

Create and organize objects in Unity Catalog

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/1-introduction>

Introduction

Completed



Your organization needs to manage data across development, testing, and production environments while enforcing clear security boundaries. Different teams require access to different datasets, and business users need a way to discover and query data without writing complex SQL. **Unity Catalog** provides a **three-layer namespace**—catalogs, schemas, and tables—that lets you organize data assets with precision while maintaining centralized governance.

Creating and organizing Unity Catalog objects establishes the foundation for your data platform. You start by designing **catalogs** that separate environments or business domains, then organize **schemas** within those catalogs to group related datasets. Within schemas, you create **tables** for structured data and **volumes** for files, applying consistent naming conventions that make your data discoverable. Advanced features like **foreign catalogs** connect external databases, while **AI/BI Genie instructions** help business users interact with your data using natural language.

By the end of this module, you'll be able to design and implement a comprehensive Unity Catalog structure that scales with your organization's needs while maintaining clear governance boundaries.

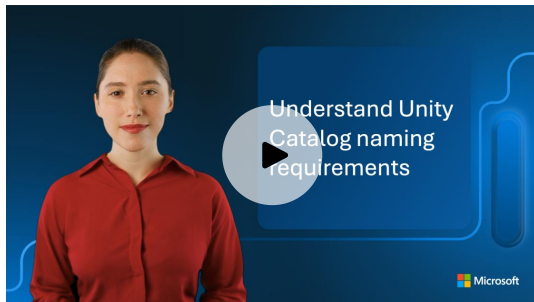
2. Apply naming conventions

Apply naming conventions

Completed

Naming conventions are fundamental to effective data governance in Azure Databricks Unity Catalog. When you apply consistent naming patterns across catalogs, schemas, and tables, you enable your team to quickly understand data organization, enforce access controls, and maintain compliance with organizational standards.

Understand Unity Catalog naming requirements



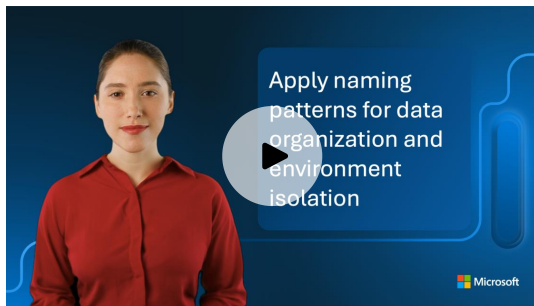
Unity Catalog imposes specific technical constraints on object names that you must follow when designing your naming conventions. These constraints ensure compatibility across the platform while supporting international characters and diverse organizational needs.

All Unity Catalog object names are limited to 255 characters and are stored in lowercase, regardless of how you create them. This means `SalesData`, `salesdata`, and `SALESDATA` all resolve to the same object. When you reference objects in queries, Unity Catalog performs case-insensitive matching, simplifying query syntax while maintaining consistency.

Certain characters are prohibited in object names. You can't use periods (`.`), spaces, forward slashes (`/`), or ASCII control characters. However, you can include hyphens and other special characters by enclosing the name in backticks. For example, ``sales-data`` is valid, while `sales.data` isn't because periods serve as namespace separators in the three-level hierarchy.

Column names follow different rules. Unity Catalog preserves the casing you specify for column names, though queries remain case-insensitive. You can use special characters in column names by enclosing them in backticks, giving you flexibility for descriptive field names while maintaining query simplicity.

Apply naming patterns for data organization



Effective naming patterns balance descriptiveness with brevity, helping your team understand data purpose without creating unwieldy identifiers. The three-level namespace structure of Unity Catalog — `catalog.schema.table` — provides natural boundaries for applying meaningful conventions.

```
sales_data (Catalog)
├── bronze (Schema - raw data)
│   ├── raw_orders (Table)
│   └── raw_customers (Table)
├── silver (Schema - cleaned / validated)
│   ├── cleaned_orders (Table)
│   └── validated_customers (Table)
└── gold (Schema - aggregated / business-level)
    ├── sales_summary (Table)
    ├── customer_lifetime_value (Table)
    └── vw_customer_stats (View example: prefix views with vw_)
```

Use **lowercase with underscores** as your primary pattern. Names like `customer_data`, `sales_summary`, and `product_inventory` are immediately readable and comply with Unity Catalog requirements. Avoid camelCase or PascalCase, as Unity Catalog converts all names to lowercase anyway, potentially creating confusion between what you specify and what the system stores.

Choose **descriptive but concise names** that clearly indicate content without excessive detail. `sales_summary_gold` effectively communicates both the domain and data quality level. In contrast, `sales_final_report_v1_updated` includes unnecessary version information that belongs in metadata or table properties rather than the name itself.

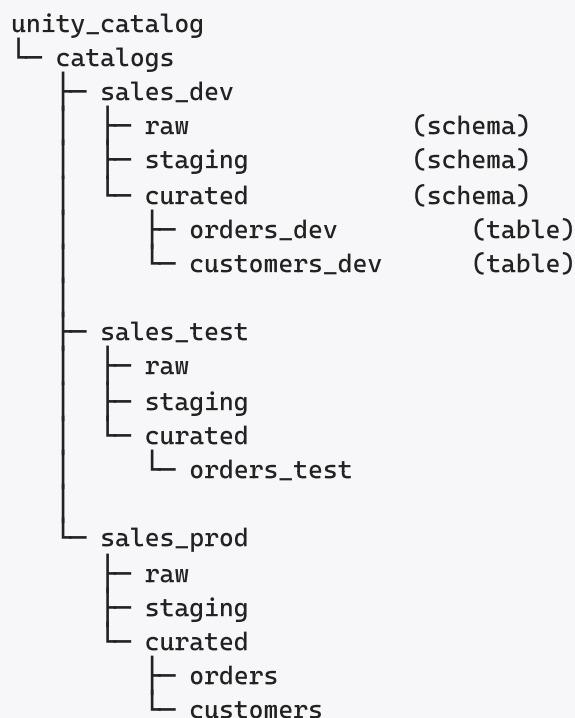
Align schema names with data layers when implementing a medallion architecture. Use **bronze** for raw ingested data, **silver** for cleaned and validated data, and **gold** for aggregated business-level data. This pattern makes the data transformation pipeline immediately apparent. For example, `sales_data.bronze.raw_orders` clearly indicates raw order data, while `sales_data.gold.monthly_revenue` represents a refined business metric.

Prefix specialized objects to distinguish them from base tables. Add `vw_` for views and materialized views (`vw_customer_stats`), and `tmp_` for temporary staging tables (`tmp_import_staging`). This convention helps you identify object types at a glance when browsing the catalog or writing queries.

Consider your **catalog naming strategy** based on whether you're organizing by business domain or environment. Domain-based catalogs like `sales_data`, `marketing_data`, and `finance_data` provide clear functional boundaries and align with data ownership models. Environment-based catalogs like `dev`, `test`, and `prod` separate workspaces by lifecycle stage, reducing risk of accidental production changes.

Design naming strategies for environment isolation

Environment isolation is critical for preventing accidental data modifications and maintaining clear separation between development, testing, and production workloads. Your naming strategy should make environment boundaries immediately obvious to anyone working with the data.



Use separate catalogs for each environment when strong isolation is your priority. Catalogs named `sales_dev`, `sales_test`, and `sales_prod` provide complete separation at the highest level of the namespace. With this approach, you grant permissions at the catalog level, making access control straightforward. Developers work exclusively in the dev catalog, while production queries reference only the `prod` catalog, eliminating cross-environment contamination risk.

Combine domain and environment in catalog names when you need both dimensions. A pattern like `{domain}_{environment}` produces catalogs such as `sales_prod`, `sales_dev`, `marketing_prod`, and `marketing_dev`. This strategy scales well across multiple business domains while maintaining environment separation. However, it increases the total number of catalogs, so you need robust catalog management practices.

Organize by layer within environment catalogs when environment separation happens at the workspace level rather than the catalog level. Within a `prod` catalog, you might have schemas named `sales_bronze`, `sales_silver`, and `sales_gold`, while a separate `dev` catalog contains

`sales_bronze`, `sales_silver`, and `sales_gold` for development work. This approach consolidates all production data under a single catalog, simplifying Unity Catalog governance while maintaining clear layer boundaries.

For **external sharing scenarios**, choose catalog names that are meaningful to external partners without exposing internal organizational structure. A catalog named `customer_analytics_shared` clearly indicates both purpose and sharing status, while `internal_sales_raw_data` signals data not intended for external access. This convention helps prevent accidental sharing of sensitive data.

External storage paths should mirror your naming conventions for consistency. Following a pattern like `abfss://container@storage.dfs.core.windows.net/env/layer/domain/table/` aligns storage organization with Unity Catalog structure. For example, `abfss://datalake@company.dfs.core.windows.net/prod/gold/sales/monthly_revenue/` makes the relationship between storage and catalog objects transparent.

Apply naming conventions for compute and development resources

Beyond data objects, consistent naming conventions for compute resources, development artifacts, and operational components help you navigate the Azure Databricks workspace efficiently and maintain clear ownership boundaries.

Clusters

Name clusters according to their purpose and environment to make resource allocation transparent. Use patterns like `{environment}_{domain}_cluster` to produce names such as `dev_sales_cluster`, `prod_etl_cluster`, and `ad-hoc_analysis_cluster`. This convention immediately identifies whether a cluster is for development experimentation or production workloads, helping you manage costs and apply appropriate access controls.

Jobs and pipelines

Structure job names using the pattern `job_{layer}_{purpose}` to align with your data transformation pipeline. Examples include `job_bronze_orders_ingestion`, `job_silver_orders_transformation`, and `job_gold_sales_aggregation`. This naming pattern makes dependencies between jobs immediately visible and helps you trace data lineage across the medallion architecture.

For Lakeflow Spark Declarative Pipelines pipelines, use the prefix `pipe_` followed by the data domain or purpose: `pipe_orders_processing`, `pipe_customer_data_cleaning`.

Name streaming pipelines to include both source and target, following patterns like `stream_{source}_to_{target}`. Examples such as `stream_kafka_to_bronze` and `stream_iot_sensor_data` make data flow explicit without requiring pipeline documentation. This convention is especially valuable when managing multiple concurrent streaming workloads.

Notebooks and repositories

Organize notebooks in folder hierarchies aligned with your project structure and data layers. A recommended pattern mirrors the medallion architecture:

```
/Repos/  
  sales_project/  
    bronze/  
      ingestion_orders/  
    silver/  
      transformation_orders/  
    gold/  
      sales_aggregation/
```

Individual notebook names should reflect their purpose and layer using patterns like `notebook_{layer}_{purpose}`. Examples include `notebook_bronze_orders_ingestion`, `notebook_silver_orders_transformation`, and `notebook_gold_sales_reporting`. This creates consistency between notebook names and the job names that execute them.

Repository names should identify the project or domain they contain. Use patterns like `repo_{project_name}` to produce names such as `repo_sales_project` and `repo_customer_analytics`. When you integrate with Git, this naming convention helps map Databricks repositories to their corresponding remote repositories.

Dashboards and monitoring

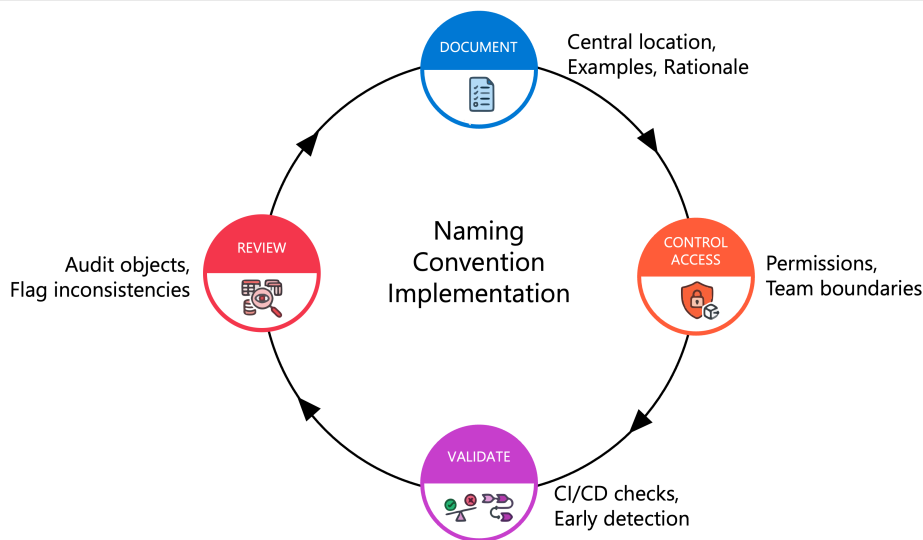
Name SQL dashboards to reflect their business purpose using patterns like `dashboard_{domain}_{purpose}`. Examples include `dashboard_sales_performance` and `dashboard_customer_trends`. This convention helps business users quickly locate relevant dashboards and distinguishes operational dashboards from exploratory analyses.

Individual SQL queries should be named descriptively using patterns like `query_{purpose}`. Names such as `query_monthly_sales_report` and `query_customer_retention_analysis` make query libraries searchable and help teams identify reusable queries for similar reporting needs.

Implement conventions across your organization

Successful implementation of naming conventions requires more than technical patterns—you need organizational alignment, documentation, and enforcement mechanisms that make conventions easy

to follow and hard to bypass.



Start by **documenting your conventions** in a central location accessible to all data team members. Include specific examples, decision criteria, and rationale for each pattern. A documented standard like "Use `{domain}_{layer}` for schema names, where domain is the business area and layer is bronze/silver/gold" removes ambiguity. Provide examples of both correct and incorrect names to illustrate the pattern clearly.

Control access through Unity Catalog permissions to limit who can create objects in specific catalogs and schemas. While you can't enforce naming patterns through permissions alone, restricting `CREATE TABLE` and `CREATE SCHEMA` privileges to specific teams helps maintain organizational boundaries. For example, grant the marketing team permissions only to the marketing catalog, reducing the risk of objects being created in the wrong location.

Validate names during CI/CD pipelines when deploying data assets through automation. Include checks in your deployment scripts that reject names violating your standards before objects are created. This automated validation catches errors early and maintains consistency across all environments.

Review existing objects regularly to identify naming inconsistencies and plan remediation. Use Unity Catalog's information schema to query object names and flag those that don't match your patterns. For newly identified legacy objects, decide whether to rename them for consistency or document them as exceptions with justification.

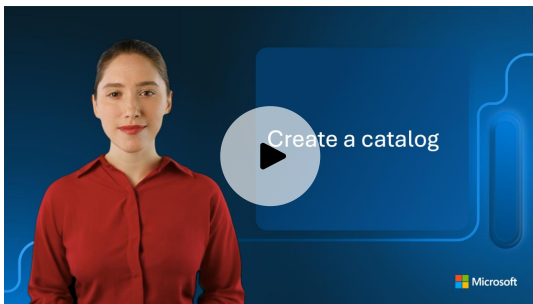
Balance flexibility with standardization based on your organization's needs. Strict enforcement works well for highly regulated environments where consistency is critical for compliance. More flexible guidelines may be appropriate for smaller teams or during periods of rapid experimentation, as long as you eventually converge on standards for production data.

3. Create catalog

Create catalog

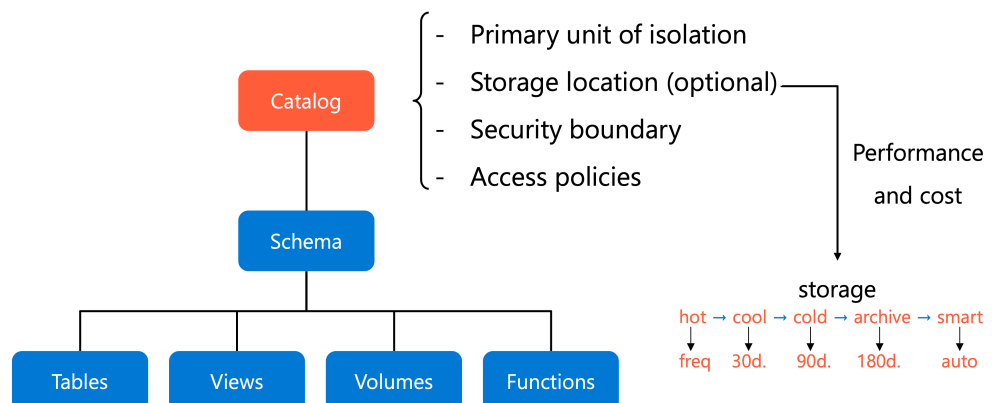
Completed

Your organization's data governance policies require clear separation between **development**, **testing**, and **production** environments. You need production data in dedicated storage, development work isolated from live systems, and sensitive customer information protected from operational data. **Unity Catalog's catalog structure** helps you meet these requirements while maintaining centralized governance and access control.



What is a catalog?

A **catalog** is the **top-level container** in Unity Catalog's **three-layer namespace**. Think of it as the foundation of your data organization, like building a house before decorating the rooms inside. Every data asset in Unity Catalog exists within this hierarchy: `catalog.schema.table`.



Catalogs serve as the primary unit of **data isolation** and organization. Each catalog can have its own **storage location**, **security boundaries**, and **access policies**. This physical separation ensures that your development experiments don't accidentally affect production data, and that sensitive information remains isolated from general-purpose datasets.

With only three layers available for organizing your data, understanding each layer's role becomes essential. Catalogs handle the highest level of separation, typically mapping to business domains, security requirements, or lifecycle stages. The schemas within those catalogs organize the interior, and volumes or tables store the actual data.

Organize data with catalogs

When you design your catalog structure, you're creating the foundation for **data governance** across your organization. Catalogs typically mirror organizational units or software development lifecycle scopes. Consider how your teams work and what data access boundaries matter most to your operations.

Environment-based isolation is one of the most common patterns. You might create separate catalogs for production, staging, and development environments. This separation prevents development queries from impacting production performance and ensures that experimental code never touches live customer data. With this pattern, your production catalog (`prod_catalog`) contains only validated, approved datasets, while your development catalog (`dev_catalog`) provides a safe space for testing and iteration.

Sensitivity-based isolation works well when compliance or privacy concerns drive your architecture. You might separate customer data from operational metrics, or isolate financial records from general business analytics. Each catalog can enforce different access policies, making it simple to grant broad access to public datasets while restricting sensitive information to authorized personnel.

Your catalog structure also affects **performance and costs**. Because each catalog can specify its own managed storage location, you can store frequently accessed production data in high-performance storage while archiving historical development data to more economical options. Regional compliance requirements become manageable too, since Unity Catalog metastores exist per region and catalogs inherit that regional isolation.

Create a catalog

Creating a catalog establishes the foundation for all data objects you'll add later. You must be a **metastore admin** or have the **CREATE CATALOG privilege** to create new catalogs. Once created, the catalog includes two automatically generated schemas: **default** for your general use and **information_schema** for system metadata.

Using **SQL**, you can create a catalog with a simple command:

```
CREATE CATALOG IF NOT EXISTS dev_catalog
COMMENT 'Development environment for data engineering experiments';
```

This creates a catalog named `dev_catalog` with a descriptive comment. The `IF NOT EXISTS` clause prevents errors if the catalog already exists, making your code more robust when run multiple times.

Using **Catalog Explorer**, you can create catalogs through the Azure Databricks UI:

1. Select **Catalog** from the left navigation.
2. Select **Create catalog**.
3. Enter a catalog name and select **Standard** as the type.
4. Optionally specify a managed storage location.
5. Select **Create**.

Create a new catalog



A catalog is the first layer of Unity Catalog's three-level namespace and is used to organize your data assets. [Learn more](#)

Catalog name*

Type*

Standard 

Storage location

Cloud storage location used for managed tables and volumes in this catalog. If not specified, it defaults to the metastore root location.

 Select external location 

sub/path

[Create a new external location](#) 

Cancel

Create

After creating the catalog, you configure **workspace bindings** to control which workspaces can access it. By default, new catalogs are accessible from all workspaces attached to your metastore. For production catalogs, you should restrict access to production workspaces only, ensuring that development environments can't accidentally query or modify production data.

Specify which workspaces can have access to this catalog.

☐ All workspaces have access

Assign to workspaces

Manage Access Level ▾

Revoke

🔍 Filter workspaces...

Workspace name	Workspace id	Access Level	Resource group	Subscription id
ms-learn	1191024883105857	Read & Write	rg-databricks	aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e

Configure managed storage

While creating a catalog without specifying storage is possible, configuring a **managed storage location** is strongly recommended. Managed storage defines where Unity Catalog stores the data files for **managed tables** and **volumes** within your catalog. Without a catalog-level storage location, managed objects fall back to the metastore's default storage, which might not align with your organizational policies or regional requirements.

To specify managed storage, you need an **external location** already configured in Unity Catalog and the **CREATE MANAGED STORAGE privilege** on that location. The storage path must point to a cloud storage container, like an Azure Data Lake Storage Gen2 container, that meets your security and compliance requirements.

```
CREATE CATALOG IF NOT EXISTS prod_catalog
MANAGED LOCATION 'abfss://prod-data@mystorageaccount.dfs.core.windows.net/catalog-'
COMMENT 'Production catalog with dedicated ADLS Gen2 storage';
```

This command creates a catalog with dedicated production storage. All managed tables and volumes created in this catalog's schemas store their data files under this location, providing complete physical isolation from other catalogs. You can even specify a subpath within an external location, giving you flexibility to organize storage containers while maintaining centralized security credentials.

Managed storage serves two critical purposes. First, it provides **physical data isolation** by storing each catalog's data in separate cloud storage paths. When you drop a managed table, Unity Catalog can safely delete the underlying files after eight days without affecting data in other catalogs. Second, it helps you meet **compliance requirements** by ensuring sensitive data resides in specific storage accounts or regions that match your regulatory obligations.

If your metastore doesn't have a default storage location configured, you must specify managed storage when creating catalogs. This requirement ensures that Unity Catalog always knows where to place managed data files, preventing confusion and maintaining clear data governance boundaries.

Catalogs form the foundation of Unity Catalog's data organization model, providing the highest level of isolation and access control. By thoughtfully designing your catalog structure around environment boundaries or sensitivity levels, you create a governance framework that scales with your organization. Now that you understand how to create and configure catalogs, you're ready to organize the interior by adding schemas and data objects within those catalogs.

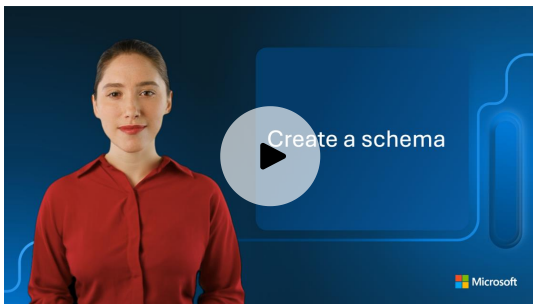
4. Create schema

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/4-create-schema>

Create schema

Completed

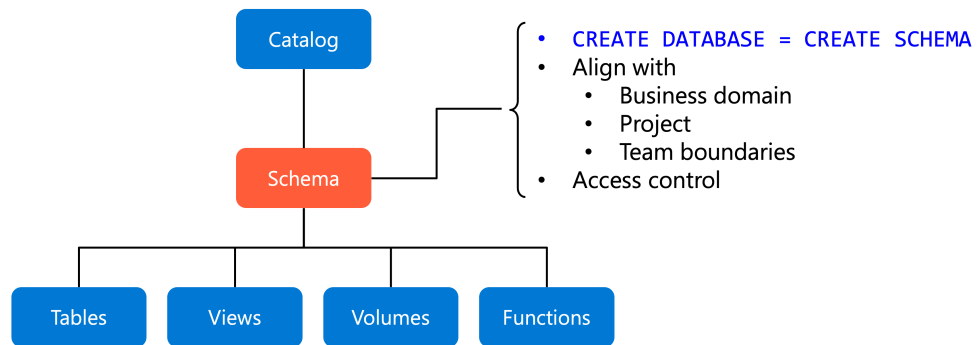
Your catalog provides the foundation for data isolation across environments, but within each catalog you need more detailed organization for your tables, views, and functions. Different teams work on different projects, departments manage distinct datasets, and various use cases require separate namespaces. **Schemas in Unity Catalog** give you this logical organization layer. They let you group related data assets within a catalog while maintaining clear boundaries and access controls.



What is a schema?

A **schema** is the **second level** in Unity Catalog's **three-layer namespace**: `catalog.schema.table`. While catalogs handle environment or sensitivity boundaries, schemas organize data within those catalogs into logical categories that typically represent a single use case, project, or team workspace.

Think of schemas as rooms within the house you built when creating your catalog. Each room serves a specific purpose and contains related items. Your production catalog might contain schemas for `customer_analytics`, `financial_reporting`, and `operations_metrics`, with each schema grouping the tables and views needed for that specific domain.



Schemas provide the organizational structure that makes your data **discoverable** and **manageable**. Instead of hundreds of tables scattered across a single namespace, you group related assets into meaningful categories. This structure helps data engineers find the datasets they need and understand how different data assets relate to each other.

In Azure Databricks, you might hear schemas called "databases." This is because `CREATE DATABASE` is an alias for `CREATE SCHEMA`. The terms are interchangeable in Databricks SQL, though "schema" aligns better with Unity Catalog's namespace hierarchy.

Organize data with schemas

When you design your schema structure within a catalog, consider how your teams work and what logical groupings make sense for your organization. Schemas typically align with **business domains**, **projects**, or **team boundaries**, creating clear ownership and access patterns.

Production Catalog: Department based schemas

- └─ `marketing_analytics`
- └─ `financial_data`
- └─ `operations_metrics`

Development Catalog: Project-based schemas

- └─ `customer_churn_model`
- └─ `pricing_optimization`
- └─ `inventory_forecasting`

Functional Organization: Stage-based Schemas

- └─ `raw_data`
- └─ `cleansed_data`
- └─ `reporting_views`

Department-based organization works well when different business units manage their own data. Within your production catalog, the marketing team might own a `marketing_analytics` schema, while the finance team maintains `financial_data`. Each department controls access to their own schema, granting read permissions to other teams as needed for cross-functional analytics.

Project-based organization suits environments where temporary initiatives need dedicated workspaces. Your development catalog might contain schemas like `customer_churn_model`,

`pricing_optimization`, and `inventory_forecasting`. When a project completes, you can archive or remove its schema without affecting other work. This pattern keeps experimental work organized and makes it easy to track which assets belong to which initiative.

Functional organization groups schemas by data processing stage or purpose. You might create schemas like `raw_data`, `cleansed_data`, and `reporting_views` within a catalog. This pattern mirrors data pipeline architecture, where each schema represents a stage in your data transformation workflow.

Your schema organization affects both **discoverability** and **access control**. Well-named schemas help users locate relevant datasets quickly. Clear schema boundaries make it straightforward to grant appropriate permissions—data engineers might have full access to the `staging` schema while analysts have read-only access to `curated_views`.

Create a schema

Creating a schema establishes a namespace for the tables, views, and functions you'll add later. You need **USE CATALOG** and **CREATE SCHEMA** permissions on the parent catalog. A metastore admin or the catalog owner can grant you these privileges.

Using SQL, you create a schema with this command:

```
CREATE SCHEMA IF NOT EXISTS prod_catalog.customer_analytics
COMMENT 'Customer behavior and segmentation analysis';
```

This creates a schema named `customer_analytics` in the `prod_catalog` catalog. The `IF NOT EXISTS` clause prevents errors if the schema already exists, making your code safe to run repeatedly. The three-part namespace (`catalog.schema.table`) requires you to specify the catalog name, ensuring schemas end up in the correct container.

Using **Catalog Explorer**, you can create schemas through the Azure Databricks workspace UI:

1. Select **Catalog** from the left navigation.
2. Select the catalog where you want to create the schema.
3. Select **Create schema** from the catalog detail pane.
4. Enter a schema name and optional description.
5. Select **Create**.

Create a new schema



A schema is the second layer of Unity Catalog's three-level namespace and organizes tables and views. [Learn more](#)

Schema name*

Storage location

Cloud storage location used for managed tables and volumes in this catalog. If not specified, it defaults to the metastore root location.

 Select external location 

sub/path

[Create a new external location](#) 

Comment

Cancel

Create

After creating the schema, you grant privileges to control who can use it and create objects within it. At minimum, users need **USE SCHEMA** permission to see the schema and query its contents. To create tables or views in the schema, users need **CREATE TABLE** or other object-specific permissions.

Each new schema includes a system-generated **information_schema** that contains metadata about the schema's objects. You can query this metadata to discover tables, views, and their structure.

Configure managed storage

While not required, specifying a **managed storage location** for your schema gives you control over where Unity Catalog stores data files for managed tables created within the schema. Without a schema-level storage location, managed tables use the catalog's managed storage (or the metastore's default if the catalog has none specified).

To configure managed storage, you need an **external location** already set up in Unity Catalog and the **CREATE MANAGED STORAGE** privilege on that location. This lets you separate data for different schemas, even within the same catalog.

```
CREATE SCHEMA IF NOT EXISTS prod_catalog.financial_data
MANAGED LOCATION 'abfss://finance@mystorageaccount.dfs.core.windows.net/schema-data'
```

```
COMMENT 'Financial transactions and reporting data with dedicated storage';
```

This command creates a schema with its own storage location in Azure Data Lake Storage Gen2. All managed tables created in this schema store their data files under this path, separate from other schemas in the catalog. This physical isolation helps you meet compliance requirements where financial data must reside in specific storage accounts with particular security configurations.

If you don't specify managed storage, your schema works the same way. Managed tables use the catalog's storage location, which simplifies management when physical isolation isn't required. You can always update the schema later to add a managed location if your requirements change.

5. Create tables and views

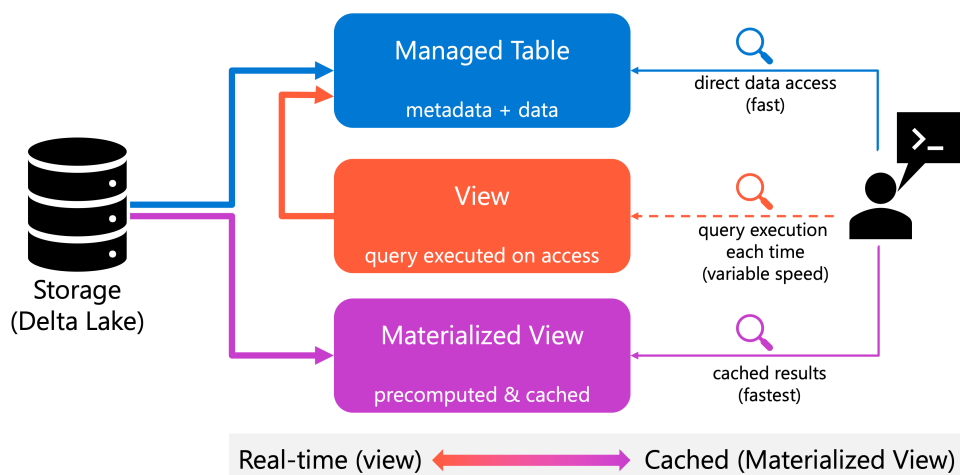
<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/5-create-tables-views-materialized>

Create tables and views

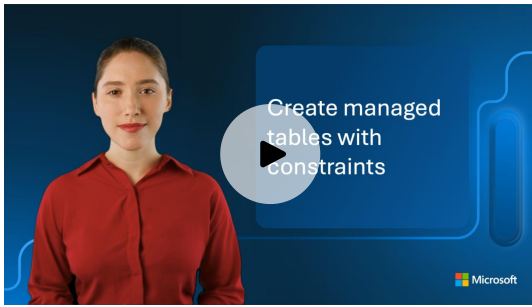
Completed

Tables and views form the foundation of your data organization in Unity Catalog. With tables, you persist data for long-term storage and analysis. With views, you simplify complex queries and control access to underlying data. With materialized views, you optimize query performance by precomputing and caching results. Understanding when and how to create each type helps you build efficient, maintainable data pipelines.

The following diagram illustrates how tables, views, and materialized views relate to each other and to the underlying data:



Create managed tables



Managed tables are the default table type in Unity Catalog. When you create a managed table, Azure Databricks manages both the table metadata and the underlying data files. This management includes automatic storage optimization, lifecycle management, and clean up when you drop the table.

To create a managed table, use the `CREATE TABLE` statement with column definitions. Unity Catalog uses a three-level namespace (`catalog.schema.table`), so you can either specify the full path or set your current catalog and schema context using `USE CATALOG` and `USE SCHEMA` statements. The following example creates a managed table for storing sales transactions:

```
-- Option 1: Use fully qualified name
CREATE TABLE production.sales.sales_transactions (
  transaction_id BIGINT,
  customer_id INT,
  product_name STRING,
  amount DECIMAL(10,2),
  transaction_date DATE
);

-- Option 2: Set current context first
USE CATALOG production;
USE SCHEMA sales;

CREATE TABLE sales_transactions (
  transaction_id BIGINT,
  customer_id INT,
  product_name STRING,
  amount DECIMAL(10,2),
  transaction_date DATE
);
```

Managed tables use the **Delta Lake** format by default, which provides ACID transactions, time travel, and schema enforcement. These features ensure data consistency and enable you to query historical versions of your data. When you drop a managed table, both the metadata and data files are deleted, preventing orphaned data in storage.

For production workloads, consider enabling features like **constraints** and **generated columns**. Constraints enforce data quality rules, while generated columns automatically compute values based on other columns. The following example shows a table with both features:

```
CREATE TABLE customer_orders (
  order_id BIGINT NOT NULL,
  customer_email STRING NOT NULL,
  order_total DECIMAL(10,2),
  tax_amount DECIMAL(10,2) GENERATED ALWAYS AS (order_total * 0.08),
  CONSTRAINT valid_total CHECK (order_total > 0)
);
```

Define primary keys and foreign keys

Primary keys and foreign keys help document table relationships and enable query optimizations in Unity Catalog. Available in Databricks Runtime 11.3 LTS and above (fully GA in Runtime 15.2+), these constraints are **informational only** and are **not enforced**. While they don't prevent invalid data, they provide valuable metadata for query optimization and data modeling.

To define a primary key, specify the `PRIMARY KEY` constraint during table creation. Primary key columns are implicitly `NOT NULL`. The following example creates a table with a single-column primary key:

```
CREATE TABLE customers (
  customer_id BIGINT NOT NULL,
  customer_name STRING,
  email STRING,
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

For composite primary keys spanning multiple columns, list all key columns in the constraint definition:

```
CREATE TABLE order_items (
  order_id BIGINT NOT NULL,
  item_id BIGINT NOT NULL,
  product_name STRING,
  quantity INT,
  price DECIMAL(10,2),
  CONSTRAINT order_items_pk PRIMARY KEY (order_id, item_id)
);
```

Foreign keys define relationships between tables by referencing primary keys in other tables. The following example creates a table with a foreign key constraint:

```
CREATE TABLE orders (
  order_id BIGINT NOT NULL PRIMARY KEY,
  customer_id BIGINT,
  order_date DATE,
  total_amount DECIMAL(10,2),
  CONSTRAINT orders_customers_fk FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

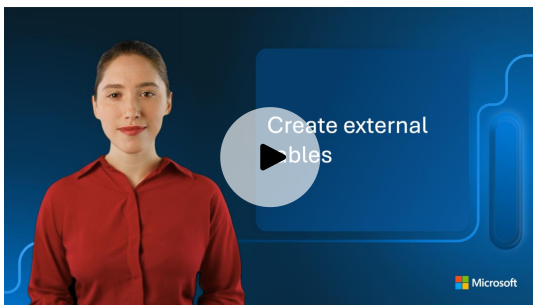
You can also add constraints to existing tables using `ALTER TABLE` statements:

```
ALTER TABLE orders
ADD CONSTRAINT orders_pk PRIMARY KEY (order_id);

ALTER TABLE order_items
ADD CONSTRAINT order_items_orders_fk FOREIGN KEY (order_id) REFERENCES orders(order_id);
```

While these constraints don't enforce referential integrity at write time, they enable the query optimizer to make better decisions about join strategies and query execution plans. This can improve performance for complex queries involving multiple table joins.

Create external tables



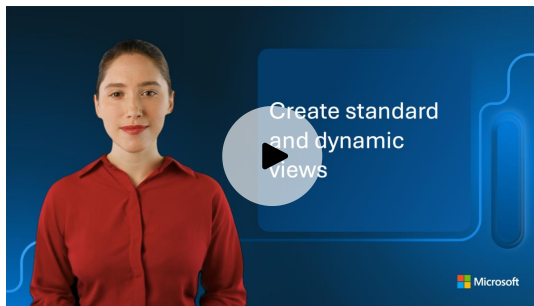
External tables reference data stored in external storage systems while registering metadata in Unity Catalog. Unlike managed tables, dropping an external table only removes the metadata—the underlying data files remain in their original location. This separation makes external tables useful when you need to access data managed by other systems or when you want to share data across multiple workspaces.

To create an external table, specify a `LOCATION` clause pointing to your data. The location must be protected by an external location configured in Unity Catalog. The following example creates an external table over CSV files:

```
CREATE EXTERNAL TABLE archived_sales
USING CSV
LOCATION 'abfss://container@storage.dfs.core.windows.net/sales/archive'
OPTIONS (
  header 'true',
  inferSchema 'true'
);
```

External tables work best when you need to query data in its original format without moving it, or when multiple systems need access to the same data. However, you lose some benefits of managed tables, such as automatic cleanup and certain Delta Lake optimizations. For new data pipelines, prefer managed tables unless you have a specific requirement for external storage.

Create standard views



Views provide a virtual layer over your tables, defined by a query that executes each time you access the view. Standard views don't store data—they compute results on demand by running the underlying query. This design makes views ideal for simplifying complex joins, encapsulating business logic, or providing consistent interfaces to frequently accessed data.

To create a view, use the `CREATE VIEW` statement with a `SELECT` query. The following example creates a view that combines customer and order information:

```
CREATE VIEW customer_order_summary AS
SELECT
    c.customer_id,
    c.customer_name,
    c.region,
    COUNT(o.order_id) AS total_orders,
    SUM(o.order_total) AS total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.customer_name, c.region;
```

Views update automatically when underlying tables change, ensuring you always query current data. This behavior differs from materialized views, which cache results and require explicit refreshes. However, complex views can be slow if they aggregate large datasets or perform expensive joins. For frequently accessed complex queries, consider materialized views instead.

You can layer views on top of other views to create multi-level abstractions. This approach helps organize complex logic into manageable pieces. At the same time, too many view layers can make troubleshooting difficult and impact performance. Balance abstraction with clarity and performance needs.

Create dynamic views for access control

Dynamic views extend standard views with fine-grained security controls. With dynamic views, you apply **row-level filters** and **column-level masks** based on the user querying the view. This capability lets you share data while controlling what different users can see, without duplicating tables or creating multiple views.

To create a dynamic view with column masking, use the `CASE` statement combined with the `is_account_group_member()` function. The following example shows how to redact email addresses for users not in the auditors group:

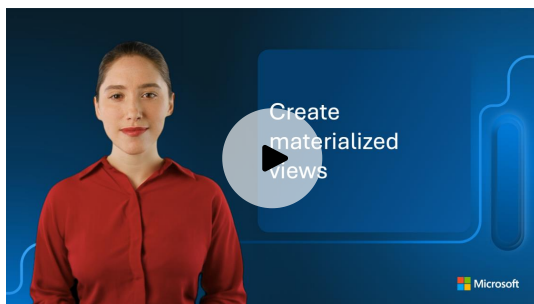
```
CREATE VIEW sales_redacted AS
SELECT
  user_id,
  CASE WHEN
    is_account_group_member('auditors') THEN email
    ELSE 'REDACTED'
  END AS email,
  country_region,
  product,
  total
FROM sales_raw;
```

For row-level security, use a `WHERE` clause with conditional logic. The following example restricts high-value transactions to managers only:

```
CREATE VIEW sales_filtered AS
SELECT
  user_id,
  country_region,
  product,
  total
FROM sales_raw
WHERE
  CASE
    WHEN is_account_group_member('managers') THEN TRUE
    ELSE total <= 1000000
  END;
```

Dynamic views require SQL warehouses, standard access mode compute, or dedicated access mode compute on Databricks Runtime 15.4 LTS or above. These security controls ensure sensitive data remains protected while still enabling broad access to analytics. Without dynamic views, you would need to create separate tables or views for each access level, increasing maintenance overhead.

Create materialized views



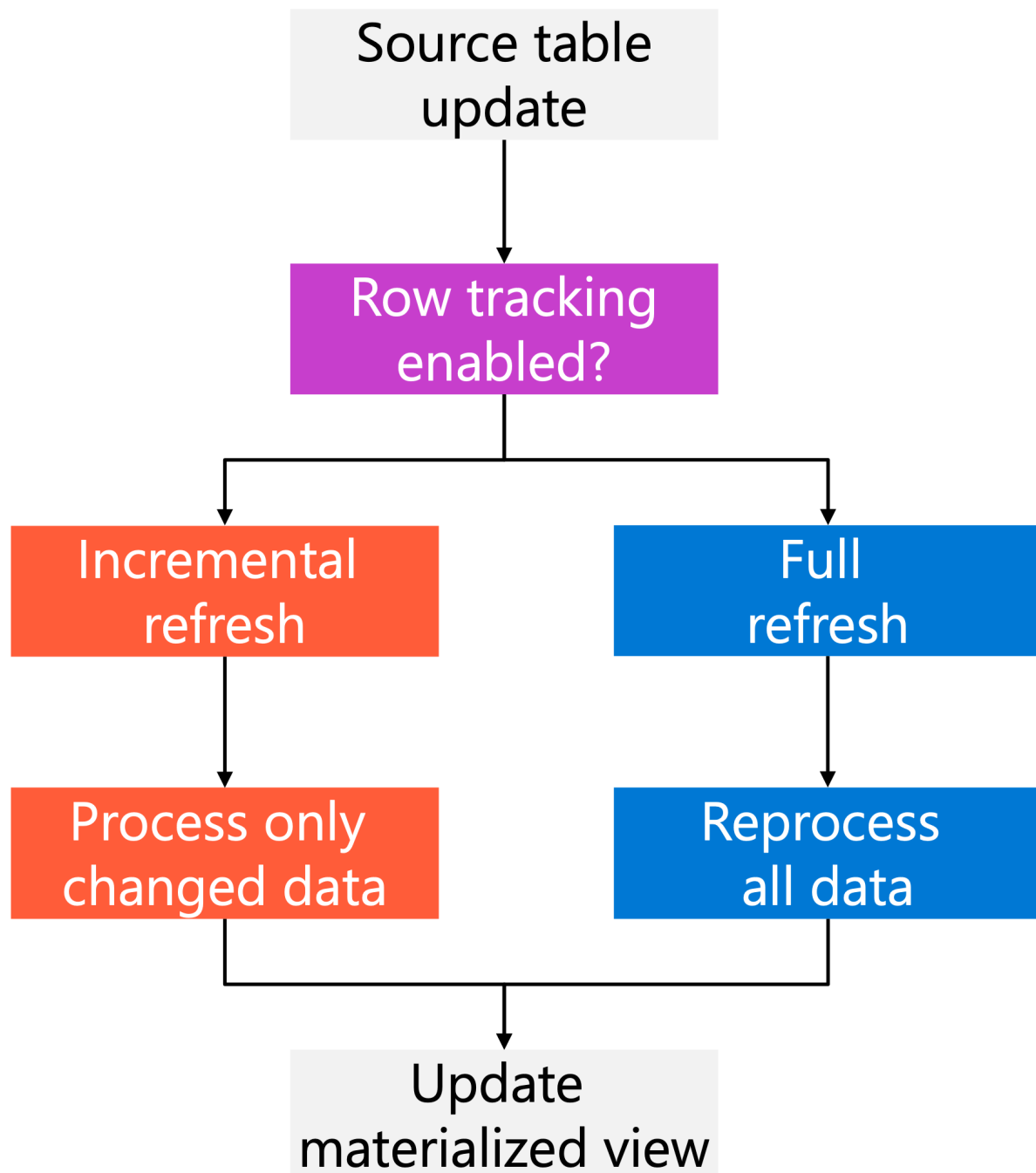
Materialized views precompute and cache query results, storing them as a Delta table. Unlike standard views that recalculate results on every query, materialized views return cached data, making them significantly faster for complex aggregations and frequently accessed queries. You refresh materialized views manually or on a schedule to reflect changes in underlying data.

To create a materialized view, use the `CREATE MATERIALIZED VIEW` statement. The following example creates a materialized view for daily sales summaries:

```
CREATE MATERIALIZED VIEW daily_sales_summary AS
SELECT
    transaction_date,
    COUNT(*) AS transaction_count,
    SUM(amount) AS total_sales,
    AVG(amount) AS average_sale
FROM sales_transactions
GROUP BY transaction_date;
```

Materialized views support two refresh strategies: **incremental** and **full**. Incremental refresh processes only changed data, which is far more efficient than reprocessing everything. To support incremental refresh, your source tables must be Delta tables with **row tracking** enabled:

The following diagram shows how materialized views handle different refresh strategies:



To enable row tracking on your source tables:

```
ALTER TABLE sales_transactions  
SET TBLPROPERTIES (delta.enableRowTracking = true);
```

Note

Row tracking requires Databricks Runtime 14.1 or above. Enabling row tracking on existing tables automatically assigns row IDs to all rows, which can take significant time for large tables

and creates multiple new table versions. This operation upgrades the table protocol version and can't be reversed.

You can schedule automatic refreshes using the `SCHEDULE` clause when creating the materialized view, or trigger refreshes when upstream data changes. The following example schedule daily refreshes:

```
CREATE MATERIALIZED VIEW daily_sales_summary
SCHEDULE EVERY 1 DAY
AS
SELECT
    transaction_date,
    COUNT(*) AS transaction_count,
    SUM(amount) AS total_sales
FROM sales_transactions
GROUP BY transaction_date;
```

Materialized views excel at improving dashboard performance and reducing compute costs for frequently run analytics. Consider materialized views when queries take more than a few seconds to run, or when multiple users query the same aggregated data repeatedly. For data that changes constantly and requires real-time accuracy, standard views might be more appropriate despite slower performance.

Best practices for tables and views

When creating tables, start with **managed tables** unless you have a specific need for external storage. Managed tables provide better integration with Unity Catalog features, automatic optimization, and simpler lifecycle management. Enable row tracking on Delta tables that serve as sources for materialized views to support efficient incremental refreshes.

For **views**, keep the underlying queries focused and avoid creating too many layers. While layering views provides logical organization, it can make debugging difficult and hide performance issues. Document the purpose of each view and its intended audience to help other team members understand your data model.

Use **dynamic views** to implement data security policies at the view level rather than duplicating tables for different user groups. This approach centralizes security logic and reduces data redundancy. Combine dynamic views with Unity Catalog permissions to create a comprehensive data governance strategy.

When choosing between standard and materialized views, consider query performance and data freshness requirements. Use standard views for queries that run quickly or when you need real-time data. Use **materialized views** for expensive aggregations or when eventual consistency is acceptable. Monitor the refresh behavior of materialized views to ensure they update incrementally rather than fully reprocessing data.

Apply **liquid clustering** to tables and materialized views instead of traditional partitioning for better query performance and easier maintenance. Liquid clustering automatically optimizes data layout based on query patterns, while partitioning requires manual configuration and can create skewed partitions. Use the `CLUSTER BY` clause when creating tables or materialized views to enable this optimization.

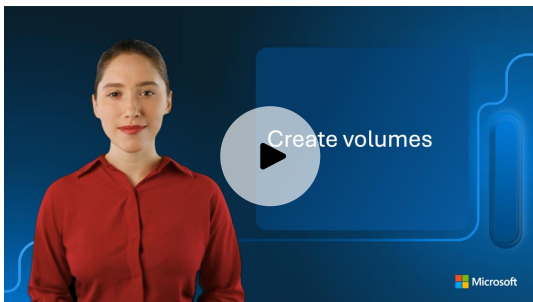
6. Create volumes

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/6-create-volumes>

Create volumes

Completed

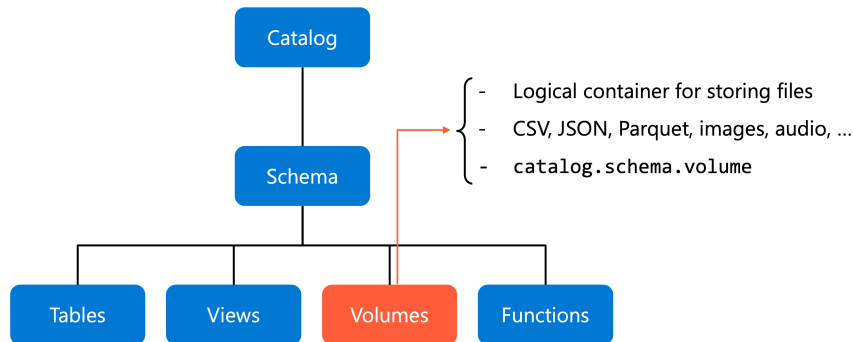
Your data engineering team needs to store machine learning models, CSV files for ingestion, and image files for analytics workloads. You already organized your data using catalogs and schemas, but you need a governed way to manage these nontabular files in cloud object storage. **Unity Catalog volumes** provide the solution by extending governance to files while maintaining the same security model you use for tables.



What are volumes?

A **volume** is a Unity Catalog object that represents a logical container for files stored in cloud object storage. Think of it as the final piece in organizing your data house—after building the foundation with catalogs and organizing the interior with schemas, volumes provide the space to store your raw files.

Volumes sit at the third level of Unity Catalog's namespace structure: `catalog.schema.volume`. Unlike tables that govern structured data, volumes govern files of any format—CSV, JSON, Parquet, images, audio files, or machine learning artifacts.



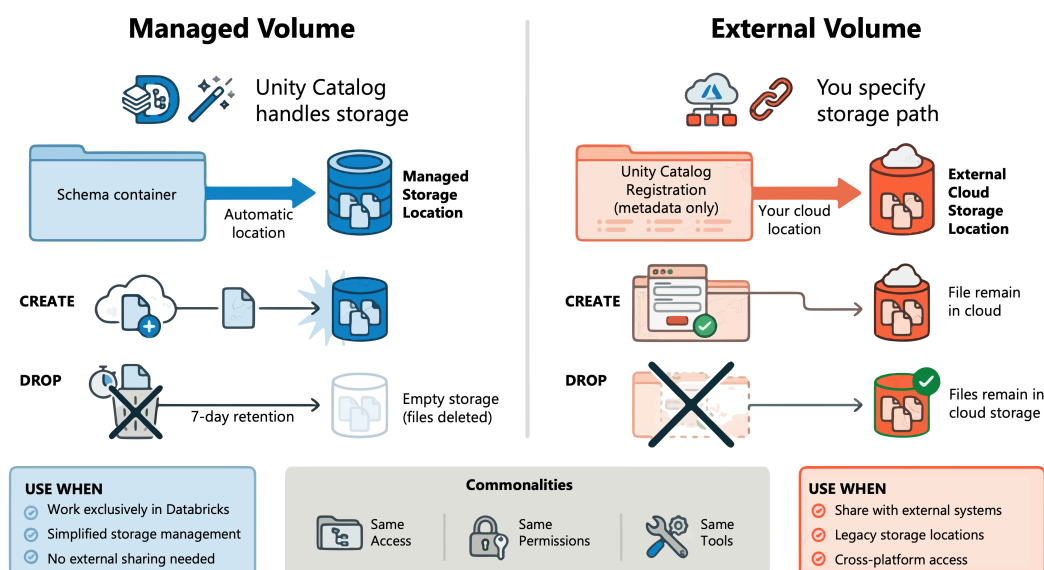
With Unity Catalog's three-layer hierarchy, understanding where volumes fit becomes essential. Volumes handle file-based data governance, providing path-based access to files while maintaining centralized security controls. This separation allows you to manage structured data through tables and unstructured data through volumes, both under the same governance framework.

Choose between managed and external volumes

When you create a volume, you choose between **managed** and **external** volumes. This decision affects where your files are stored and what happens when you delete the volume.

Managed volumes offer the simplest approach. Unity Catalog handles storage location and lifecycle management automatically. When you create a managed volume, Azure Databricks stores files in the managed storage location associated with your schema. When you drop a managed volume, Unity Catalog marks the files for deletion after a seven-day retention period.

External volumes provide governance for existing cloud storage locations. You specify the storage path when creating the volume, and the files remain in that location throughout the volume's lifecycle. When you drop an external volume, the files remain in cloud storage—only the Unity Catalog registration is removed.



Consider managed volumes when you work exclusively within Azure Databricks and want simplified storage management. Your data engineering pipelines can write files directly to volumes without worrying about underlying storage paths or credentials.

Consider external volumes when you need to share data between Azure Databricks and other systems. Perhaps your machine learning models must be accessed by external applications, or you need to add governance to files already produced by legacy systems. External volumes allow you to maintain existing storage locations while adding Unity Catalog's security controls.

The technical implementation remains nearly identical regardless of your choice. You access files using the same path format, apply the same permissions, and use the same tools. The main difference lies in storage control and data lifecycle management.

Create a volume

Creating a volume establishes a governed storage location for your files. You must have **CREATE VOLUME** privilege on the schema and **USE** privileges on both the schema and catalog. For external volumes, you also need **CREATE EXTERNAL VOLUME** privilege on the target external location.

Using SQL, create a managed volume with this syntax:

```
CREATE VOLUME IF NOT EXISTS dev_catalog.bronze_schema.landing_files
COMMENT 'Landing area for CSV files from external systems';
```

This creates a volume named `landing_files` in the specified schema. The `IF NOT EXISTS` clause prevents errors if the volume already exists, making your code resilient to repeated execution.

For external volumes, add the `LOCATION` clause to specify the cloud storage path:

```
CREATE EXTERNAL VOLUME prod_catalog.silver_schema.ml_models
LOCATION 'abfss://models@mystorageaccount.dfs.core.windows.net/production/';
```

The location must point to a path within an external location that Unity Catalog already governs. This ensures that Unity Catalog maintains security control even when the volume references existing storage.

Using Catalog Explorer, create volumes through the UI:

1. Select **Catalog** from the left navigation.
2. Navigate to the target schema and select it.
3. Select **Create > Volume**.
4. Enter a volume name.
5. Choose **Managed** or **External** as the volume type.
6. For external volumes, select an external location and specify the subdirectory path.
7. Select **Create**.

Create a new volume



Volumes are locations for storing files in Unity Catalog.

Volume name*

Volume type [Learn more](#)

- ☒ **Managed volume**
Files are stored in a location managed by Unity Catalog.
- ☐ **External volume**
Files are stored in an external location that you configure.

Choose catalog and schema*

Volumes are organized under a catalog and a schema. Select existing or create new.

 databricks_learn  

 adventure-works  

Cancel

Create

After creation, your volume is ready to store files. The volume inherits permissions from its parent schema, though you can grant specific permissions to individual users or groups as needed.

Access files in volumes

Once you create a volume, you access files using a standard path format that works across all Azure Databricks tools and languages. The path follows this pattern:

```
/Volumes/<catalog>/<schema>/<volume>/<path>/<filename>
```

This POSIX-style path works with Apache Spark, pandas, SQL, and other frameworks without requiring cloud storage credentials or connection strings. For example, to read a CSV file stored in your volume:

```
df = spark.read.format("csv").load("/Volumes/dev_catalog/bronze_schema/landing_files/data.csv")
```

The same path works in pandas:

```
import pandas as pd
df = pd.read_csv('/Volumes/dev_catalog/bronze_schema/landing_files/data.csv')
```

You can also query files directly using SQL:

```
SELECT * FROM csv.`/Volumes/dev_catalog/bronze_schema/landing_files/data.csv`;
```

For file management operations, use `dbutils.fs` commands in notebooks:

```
# List files in a volume
dbutils.fs.ls("/Volumes/dev_catalog/bronze_schema/landing_files")

# Copy a file
dbutils.fs.cp(
    "/Volumes/dev_catalog/bronze_schema/landing_files/source.csv",
    "/Volumes/dev_catalog/silver_schema/processed/destination.csv"
)
```

External volumes support an additional access pattern. Users with appropriate permissions can access files using cloud storage URIs directly, such as

`abfss://container@account.dfs.core.windows.net/path/file.csv`. However, Unity Catalog still governs this access through volume permissions, not external location permissions.

Manage volume permissions

Volume permissions control who can read and write files. Unity Catalog provides four key privileges for volumes:

- **READ VOLUME:** Allows listing and reading files in the volume
- **WRITE VOLUME:** Allows creating, updating, and deleting files
- **CREATE VOLUME:** Allows creating new volumes in a schema
- **CREATE EXTERNAL VOLUME:** Allows creating external volumes using an external location

To grant read access to a data engineers group:

```
GRANT READ VOLUME ON VOLUME dev_catalog.bronze_schema.landing_files TO `data-engineers`
```

To grant full read and write access to data engineers:

```
GRANT READ VOLUME, WRITE VOLUME ON VOLUME dev_catalog.bronze_schema.landing_files TO `data-engineers`
```

Permissions cascade from parent objects. If you grant `READ VOLUME` at the catalog level, users can read files from all volumes in that catalog. This makes it simple to set up broad access policies while restricting sensitive data through more specific grants.

Note

Unity Catalog volumes don't support folder-level or subdirectory-level ACLs. Permissions apply to the entire volume. If you need different access controls for different sets of files, create separate volumes for each security boundary.

Remember that reading or writing files requires both volume permissions and the necessary **USE** privileges on the parent catalog and schema. Unity Catalog enforces this hierarchy to maintain consistent access control across your data assets.

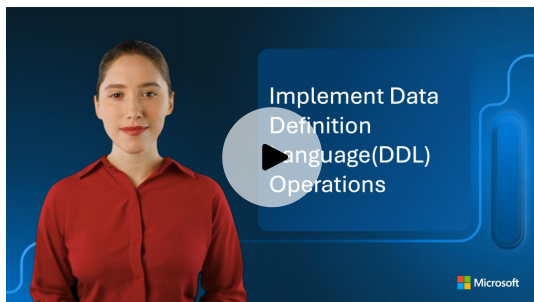
7. Implement DDL operations

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/7-implement-data-definition-language-operations>

Implement DDL operations

Completed

Your data engineering team needs to manage complex data transformations, enforce data quality rules, and maintain streaming pipelines across multiple environments. You need **reusable functions** to standardize calculations, **stored procedures** to orchestrate multi-step operations, and **streaming tables** to process real-time data. Unity Catalog's **Data Definition Language (DDL) operations** provide the tools to create and modify these objects while maintaining governance and version control.



Create functions for reusable logic

Functions encapsulate business logic that you can reuse across queries and transformations. Unity Catalog supports both **scalar functions** that return single values and **table-valued functions** that return entire result sets. You can write functions in **SQL** or **Python**, depending on your team's expertise and the complexity of the logic.

When you create a function in Unity Catalog, it becomes a governed object that you can share across teams and workspaces. With Unity Catalog, you define functions once and invoke them anywhere within your data platform. The catalog tracks function ownership, dependencies, and usage patterns, providing visibility into how your organization uses shared logic.

SQL scalar functions work well for simple calculations and transformations. You define the function with parameters, specify the return type, and provide the SQL expression that computes the result:

```
CREATE FUNCTION dev_catalog.finance_schema.calculate_discount(price DOUBLE, discount_rate DOUBLE)
RETURNS DOUBLE
COMMENT 'Calculate discounted price'
RETURN price * (1 - discount_rate);
```

This function accepts two numeric parameters and returns the discounted price. You can now call this function from any query within your Unity Catalog environment, ensuring consistent discount calculations across all reports and analytics.

Python functions handle more complex logic that requires external libraries or advanced data processing. Unity Catalog executes Python functions using serverless or pro SQL warehouses:

```
CREATE FUNCTION dev_catalog.analytics_schema.normalize_text(input_text STRING)
RETURNS STRING
LANGUAGE PYTHON
COMMENT 'Normalize text using Python'
AS $$
    return input_text.lower().strip() if input_text else None
$$;
```

Table-valued functions return complete result sets rather than scalar values. This approach works well when you need to transform arrays into rows or generate multiple records from a single input:

```
CREATE FUNCTION dev_catalog.utils_schema.split_to_rows(input_array ARRAY<STRING>)
RETURNS TABLE(value STRING)
RETURN SELECT explode(input_array) AS value;
```

You can then query this function like a table: `SELECT * FROM dev_catalog.utils_schema.split_to_rows(array('a', 'b', 'c'))`.

You can also create table-valued functions using **Python**, which is useful when you need more complex processing logic:

```
CREATE FUNCTION dev_catalog.utils_schema.generate_sequence(start INT, end INT)
RETURNS TABLE(num INT)
LANGUAGE PYTHON
COMMENT 'Generate a sequence of numbers from start to end'
AS $$
    return [(i,) for i in range(start, end + 1)]
$$;
```

Temporary functions exist only within your current session and don't persist in the catalog. This approach works well for one-time analyses or testing new logic before creating a permanent function:

```
CREATE TEMPORARY FUNCTION calculate_tax(amount DOUBLE)
RETURNS DOUBLE
RETURN amount * 0.08;
```

Temporary functions don't require catalog or schema qualifiers because they exist only in your session scope.

Create procedures for multi-step operations

SQL stored procedures let you define sequences of SQL statements that run together, with support for variables, control flow, and error handling. Procedures become particularly valuable when you need to orchestrate complex data operations that involve multiple steps, conditional logic, or iterative processing.

With procedures, you can encapsulate entire workflows—like data validation, transformation, and loading—into a single callable unit. You create procedures using the `CREATE PROCEDURE` statement, which requires Unity Catalog and Databricks Runtime 17.0 or above:

```
CREATE PROCEDURE dev_catalog.etl_schema.refresh_customer_summary(
  IN source_date DATE,
  OUT rows_processed INT
)
LANGUAGE SQL
SQL SECURITY INVOKER
COMMENT 'Refresh customer summary table for specified date'
BEGIN
  DECLARE row_count INT;

  DELETE FROM dev_catalog.analytics_schema.customer_summary
  WHERE summary_date = source_date;

  INSERT INTO dev_catalog.analytics_schema.customer_summary
  SELECT customer_id, summary_date, SUM(amount) as total
  FROM dev_catalog.sales_schema.transactions
  WHERE transaction_date = source_date
  GROUP BY customer_id, summary_date;

  SET row_count = (SELECT COUNT(*) FROM dev_catalog.analytics_schema.customer_summary
    WHERE summary_date = source_date);
  SET rows_processed = row_count;
END;
```

This procedure accepts an input date parameter and returns the number of rows processed through an output parameter. You call procedures using the `CALL` statement:

```
CALL dev_catalog.etl_schema.refresh_customer_summary(DATE '2024-01-15', :rows);
```

Procedures support **IN** parameters for inputs, **OUT** parameters for outputs, and **INOUT** parameters that serve both purposes. The `SQL SECURITY INVOKER` clause specifies that the procedure runs with

the permissions of the user calling it, ensuring proper access control.

Modify catalog ownership and properties

As your data organization evolves, you need to transfer ownership of catalogs or modify their properties. The `ALTER CATALOG` statement provides these capabilities, letting you change catalog owners, apply tags for metadata management, or enable predictive optimization.

Transferring ownership ensures that the right team or principal manages each catalog. You might transfer ownership when reorganizing teams or delegating catalog management to a different group:

```
ALTER CATALOG dev_catalog SET OWNER TO `data_engineering_team`;
```

This command transfers ownership to a group named `data_engineering_team`. Only the current owner, a metastore admin, or a user with the `MANAGE` privilege can transfer ownership.

Applying tags helps you organize and classify catalogs according to your organization's taxonomy. Tags provide searchable metadata that supports governance and discovery:

```
ALTER CATALOG dev_catalog SET TAGS (  
  environment = 'development',  
  cost_center = 'engineering',  
  data_classification = 'internal'  
);
```

Enabling predictive optimization automates maintenance tasks like compaction and vacuum operations. Unity Catalog monitors usage patterns and applies optimizations when appropriate:

```
ALTER CATALOG prod_catalog ENABLE PREDICTIVE OPTIMIZATION;
```

Tables within the catalog inherit this setting unless you explicitly override it at the table level.

Modify table structure and constraints

As your data requirements change, you need to modify table schemas without rebuilding entire datasets. The `ALTER TABLE` statement supports adding columns, renaming columns, dropping columns, managing constraints, and changing column types—all while preserving your existing data.

Adding columns extends your table schema to accommodate new data fields. When you add a column to an existing Delta table, Unity Catalog populates it with `NULL` values for existing rows:

```
ALTER TABLE dev_catalog.sales_schema.orders  
ADD COLUMN shipping_carrier STRING  
COMMENT 'Name of the shipping carrier';
```

Renaming columns updates column names without moving data. This operation requires Delta Lake column mapping, which you enable by setting table properties:

```
ALTER TABLE dev_catalog.sales_schema.orders
RENAME COLUMN customer_email TO email_address;
```

Dropping columns removes columns from your table schema. You must drop any dependent constraints or generated columns first:

```
ALTER TABLE dev_catalog.sales_schema.orders
DROP COLUMN temporary_field;
```

Changing column types adapts your schema to new requirements. You can widen types (like `INT` to `BIGINT`) or change compatible types while Unity Catalog handles data conversion:

```
ALTER TABLE dev_catalog.sales_schema.orders
ALTER COLUMN quantity TYPE BIGINT;
```

These operations work on Unity Catalog tables and require the `MODIFY` privilege.

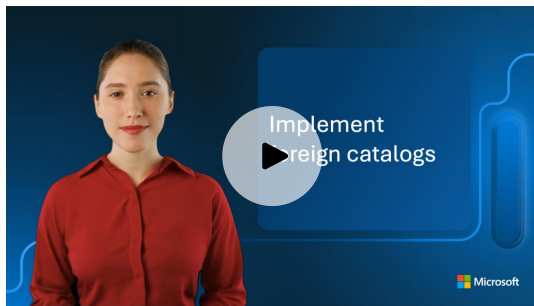
8. Implement foreign catalog

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/8-implement-foreign-catalog>

Implement foreign catalog

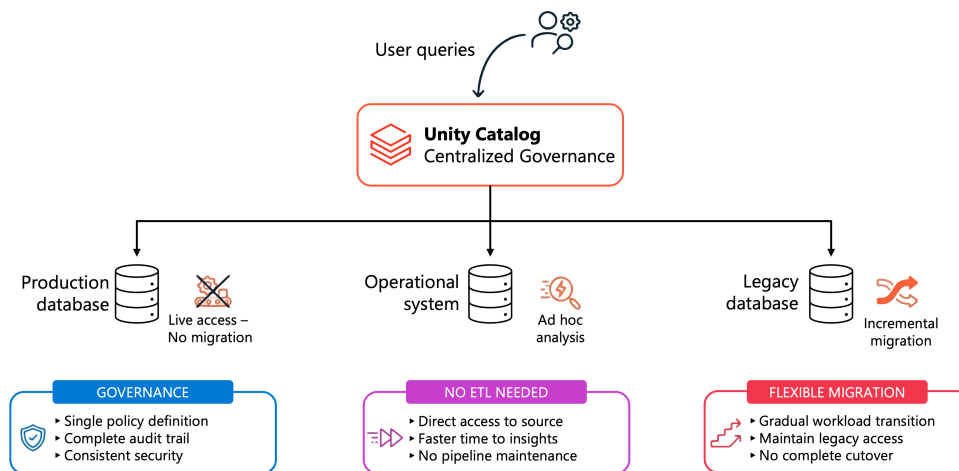
Completed

Your analytics team needs immediate access to customer transaction data stored in a PostgreSQL database. Moving terabytes of operational data into Azure Databricks for exploratory analysis would take weeks and consume significant resources. At the same time, your organization requires **centralized governance** for all data access, regardless of where the data resides. **Foreign catalogs in Unity Catalog** solve this challenge by enabling you to query external databases directly while maintaining Unity Catalog's access control, lineage tracking, and audit capabilities.



Why implement foreign catalogs?

Foreign catalogs extend Unity Catalog's governance to external data systems without requiring data migration. This approach becomes essential when you need to balance operational efficiency with security and compliance requirements.



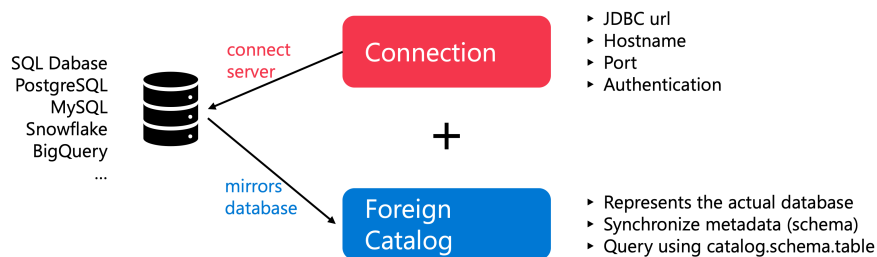
Governance and auditability remain consistent across your data estate. With foreign catalogs, your security team defines access policies once in Unity Catalog, and those policies apply whether users query data in Databricks-managed tables or external databases. Every query against foreign data appears in Unity Catalog's audit logs, providing complete visibility into who accessed what data and when. This centralized governance eliminates the complexity of managing permissions across multiple systems.

External data access without ETL accelerates time to insights. You can run proof-of-concept analyses on production databases, create ad hoc reports from live operational systems, or validate data quality before committing to a full migration. The data stays in its original location, avoiding the time and cost of building and maintaining ETL pipelines for exploratory work.

Incremental migration support provides flexibility during transitions to Unity Catalog. Organizations moving from legacy systems can implement foreign catalogs to maintain access to data that hasn't migrated yet, allowing workloads to transition gradually rather than requiring a complete cutover.

Understand connections and foreign catalogs

A **connection** and a **foreign catalog** work together to enable external data access, but they serve different purposes in the architecture. Understanding this relationship is essential before you begin implementation.



A **connection** stores the technical details needed to reach an external database system. Think of it as a registered credential that specifies where the database lives and how to authenticate to it. The connection contains the JDBC URL, hostname, port, and authentication credentials for systems like PostgreSQL, MySQL, Snowflake, or BigQuery. You create the connection once and can use it to establish multiple foreign catalogs if needed.

A **foreign catalog** represents the actual database you want to query. It mirrors the structure of an external database within Unity Catalog's namespace, making external schemas and tables accessible through standard SQL queries. When you create a foreign catalog, you specify which connection to use and which external database to mirror. Unity Catalog then synchronizes the metadata, allowing you to reference external tables using the familiar `catalog.schema.table` naming pattern.

This separation between connection and foreign catalog provides important flexibility. A single connection to your PostgreSQL server might support multiple foreign catalogs, each representing a different database on that server. The connection handles authentication, while the foreign catalog determines what data becomes visible in Unity Catalog.

Configure a connection to the external database

Before you can create a foreign catalog, you must establish a connection that defines how Azure Databricks reaches your external database. The connection stores credentials securely and validates network connectivity to the target system.

Prerequisites must be in place before creating a connection. You need either the **CREATE CONNECTION** privilege on the Unity Catalog metastore or metastore admin permissions. Your Azure Databricks compute must use Runtime 13.3 LTS or above with Standard or Dedicated access mode. Network connectivity between your Azure Databricks workspace and the external database is essential; firewalls and network security groups must allow traffic on the database port.

Using SQL, you create a connection by specifying the database type and configuration options. This example creates a connection to a PostgreSQL database:

```
CREATE CONNECTION postgresql_prod TYPE postgresql
OPTIONS (
  host 'prod-db.example.com',
  port '5432',
  user secret ('prod-secrets', 'postgres-user'),
  password secret ('prod-secrets', 'postgres-password')
);
```

This command establishes a connection named `postgresql_prod` that points to a PostgreSQL server. The `secret` function references credentials stored in Databricks secrets rather than embedding passwords directly in the SQL statement, following security best practices. Each database type (MySQL, Snowflake, BigQuery) requires specific options appropriate to that system.

For **Azure SQL Database**, you would create a connection like this:

```
CREATE CONNECTION azuresql_prod TYPE SQLSERVER
OPTIONS (
  host 'myserver.database.windows.net',
  port '1433',
  user secret ('prod-secrets', 'azuresql-user'),
  password secret ('prod-secrets', 'azuresql-password')
);
```

Using **Catalog Explorer**, you can create connections through the Azure Databricks interface:

1. In the navigation menu, select **Catalog**.
2. Select the **Add** icon and choose **Add a connection**.
3. Enter a name for your connection, such as `postgresql_prod`.
4. Select the connection type (for example, PostgreSQL, MySQL, or Snowflake).
5. Provide the required connection details including host, port, and credentials.
6. (Optional) Select **Test connection** to verify connectivity.
7. Select **Create connection**.

Validation confirms that Azure Databricks can reach your database. After creating the connection, test it by attempting a simple metadata query. If the connection fails, verify network rules, credentials, and that the database server accepts connections from Azure Databricks IP addresses.

Create a foreign catalog using the connection

With a validated connection in place, you're ready to create the foreign catalog that makes external tables accessible through Unity Catalog. This step establishes the mapping between your external database and Unity Catalog's namespace.

Permissions required include the **CREATE CATALOG** privilege on the metastore and either ownership of the connection or the **CREATE FOREIGN CATALOG** privilege on that specific connection. These granular permissions allow you to control who can expose external data through Unity Catalog.

Using SQL, you create a foreign catalog by referencing the connection and specifying which external database to mirror:

```
CREATE FOREIGN CATALOG IF NOT EXISTS prod_customer_data
USING CONNECTION postgresql_prod
OPTIONS (database 'customers');
```

This command creates a catalog named `prod_customer_data` that mirrors the `customers` database from your PostgreSQL server. The `IF NOT EXISTS` clause prevents errors when running the script multiple times. Unity Catalog immediately synchronizes metadata from the external database, making its schemas and tables visible in your workspace.

Using Catalog Explorer, you can create the foreign catalog during connection setup or afterward:

1. Select **Catalog** and then select **Create a catalog**.
2. Choose **Foreign catalog** as the catalog type.
3. Enter a name like `prod_customer_data`.
4. Select the connection you created earlier.
5. Specify the external database name (for example, `customers`).
6. Select **Create**.

Create a new catalog



A catalog is the first layer of Unity Catalog's three-level namespace and is used to organize your data assets. [Learn more](#)

Catalog name*

Type*

Foreign 

Connection*

Select connection 

[Create a new connection](#) 

Cancel

Create

Metadata synchronization happens automatically each time you interact with the foreign catalog. When you query `SELECT * FROM prod_customer_data.public.transactions`, Unity Catalog refreshes its view of the external schema structure before executing the query. This ensures you

always see current metadata, even if database administrators add new tables or columns in the external system.

While automatic synchronization is optimal for most workloads, it can impact performance for high-frequency queries or when external systems are accessed by tools outside Databricks. For these scenarios, you can manually refresh metadata on a schedule using the `REFRESH FOREIGN` command. This approach allows queries to run faster using cached metadata rather than refreshing on every query. You can refresh at different levels of granularity:

```
-- Refresh an entire catalog
REFRESH FOREIGN CATALOG prod_customer_data;

-- Refresh a specific schema
REFRESH FOREIGN SCHEMA prod_customer_data.public;

-- Refresh a specific table
REFRESH FOREIGN TABLE prod_customer_data.public.transactions;
```

Schedule these refresh operations using Lakeflow Jobs to run at intervals that match how frequently your external schema changes. This proactive approach is useful immediately after creating a foreign catalog, when the first query would otherwise trigger a full metadata refresh.

Grant access and query foreign data

Foreign catalogs appear in Unity Catalog alongside your standard catalogs, making them subject to the same permission model. You control access by granting appropriate privileges to users and groups.

Grant read privileges to users who need to query foreign data:

```
GRANT USE CATALOG ON CATALOG prod_customer_data TO `data-analysts`;
GRANT USE SCHEMA ON SCHEMA prod_customer_data.public TO `data-analysts`;
GRANT SELECT ON TABLE prod_customer_data.public.transactions TO `data-analysts`;
```

These grants follow Unity Catalog's hierarchical permission model. Users need `USE CATALOG` on the catalog, `USE SCHEMA` on the schema, and `SELECT` on specific tables. This granular control lets you expose some foreign tables while restricting others, even though they all exist in the same external database.

Query foreign tables using standard SQL syntax. Once permissions are in place, users query foreign data exactly as they would query native Unity Catalog tables:

```
SELECT
  customer_id,
  SUM(transaction_amount) AS total_spent
FROM prod_customer_data.public.transactions
WHERE transaction_date >= '2024-01-01'
GROUP BY customer_id
ORDER BY total_spent DESC
LIMIT 100;
```

Azure Databricks translates this query into SQL appropriate for PostgreSQL, pushes the query down to the external database, and returns results. The external database performs the aggregation and filtering, leveraging its own compute resources while Azure Databricks handles the final result set.

Query execution uses remote compute in the external system. Understanding this behavior helps you optimize performance. Complex joins, aggregations, and filters are pushed down to the external database when possible, reducing data transfer. However, if your result set is very large, the Databricks executor retrieving data might run out of memory. Monitor query performance and consider using materialized views to cache frequently accessed external data locally.

9. Configure AI/BI Genie instructions

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/9-configure-ai-bi-genie-instructions>

Configure AI/BI Genie instructions

Completed

Making your data discoverable involves more than organizing it into catalogs and schemas. **AI/BI Genie** in Azure Databricks enables business users to explore data through **natural language questions**, but it needs guidance to interpret your organization's terminology and business logic.

When you configure AI/BI Genie instructions properly, you transform Unity Catalog metadata into a **knowledge store** that helps both humans and AI systems understand your data.

The screenshot shows the Databricks AI/BI Genie interface. At the top, there's a header with 'Microsoft Azure' and 'databricks' logos, a search bar, and a user profile. Below the header, the 'My Genie Space' section is visible, featuring buttons for 'New chat', 'Configure', 'Benchmarks', 'Monitoring', and 'Share'. A chat history shows two queries. The first query, 'how many products do I have in my catalog?', is answered with 'You have a total of 606 products in your catalog.' and a table with one row showing 'product_count' as 606. The second query, 'what is the average price?', is answered with 'The average price of products in your catalog is 747.66.' and a table with one row showing 'average_price' as 747.66. At the bottom, there's a text input field 'Ask your question...' and a disclaimer 'Always review the accuracy of responses.'

Microsoft Azure | databricks | Search | databricks-learn | User Profile

My Genie Space

+ New chat | Configure | Benchmarks | Monitoring | Share

how many products do I have in my catalog?

Analysis (click to view) | dim_product

You have a total of **606 products** in your catalog.

1 row | Download | Show code

	1.2 product_count
1	606

Was this correct? [Yes] [No] [Feedback]

what is the average price?

Analysis (click to view) | dim_product

The **average price** of products in your catalog is **747.66**.

1 row | Download | Show code

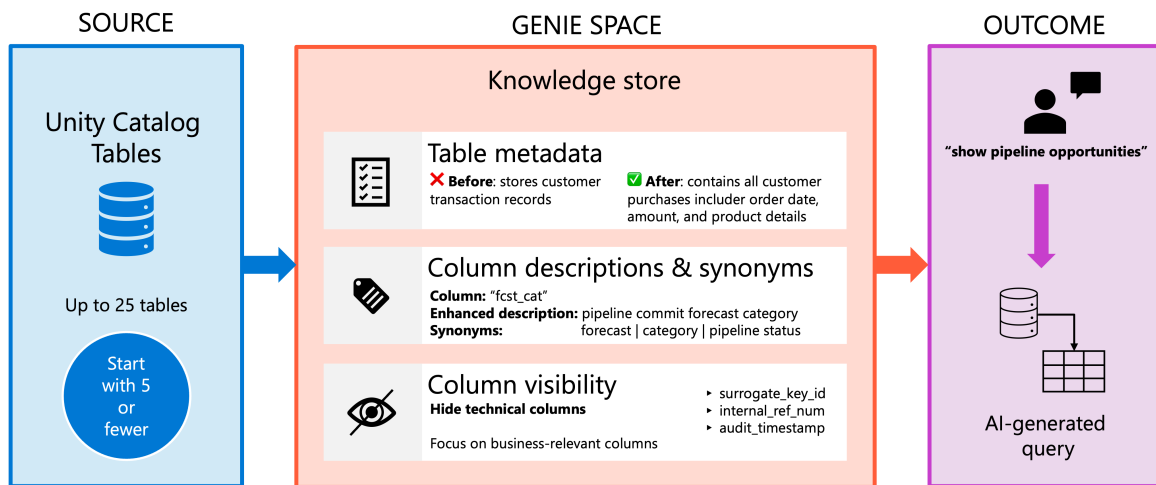
	1.2 average_price
1	747.66

Ask your question... [Send]

Always review the accuracy of responses.

Configure the knowledge store

The **knowledge store** is your primary tool for teaching AI/BI Genie about your data. You can customize **metadata** at the **space level** without altering the underlying Unity Catalog objects. This gives you flexibility to refine descriptions and add context specific to how business users interact with the data.



When you create a Genie space, you start by selecting up to **25 tables or views** from Unity Catalog. These data objects form the foundation of your space. The key is choosing tables that answer a focused set of business questions rather than including everything available. A sales manager might need access to opportunity and account tables, while a logistics manager requires shipment and inventory data. Keep your initial selection small. **Five tables or fewer** works well. Expand only as users request additional data.

After adding tables, you enhance their discoverability by editing metadata. Select **Configure > Data** in your Genie space, then select a table to view its columns. Each table includes a **description field** where you can explain its purpose using **business language**. Instead of technical descriptions like "stores customer transaction records," write "contains all customer purchases including order date, amount, and product details." This context helps Genie generate accurate SQL when users ask questions about purchases or orders.

Column-level metadata requires similar attention. Unity Catalog preserves column names and descriptions from the source tables, but you can refine them within the Genie space. Suppose you have a column named `fcst_cat` that stores forecast categories. In the space, you update its description to "indicates whether an opportunity is in the Pipeline, Best Case, or Commit forecast category" and add **synonyms** like "forecast," "category," and "pipeline status." When users ask about pipeline opportunities, Genie matches their language to your column using these synonyms.

You can also **hide columns** that don't serve business users. Technical columns like surrogate keys, internal identifiers, or audit timestamps often confuse both users and AI. Select the columns you want to hide and choose **Hide selected columns** from the **Actions** menu. This keeps Genie focused on columns that matter for business analysis.

Define data relationships and prompt matching

With metadata in place, you help Genie understand how your tables connect and what values they contain. These configurations allow Genie to write accurate **JOIN statements** and match user questions to specific data values.

Unity Catalog can store **foreign key relationships** between tables, and Genie uses these to construct **JOIN** statements automatically. If your tables don't have foreign keys defined in Unity Catalog, or if you need more complex join logic, define relationships in the knowledge store. Navigate to the **Joins** section and specify how tables relate to each other.

For a sales pipeline dataset, you define a join between an **opportunity** table and an **accounts** table. Select both tables, enter the join condition `opportunity.accountid = accounts.id`, and specify the relationship type as **Many to one** because multiple opportunities can belong to a single account. This teaches Genie how to combine these tables when users ask questions like "show me opportunities by account region."

Prompt matching complements relationship definitions by giving Genie visibility into your actual data. When enabled, Genie collects representative values from columns, helping it match user prompts to real values in your tables. If a user asks about "opportunities in California" but your data uses state codes like "CA," prompt matching allows Genie to make that connection. Two components control how matching works: **format assistance** provides representative values for all eligible columns so Genie understands data types and formatting patterns, while **entity matching** curates distinct value lists for categorical columns. The system automatically enables prompt matching when you add tables, but you can manage which columns participate.

Entity matching is most useful for categorical columns where users are likely to reference specific entries—such as states, product categories, and region names. Genie can store up to **1,024 distinct values** per column across **120 columns** in your space. To enable entity matching for a column, select the pencil icon next to the column name, expand **Advanced settings**, and turn on **Entity matching**. As your data changes over time, refresh prompt matching data to keep the knowledge store current.

Add SQL instructions and examples

Beyond metadata and relationships, you teach Genie how to answer business questions by providing **SQL instructions**. Two types of instructions serve different purposes: **SQL expressions** define reusable business concepts, while **example SQL queries** show how to handle specific question patterns.

SQL expressions work best for common business terms that appear across many questions. Navigate to **Configure > Instructions > SQL Expressions** to define measures, filters, and dimensions. A measure might calculate gross margin as `(revenue - cost) / revenue`, while a dimension could extract the fiscal quarter from a date field. When you define these expressions with clear names and synonyms, Genie can incorporate them into generated queries without you writing complete SQL examples.

Example SQL queries address more complex scenarios where users ask multi-part questions or use domain-specific phrasing. These queries appear in the **SQL Queries** tab under **Instructions**. For each example, you write both the question users might ask and the SQL that answers it. If users frequently ask "What's our current pipeline by region?", provide that exact phrasing along with SQL that joins opportunity and account tables, filters for open opportunities in the pipeline forecast category, and groups results by region.

```
-- What's our current pipeline by region?
SELECT
  accounts.region AS Region,
  SUM(opportunity.amount) AS Pipeline
FROM opportunity
JOIN accounts ON opportunity.accountid = accounts.id
WHERE
  opportunity.forecastcategory = 'Pipeline'
  AND opportunity.stagename NOT LIKE '%closed%'
GROUP BY accounts.region;
```

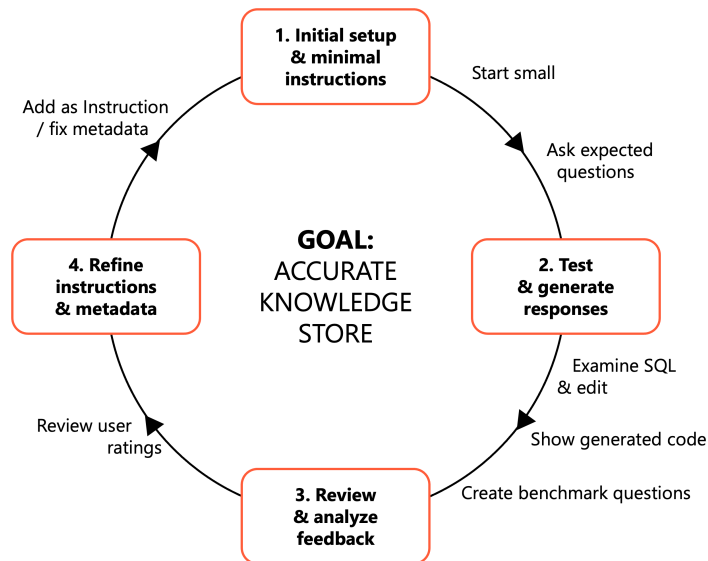
Genie can either use example queries directly or learn from them to answer similar questions. When you add **parameters** to example queries using the syntax `:parameter_name`, Genie substitutes user-provided values at runtime. Parameterized example queries that match the user's exact question are marked as **Trusted** to give users confidence in the results. Non-parameterized example queries are used as context to guide Genie in generating similar SQL, but aren't marked as Trusted.

The **Text** tab holds general instructions that apply globally across all questions. Use this sparingly for context that doesn't fit into other formats. Examples include explaining that your fiscal year starts in February or that all timestamps should be converted to a specific timezone. Too many general instructions can dilute their effectiveness, so focus on essential business rules that Genie can't infer from your data.

You can add up to **100 instructions** total across SQL queries, SQL functions, and the general text instructions block. Each example SQL query counts as one, each SQL function counts as one, and the entire general instructions text block counts as one. SQL expressions have a separate limit of **200** and do not count toward the 100-instruction total. These limits encourage you to prioritize quality over quantity.

Apply iterative refinement

Configuring AI/BI Genie is an **ongoing process**, not a one-time setup. Start with minimal instructions and expand based on actual user questions and feedback.



After configuring your initial space, test it yourself by asking questions you expect business users to pose. Review the generated SQL carefully to identify where Genie misinterprets data or business terminology.

When you find incorrect responses, use the **Show generated code** option to examine the SQL, edit it as needed, and save it as a new instruction by clicking **Add as instruction**. This teaches Genie how to handle similar questions in the future. You can also create **benchmark questions** that test Genie's accuracy systematically. Add common questions with their expected SQL answers, then run benchmarks regularly as you refine the space to measure improvement.

Encourage business users to provide feedback through the built-in response rating mechanism. When users mark responses as correct or incorrect, you can review their feedback in the **Monitoring** tab and adjust instructions accordingly. This collaborative refinement ensures your Genie space evolves to meet real business needs rather than assumptions about how users might ask questions.

As you iterate, watch for patterns in failed queries. If Genie consistently uses the wrong table or column, review your metadata and descriptions. If it can't match user language to data values, check that prompt matching is enabled for relevant columns. If joins are incorrect, verify your relationship definitions. Each refinement makes the knowledge store more accurate and the space more valuable to users.

10. Exercise - Create and Organize Objects in Unity Catalog

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/10-exercise-create-organize-objects-unity-catalog>

Exercise - Create and Organize Objects in Unity Catalog

Completed

Now it's your chance to create and organize objects in Unity Catalog. In this lab, you build a complete Unity Catalog namespace for a university data platform — creating a catalog, medallion schemas, managed tables with primary and foreign key constraints, views, a volume, and a reusable SQL function. You practice DDL operations such as adding columns and applying governance tags, and explore how Unity Catalog organizes and governs structured data at every layer of the medallion architecture. By the end, you'll have a fully structured, query-ready environment that reflects real-world data engineering practices on Azure Databricks.

Note

To complete this lab, you need an [Azure subscription](#) in which you have administrative access.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

11. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/11-knowledge-check>

Knowledge check

Completed

12. Summary

Summary

Completed

You've learned how to create and organize Unity Catalog objects to build a comprehensive data governance framework. Starting with catalogs for environment isolation, you progressed through schemas for logical organization, then created tables, views, and volumes for storing and accessing data. You explored advanced capabilities like foreign catalogs for external database integration and AI/BI Genie instructions for natural language data discovery.

The three-layer namespace structure—catalogs, schemas, and objects—provides the flexibility to organize data according to your organization's needs while maintaining centralized security controls. Managed tables and volumes simplify data lifecycle management, while external objects let you govern existing data without migration. Dynamic views enable row-level and column-level security, and materialized views improve performance for complex analytics queries.

Effective naming conventions and well-configured metadata make your data discoverable and maintainable over time. As your organization's data platform grows, these foundational organizational patterns scale to accommodate new teams, projects, and use cases while preserving the governance boundaries you've established.

Apply these concepts in your own Azure Databricks environment by starting small—create a catalog for development work, add a schema for your current project, and experiment with different table and view types. As you gain confidence, expand your catalog structure and refine your naming conventions to match your organization's governance requirements.