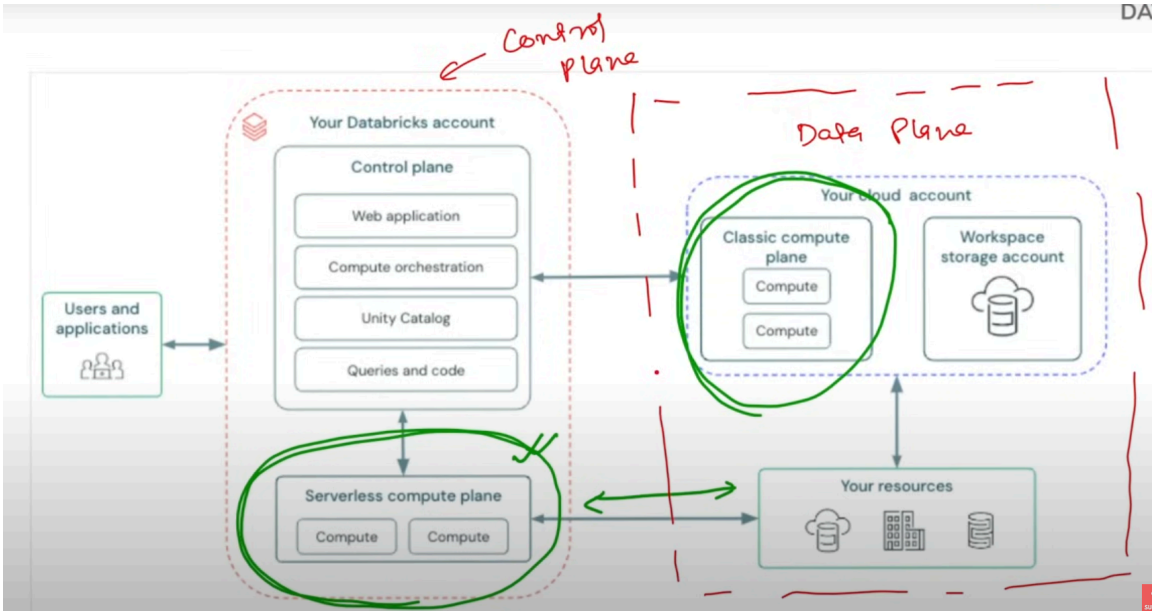
Monitoring and Observability

Check this link

Databricks Serverless Compute

Databricks Serverless Compute

Architecture



Unlike Classic Architecture, the compute is not on the data plane, its on the compute plane managed by Databricks.

Databricks spins up clusters in the same region as the workspace to reduce latency.

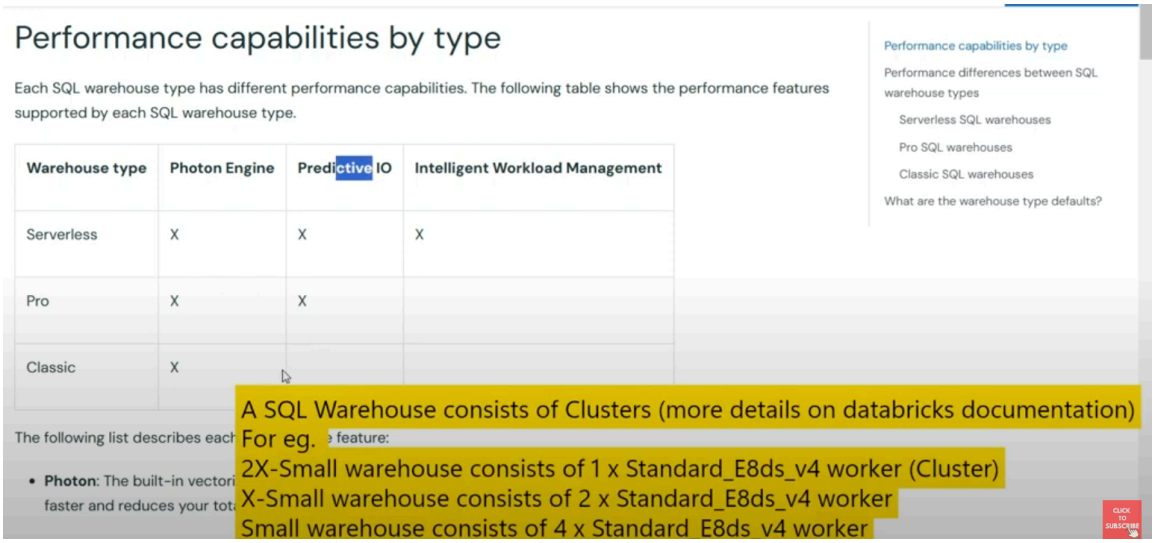
The VM's used for one customer is not reused again.

We need to enable serverless compute in account console to create serverless cluster.

Databricks Warehouses

Databricks SQL Warehouses

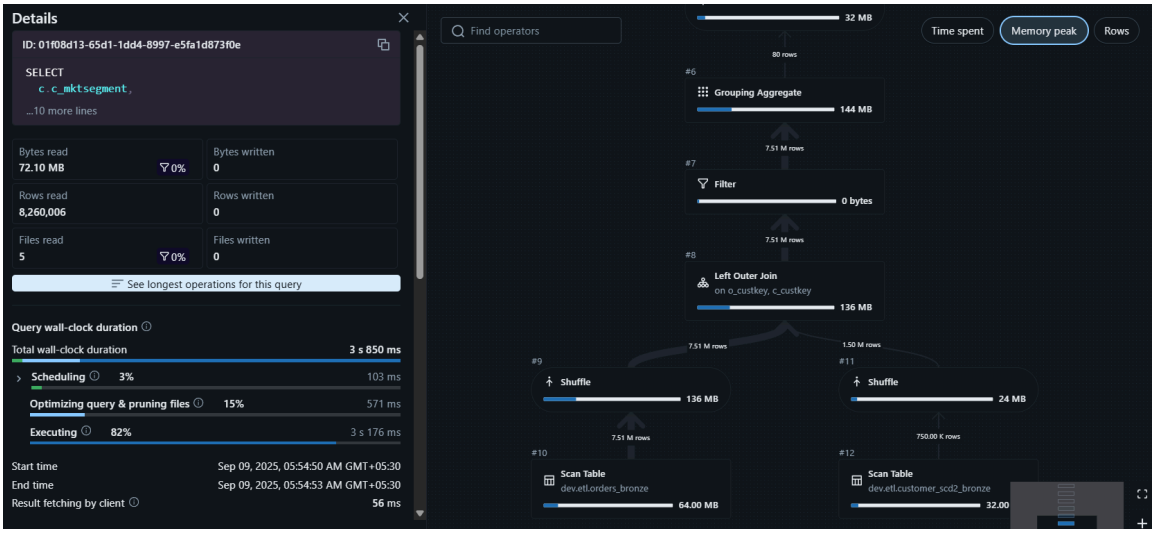
Types of SQL Warehouses



Sample SQL Query and Monitoring

```
SELECT c.c_mktsegment,count(o.o_orderkey) total_orders
FROM
dev.etl.orders_bronze o
LEFT JOIN dev.etl.customer_scd2_bronze c
ON o.o_custkey = c.c_custkey
WHERE c.__END_AT is null
group by c.c_mktsegment
```

Query Monitoring


Details
ID: 01f08d13-65d1-1dd4-8997-e5fa1d873f0e
SELECT
c.c_mktsegment,
...10 more lines
Bytes read
72.10 MB
0%
Bytes written
0
Rows read
8,260,006
Rows written
0
Files read
5
0%
Files written
0
See longest operations for this query
Query wall-clock duration
Total wall-clock duration
3 s 850 ms
Scheduling
3%
103 ms
Optimizing query & pruning files
15%
571 ms
Executing
82%
3 s 176 ms
Start time
Sep 09, 2025, 05:54:50 AM GMT+05:30
End time
Sep 09, 2025, 05:54:53 AM GMT+05:30
Result fetching by client
56 ms
Find operators
Time spent
Memory peak
Rows
32 MB
80 rows
#6
Grouping Aggregate
144 MB
7.51 M rows
#7
Filter
0 bytes
7.51 M rows
#8
Left Outer Join
on o_custkey, c_custkey
136 MB
7.51 M rows
1.50 M rows
#9
Shuffle
136 MB
#11
Shuffle
24 MB
7.51 M rows
750.00 K rows
#10
Scan Table
dev.etl.orders_bronze
64.00 MB
#12
Scan Table
dev.etl.customer_scd2_bronze
32.00

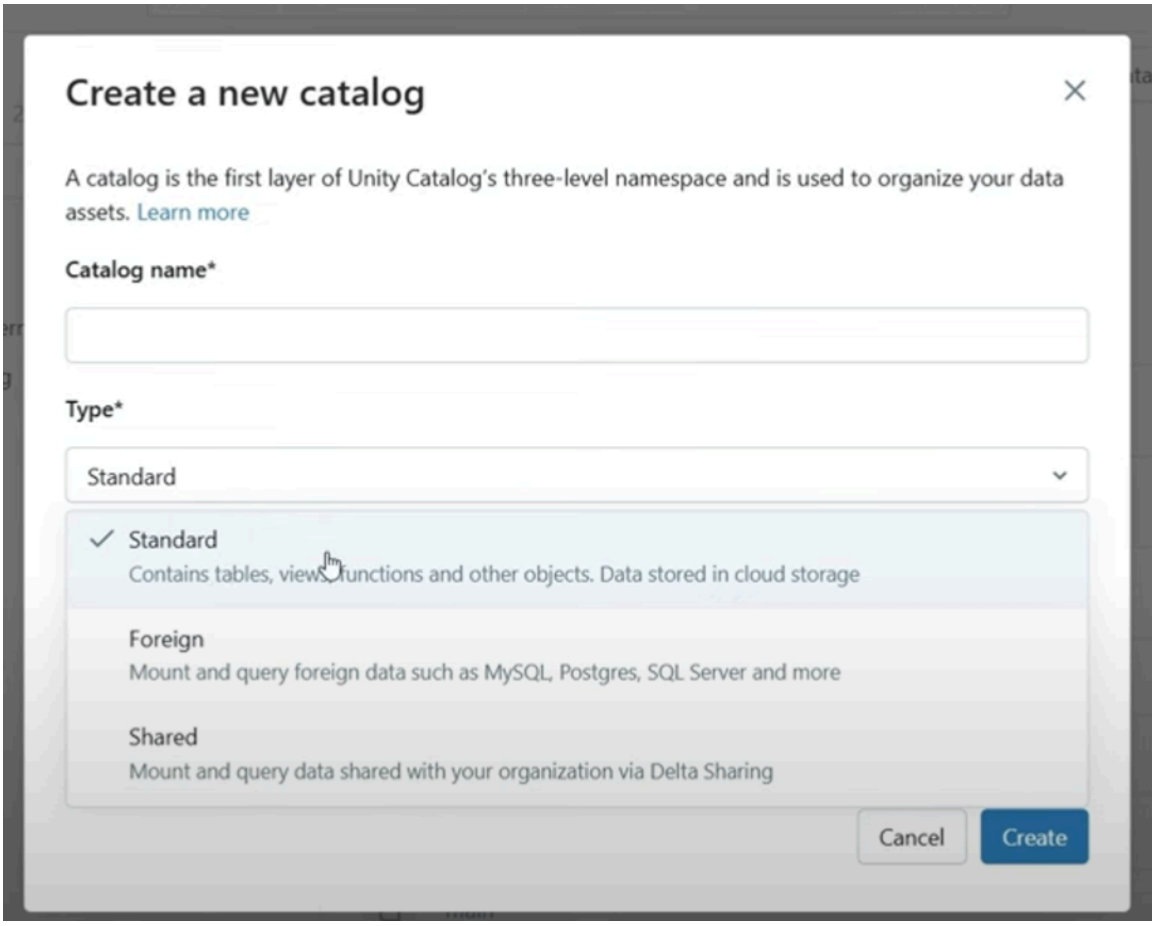
Databricks Lakehouse Federation

Lakehouse Federation in Databricks

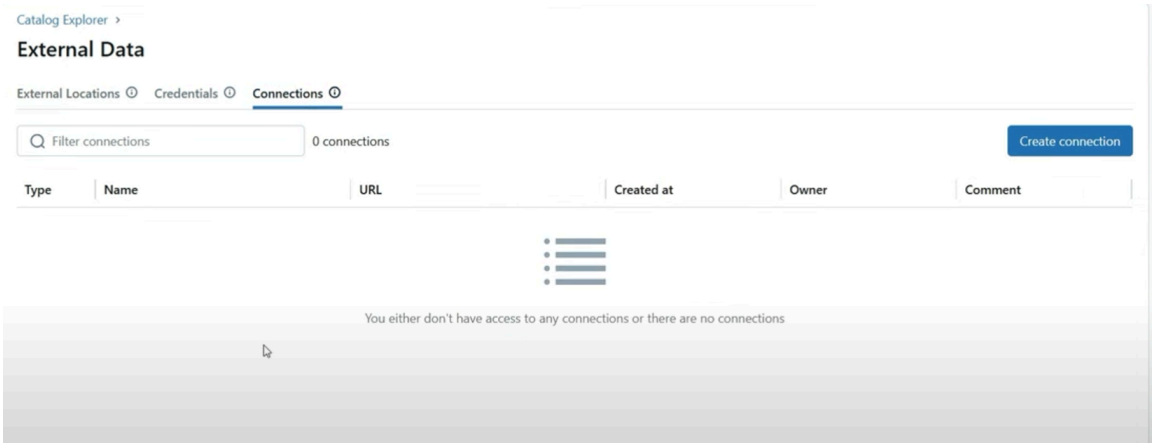
This feature allows us to query external system data without replicating storage on databricks.

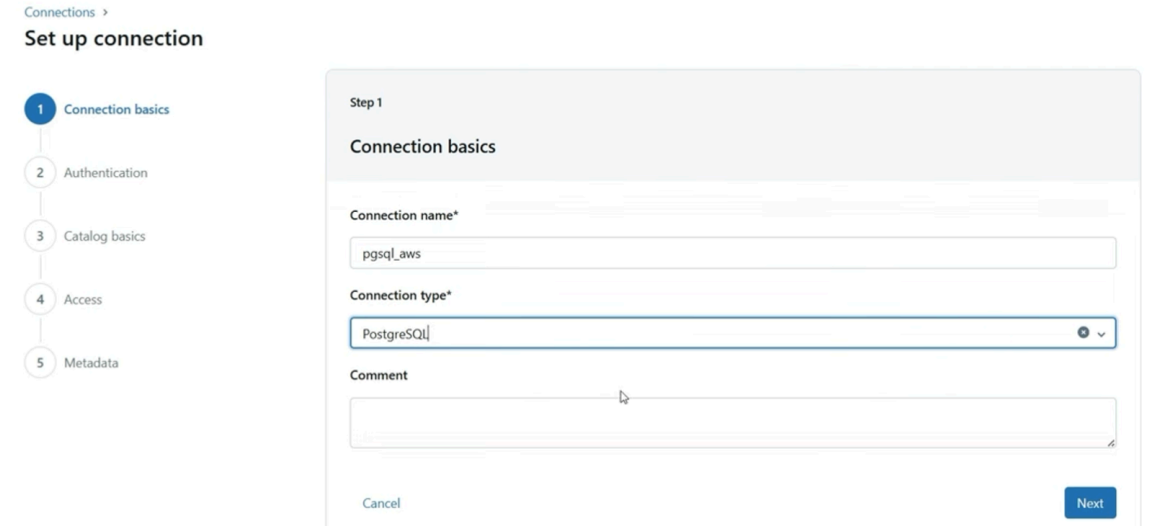
Steps to Setup Federation

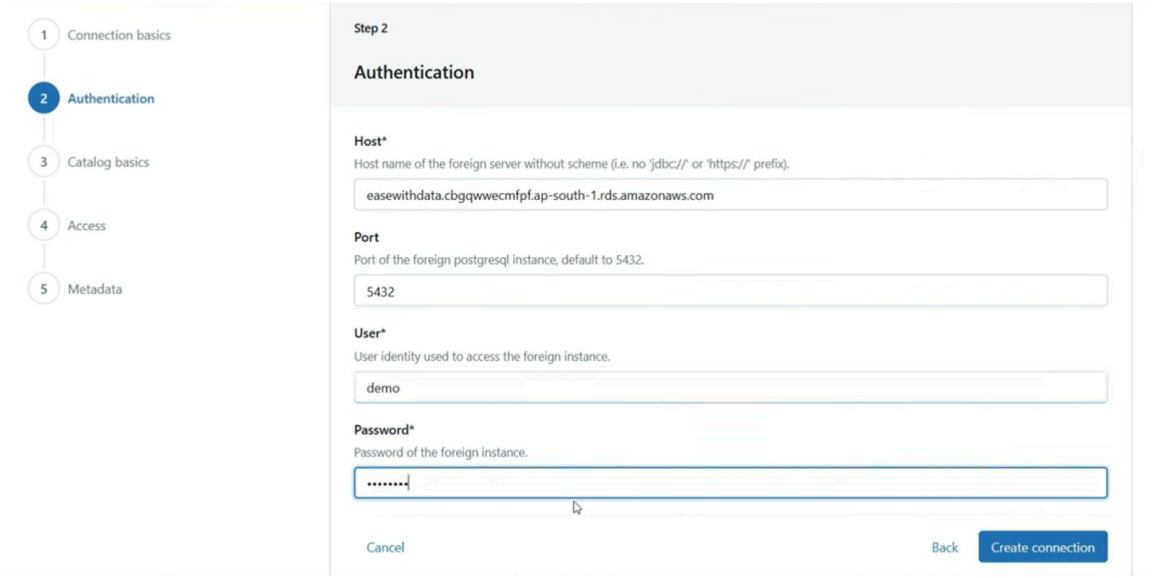
We need the external location to setup connection.

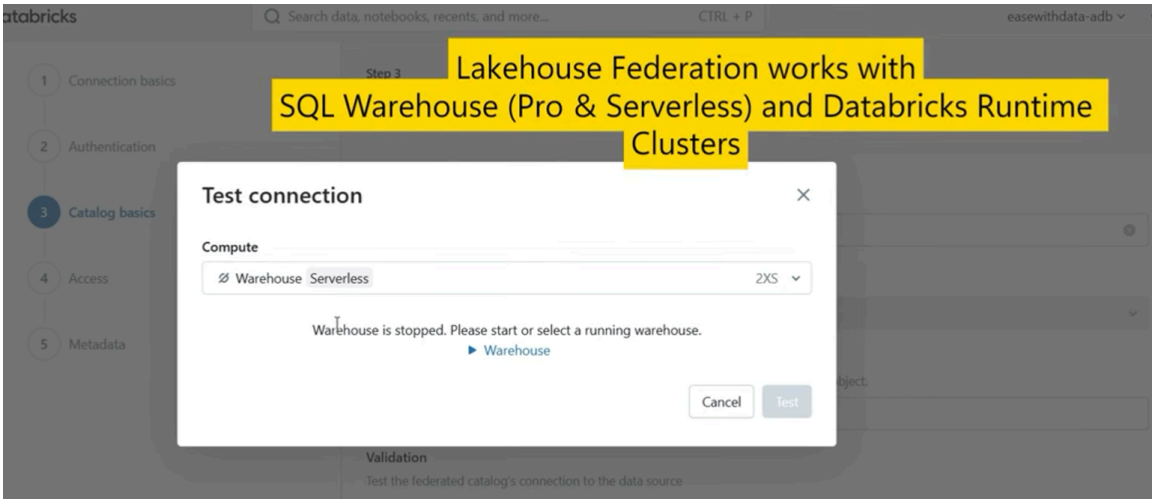


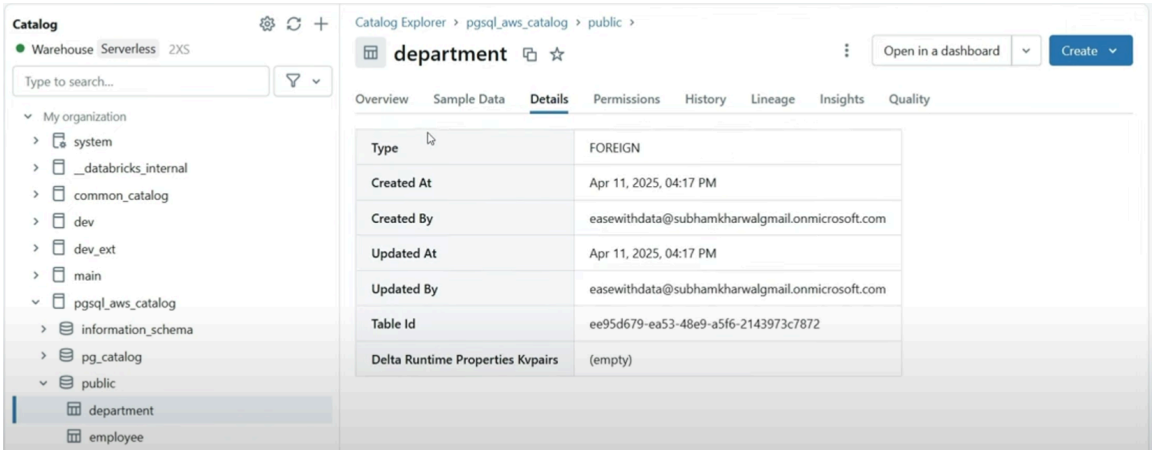
Create a new Connection.


Catalog Explorer ›
External Data
External Locations
Credentials
Connections
Filter connections
0 connections
Create connection
Type
Name
URL
Created at
Owner
Comment
You either don't have access to any connections or there are no connections


Connections ›
Set up connection
1 Connection basics
2 Authentication
3 Catalog basics
4 Access
5 Metadata
Step 1
Connection basics
Connection name*
pgsql_aws
Connection type*
PostgreSQL
Comment
Cancel
Next


1 Connection basics
2 Authentication
3 Catalog basics
4 Access
5 Metadata
Step 2
Authentication
Host*
Host name of the foreign server without scheme (i.e. no 'jdbc://' or 'https://' prefix).
easewithdata.cbgqwwecmfpf.ap-south-1.rds.amazonaws.com
Port
Port of the foreign postgresql instance, default to 5432.
5432
User*
User identity used to access the foreign instance.
demo
Password*
Password of the foreign instance.
Cancel
Back
Create connection





- Foreign catalogs are read only.
- Any catalog that doesnt come under Unity Catalog scheme of things is Foreign Catalog.
- We can manage permissions and also check lineage of tables of Foreign Catalogs.

Creating Catalogs using SQL

Run the following SQL command in a notebook or the SQL query editor. Items in brackets are optional. Replace the placeholder values:

- `<catalog-name>`: Name for the catalog in Azure Databricks.
- `<connection-name>`: The connection object that specifies the data source, path, and access credentials.
- `<database-name>`: Name of the database you want to mirror as a catalog in Azure Databricks. Not required for MySQL, which uses a two-layer namespace.
- `<external-catalog-name>`: *Databricks-to-Databricks* only: Name of the catalog in the external Databricks workspace that you are mirroring. See Create a foreign catalog.

SQL Copy

```
CREATE FOREIGN CATALOG [IF NOT EXISTS] <catalog-name> USING CONNECTION <connection-name>
OPTIONS (database '<database-name>');
```

Metrics Views in Databricks

Metric views compared to standard views

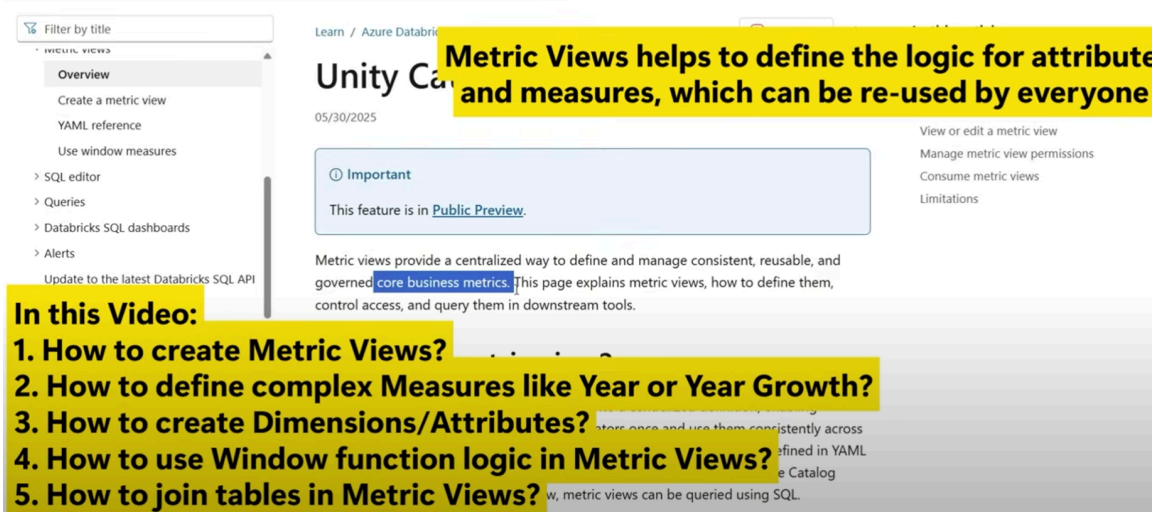
Standard views are typically designed to answer a specific business question. They often include aggregation logic and dimension groupings that must be specified when the view is created. This can cause issues when a user wants to query a dimension not included in the original view. More complex measures, such as ratios or distinct counts, often cannot be re-aggregated without returning incorrect results.

Metric views allow authors to define measures and dimensions independently of how users filter and group the data. This separation allows the system to generate the correct query based on the user's selection, while preserving consistent, centralized metric logic.

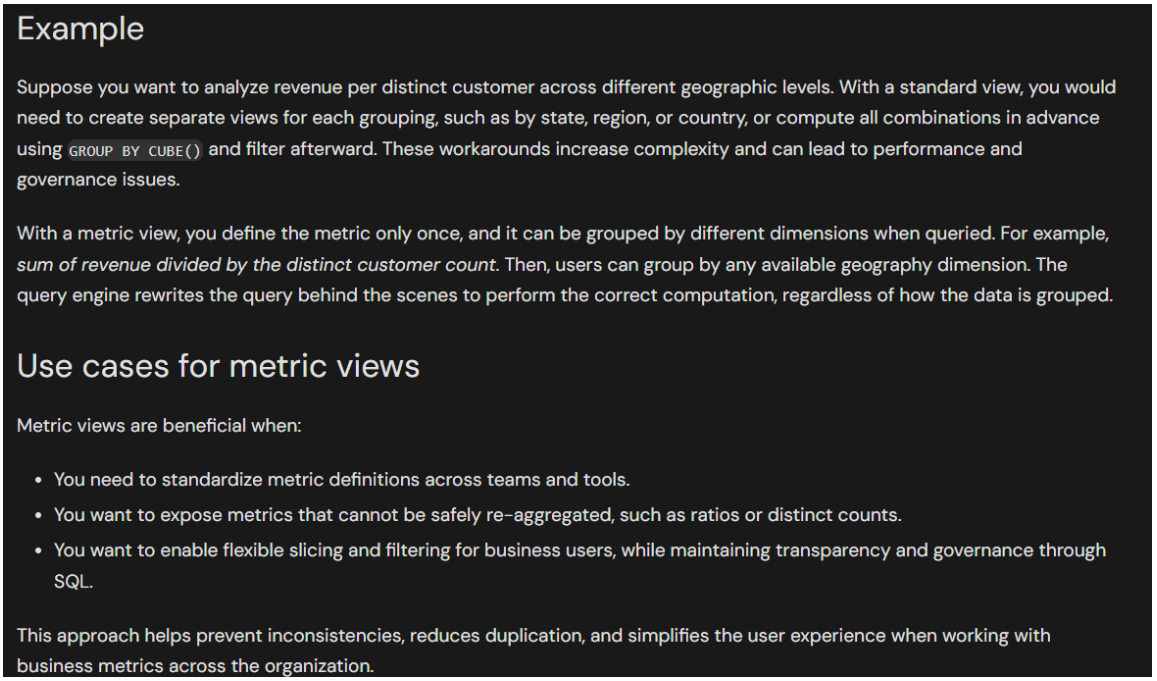
Example

Suppose you want to analyze revenue per distinct customer across different geographic levels. With a standard view, you would need to create separate views for each grouping, such as by state, region, or country, or compute all combinations in advance using `GROUP BY CUBE()` and filter afterward. These workarounds increase complexity and can lead to performance and governance issues.

With a metric view, you define the metric only once, and it can be grouped by different dimensions when queried. For example, *sum of revenue divided by the distinct customer count*. Then, users can group by any available geography dimension. The query engine rewrites the query behind the scenes to perform the correct computation, regardless of how the data is grouped.



Example Use Case



How to Create Metrics Views in Databricks?

Measures are quantitative values like sales, revenue etc.

Dimensions are used to support measures like year, month etc. Eg: sales per year, revenue per month

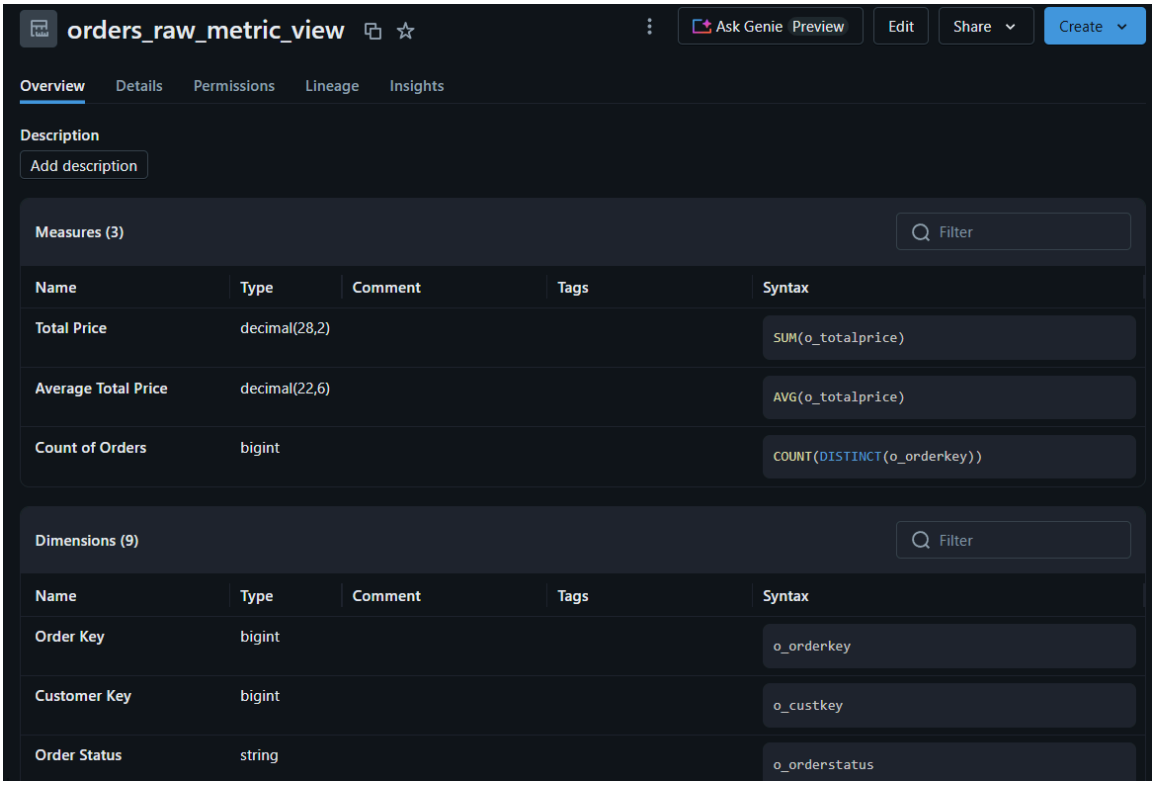
Any joins in metric views are left outer joins.

Metric views are defined in yaml format

Eg:

```yaml
version: 0.1

source: dev.bronze.orders_raw

dimensions:
  - name: Order Key
    expr: o_orderkey

  - name: Customer Key
    expr: o_custkey

  - name: Order Status
    expr: o_orderstatus

  - name: Order Status Readable
    expr: >
      case
        when o_orderstatus = 'O' then 'Open'
        when o_orderstatus = 'F' then 'Fulfilling'
```

```
        when o_orderstatus = 'P' then 'Processing'
      end

  - name: Order Date
    expr: o_orderdate

  - name: Order Year
    expr: DATE_TRUNC('year',o_orderdate)

  - name: Order Priority
    expr: o_orderpriority

  - name: Clerk
    expr: o_clerk

  - name: Ship Priority
    expr: o_shippriority

measures:
  - name: Total Price
    expr: SUM(o_totalprice)

  - name: Average Total Price
    expr: AVG(o_totalprice)

  - name: Count of Orders
    expr: COUNT(DISTINCT(o_orderkey))
```



Name	Type	Comment	Tags	Syntax
Order Key	bigint			o_orderkey
Customer Key	bigint	Add comment	Add tags	o_custkey
Order Status	string			o_orderstatus
Order Status Readable	string			case when o_orderstatus = 'O' then 'Open' ... 3 more lines
Order Date	date			o_orderdate
Order Year	timestamp			DATE_TRUNC('year', o_orderdate)
Order Priority	string			o_orderpriority
Clerk	string			o_clerk
Ship Priority	int			o_shippriority

Querying the metric view

```
SELECT (`Order Status Readable`),MEASURE(`Total Price`) FROM orders_raw_metric_view GROUP BY (`Order Status Readable`);
```

Raw results

	Order Status Readable	measure(Total Price)
1	Open	549645446554.61
2	Processing	35384096972.62
3	Fulfilling	549905178604.88
4	null	-2.1

Window Functions

```yaml
measures:
  - name: Total Price
    expr: SUM(o_totalprice)

  - name: Average Total Price
    expr: AVG(o_totalprice)

  - name: Count of Orders
    expr: COUNT(DISTINCT(o_orderkey))

  - name: Total Orders Current Year
    expr: COUNT(DISTINCT o_orderkey)
    window:
      - order: Order Year
        range: current
```

```
        semiadditive: last

  - name: Total Orders Last Year
    expr: COUNT(DISTINCT o_orderkey)
    window:
      - order: Order Year
        range: trailing 1 year
        semiadditive: last
```

Final Metric Views with Joins

```
version: 0.1

source: dev.bronze.orders_raw

joins:
  - name: cust_dim
    source: dev.bronze.customer_raw
    on: cust_dim.c_custkey = source.o_custkey

dimensions:
  - name: Order Key
    expr: o_orderkey

  - name: Customer Key
    expr: o_custkey

  - name: Order Status
    expr: o_orderstatus

  - name: Order Status Readable
    expr: >
      case
        when o_orderstatus = 'O' then 'Open'
        when o_orderstatus = 'F' then 'Fulfilling'
        when o_orderstatus = 'P' then 'Processing'
      end

  - name: Customer Market Segment
    expr: cust_dim.c_mktsegment

  - name: Order Date
    expr: o_orderdate

  - name: Order Year
    expr: DATE_TRUNC('year',o_orderdate)

  - name: Order Priority
    expr: o_orderpriority

  - name: Clerk
    expr: o_clerk

  - name: Ship Priority
    expr: o_shippriority
```

```
measures:
  - name: Total Price
    expr: SUM(o_totalprice)

  - name: Average Total Price
    expr: AVG(o_totalprice)

  - name: Count of Orders
    expr: COUNT(DISTINCT(o_orderkey))

  - name: Total Orders Current Year
    expr: COUNT(DISTINCT o_orderkey)
    window:
      - order: Order Year
        range: current
        semiadditive: last

  - name: Total Orders Last Year
    expr: COUNT(DISTINCT o_orderkey)
    window:
      - order: Order Year
        range: trailing 1 year
        semiadditive: last

  - name: Year on Year Growth
    expr: MEASURE(`Total Orders Current Year`)-MEASURE(`Total Orders Last
Year`)
```

Clear Example of Metric View vs Standard View

---

1. Standard View Approach

If you only had **standard views**, you'd need **separate SQL queries** (or separate views) for each grouping level.

(a) Group by **State**

```
SELECT
    state,
    SUM(revenue) / COUNT(DISTINCT customer_id) AS revenue_per_customer
FROM orders
GROUP BY state;
```

(b) Group by **Region**

```
SELECT
    region,
```

```
    SUM(revenue) / COUNT(DISTINCT customer_id) AS revenue_per_customer
FROM orders
GROUP BY region;
```

(c) Group by **Country**

```
SELECT
    country,
    SUM(revenue) / COUNT(DISTINCT customer_id) AS revenue_per_customer
FROM orders
GROUP BY country;
```

 Notice: you either write **multiple queries/views**, or you compute all combinations in one big query with `GROUP BY CUBE(country, region, state)` (which can be heavy).

---

2. Metric View Approach

With a **metric view**, you define the **metric once**:

```
CREATE OR REPLACE METRIC VIEW revenue_metrics
AS SELECT
    country,
    region,
    state,
    SUM(revenue) AS total_revenue,
    COUNT(DISTINCT customer_id) AS distinct_customers
FROM orders
GROUP BY country, region, state;
```

- **Metric defined:**

```
revenue_per_customer = total_revenue / distinct_customers
```

Now, when analysts query this metric view, they don't need to rewrite SQL for each grouping. The query engine rewrites it under the hood.

Example Queries on Metric View

(a) Group by State

```
SELECT
    state,
    total_revenue / distinct_customers AS revenue_per_customer
FROM revenue_metrics;
```

(b) Group by Region

```
SELECT
    region,
    SUM(total_revenue) / SUM(distinct_customers) AS revenue_per_customer
FROM revenue_metrics
GROUP BY region;
```

(c) Group by Country

```
SELECT
    country,
    SUM(total_revenue) / SUM(distinct_customers) AS revenue_per_customer
FROM revenue_metrics
GROUP BY country;
```

 Difference: You don't redefine the metric — you just group by a different **dimension** (state, region, country). Databricks rewrites the query correctly to maintain **metric consistency** (so that KPIs mean the same thing everywhere).

 In short:

- **Standard views** = you must predefine each grouping (state/region/country).
- **Metric views** = define the metric once (`revenue_per_customer`) and reuse across any dimension.

Streaming and Materialized Views in Databricks SQL

There is no auto option for incremental load while developing streaming tables and views in SQL, hence additional option has to be provided.

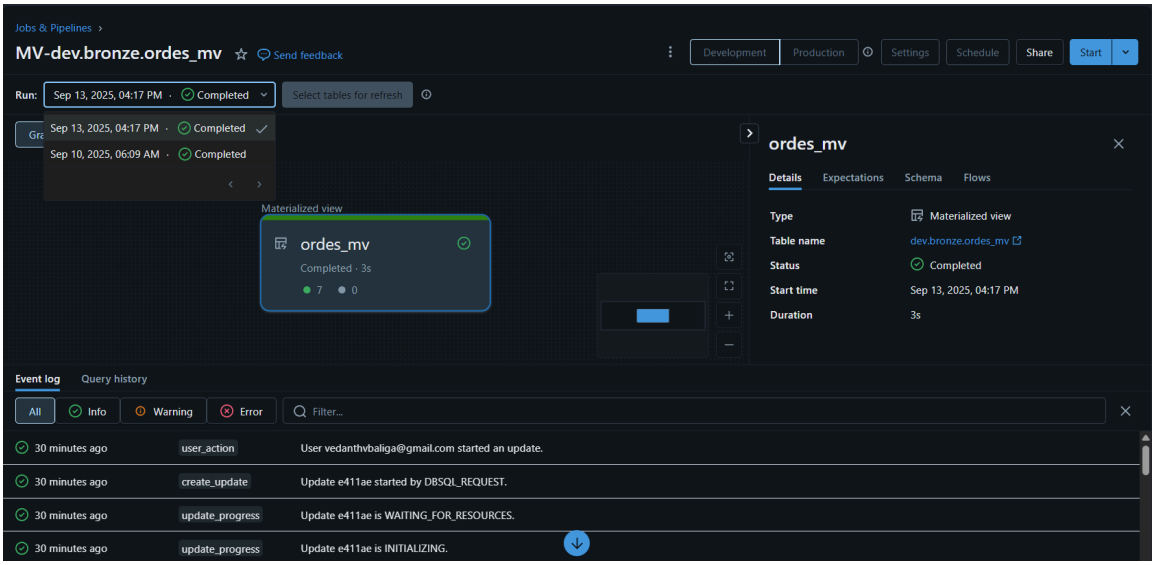
```
CREATE OR REFRESH STREAMING TABLE dev.bronze.orders_st
AS
SELECT * FROM
STREAM read_files(
  "/Volumes/dev/bronze/landing/input/",
  format => 'csv',
  includeExistingFiles => false
)
```

```
CREATE OR REPLACE MATERIALIZED VIEW dev.bronze.ordes_mv
-- SCHEDULE EVERY 4 HOURS
AS
SELECT Country,sum(UnitPrice) as agg_total_price FROM dev.bronze.orders_st
group by Country
```

Replacing the materialized view does not refresh entire data, just incrementally. The group aggregate is also calculated incrementally.

```
CREATE OR REPLACE MATERIALIZED VIEW dev.bronze.ordes_mv
-- SCHEDULE EVERY 4 HOURS
AS
SELECT Country,sum(UnitPrice) as agg_total_price FROM dev.bronze.orders_st
group by Country
```

Every run of the query on Serverless Warehouse spins up DLT job in background.


Jobs & Pipelines
MV-dev.bronze.ordes_mv
Send feedback
Development
Production
Settings
Schedule
Share
Start
Run:
Sep 13, 2025, 04:17 PM · Completed
Select tables for refresh
Sep 13, 2025, 04:17 PM · Completed
Sep 10, 2025, 06:09 AM · Completed
Materialized view
ordes_mv
Completed · 3s
7
0
ordes_mv
Details
Expectations
Schema
Flows
Type
Materialized view
Table name
dev.bronze.ordes_mv
Status
Completed
Start time
Sep 13, 2025, 04:17 PM
Duration
3s
Event log
Query history
All
Info
Warning
Error
Filter...
30 minutes ago
user_action
User vedanthvbaliga@gmail.com started an update.
30 minutes ago
create_update
Update e411ae started by DBSQL_REQUEST.
30 minutes ago
update_progress
Update e411ae is WAITING_FOR_RESOURCES.
30 minutes ago
update_progress
Update e411ae is INITIALIZING.

✅ Steps to Setup Databricks CLI on Linux

1. Install Python & pip

Databricks CLI is a Python package. Check if you have Python 3 installed:

```
python3 --version
```

If not, install it:

```
sudo apt update
sudo apt install python3 python3-pip -y
```

2. Install Databricks CLI

Run:

```
pip3 install databricks-cli --upgrade
```

Check installation:

```
databricks --version
```

3. Generate a Personal Access Token (PAT) in Databricks

1. Go to your **Databricks workspace** in the browser.
2. Click on your username (top right) → **User Settings**.
3. Under **Access Tokens**, click **Generate New Token**.
4. Copy the token (you won't see it again).

4. Configure Databricks CLI

Run:

```
databricks configure --token
```

It will ask:

- **Databricks Host (URL)** → Example:
- AWS: `https://<workspace-url>.cloud.databricks.com`
- Azure: `https://<workspace-name>.azuredatabricks.net`
- **Token** → Paste the token you generated.

---

5. Test the CLI

Run a test command to list clusters:

```
databricks clusters list
```

If successful, you'll see details of your clusters.

---

6. (Optional) Store Multiple Profiles

You can save multiple workspace logins using profiles in `~/.databricks/config`. Example:

```
[DEFAULT]
host = https://myworkspace.azuredatabricks.net
token = dapi123abc

[staging]
host = https://staging-workspace.azuredatabricks.net
token = dapi456def
```

Then use:

```
databricks --profile staging clusters list
```

---

7. Use CLI for Common Tasks

Examples:

```
# List jobs
databricks jobs list

# Upload a Python file to DBFS
databricks fs cp myscript.py dbfs:/FileStore/scripts/myscript.py

# Run a job
databricks jobs run-now --job-id <job_id>
```