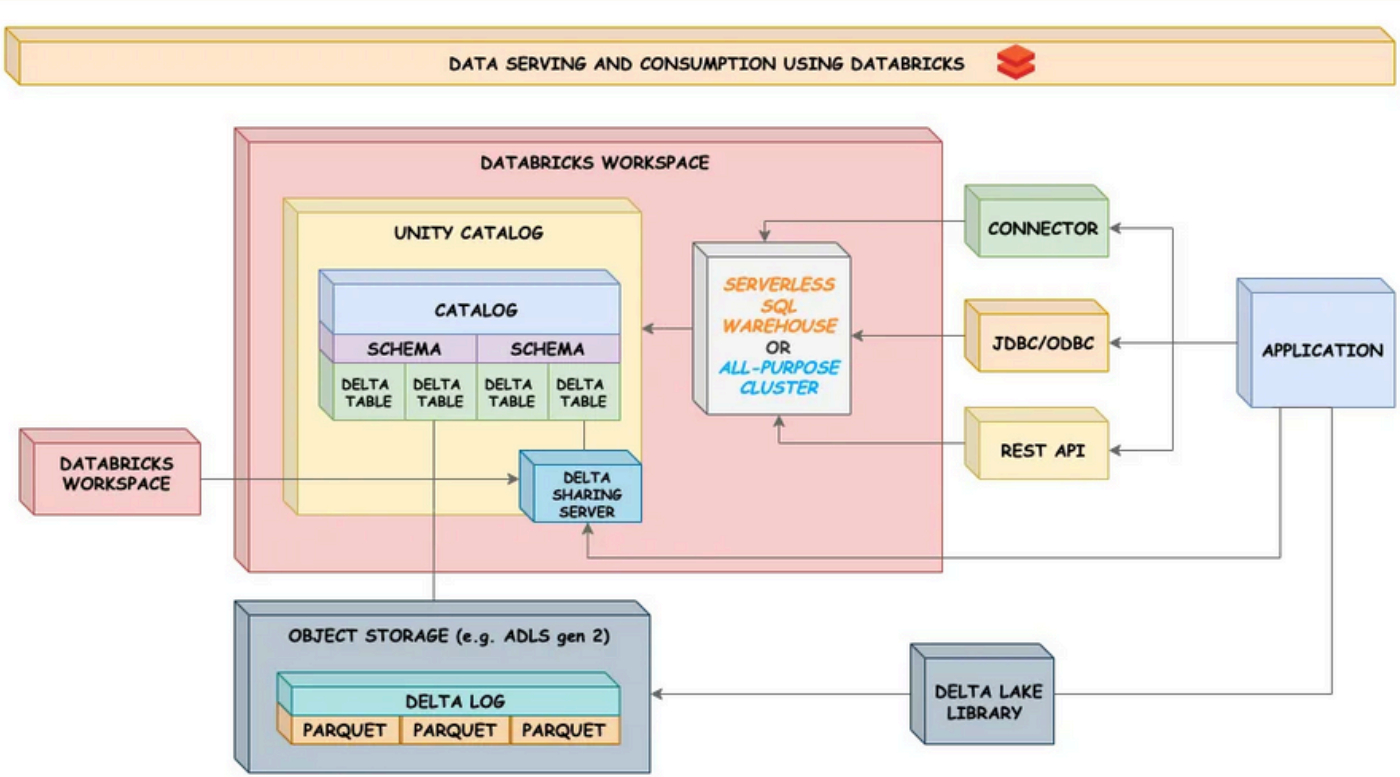
Making Databricks Data Accessible to End Data Serving and Consumption Using Databricks

Ganesh R

Azure Data Engineer



Data Serving and Consumption Using Databricks

There are many articles and resources on how to process data within Databricks, but far fewer that discuss how to achieve the end goal: making data available to end-users or applications.

- What options do we have?
- What are the pros and cons of each option? When should you choose one over
- another?

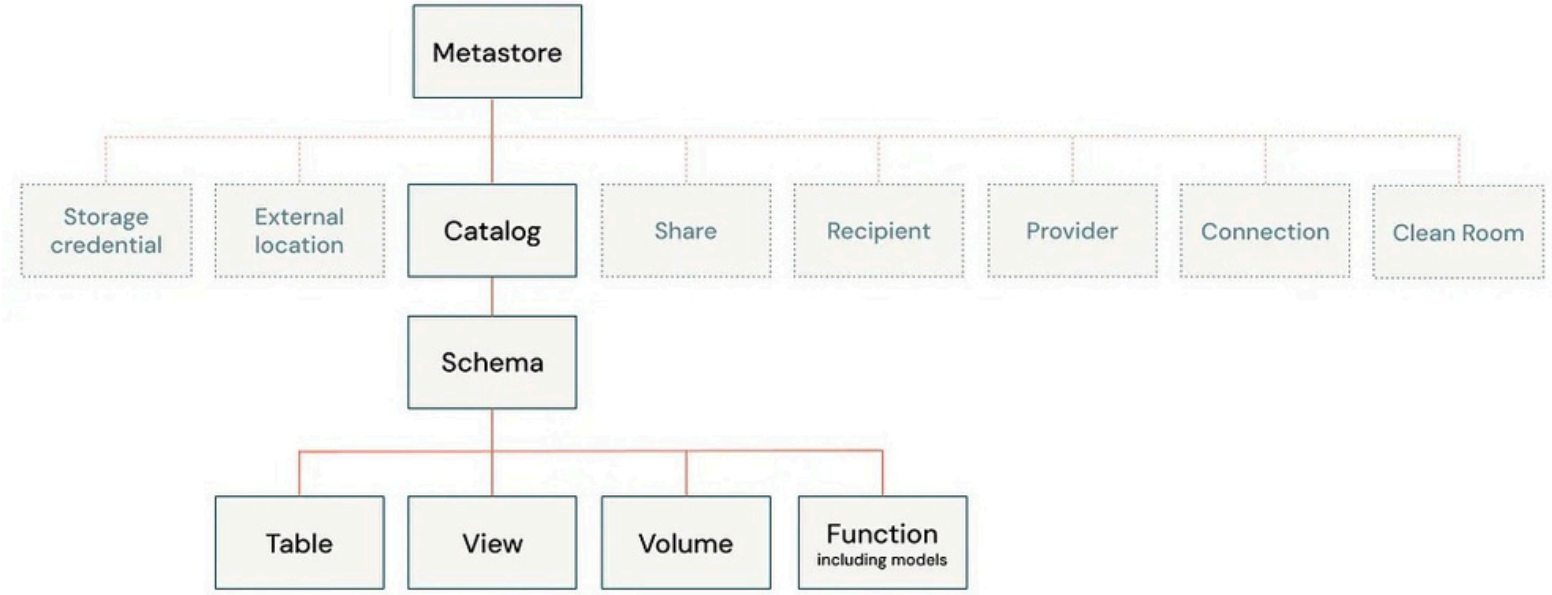
On the one hand, there are built-in solutions like (AI/BI) dashboards and, more recently, Databricks Apps. These are designed to make data consumption as easy as possible while keeping users within the Databricks ecosystem. But what if a company already has established tools like PowerBI, needs more extensive capabilities, or has higher requirements?

In this article, we will cover all the available options to help you choose the right one for your organization.

Data Lifecycle and Governance in Databricks

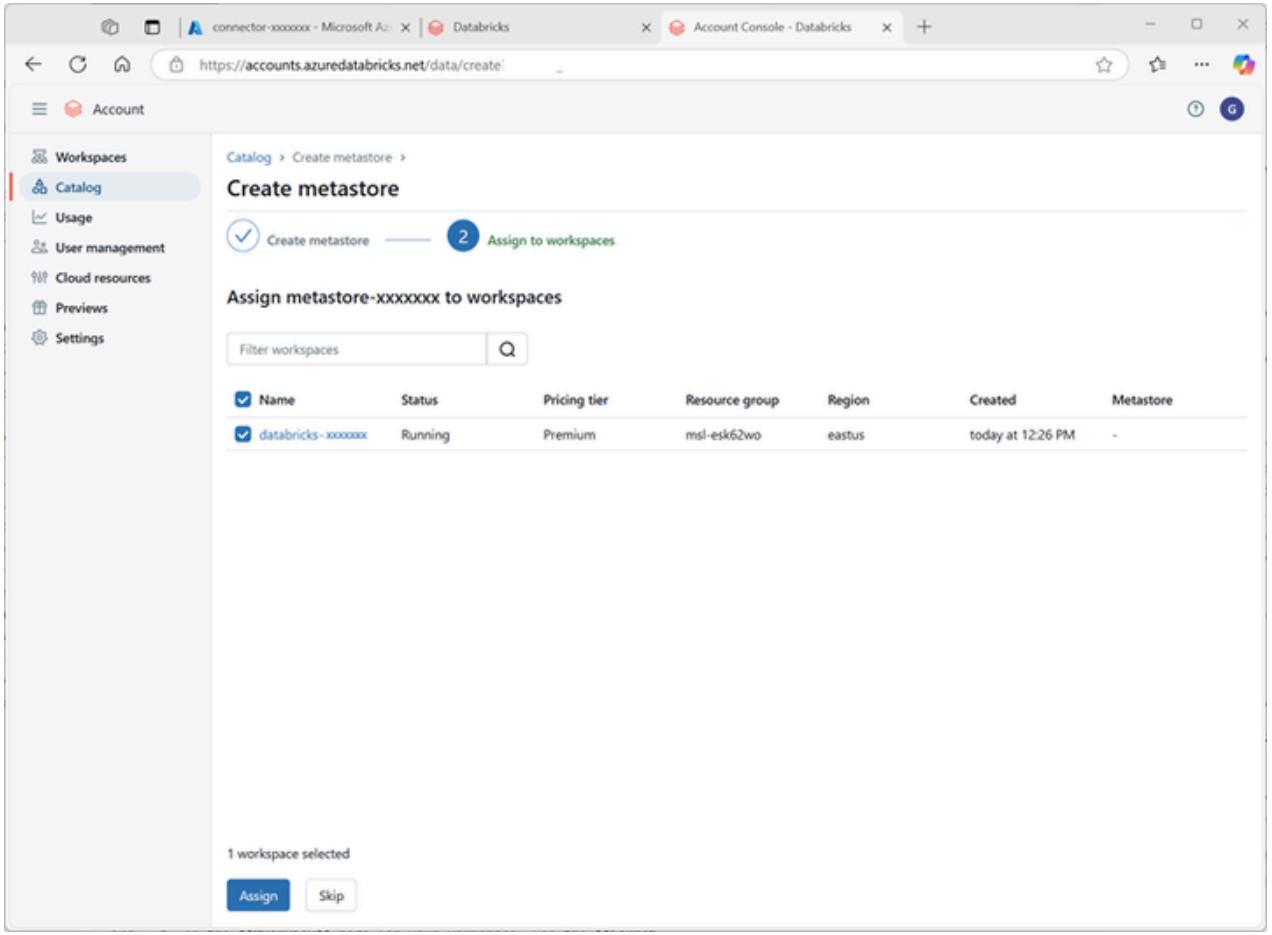
To understand the full range of options, let's start by examining what happens within Databricks itself. Data is ingested from various sources — such as object stores, databases, or streaming services — and then processed within Databricks. Two key considerations here are data governance and storage, as these factors significantly impact how data can be consumed and managed within Databricks. Since Unity Catalog represents the future of the Databricks platform, we will focus on the options available when Unity Catalog is enabled.

To understand how data can be accessed within Databricks, we need to examine Unity Catalog's governance model. Because the entire setup can be somewhat complex, we'll explain in detail what the implications are.

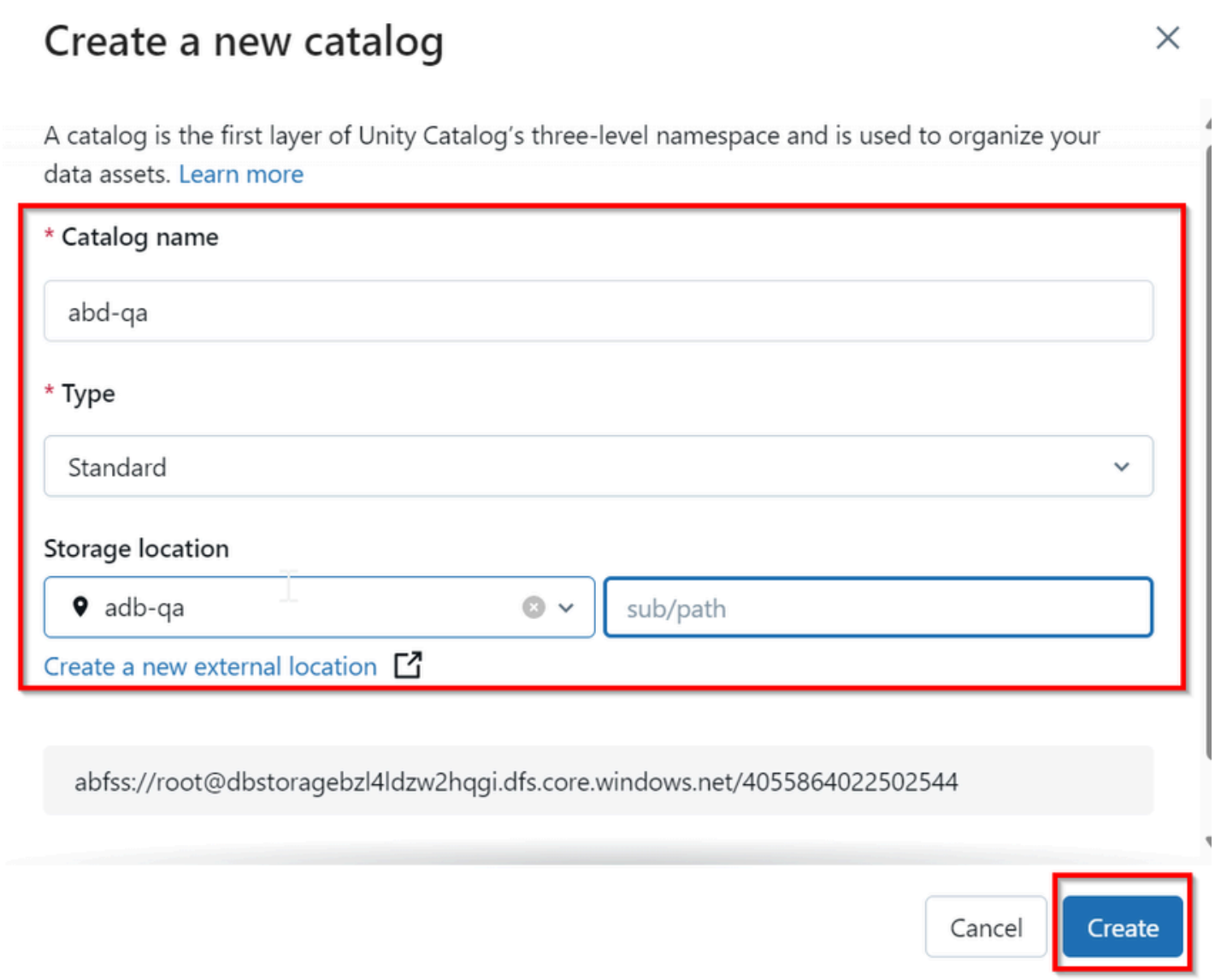


Unity Catalog Hierarchy

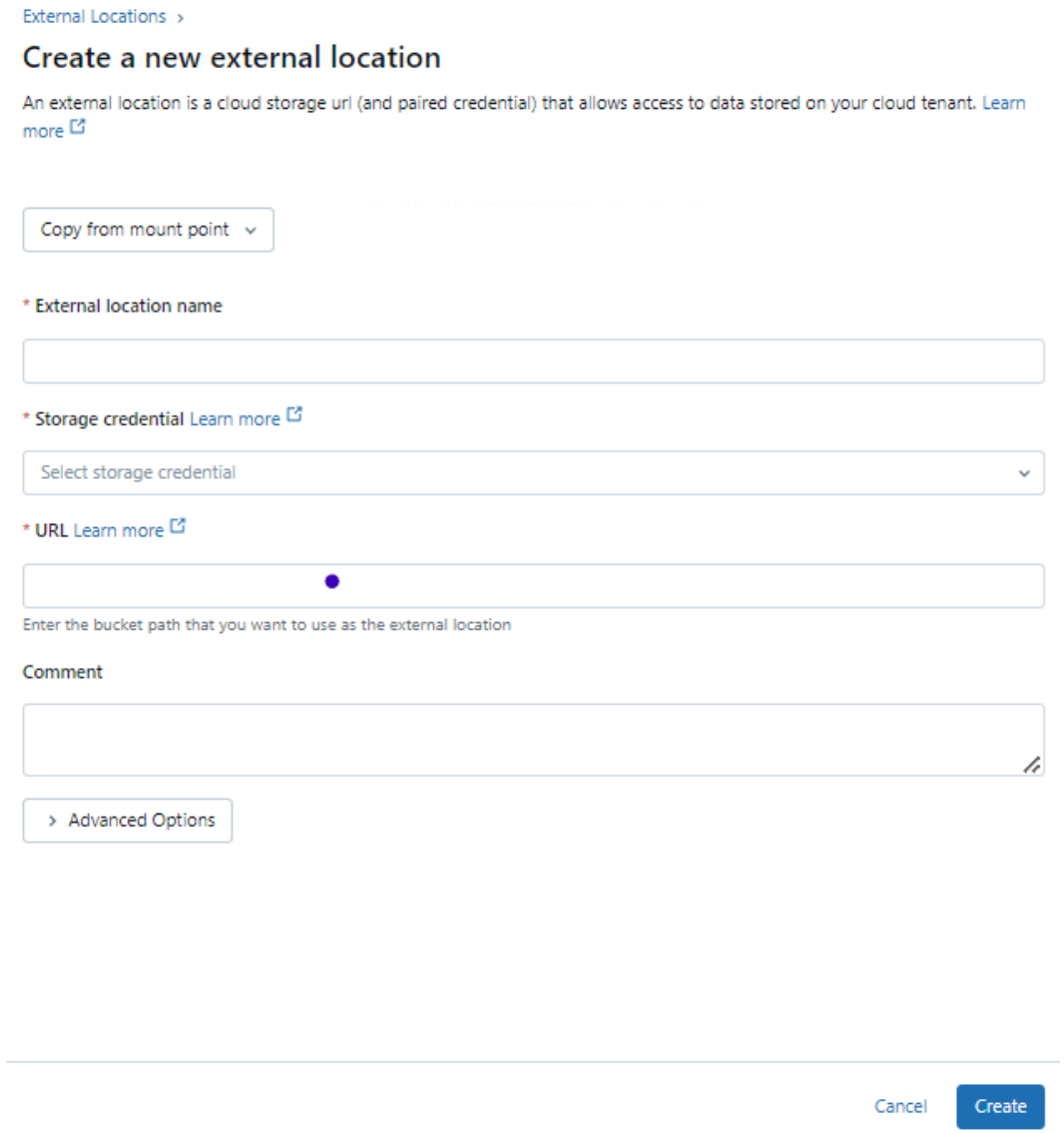
The foundation of Unity Catalog is the metastore. Each Databricks account can have a single metastore per region (with all Databricks workspaces within a cloud tenant belonging to a central Databricks account). When creating a metastore, we can specify a root location. If this isn't specified, we'll need to define a location for each new catalog we create.



This is important because if we choose to specify a location, when we then create catalogs, schemas and tables WITHOUT specifying a location, this is where they will be stored.



When we then want to create a catalog, we can specify a storage location but it's not simply a path to an object store. It is an "external location". An external location combines a "storage credential" (an identity) with a "storage path".

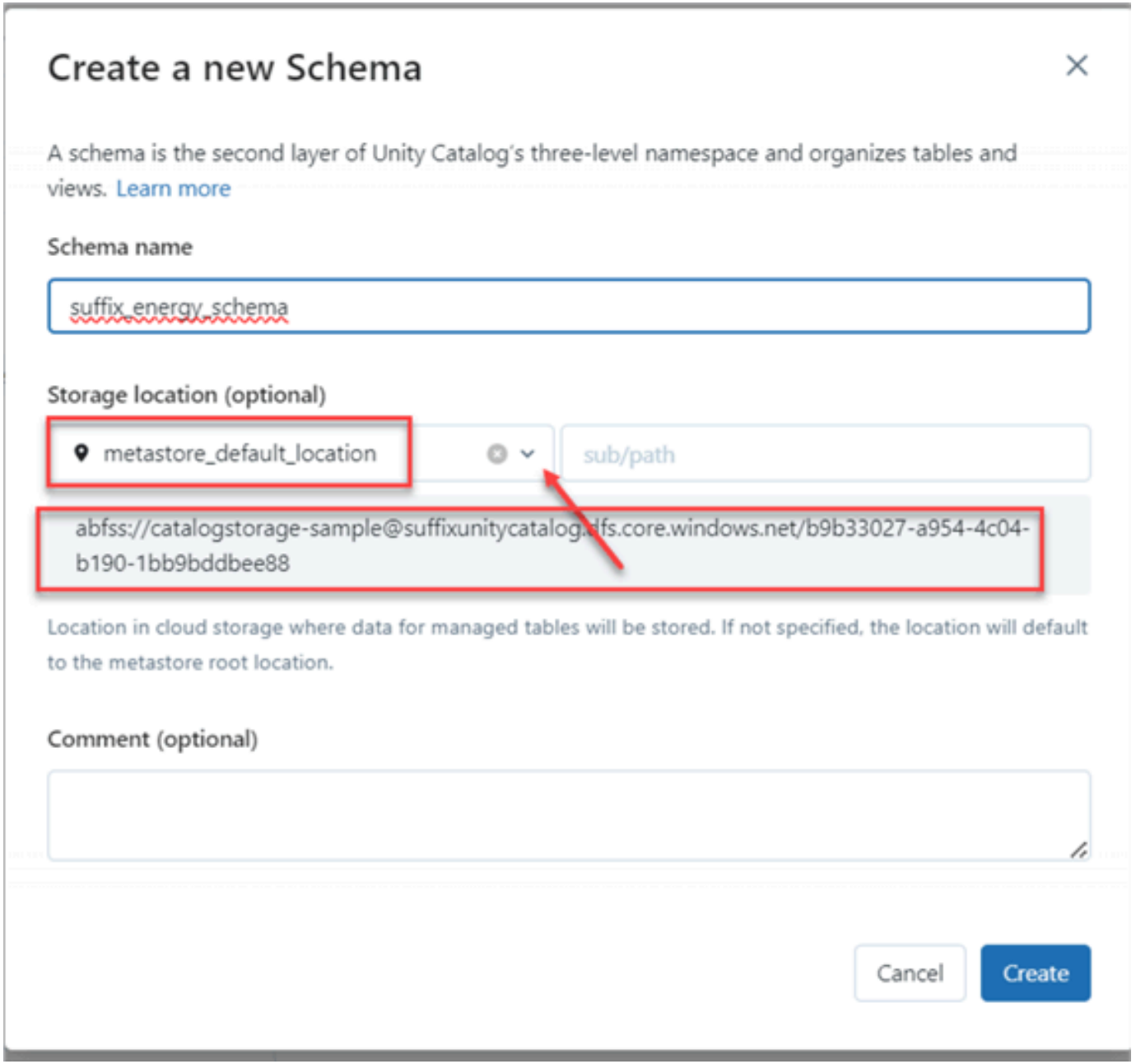


Previously, with the Hive Metastore, we really only had two options:

1. Creating a table without a location which resulted in a managed table
2. Creating a table with a location which resulted in an external table

In this setup, the location was a path to an object store:

```
abfss://container@storageaccount.dfs.core.windows.net/table_name
```



Now, Unity Catalog enables us to specify external locations for each object level — metastore, catalog, schema, and table. This means we can create a catalog and schema within such an external location and if we then don't specify a location when we create a table within these catalogs and schemas, it will still be a MANAGED TABLE.

Cmd 1

```
import pyspark.sql.functions as F
df = spark.range(10).withColumn("c", F.rand())
df.write.format("delta").save("/tmp/aott-tdb")
```

▸ (4) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [id: long, c: double]

Command took 4.63 seconds -- by

Cmd 2

```
spark.sql("CREATE TABLE aott_tdb   USING DELTA  LOCATION '/tmp/aott-tdb'")
```

▸ (3) Spark Jobs

Out[2]: DataFrame[]

Command took 1.42 seconds -- by

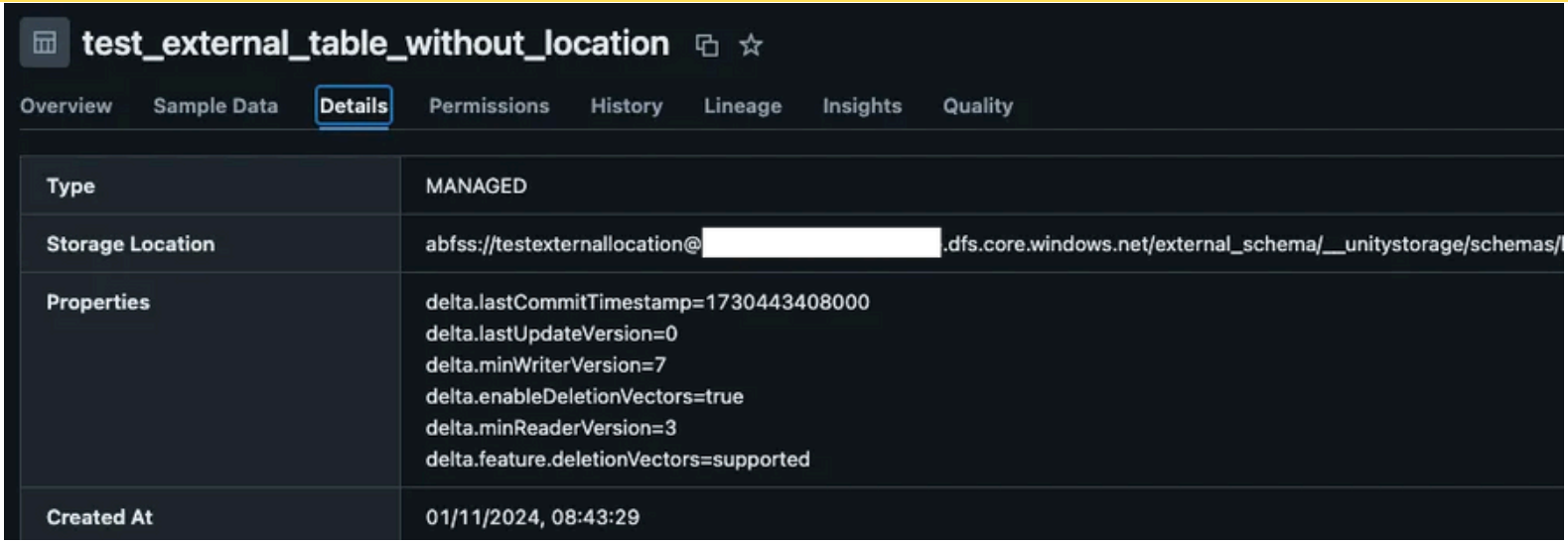
Cmd 3

```
%sql
select * from aott_tdb
```

▸ (3) Spark Jobs

	id	c
1	3	0.6383800654042262
2	4	0.044033017824975595
3	8	0.7795797277
4	9	0.5297061594258456
5	5	0.21534697502160371

Copy to clipboard



This configuration gives organizations flexibility over the data storage location while benefiting from Unity Catalog's governance capabilities. However, it can be somewhat confusing to understand because when I first heard "external location", I assumed that all the tables would be external tables.

In conclusion, the only factor that decides whether a table is managed or unmanaged (external) is whether a location is specified for the table like this:

Why Is the Managed Table Faster?

From the test results, the managed table query was nearly 6 times faster than the external table query. Several key factors contribute to this difference:

1. Metadata Optimization

Managed tables leverage Databricks' internal metadata caching, DELTA and automatic statistics collection, making queries significantly faster.

External tables rely on the raw storage format, requiring additional lookups, which increases query overhead.

2. Partition Pruning and Indexing

Partition pruning is automatically optimized for managed tables.

External tables may require manual partitioning optimizations, as their metadata is stored externally.

3. Data Caching and IO Handling

Managed tables store data in Delta format, benefiting from Databricks' caching mechanisms.

External tables reference raw Parquet files, which don't leverage the same optimizations.

4. Storage Layer Optimization

Managed table data is structured and optimized for Databricks queries.

External tables access raw files directly from Azure Storage, leading to higher network latency and additional read operations.

Feature	Managed Table ✅	External Table 🌍
Performance	⚡ Fastest	🐢 Slower
Metadata Handling	✅ Automatic	❌ Manual
Storage Control	🔒 Databricks-managed	🌎 External Storage
Auto-Optimization	✅ Yes	❌ No
Delete Impact	🚀 Data + Metadata deleted	❗ Only metadata deleted

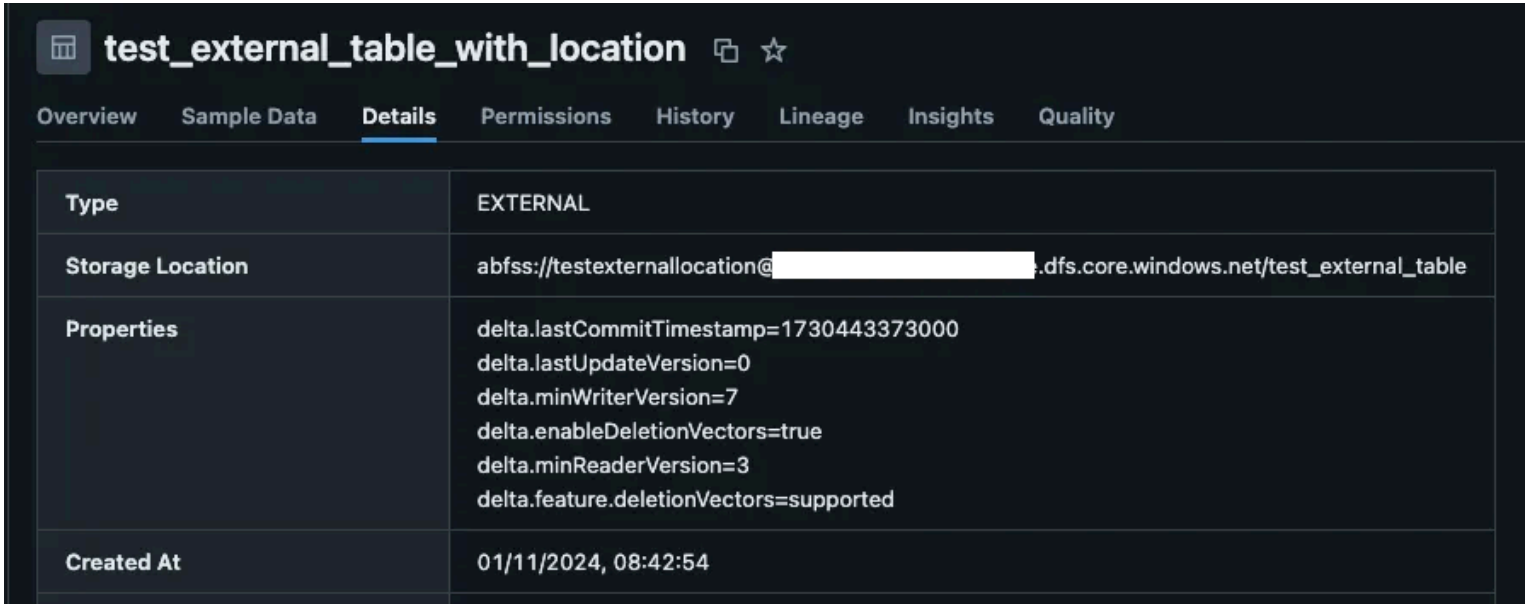
2 minutes ago (6s) 1

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

table_schema = StructType(
    [
        StructField("id", IntegerType(), True),
        StructField("name", StringType(), True),
        StructField("age", IntegerType(), True),
    ]
)

empty_df = spark.createDataFrame([], schema=table_schema)
empty_df.createOrReplaceTempView("empty_sql_df")

spark.sql(f"""
    CREATE OR REPLACE TABLE test_external_catalog.test_external_schema.test_external_table_with_location
    LOCATION 'abfss://testexternallocation@[redacted].dfs.core.windows.net/test_external_table'
    AS (SELECT * FROM empty_sql_df)
""")
```



test_external_table_with_location

Overview | Sample Data | **Details** | Permissions | History | Lineage | Insights | Quality

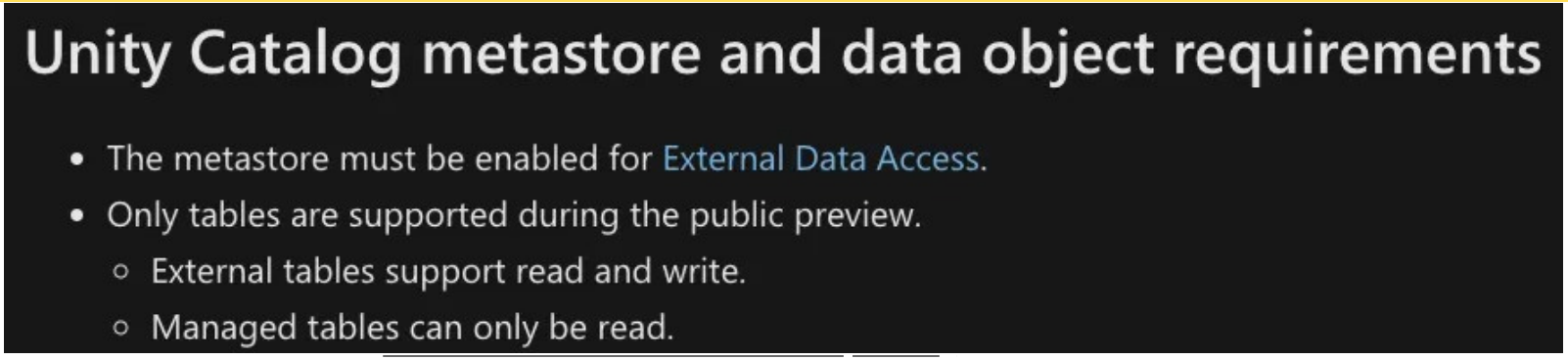
Type	EXTERNAL
Storage Location	abfss://testexternallocation@[redacted].dfs.core.windows.net/test_external_table
Properties	delta.lastCommitTimestamp=1730443373000 delta.lastUpdateVersion=0 delta.minWriterVersion=7 delta.enableDeletionVectors=true delta.minReaderVersion=3 delta.feature.deletionVectors=supported
Created At	01/11/2024, 08:42:54

But what does “managed table” or “external table” actually mean? And what does it mean for a table to be part of the Unity Catalog governance model?

In the second example, we created an external table that is still part of Unity Catalog’s governance model. As long as we specify the full structure, such as <catalog-name>.<schema-name>.<table-name>, when creating a table, it will be part of the Unity Catalog governance model but still be stored in the location we specify on the metastore / catalog / schema level.

This means that if we delete the external table inside of Databricks, the underlying data is not deleted (for managed tables it should be deleted within max. 30 days from the cloud provider). However, we can still enforce governance on to the table.
Why is all of this important?

This distinction is crucial because it impacts data accessibility and governance. For example, when we want to control how data can be accessed via Unity Catalog, there are differences between external and managed tables:



Control external access to data in Unity Catalog

In the end, it also boils down to having control over where the data is stored and how it is managed and whether we want to rely on Datbaricks's governance system and use Databricks compute resources to further process the data. It's important to know that we can still have external tables that are part of Unity Catalog's governance model and managed tables that are stored where we want them to be.

However, other features, like Liquid Clustering, require tables to be managed. This is because, for managed tables, Databricks controls the lifecycle and file layout of the table. When using managed tables, it is not recommended to directly access and manipulate the underlying files.

What To Do With Processed Data

Returning to the data lifecycle, data in Databricks is typically stored in Delta tables and processed through different stages of quality and aggregation, following the medallion architecture. Once data reaches the desired state, we have two primary routes we can go:

External System Integration:

- Writing data out to an external system can be ideal if Databricks is only part of a larger data pipeline. For example, if the data pipeline includes multiple systems, Databricks may write out the processed data to the next one in the pipeline. Or, the next system in line could access and even manipulate the raw files (if we are working with external tables).
- Another common scenario is when a primary analytics environment outside of Databricks governs the data.
- External systems may also be preferable for requirements like ultra-low latency, which a dedicated SQL warehouse could better support.

In-Platform:

- Data can remain within the Databricks ecosystem for consumption. This option is especially relevant when users primarily work within Databricks or when tools and dashboards within Databricks can fulfill all needs.

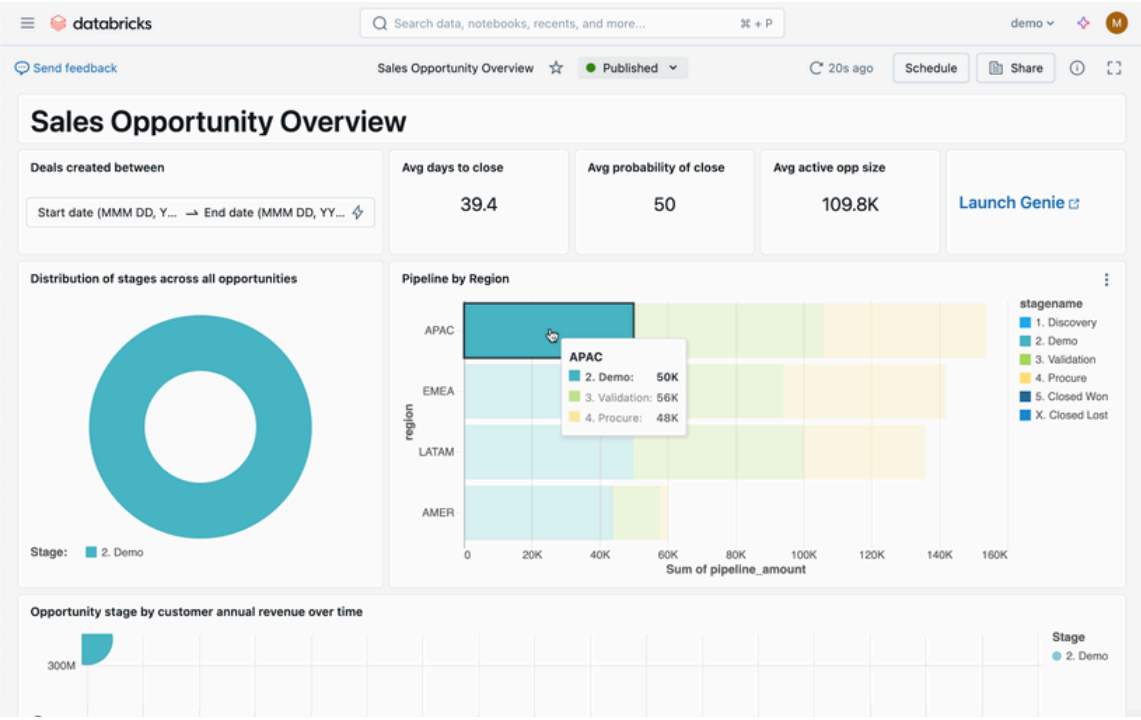
As writing out the data into other system is dependent on the target system, we will continue focusing on the options we have if data remains within the platform.

Options for Data Consumption Within Databricks

Built-In Consumption Tools

Databricks (AI/BI) Dashboards

Databricks offers built-in capabilities for creating dashboards and visualizations using Databricks SQL. We can write SQL queries against the data and present the results in various chart formats. This approach allows for easy integration and real-time data access since everything stays within the platform.



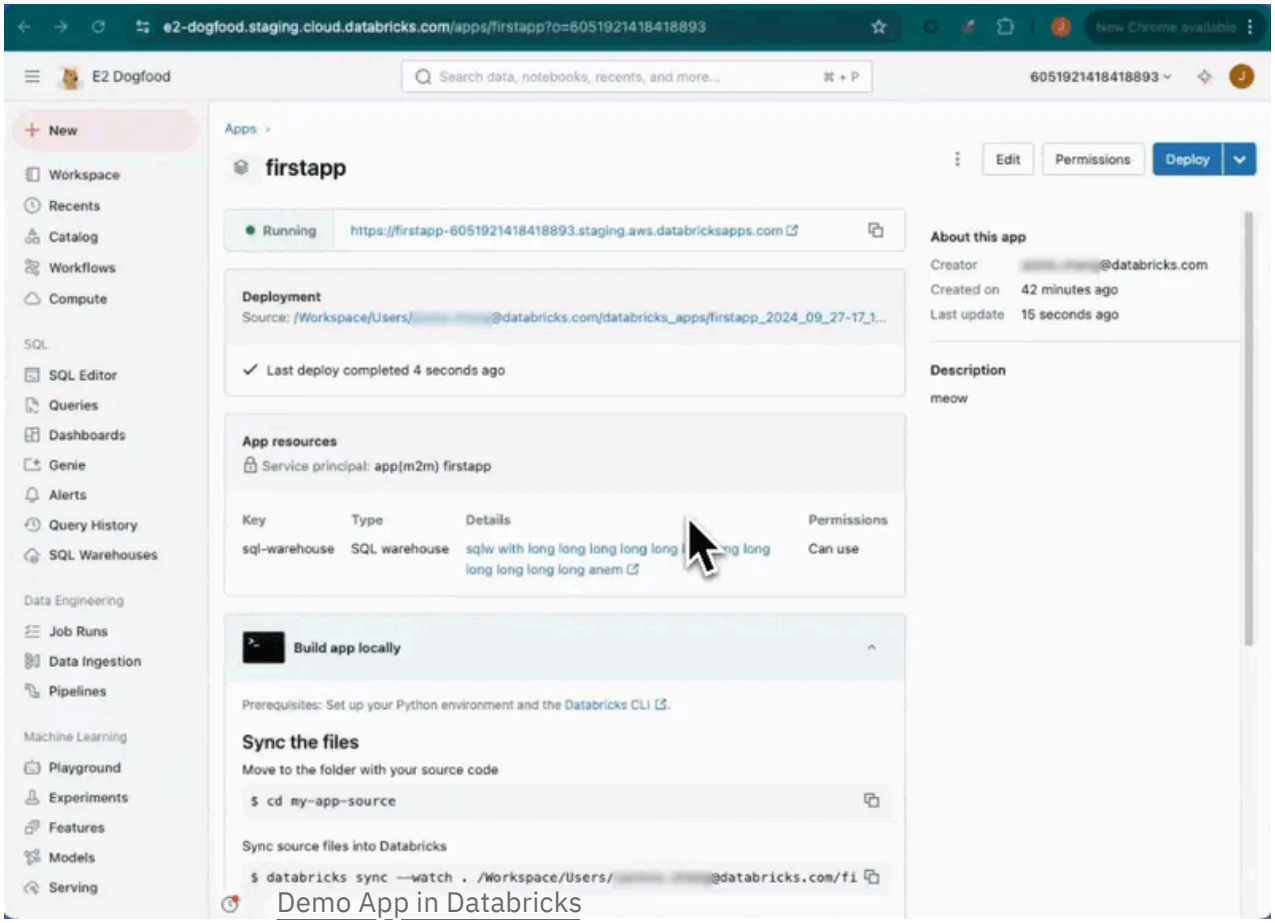
Sample Retail Revenue & Supply Chain Dashboard in Databricks

Using these dashboards, teams can explore data and share findings — all without leaving Databricks. It's a straightforward way to make data accessible, especially if ~~the reporting~~ needs are relatively simple and users are already familiar with the platform.

Databricks Apps

Another option is to build custom applications within Databricks. This could involve using libraries like Streamlit or Plotly to create interactive data applications. This approach offers a high degree of customization and interactivity. It's particularly useful when we need to build specialized

analytics applications or data exploration tools where dashboards are not enough.



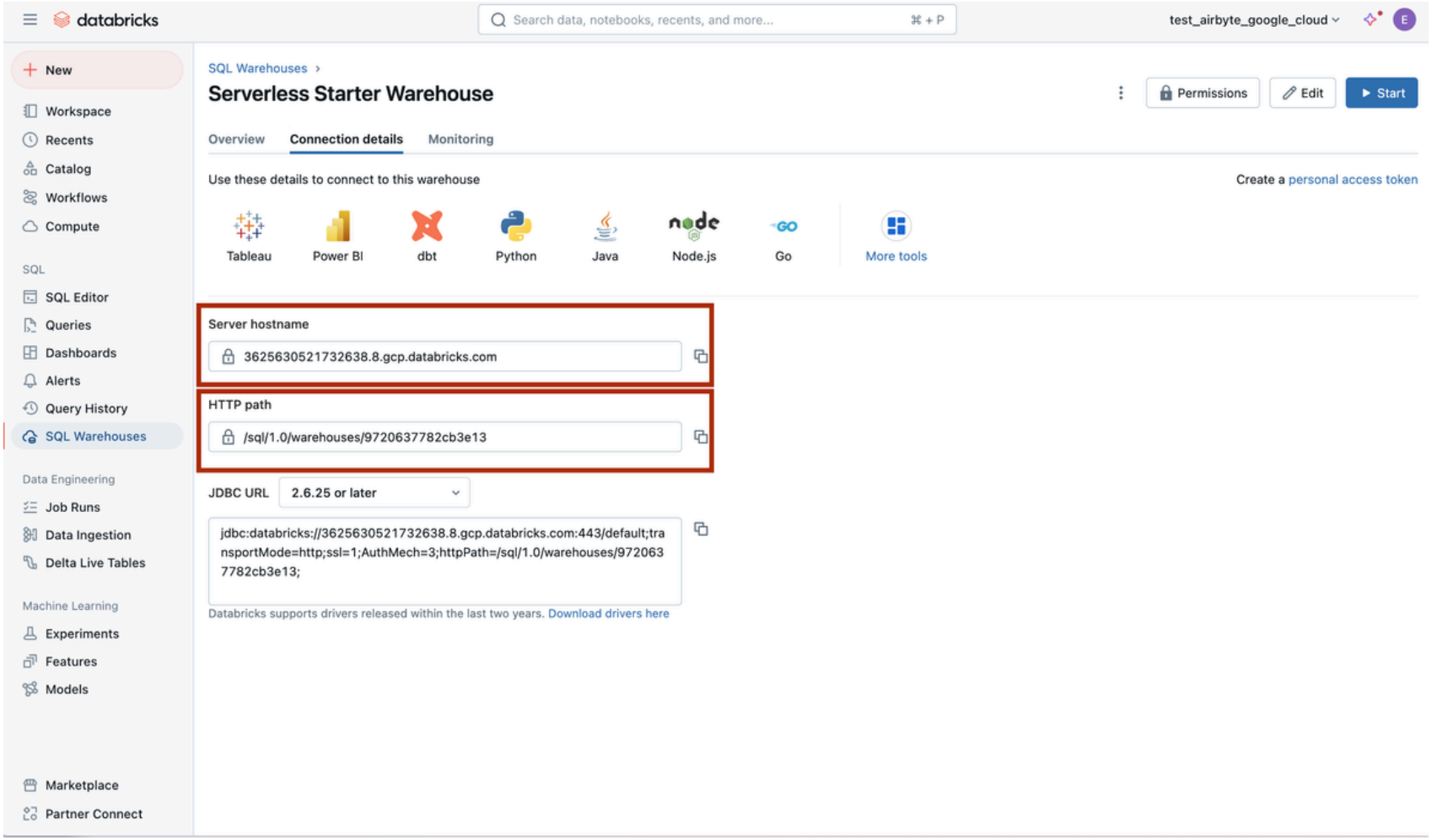
Demo App in Databricks

The benefit of both Dashboards and Apps is that they handle one of the most dreaded parts of developing an application which is governance, security and hosting. By being directly integrated into the Databricks ecosystem, we can take advantage of the systems we already have in place without requiring additional infrastructure or governance layers. Nonetheless, as Apps are still a relatively new feature. We still have to test their viability in relation to value/cost.

Integration with External Applications

If the built-in tools don’t meet all requirements, or if the users prefer other applications, Databricks provides several ways to integrate with external tools while keeping data within the platform. The main choice here is whether we want to use one of Databricks’s compute option (and implicitly it’s governance system) to access the data or if we want to access the raw data from our cloud provider.

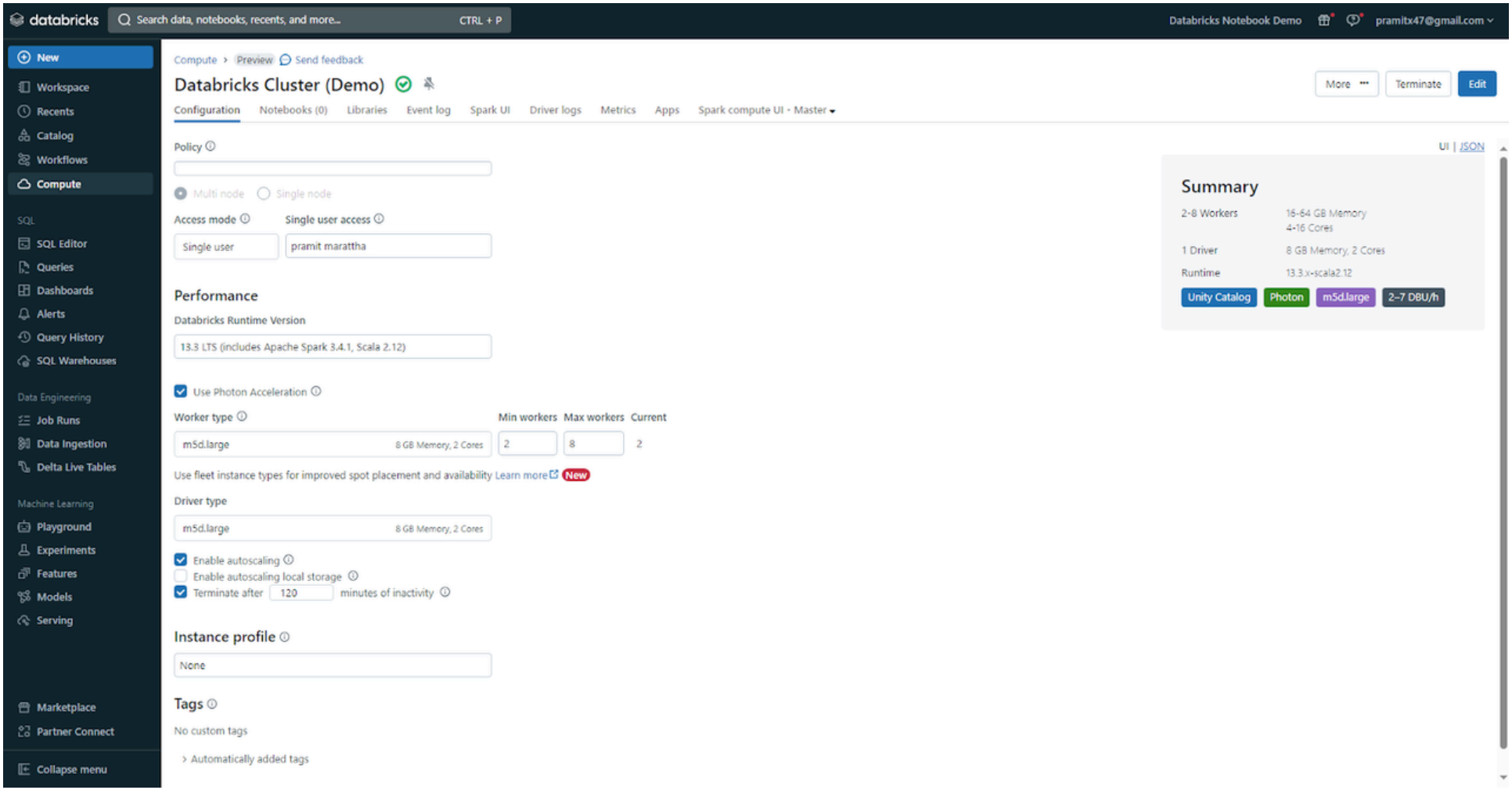
Using SQL Warehouses or All-Purpose Clusters Databricks offers SQL Warehouses (formerly SQL Endpoints) and All- Purpose Clusters that serve as compute engines for executing queries against the data. SQL Warehouses are optimized for SQL workloads and come with Photon, Databricks’ high-performance query engine, enabled by default.



SQL Warehouse Connection Details

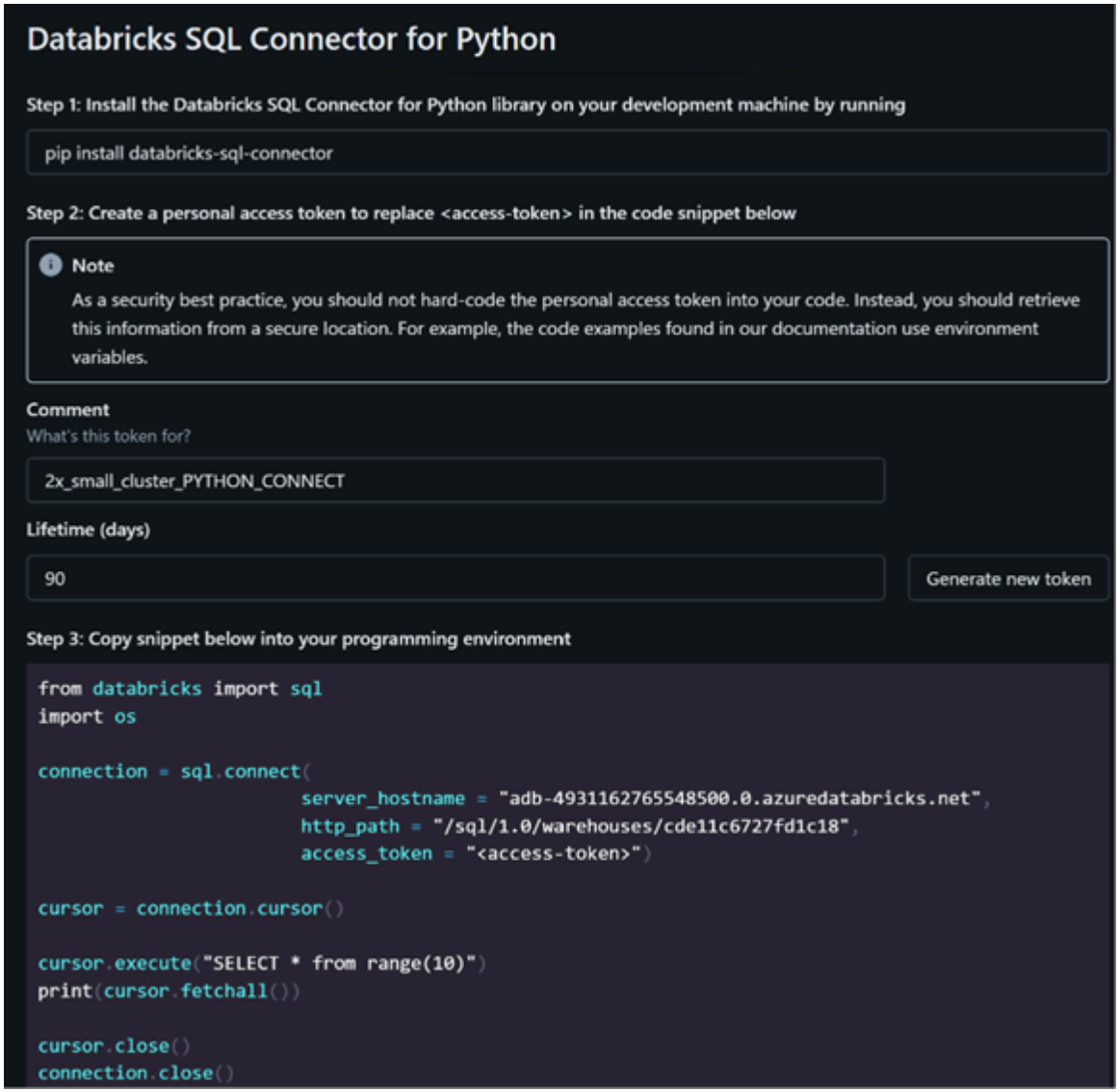
Both support JDBC and ODBC connections, making them compatible with many business intelligence tools (for these we can also just use Partner Connect for a simplified connection process) like Power BI, Tableau, but also

with custom applications. This means users can continue using their preferred tools while accessing data directly from Databricks.



All-Purpose Cluster Connection Details

For Python applications, Databricks also provides a Python SQL Connector.



This simplifies the process of connecting to SQL Warehouses from Python code, enabling developers to execute queries and retrieve data within their applications. Alternatively, we can use the Databricks REST API to execute SQL statements and fetch the results.

```
databricks api post /api/2.0/sql/statements \
--profile <profile-name> \
--json '{
  "warehouse_id": "'"$DATABRICKS_SQL_WAREHOUSE_ID"'",
  "catalog": "samples",
  "schema": "tpch",
  "statement": "SELECT l_orderkey, l_extendedprice, l_shipdate FROM lineitem WHERE l_extendedprice > :extended
  "parameters": [
    { "name": "extended_price", "value": "60000", "type": "DECIMAL(18,2)" },
    { "name": "ship_date", "value": "1995-01-01", "type": "DATE" },
    { "name": "row_limit", "value": "2", "type": "INT" }
  ]
}' \
> 'sql-execution-response.json' \
&& jq . 'sql-execution-response.json' \
&& export SQL_STATEMENT_ID=$(jq -r .statement_id 'sql-execution-response.json') \
&& export NEXT_CHUNK_INTERNAL_LINK=$(jq -r .result.next_chunk_internal_link 'sql-execution-response.json') \
&& echo SQL_STATEMENT_ID=$SQL_STATEMENT_ID \
&& echo NEXT_CHUNK_INTERNAL_LINK=$NEXT_CHUNK_INTERNAL_LINK
```

Example SQL Statement via REST API

Understanding the Difference Between Using the REST API and the Python Connector

When integrating programmatically, we might wonder whether to use the Databricks REST API or connect directly via Python using the Python SQL Connector. The choice between these two methods revolves around how we interact with the Databricks platform and the level of abstraction each provides.

Using the REST API

Using the REST API involves making HTTP requests to Databricks endpoints to execute SQL statements. This method is language-agnostic, accessible from any programming language that can make HTTP requests, making it flexible for integrating with various systems and applications.

However, the REST API often requires managing asynchronous operations. We might need to poll for the status of a query execution and handle responses accordingly. We'll also need to manually handle aspects like constructing HTTP requests, parsing JSON responses, managing authentication tokens, and error handling.

The REST API is suitable when:

- Working in an environment where a direct database connection isn't possible.
- In need for a lightweight method to execute queries without setting up database drivers.
- We're integrating with systems that rely on HTTP protocols.

Connecting Directly via Python

Connecting directly via Python involves using the Databricks SQL Connector for Python. From the official pypi documentation:

> The Databricks SQL Connector for Python allows you to develop Python applications that connect to Databricks clusters and SQL warehouses. It is a Thrift- based client with no dependencies on ODBC or JDBC. It conforms to the Python DB API 2.0 specification and exposes a SQLAlchemy dialect for use with tools like pandas and alembic which use SQLAlchemy to execute DDL. Use pip install dependencies.
>
> databricks-sql-connector[sqlalchemy] to install with SQLAlchemy's pip install databricks-sql-connector[alembic] will install alembic's dependencies.
>
> This connector uses Arrow as the data-exchange format, and supports APIs to directly fetch Arrow tables. Arrow tables are wrapped in the ArrowQueue class to provide a natural API to get several rows at a time.

```
pip install databricks-sql-connector[sqlalchemy]
```

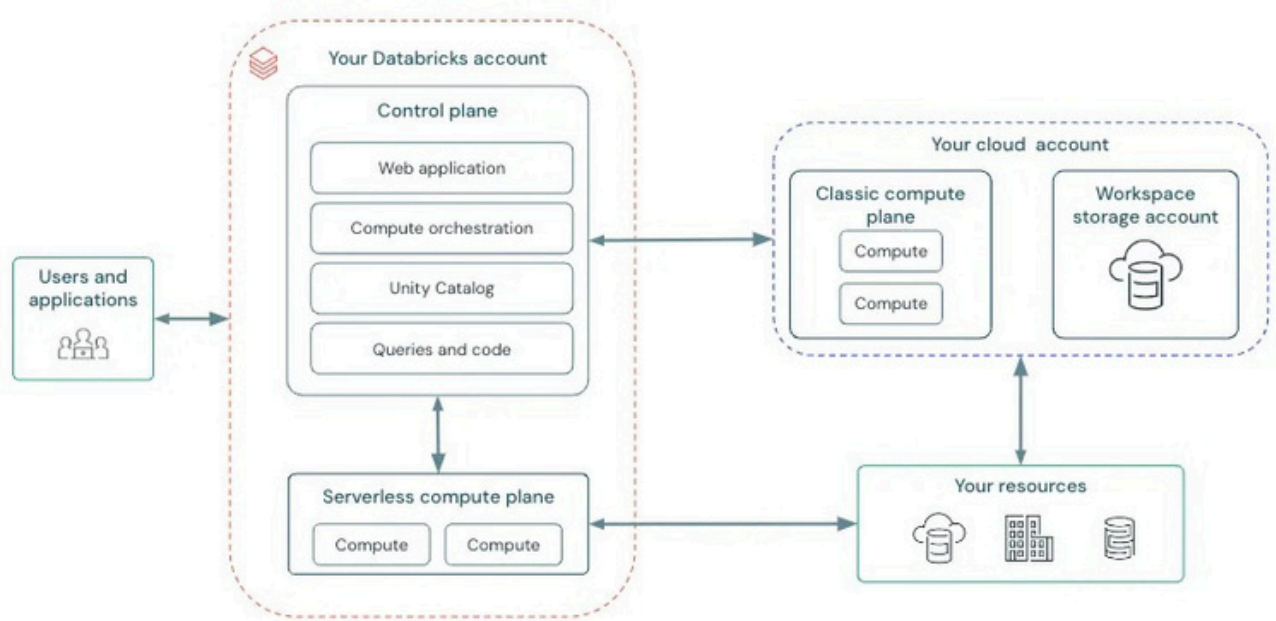
By using the Python SQL Connector, we can establish a direct connection to Databricks using a connection string that includes details like the server hostname, HTTP path, and authentication credentials. This method allows us to use standard database API interfaces, providing methods for executing queries, fetching results, and handling transactions. Query execution is typically synchronous, and we receive results directly in the Python application as Python objects, such as lists or dictionaries.

Direct Python connections are ideal when:

- We're developing an application or script in Python that needs to interact with the SQL warehouse.
- We prefer working with database connections using familiar Python libraries.
- We want to leverage Python's data processing capabilities alongside queries.

Understanding the Networking Differences Between SQL Warehouses and All-Purpose Clusters

The choice between SQL Warehouses and All-Purpose Clusters also involves networking considerations. In the Databricks architecture, serverless resources like SQL Warehouses operate within the Databricks account, while classic compute resources, such as All-Purpose Clusters, run within a customer's cloud account.



Azure Databricks Architecture Overview

For large organizations, establishing private connections to and from Databricks is essential to maintain security and compliance. However, the setup process differs depending on the type of resource:

- Serverless Resources (SQL Warehouses): Because serverless resources are managed within the Databricks account, creating private connections requires setting up Network Connectivity Configurations (NCCs) and using services like Private Link to establish private access paths between Databricks and external applications. This setup enables data access without routing over the public internet, but it introduces additional setup complexity. External applications must also be configured to connect through these private paths, which can impact your choice of tools depending on whether they support such network configurations.
- All-Purpose Clusters: Operating within the a customer's cloud environment, these clusters enable more flexible, customized networking setups, such as configuring VPCs, private subnets, and security groups. This gives organizations full control over ingress and egress traffic, often making it easier to integrate with other resources already in the cloud environment without requiring additional services like NCCs.

One important point to mention here is that serverless resources are still created in the same region as the workspace, providing regional consistency: "Azure Databricks creates a serverless compute plane in the same Azure region as your workspace's classic compute plane." (Link)

Reading the Raw Parquet Files / Delta Tables In addition to using Databricks' compute resources, another option is to directly access the raw parquet files / delta tables stored in cloud storage with popular libraries like DuckDB or delta-rs. This method bypasses Databricks' governance layer, which may be appropriate in specific cases where governance requirements are minimal, or direct access is necessary.

Reasons for Accessing Raw Files Directly

There are scenarios where direct access to raw files can provide some benef its:

- External Tool Integration: When integrating with external systems direct access may simplify data sharing without the need for Databricks compute resources.
- Cost Optimization: If the requirement is simply to retrieve data without additional computation, this approach can reduce costs associated with Databricks clusters.

Key Considerations

1. Authentication and Permissions: Accessing raw files requires configuring permissions directly on the cloud storage, rather than relying on Databricks' role-based access controls.

2. Delta Format Compatibility: If data is stored in Delta format, we have to ensure that the external tool can handle the Delta Lake format and metadata, as not all features are available outside of Databricks (e.g. DuckDB recently added native support for Delta Lake in collaboration with Databricks — Demo on YouTube)

3. Data Organization and Structure: Without the structure provided by Unity Catalog, it becomes essential to manage and maintain the directory paths and file organization manually.

Appropriate Use Cases

Direct access to raw parquet files may be beneficial when:

- Data Governance Requirements are Minimal: For cases where centralized governance isn't essential, this approach offers a straightforward alternative.
- Lightweight Data Access Needs: When performing simple read operations that don't require Databricks' compute capabilities.

In summary, while bypassing Databricks' governance layer may not always align with best practices, directly accessing raw parquet files provides flexibility and cost-efficiency in certain data workflows.

External Access to Data in Unity Catalog

Until now we have discussed:

- How to use Databricks compute options and Unity Catalog governance to access data How to access the raw data / delta tables generated inside of Databricks and stored in
- cloud storage with external engines (WITHOUT Unity Catalog)

In this chapter we will talk about how to use external processing engines to retrieve data WITH Unity Catalog, maintaining governance and compliance.

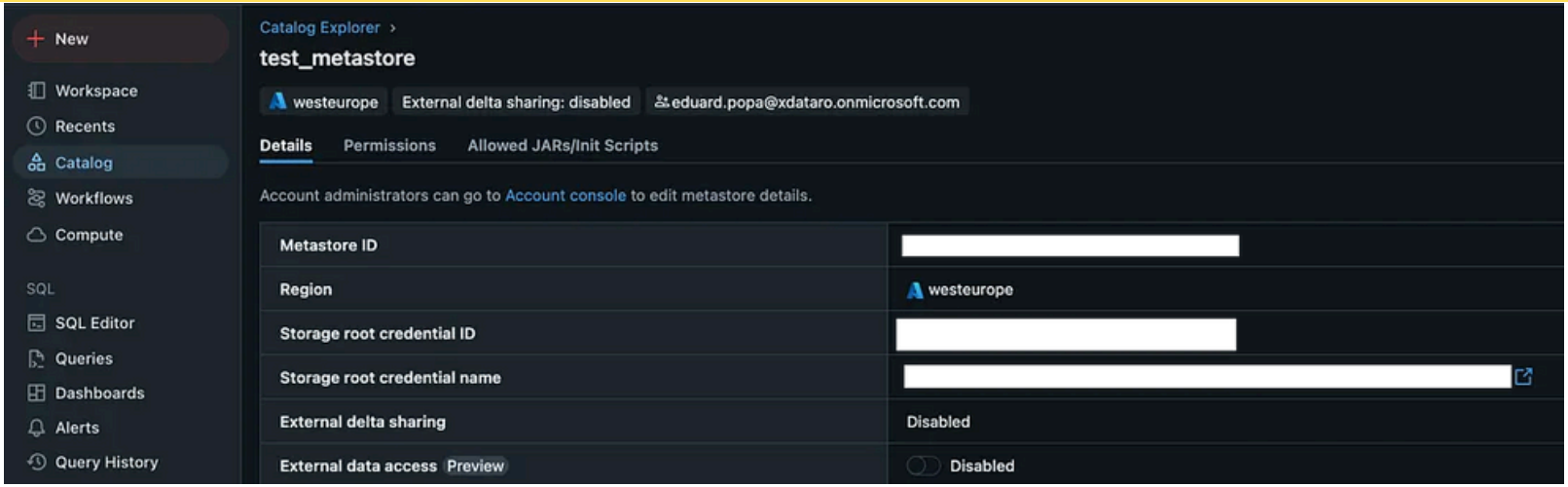
Overview of Credential Vending and Granting External Engine Access For external engines — such as Apache Spark, Trino, Microsoft Fabric, and Iceberg API clients — to access data stored within Unity Catalog, Azure Databricks provides a feature known as credential vending. This feature allows a Databricks principal (a user, group, or service principal) to request a

short-lived credential that enables external data access. This credential contains a temporary access token and a cloud storage URL, allowing the external engine to read from (and potentially write to) Unity Catalog- managed storage locations.

Credential vending ensures that all access remains governed, time-bound, and restricted to authorized principals.

Requirements for Enabling External Data Access in Unity Catalog Metastore Configuration:

- The Unity Catalog metastore must explicitly have External Data Access enabled. This setting is off by default to prevent unintended data exposure. Only a metastore admin can enable this setting in the Catalog pane under Metastore settings.



Metastore Configuration

Networking:

- Databricks Workspace Access: The external engine must have network access to the Databricks workspace where Unity Catalog is managed. This could involve configurations like IP access lists or Azure Private Link.
- Cloud Storage Access: Temporary credentials allow external engines to access cloud storage where Unity Catalog tables are stored. We have to configure cloud storage access controls (firewalls, access control lists) to permit only authorized external engines.

Permission Requirements:

- The requesting principal must have the EXTERNAL USE SCHEMA privilege on the schema containing the table.
- They must also have SELECT privileges on the table, along with USE CATALOG and USE SCHEMA privileges for the catalog and schema that hold the table.
- Importantly, schema owners do not have the EXTERNAL USE SCHEMA privilege by default, and it must be explicitly granted by the catalog owner to avoid unauthorized data access.

Table Type Limitations:

- During the Public Preview phase, only tables are eligible for external access — views, materialized views, Delta Live Tables streaming tables, and other advanced table types like Lakehouse federated tables are unsupported.
- External tables support both read and write access, while managed tables are read-only for external engines.

Requesting Temporary Credentials for External Data Access Once the setup is complete, authorized users can request a temporary credential using the Unity Catalog REST API. This credential grants

temporary access to the specified table, allowing the external engine to perform read or write operations as permitted.

Here's an example of how to request a temporary credential using a REST API call:

```
curl -X POST -H "Authentication: Bearer $OAUTH_TOKEN" \ https://<workspace-instance>/api/2.1/unity-catalog/temporary-table-credentials \ -d '{"table_id": "<table-identifier>", "operation_name": "<READ|READ_WRITE>"}'
```

This API request will generate a temporary access token and cloud storage URL, which can be provided to the external engine. Only tables with HAS_DIRECT_EXTERNAL_ENGINE_READ_SUPPORT or HAS_DIRECT_EXTERNAL_ENGINE_WRITE_SUPPORT (identified by the include_manifest_capabilities option in the ListTables API) are eligible for this credential vending process.

Key Considerations for External Data Access

1. Security and Compliance: Credential vending ensures that access to data remains temporary and controlled, with permissions limited to authorized entities.

2. Governance Impact: Although external engines can access Unity Catalog- governed data, it's essential to recognize that certain governance features, such as column-level lineage or table lifecycle management, are only available when data is processed within Databricks.

3. Temporary Credential Management: As temporary credentials are short- lived, external systems may require credential renewal mechanisms for long-running workloads or automated data integrations.

4. Cost Optimization: By using external engines for specific workloads, organizations can optimize costs by reducing reliance on Databricks clusters for processing needs.

In summary, enabling external access to Unity Catalog data provides flexibility for multi-platform integration while preserving governance.

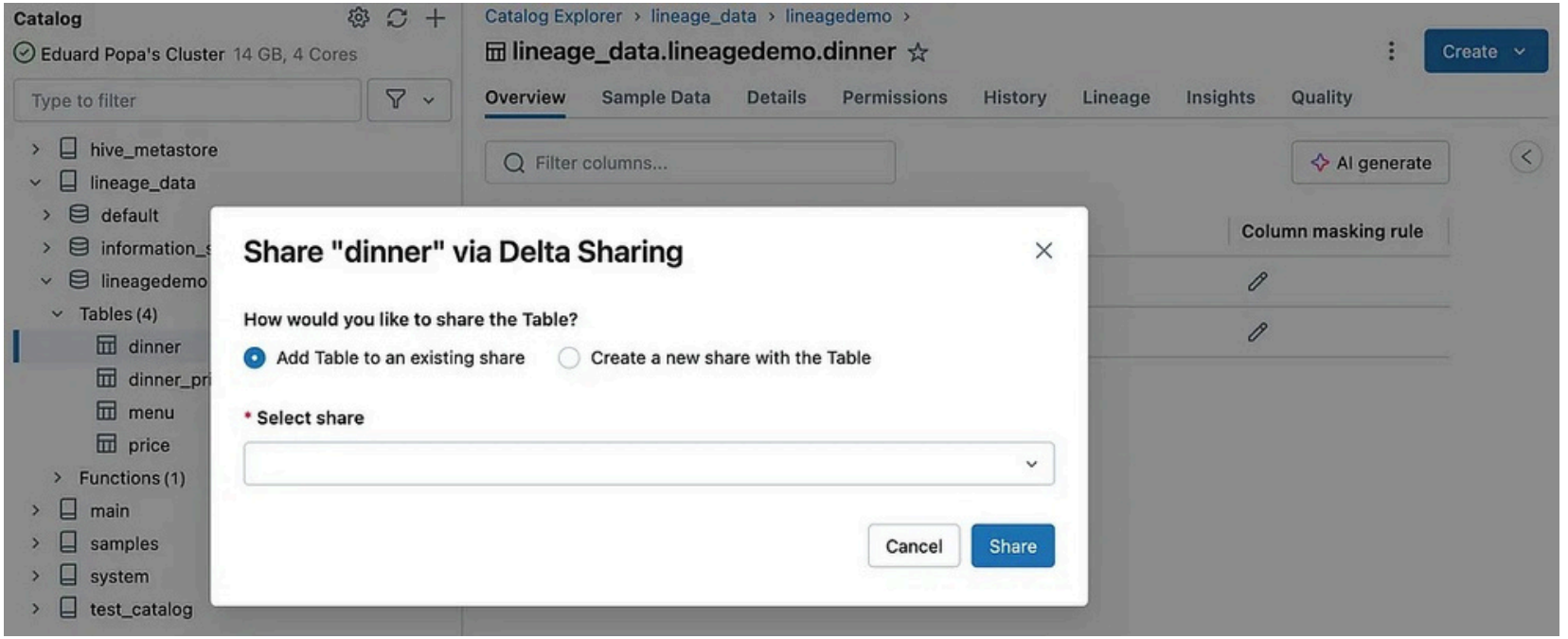
Delta Sharing

Delta Sharing is a protocol developed by Databricks that facilitates secure, scalable, and platform-agnostic data sharing. It's designed to accommodate various data platforms and enable organizations to share data and AI assets across cloud environments and with external partners. This feature also

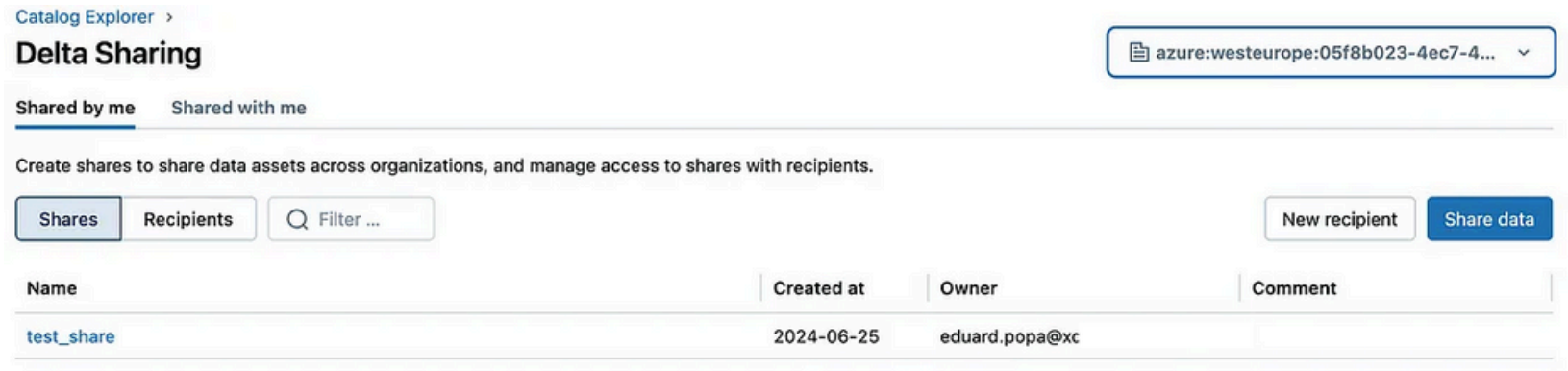
serves as the foundation for the Databricks Marketplace and in this chapter we will take a look at how Delta Sharing works, its key components, and its practical applications.

How Delta Sharing Works

Delta Sharing is structured around an open protocol that allows data exchange across platforms, even when the recipient doesn't use Databricks. It leverages a Delta Sharing server to grant recipients secure, read-only access to data, supporting both on-Databricks and off-Databricks data exchanges. Even though Delta Sharing is open source, self-hosting a Delta Sharing Server can be quite cumbersome. Now, however, it can be easily configured through the UI.



Delta Sharing Unity Catalog



Delta Sharing Shares in Unity Catalog

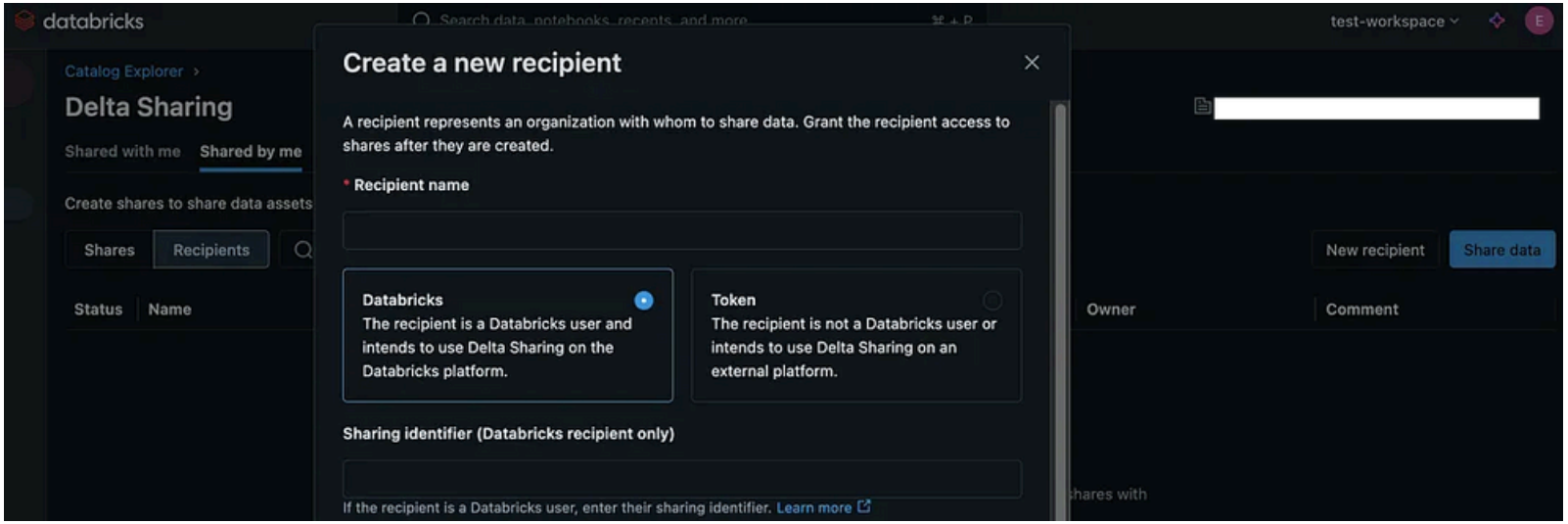
Delta Sharing is available in three distinct modes:

- Databricks-to-Databricks Sharing: Enables sharing between Databricks workspaces that are both Unity Catalog-enabled. This mode provides integrated data governance, auditing, usage tracking, and additional assets, such as Unity Catalog volumes, notebooks, and AI models.
- Open Sharing Protocol: Allows sharing of data with users on any computing platform, even if they aren't on Databricks. Providers can use Unity Catalog for data management while extending access to users on different platforms.

- Customer-Managed Open-Source Server: For organizations that require even broader compatibility, Delta Sharing can be deployed as an open- source project to support data sharing from virtually any platform to any platform.

Key Concepts in Delta Sharing: Shares, Providers, and Recipients Delta Sharing revolves around three core concepts:

- Shares: A share is a read-only collection of Delta tables or partitions that the provider makes accessible to recipients. In Databricks-to-Databricks sharing, shares can also include views, Unity Catalog volumes, models, and notebooks. These shares are registered as securable objects within Unity Catalog, and providers can add or remove data assets and manage recipient access at any time.
- Providers: A provider is an entity that offers data to recipients. Providers need at least one Unity Catalog-enabled workspace to manage shares and recipients. In the context of Unity Catalog, a provider is also a securable object representing the data-sharing organization.
- Recipients: Recipients are the entities that access data shared by providers. In Unity Catalog, recipients are securable objects associated with credentials that grant access to one or more shares. Providers must explicitly define recipients for each Unity Catalog metastore, making it possible to manage permissions across multiple shares.



Creating Delta Sharing Recipient

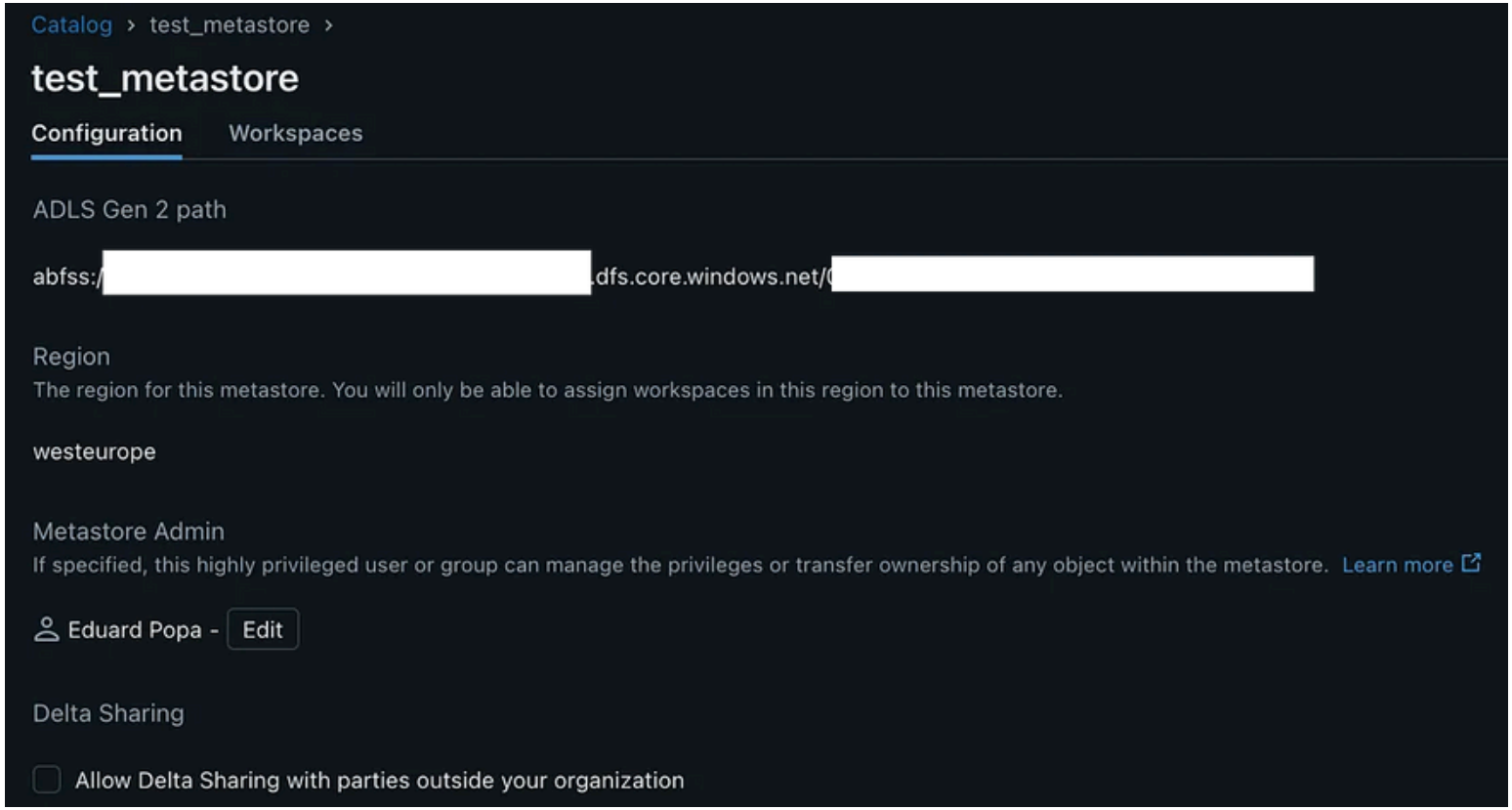
Open Sharing vs. Databricks-to-Databricks Sharing The choice between open sharing and Databricks-to-Databricks sharing depends on the recipient's environment and the assets being shared:

- Open Sharing: This method allows providers to share data with any user or platform, using token-based credentials to secure the data. Recipients can access the data in popular tools like Power BI, Apache Spark, and Pandas.

Databricks-to-Databricks Sharing: Designed for recipients on Unity Catalog-enabled Databricks workspaces, this approach doesn’t require tokens. Instead, access is managed directly through Databricks, providing a fully integrated sharing experience. This method also supports sharing additional Databricks assets, such as views, volumes, models, and notebooks, which are unavailable in open sharing.

How Providers Set Up Delta Sharing Setting up Delta Sharing involves configuring the Unity Catalog metastore to enable data sharing and defining shares and recipients:

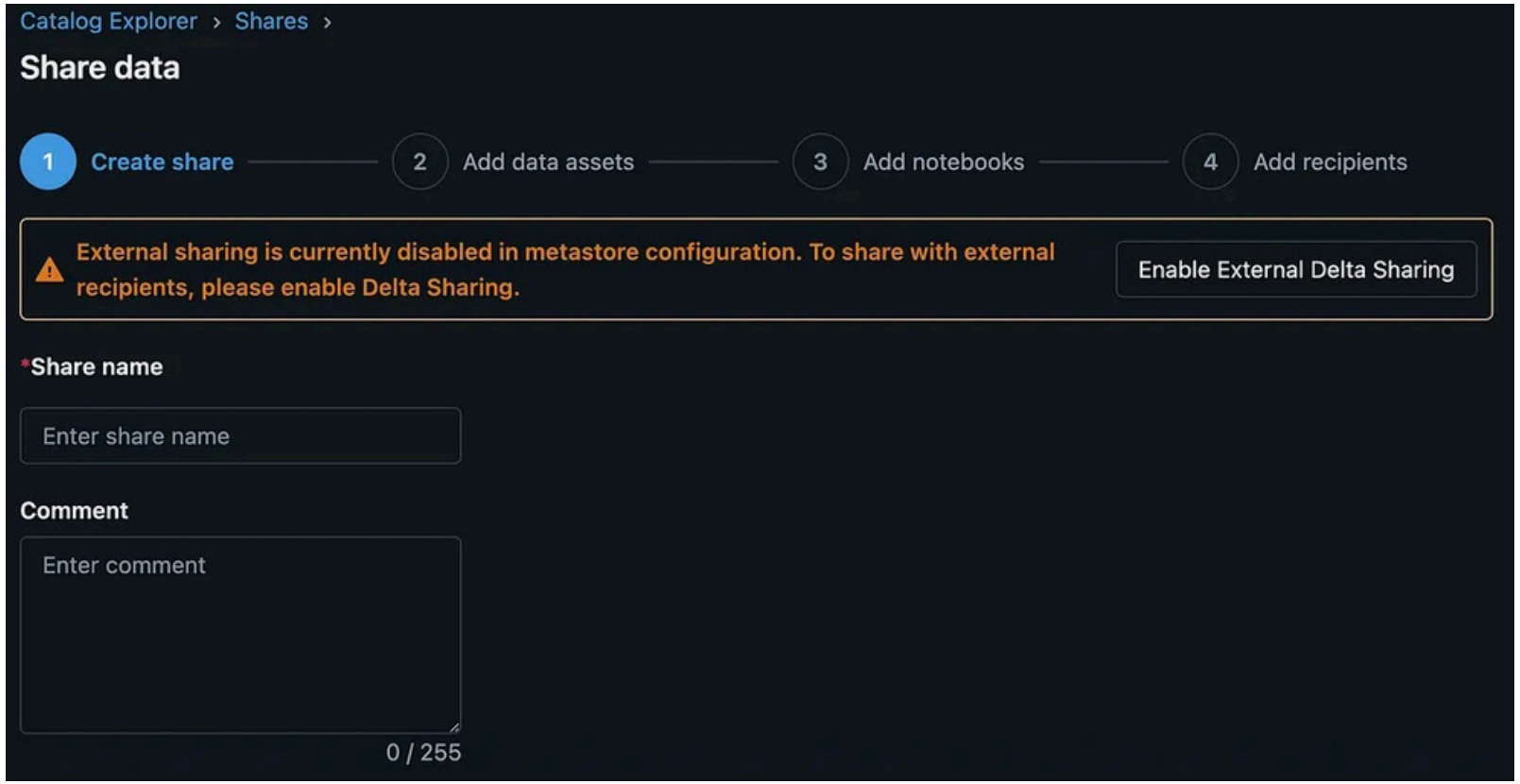
- Enable Delta Sharing on the Metastore: A Databricks account admin enables Delta Sharing within the Unity Catalog metastore where the data is stored.



Enabling Delta Sharing on the Metastore

- Create and Configure Shares: Providers can add Delta or Parquet tables to shares, along with views, volumes, models, and notebooks if using Databricks-to-Databricks sharing.
- Define Recipients: For each metastore, providers define recipients who are granted access to specific shares. In open sharing, token-based credentials are generated for the recipient, while Databricks-to- Databricks recipients are identified through a sharing identifier.

- Grant Access: Providers can manage access at any time, assigning or revoking permissions for recipients on each share.
- Distribute Access Information: In open sharing, providers send recipients an activation link or access token via a secure channel. For Databricks-to-Databricks sharing, recipients automatically gain access within their Databricks workspace.



Setting Up Delta Sharing

Recipient Access and Usage

Recipients access Delta Sharing assets in a read-only format. Access methods differ based on the sharing protocol:

- Open Sharing: Recipients authenticate using token-based credentials and can connect through various tools and languages, such as Spark, Pandas, and Power BI.
- Databricks-to-Databricks Sharing: Recipients access the data directly in their Unity Catalog-enabled workspace. This approach also enables recipients to grant or deny access to other users within their organization, streamlining access management.

Changes made by the provider to shared data assets are propagated in near real-time to the recipient, ensuring consistency across shared data.

Monitoring and Auditing Shared Data Access

Delta Sharing incorporates auditing features that allow providers and recipients to track data-sharing activity. Azure Databricks audit logs and system tables offer insights into share creation, recipient management, and data access.

Advanced Features in Databricks-to-Databricks Sharing

The Databricks-to-Databricks protocol provides several advanced sharing options, including:

- Sharing Volumes and Models: Unity Catalog volumes and models can be shared along with tables and views, making Databricks-to-Databricks sharing ideal for complex data and AI use cases.
- Notebook Sharing: Notebooks can be shared in read-only format, allowing recipients to clone and modify copies as needed.
- Dynamic View Sharing: Row and column access can be restricted at the view level, enabling data governance based on recipient properties.
- Streaming Support: Recipients can use shared tables with history as streaming sources, facilitating incremental processing with low latency.

Delta Sharing in Practice: Use Cases and Benefits

Delta Sharing's flexibility and security make it well-suited for a range of data-sharing scenarios:

- Cross-Organization Data Collaboration: Organizations can securely exchange data with external partners, clients, or vendors, regardless of the computing platform.

- Multi-Cloud and Hybrid Architectures: Delta Sharing supports various cloud and on-premises data architectures, allowing organizations to integrate data from multiple environments.
- Cost-Efficiency: By eliminating the need for data replication, Delta Sharing minimizes data egress costs, providing a cost-effective solution for regional data exchanges.

Key Considerations

- Security and Compliance: Delta Sharing's token-based security, credential revocation, and activity auditing provide a strong security foundation.
- Egress Costs: While Delta Sharing avoids in-region egress fees, cross- region and cross-cloud data sharing may still incur additional costs.

  Providers should monitor and manage these costs as necessary.
- Feature Limitations: Some advanced Delta Lake features and object types (such as materialized views) are not supported in open sharing. Databricks-to-Databricks sharing supports these capabilities, but open sharing has some restrictions.

Follow for more content like this Azure Cloud for Data Engineering

Ganesh R

Azure Data Engineer

https://www.linkedin.com/in/rganesh0203/

In conclusion, making Databricks data accessible to end users is about understanding all the available options and considering both user needs and the technical framework. For teams already invested in Databricks, the in- platform options — such as Databricks dashboards and apps — provide a straightforward way to deliver data insights directly into users' hands. However, if your organization relies on external tools like Power BI, or if you need more control over where the data resides, the external integration options allow you to bridge Databricks data with your existing infrastructure.

Lastly, it's essential to keep in mind your organization's security and governance requirements, ensuring data access is both secure and compliant. I hope you found this article helpful and look forward to seeing you in the next one!

. . .