

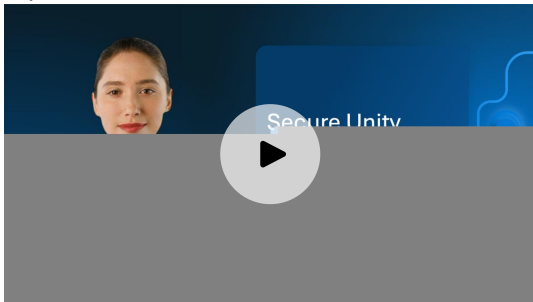
Secure Unity Catalog objects

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/1-introduction>

Introduction

Completed



Securing data is fundamental to enterprise data platforms, yet traditional approaches often fragment governance across multiple systems and require constant maintenance. As organizations scale their analytics workloads on Azure Databricks, they need unified security that protects sensitive information while maintaining operational efficiency. **Unity Catalog** addresses this challenge by providing centralized governance for data assets, compute resources, and access controls.

When you work with Unity Catalog, security operates at multiple layers. You control who can access catalogs, schemas, and tables through **granular permissions**. You enforce **row-level and column-level restrictions** to ensure users see only the data they're authorized to view. You authenticate data access through **service principals** or **managed identities**, eliminating hardcoded credentials from your code. You retrieve sensitive configuration values from **Azure Key Vault** without exposing secrets in notebooks or job configurations.

Understanding how these security mechanisms work together enables you to build data solutions that balance accessibility with protection. You need to know when inherited permissions simplify administration and when explicit grants provide necessary control. You must choose between row and column security functions versus dynamic views based on your governance requirements. You should recognize when managed identities offer advantages over service principals for authenticating storage access.

This module guides you through Unity Catalog's security model, from understanding the **query lifecycle** to implementing **authentication strategies**. You explore access control patterns that scale across your organization, fine-grained filtering techniques that protect sensitive data, and credential management approaches that strengthen your security posture. By the end, you'll be equipped to design and implement comprehensive security for your Azure Databricks environment.

2. Understand query lifecycle

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/2-understand-query-lifecycle>

Understand query lifecycle

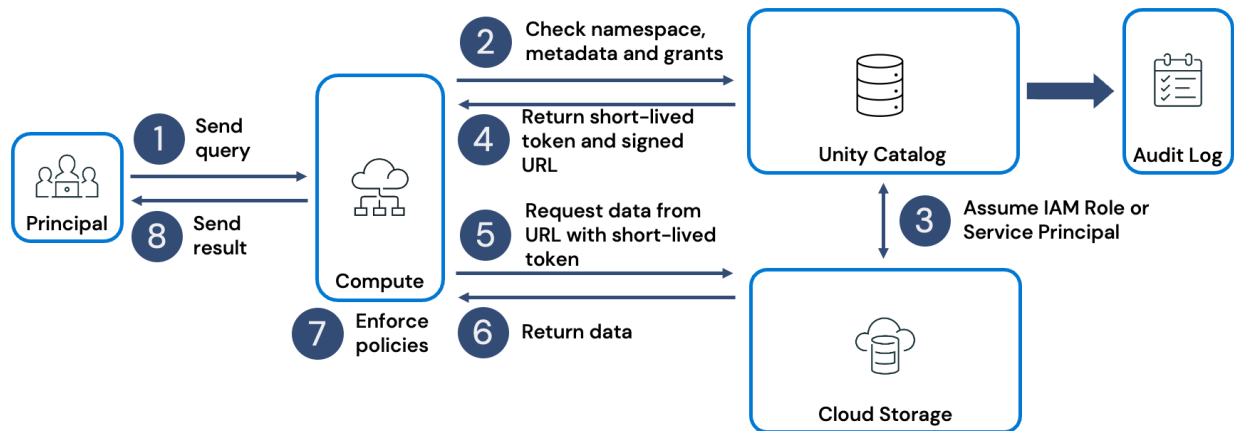
Completed

Let's examine how Unity Catalog manages query execution, with particular attention to security and administrative controls. We'll walk step by step through what happens when a query is submitted against a Unity Catalog table, covering the interactions between compute, the control plane, the data plane, and cloud storage.

The query lifecycle in Unity Catalog



The following diagram illustrates the overall flow of a query as it moves through different components in Unity Catalog.



Step 1: Query submission

The lifecycle begins when a principal (a user or service identity) issues a query. Queries can originate in different ways:

- A data scientist might use an **all-purpose cluster** for interactive Python or SQL workloads.
- A **service principal** might run a scheduled pipeline or job on a **job cluster**.
- A **data analyst** might send a query through **Databricks SQL**, using a SQL warehouse.
- A **business intelligence (BI) tool** connected to a SQL warehouse could also generate the request.

In each case, the compute resource—cluster or warehouse—receives the query and begins execution.

Step 2: Request validation in Unity Catalog

The compute resource forwards the request to **Unity Catalog**. Unity Catalog acts as the control plane for security and governance. It records the request in the **audit log** and checks whether the principal has the necessary permissions to access the objects referenced in the query. If permissions are denied, the query is blocked and the denial is logged. If access is granted, the request moves forward.

Step 3: Assuming cloud credentials

For every object referenced in the query, Unity Catalog assumes the appropriate **cloud credential** associated with that object. These credentials are configured by a cloud administrator.

- For **managed tables**, the credential typically points to the cloud storage tied to the Unity Catalog metastore.
- For **external tables or files**, the credential corresponds to an external location governed by a defined storage credential.

Note

In Azure, the preferred way to grant Unity Catalog access to Azure Data Lake Storage (ADLS Gen2) is via a **Managed Identity + Access Connector** (versus relying purely on service principals). They have the benefit of allowing Unity Catalog to access storage accounts protected by network rules, which isn't possible using service principals, and they remove the need to manage and rotate secrets.

Step 4: Issuing scoped tokens

Once credentials are validated, Unity Catalog generates a **scoped temporary access token** for each object. Along with the token, it provides a secure access URL. This allows the compute resource to read directly from storage without exposing long-lived credentials.

Step 5: Data access from storage

The compute resource (cluster or SQL warehouse) uses the token and URL provided by Unity Catalog to request data directly from the underlying ADLS Gen2 endpoints (via `abfss://` or `dfs.core.windows.net` URL)

Note

If your ADLS account has firewall or virtual network restrictions, you must explicitly allow the Azure Databricks access connector / managed identity to access the storage (in addition to allowing the compute nodes). If the storage account is locked down, even a valid token may be rejected if the identity is not allowed via firewall rules.

Step 6: Data transfer

Cloud storage returns the requested data to the compute resource. This process is repeated for every object referenced in the query.

Step 7: Fine-grained filtering

Unity Catalog enforces **row- and column-level filters** on the compute resource itself. This ensures that principals only see the exact subset of data they're entitled to access.

- **Row-level filtering** introduces conditions that are evaluated at query execution time inside the compute engine. Even though the underlying storage system may return a broader dataset, Unity Catalog ensures that only the rows matching the defined conditions are made visible to the requesting principal. This allows multiple users to query the same table but each receives a personalized, restricted view of the data. A row filter function is basically a predicate (a SQL expression or user-defined function) that is evaluated during query execution.
- **Column-level filtering** operates similarly but at the attribute level. Specific columns can be masked or transformed so that sensitive information is hidden or anonymized, depending on

the privileges of the principal. This makes it possible to share datasets broadly while ensuring that confidential fields are only fully visible to those with appropriate authorization.

Step 8: Returning the result

The filtered query result is returned from the compute resource to the calling user, job, or application.

How queries work with Apache Hive



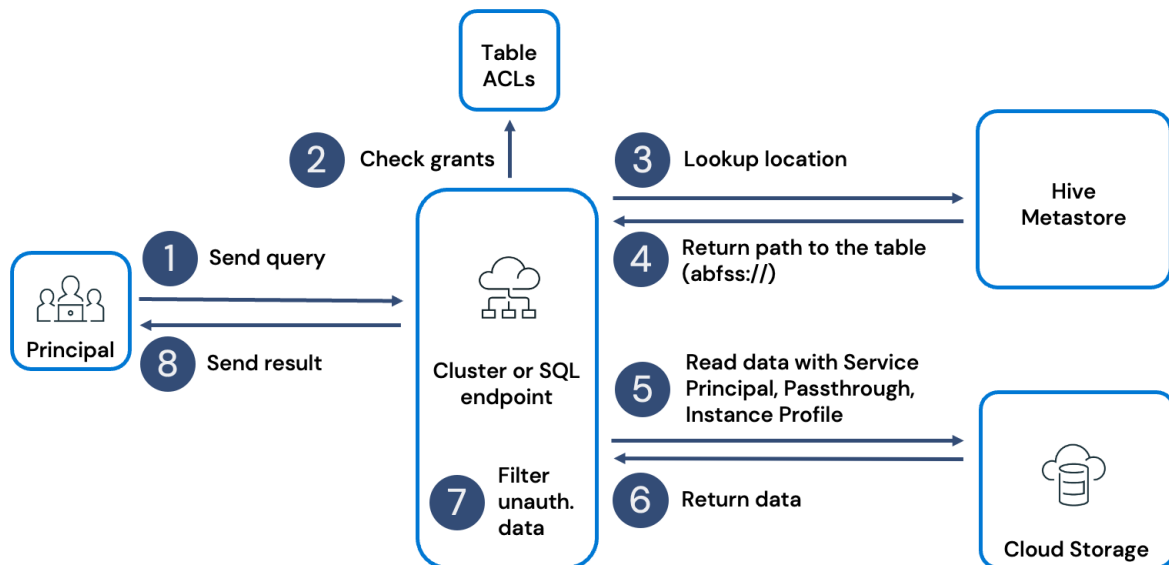
Unity Catalog was designed to streamline governance, but it also maintains compatibility with the legacy Hive metastore. When a workspace is assigned to a Unity Catalog metastore, Hive appears as a special catalog named **hive_metastore**. Tables stored there can still be queried by referencing this catalog in the namespace.

Note

Hive metastore table access control is a legacy data governance model.

The lifecycle of a query against a Hive table differs from Unity Catalog in several ways:

- Hive doesn't provide centralized, fine-grained governance. Instead, administrators often need to manage **service accounts, secrets, or instance profiles** for authentication and authorization.
- Access to data may involve setting up **mount points, passthrough authentication, or storage policies**, each requiring additional configuration.
- Audit logging and access controls are less integrated compared to Unity Catalog, which can lead to inconsistencies in security enforcement.



Step 1: Query submission

A principal (user or service) sends a query to a cluster or SQL endpoint. This is the starting point for all Hive queries.

Step 2: Access check

The cluster checks the query against **table access control lists (ACLs)**. These ACLs determine whether the requesting principal has the right to access the specified table.

Step 3: Location lookup

If access is granted, the cluster consults the **Hive metastore** to find the storage location of the table. The metastore contains metadata about tables, including their schemas and storage paths.

Step 4: Path returned

The Hive metastore returns the **storage path** to the cluster, usually as a URI (for example, `abfss://...`). This path points to the physical location of the table data in cloud storage.

Step 5: Data request

The cluster attempts to read data from the cloud storage. At this stage, it must authenticate using one of several mechanisms, such as a **service principal**, **passthrough authentication**, or an **instance profile**. Administrators often need to configure these credentials in advance, along with secrets and mount points.

Step 6: Data returned

Cloud storage sends the requested data back to the cluster.

Step 7: Enforcement of policies

The cluster enforces any last-mile filtering to remove data the principal shouldn't see. This may involve applying row- or column-level restrictions at query time.

Step 8: Result delivery

The final query result is sent back to the principal who issued the request.

In practice, this means querying Hive involves more manual administrative steps and ongoing maintenance. Unity Catalog reduces this overhead by automating credential handling, providing scoped tokens, and enforcing policies consistently across all objects.

3. Implement access control strategies

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/3-implement-access-control-strategies>

Implement access control strategies

Completed



Unity Catalog uses a privilege model to control access to securable objects such as catalogs, schemas, tables, and views. Grants define *who* (the principal, such as a group or user) can perform *what action* (a privilege like SELECT, MODIFY, or CREATE) on *which object* (such as a table). Privileges can be assigned explicitly at each level of containment or inherited from higher levels like schemas or catalogs. Understanding these patterns is key to designing secure and maintainable permission structures.

On Azure, a Unity Catalog **metastore** is a top-level container for all securable objects and is tightly integrated with Azure Databricks workspaces:

- **Region Bound:** A metastore is created in a specific Azure region and can only be assigned to workspaces in the same region.

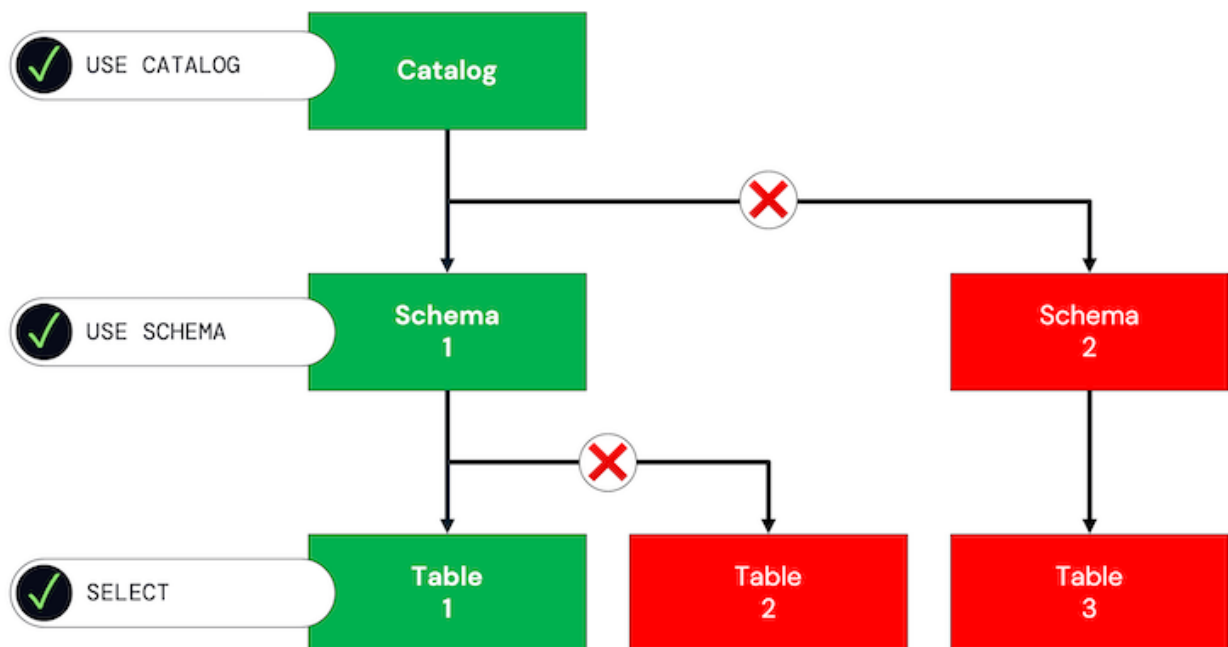
- **One per Workspace:** Each Azure Databricks workspace can be linked to only one metastore at a time.

Because of this region binding, organizations often create multiple metastores (one per region) if they operate across different Azure geographies. Data governance policies should account for this boundary.

Querying a table with explicit permissions at each level

When a user or group needs to query a table, it isn't enough to grant `SELECT` directly on the table. Unity Catalog requires permissions at all levels of containment. That means the grantee needs:

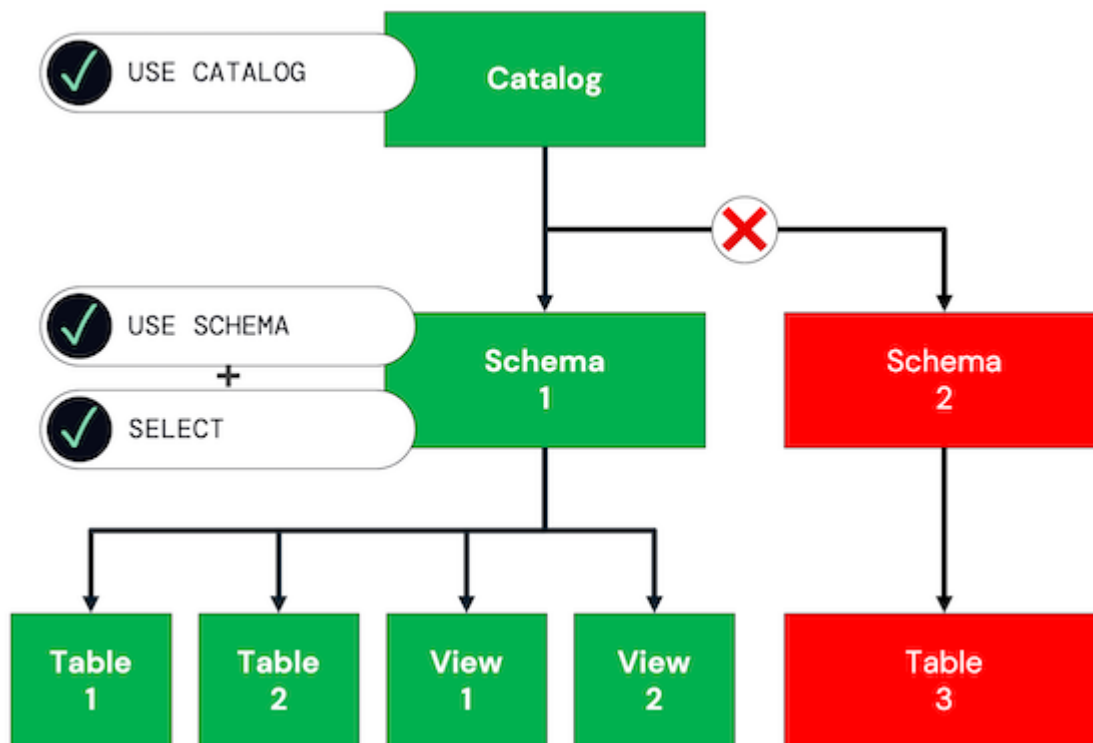
1. **USE CATALOG** on the catalog that contains the schema,
2. **USE SCHEMA** on the schema that contains the table, and
3. **SELECT** on the table itself.



If any of these privileges are missing, the user can't successfully query the table. This approach is called as the *explicit privilege model* because permissions are granted individually at each level. It's inherently secure because access to other objects is never granted unintentionally, but it requires more administrative effort. For example, if there are thousands of tables in a schema, each table must be explicitly granted, which is time-consuming to maintain.

Querying a table with inherited permissions from schema

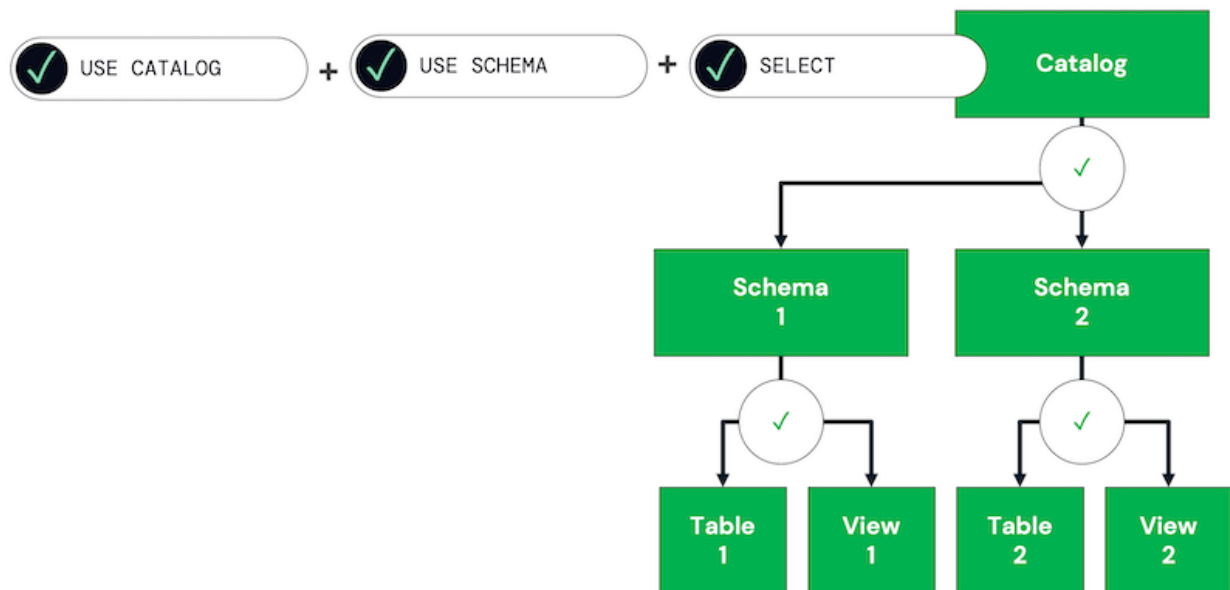
Privilege inheritance reduces administrative overhead by granting permissions at a higher level. Instead of assigning `SELECT` on each table individually, an administrator can grant **SELECT on the schema**. Combined with **USE SCHEMA** and **USE CATALOG** privileges, this automatically gives the grantee `SELECT` on all tables and views within that schema, including any created in the future.



This approach simplifies management, especially when large numbers of tables exist. However, it's less restrictive than the explicit model because all current and future objects within the schema become accessible. Access is still limited to that schema, though—other schemas remain inaccessible unless granted separately.

Querying a table with inherited permissions from catalog

Inheritance can also extend from the catalog level. By granting **USE CATALOG** and **ALL PRIVILEGES** (or **SELECT**) on the catalog, the grantee automatically inherits the necessary permissions on all schemas, tables, and views within that catalog. This includes any schemas or objects created in the future.

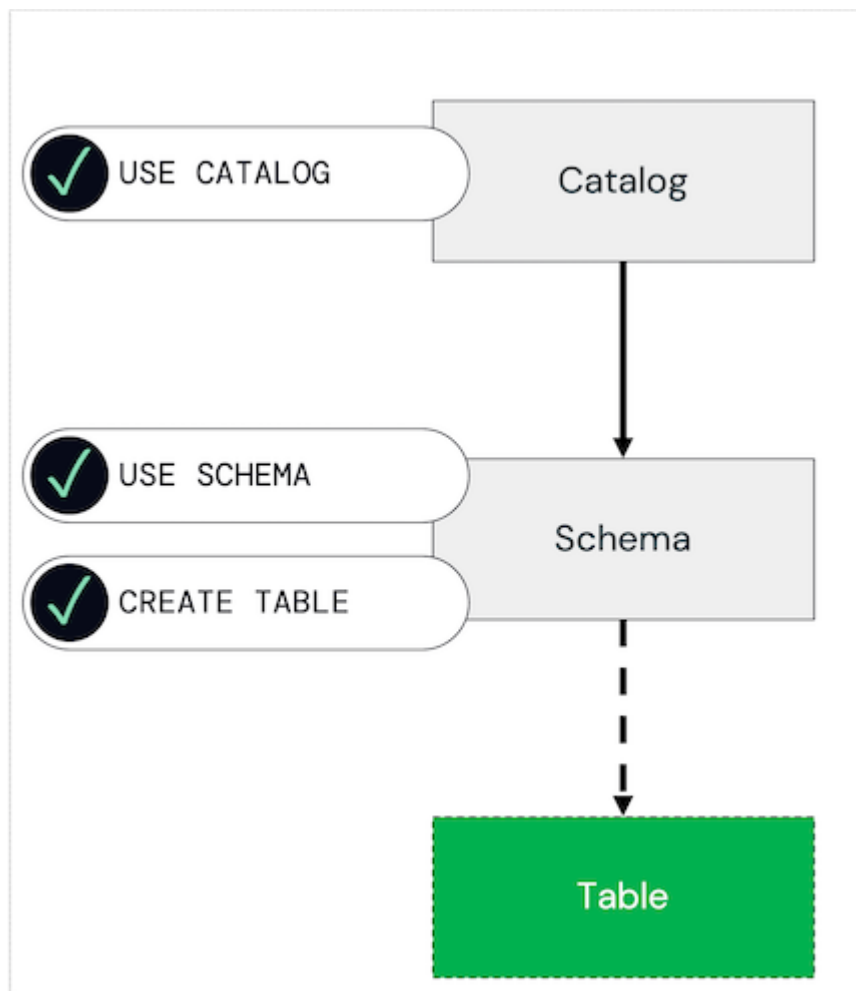


This model is the most permissive because it opens access across the entire catalog. It's convenient when broad access is desired, such as giving a group of analysts the ability to query everything within a specific catalog. However, administrators need to carefully assess whether all objects in the catalog should be exposed in this way, since there's no granularity at the schema or table level when inheriting from the catalog.

Creating a table: explicit at schema

Creating new tables requires more than SELECT permissions. At a minimum, the grantee must have:

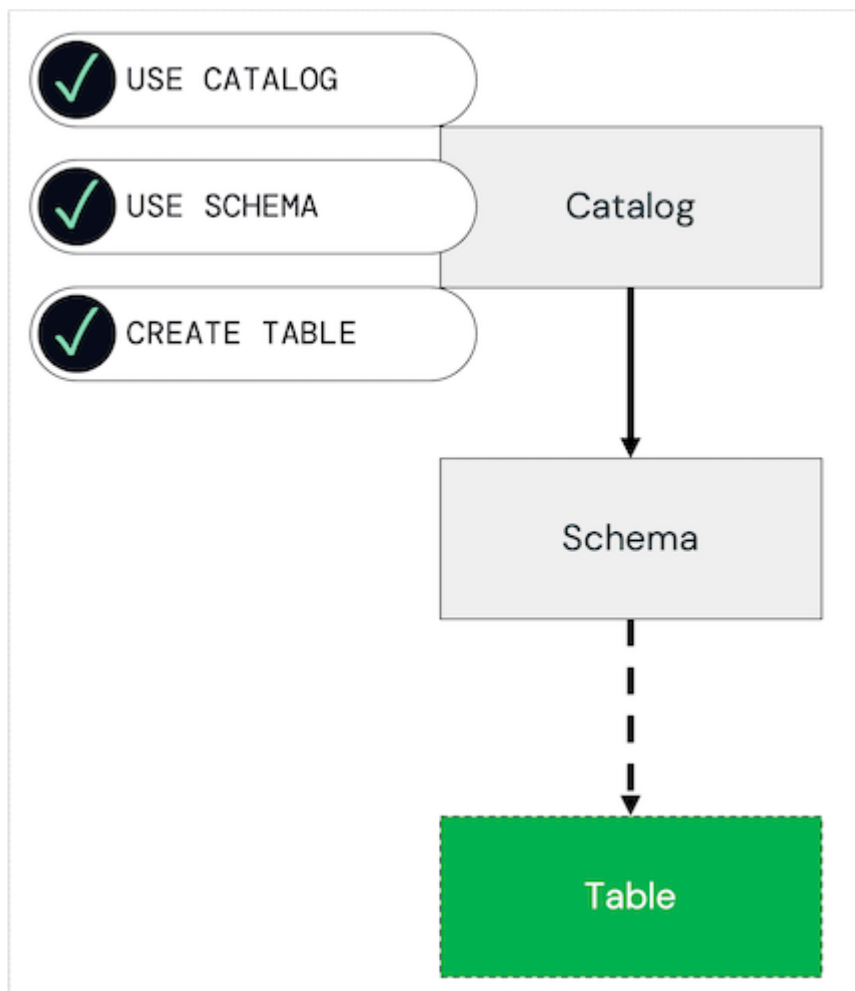
- **USE CATALOG** on the containing catalog,
- **USE SCHEMA** on the target schema, and
- **CREATE TABLE** on that schema.



With these permissions, the grantee can create new tables in that specific schema. This explicit setup restricts table creation to one schema and avoids granting unnecessary privileges elsewhere. It's a common choice when only certain schemas should allow new object creation.

Creating a table: inherited from catalog

In more permissive environments, table creation rights can be inherited from the catalog level. By granting **USE CATALOG**, **USE SCHEMA**, and **CREATE TABLE** at the catalog, the grantee automatically has the ability to create tables in any schema within that catalog, including schemas that may be added later.



This approach is easier to maintain but comes with greater risk, since it enables table creation everywhere within the catalog. It's best suited for catalogs designed for collaborative or exploratory work, where unrestricted creation is acceptable.

Granting permissions with ANSI SQL

Unity Catalog follows the ANSI SQL Data Control Language (DCL) standard for granting and revoking privileges. This means you can manage permissions with familiar statements like **GRANT** and **REVOKE**. For example, if you want to allow the group *analysts* to query a specific table, you can issue:

```
GRANT SELECT ON TABLE sales TO `analysts`;
```

If you want that same group to traverse the containing schema and catalog, you would also need:

```
GRANT USE SCHEMA ON SCHEMA finance TO `analysts`;  
GRANT USE CATALOG ON CATALOG corpdata TO `analysts`;
```

To permit table creation within a schema, you can grant:

```
GRANT CREATE TABLE ON SCHEMA finance TO `analysts`;
```

Permissions can be revoked in a similar way:

```
REVOKE SELECT ON TABLE sales FROM `analysts`;
```

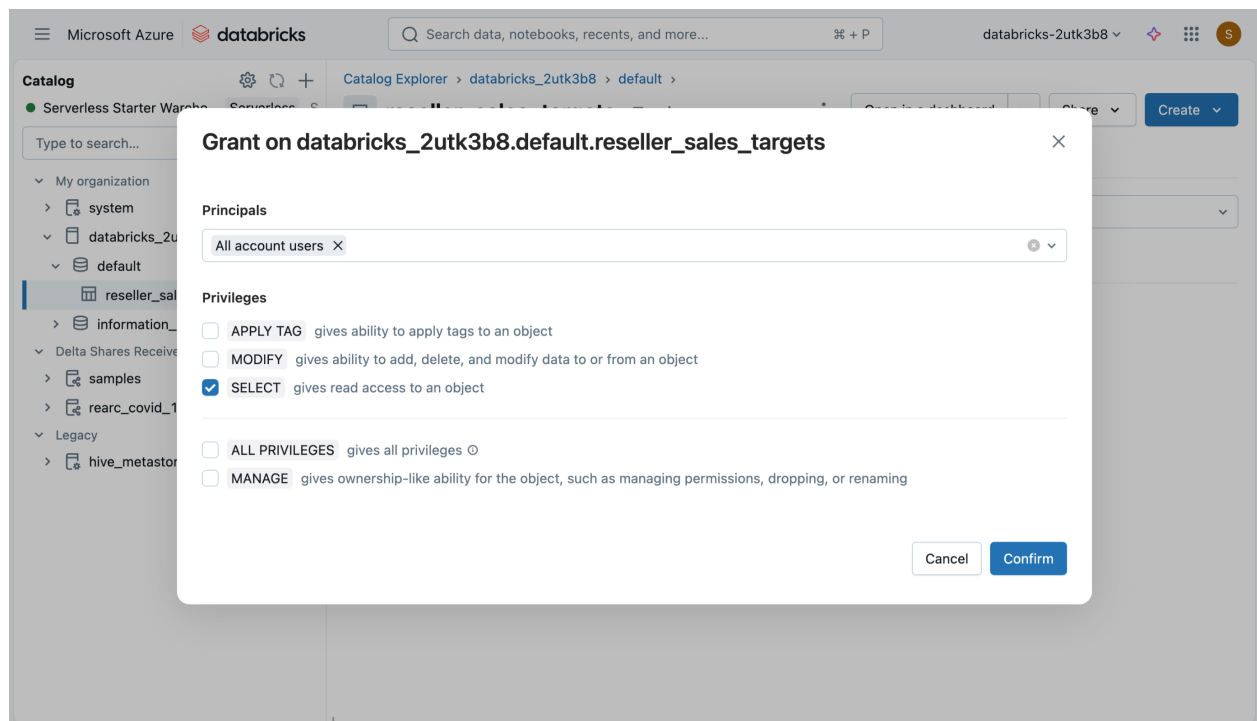
These SQL statements can be run from a Databricks notebook, the SQL editor in the workspace, or in Databricks SQL.

Granting permissions with the Azure Databricks user interface

For those people who prefer a graphical approach, Unity Catalog permissions can also be managed directly in the Azure Databricks UI. In **Catalog Explorer** (previously referred to as Data Explorer), you can navigate through catalogs, schemas, and tables. Each object has a **Permissions** tab where you can add users, groups, or service principals as principals, and assign privileges like SELECT, USE SCHEMA, or ALL PRIVILEGES.

For example, to grant SELECT on a table through the UI:

1. Open **Catalog Explorer**.
2. Navigate to the catalog and schema that contain the target table.
3. Select the table, then go to the **Permissions** tab.
4. Add the group (for example, *analysts*) and assign the **SELECT** privilege.



You can repeat the same process at the schema or catalog level to assign broader permissions. The UI also clearly shows inherited privileges, making it easier to audit access across levels.

Verify granted privileges

After granting privileges, you should verify that the correct permissions are in place. Unity Catalog provides the **SHOW GRANTS** command to display all privileges on an object or for a specific principal.

To see all grants on a table, use the SHOW GRANTS command with the object type and name:

```
-- Show all privileges granted on a table
SHOW GRANTS ON TABLE sales.reporting.monthly_revenue;
```

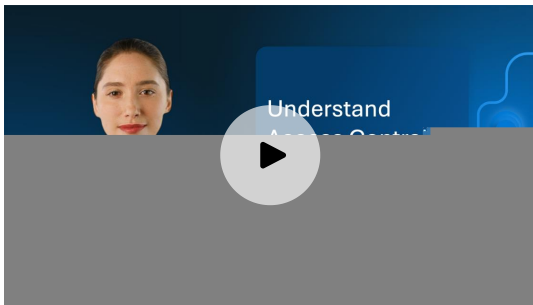
This command returns a list showing each principal, the privileges they hold, and the object those privileges apply to. You can also check grants on schemas, catalogs, or other securable objects using the same pattern.

If you want to see privileges for a specific principal, include the principal name in the command:

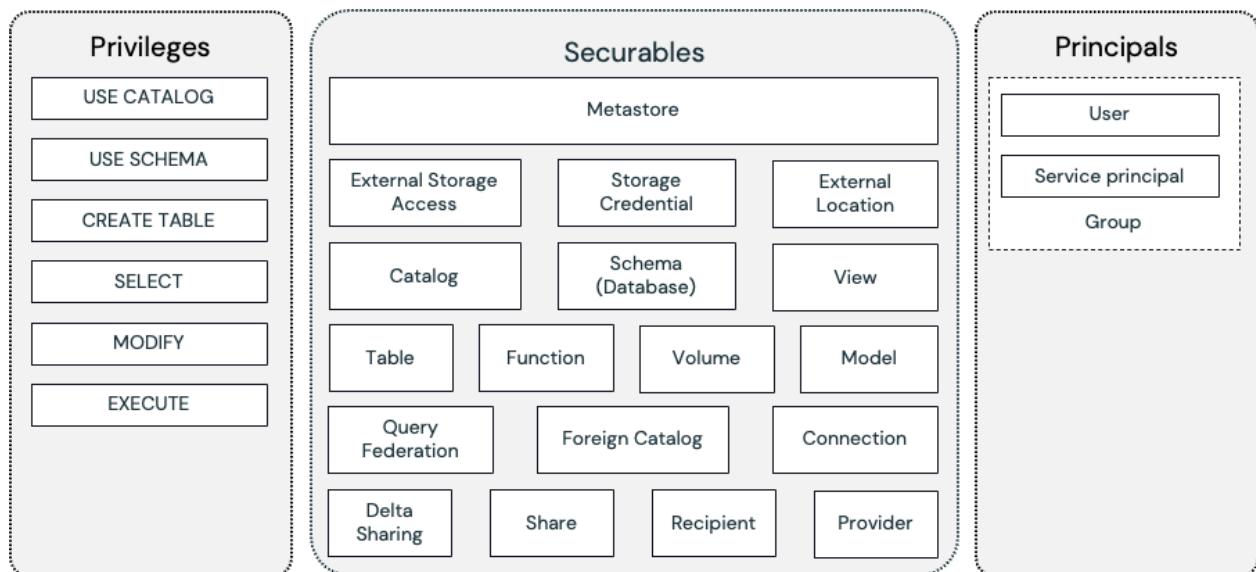
```
-- Show privileges for a specific group
SHOW GRANTS 'finance-team' ON TABLE sales.reporting.monthly_revenue;
```

When reviewing privileges, remember that **inherited grants** might not appear in the output for the specific table. If a user has SELECT granted at the schema or catalog level, that privilege applies to the table through inheritance, but SHOW GRANTS on the table itself might not display it. To get a complete picture, check grants at all levels of the hierarchy.

Understand Access Control Lists (ACLs)



Access Control Lists (ACLs) are the backbone of Unity Catalog's authorization model: they explicitly associate a *principal* with a *privilege* on a *securable object*. To understand how ACLs work, we can break down those three pieces—privileges, securables, principals—and then see how they interact in a typical grant.



Privileges

Privileges are the specific operations or actions that can be performed on securable objects. In Unity Catalog, different kinds of objects support different privilege sets. For example, for a **table** the allowed privileges include `SELECT`, `MODIFY`, `MANAGE`, `ALL PRIVILEGES` (which expands to all applicable privileges) among others. For a **schema**, privileges like `USE SCHEMA`, `CREATE TABLE`, `ALL PRIVILEGES`, and `MANAGE` apply. At the **catalog** level, you see privileges such as `USE CATALOG`, `CREATE SCHEMA`, `BROWSE`, `ALL PRIVILEGES`, and more.

Because not every privilege makes sense for every object (for example, `MODIFY` on a view is meaningless), Unity Catalog limits which privileges are valid on which securable types. The `ALL PRIVILEGES` shorthand is powerful in that it gives "every applicable privilege" on an object, and as new privileges are added over time, the `ALL PRIVILEGES` grant automatically covers those new privileges as well.

Securables

Securables are the objects to which you can grant or revoke privileges. In Unity Catalog, securables are hierarchical: the metastore (or catalog) is at the top, followed by catalogs, then schemas, then tables/views, and other object types such as functions, volumes, external locations, etc.

Because of this hierarchy, privileges can be inherited (if permitted) from higher levels down to lower levels.

Principals

Principals are the "who" in an ACL—the identity to which privileges are granted. In Unity Catalog / Azure Databricks, a principal can be a **user**, a **service principal** (for programmatic identities), or a **group**. Groups are often preferred for manageability: you assign privileges to groups, and then users inherit those privileges by membership, rather than granting to each individual.

The syntax for naming principals in SQL uses backticks when special characters exist (for example, an "@" in a user name or dashes in a service principal ID). For instance:

```
GRANT SELECT ON TABLE t TO `alice@company.com`;
```

Or

```
GRANT SELECT ON TABLE t TO `aaaaaaaa-bbbb-cccc-1111-222222222222`; -- service principal
```

Also worth noting: `account users` is a built-in principal representing all users of the account; you can grant privileges to `account users` (though not to the `users` workspace-local group) in Unity Catalog.

When you put these pieces together, the ACL is simply a statement like: *grant this privilege to that principal on this securable object*. At runtime, when a user tries to perform an operation (say select from a table), the system checks the ACLs along the hierarchy (catalog, schema, table) and evaluates whether the user (via group membership, explicit grants, or inherited grants) has the required combination of permissions to succeed.

Microsoft Entra ID integration

On Azure Databricks, Unity Catalog identities are managed through **Microsoft Entra ID**. This means that all users, service principals, and groups referenced in grants must exist in Microsoft Entra ID. Key points include:

- **Users and Groups:** Unity Catalog does not support workspace-local users or groups. You must provision and manage them in Microsoft Entra ID, and Unity Catalog syncs them into the Azure Databricks account.
- **Service Principals:** Programmatic identities are registered as Microsoft Entra applications and can be granted privileges in Unity Catalog. Credentials are typically stored in Azure Key Vault or accessed through managed identities.
- **Account Users:** The special built-in principal `account users` represents all Microsoft Entra ID users in the Azure Databricks account, and can be used when broad access is acceptable.

Because Unity Catalog relies on Microsoft Entra ID, administrators should coordinate closely with their Microsoft Entra ID administrators to ensure that group memberships and role assignments remain accurate.

4. Understand fine-grained access control

Understand fine-grained access control

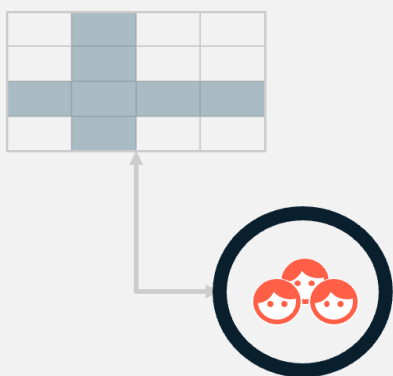
Completed



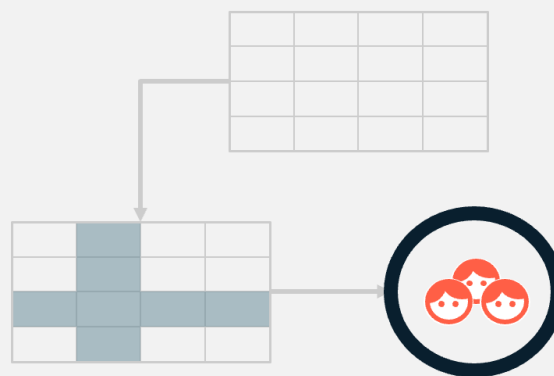
In the previous module, you looked at granting or denying access to entire tables. That works when every authorized user can see every row and column, but in real scenarios you often need to expose only part of a dataset. You might need to hide rows that a user isn't entitled to view, redact or mask specific columns that contain sensitive attributes, or partially transform values so that they remain useful without leaking full detail. Rather than duplicating tables or maintaining multiple extract copies, you can apply fine-grained access control. In Databricks Unity Catalog, you implement fine-grained control using one of two approaches:

- **Row and Column Security:** the preferred, modern method, composed of column masking and row filtering.
- **Dynamic Views:** the earlier pattern that still has valid use cases.

Row and Column Security



Dynamic View



In this unit, you'll learn what each approach is, when to use it, and why one is generally favored today. You'll explore the specific SQL syntax in the next unit, so here we stay at the conceptual level.

Why fine-grained control matters

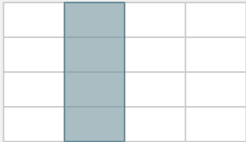
You frequently have a single canonical table that must serve different audiences: application users, data analysts, compliance reviewers, or support staff. Creating separate physical tables for each audience leads to data duplication, stale copies, extra orchestration, and security drift. Views can help, but managing many views—each encoding slightly different masking or filtering—becomes difficult over time. Fine-grained access control lets you keep one authoritative table while tailoring what each principal actually sees at query time.

Typical objectives you'll address:

- **Column masking:** Show a column's presence but hide (or transform) its sensitive values for most users.
- **Row filtering:** Suppress entire records that a user or group shouldn't see.
- **Partial transformation:** Preserve analytical utility (for example, last two digits, domain of an email) without exposing full identifiers.

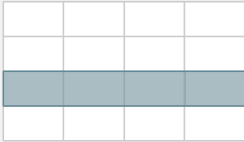
Limit Access to Columns

Omit column values from output



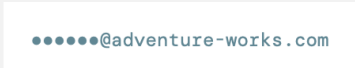
Limit Access to Rows

Omit rows from output



Data Masking

Obscure data



Row and column security

Row and Column Security is implemented through two core concepts:

- **Column masking:** You associate a masking function with a specific column. When a user queries the table, Unity Catalog invokes that function to decide whether to return the real value, a redacted placeholder, or a transformed version. You typically define one function per masked column.
- **Row filtering:** You associate a filter function with the table. That function returns a boolean determining whether each row is visible to the querying principal. One function can govern all row-level rules for that table.

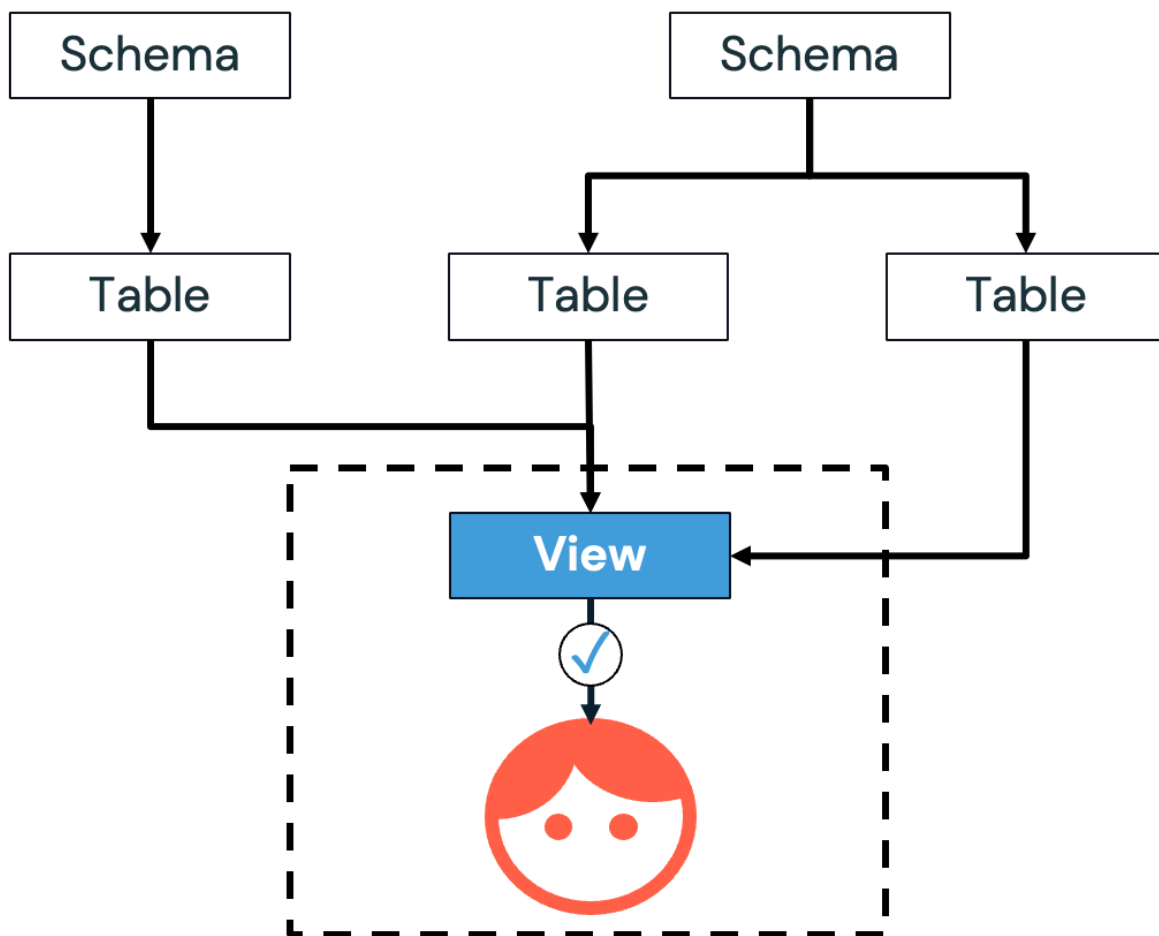
You create user-defined SQL functions that contain the decision logic (for example, checking the querying user's group membership). Because these functions are securable objects just like tables, you also decide where they live (schema/catalog) and who can alter or execute them. You then bind them directly to the table's metadata. At that point, anyone querying the table benefits from a single authoritative definition of masking and filtering—no extra layer required.

Key characteristics you should internalize:

- **Centralized logic:** Rules live alongside the table definition, reducing drift between data and policy.
- **Combinatorial flexibility:** A single set of masking and filtering functions handles all user combinations; you don't create separate artifacts per audience.
- **Performance alignment:** Operating at the table layer avoids the overhead of stacking multiple view abstractions.
- **Operational simplicity:** Fewer named objects to permission, audit, and lifecycle.

Dynamic views

Dynamic Views predate Row and Column Security. These have been with Azure Databricks for some time, and though they're no longer the preferred method for controlling access to rows and columns, they still have their place. You define a view that selects from one or more underlying tables and encodes conditional logic: CASE expressions to mask sensitive columns, predicates to exclude rows, or transformations that partially obfuscate values. Any user with permission on the view can query it without having direct access to the underlying tables, assuming the view owner has that access. This separation lets you shield source objects while presenting a curated projection.



You might create multiple dynamic views over the same table to serve different user cohorts (for example, a support view with partially masked identifiers and an analyst view with additional fields). Each view becomes its own securable object with its own ownership and grants.

Characteristics to note:

- **Level of separation:** Users can be isolated from source tables entirely.
- **Multi-table composition:** You can assemble data from several tables and apply masking/filtering in one place.
- **Read-only exposure:** Views can't themselves be updated; any data modification still requires access to the base tables.
- **Object proliferation risk:** Each audience-specific variant typically requires a new view.

Why row and column security is preferred

You should default to Row and Column Security because:

- **Reduced object sprawl:** One table plus a small set of functions replaces a potentially expanding family of views.
- **Easier maintenance:** Policy changes occur where the data resides, lowering chances of unsynchronized logic across parallel views.
- **Performance advantages:** Enforcement happens at the source, avoiding extra resolution layers and enabling better optimization.
- **Clearer governance:** Auditing and lineage focus on the primary table rather than a web of derivative objects.
- **Flexible combinations:** You avoid creating a new physical or logical artifact for every unique mix of masking and filtering rules.

When dynamic views still make sense

You should consider dynamic views instead (or in addition) when:

- Source tables are intentionally treated as **read-only interfaces**; you want an explicit contract boundary between producers and consumers.
- You need to present a logical dataset assembled from **multiple tables** with consistent masking and filtering logic applied across joins.
- **Organizational policy** requires strict separation such that consumers shouldn't even know the exact structure or existence of underlying tables.
- **Existing legacy patterns** already use dynamic views and a phased migration strategy is necessary.

Comparing row and column security with dynamic views

Here's your comparison table with **Row and Column Security** vs **Dynamic Views** side by side, so you can see the trade-offs and decide which approach best fits your needs.

Dimension	Row and Column Security	Dynamic Views
Maintainability	Minimizes moving parts by enforcing security at the table level.	More objects to manage if multiple views are created.
Flexibility	Limited to what's stored in the table.	Can combine multiple tables directly for flexible compositions.
Object governance	Fewer objects overall; simplifies grants and audits.	More objects to govern; grants may need to be managed at both view and table level.
Read/write patterns	Supports seamless querying and updating with masking or filtering applied at the table.	Typically read-only; write patterns require direct table access.
Abstraction boundary	Exposes storage design more directly.	Provides a presentation layer decoupled from underlying storage.

Typical workflow

Follow these steps to design the right approach and keep your model both secure and maintainable:

1. Identify sensitive columns and row-level segmentation rules (for example, region-based, role-based).
2. Decide if you truly need multi-table assembly or strict abstraction. If not, prefer Row and Column Security.
3. For Row and Column Security: design minimal user-defined functions—one per masked column and optionally one row-filter function—that evaluate the querying principal.
4. Attach these functions to the table so that queries automatically enforce the logic.
5. For Dynamic Views (when justified): craft a single view that applies CASE expressions and WHERE predicates; create additional views only when audiences require materially different exposures.
6. Periodically review whether evolving requirements still justify each dynamic view or whether consolidation into Row and Column Security is now feasible.

Common pitfalls

A few common pitfalls can undermine the effectiveness of a security model. One of the most frequent mistakes is **duplicating tables** for each audience. This approach quickly creates synchronization challenges and complicates governance, as any schema or data change must be applied consistently across all copies. Another recurring issue is the creation of **multiple, nearly identical views** instead of implementing generalizable masking logic. This not only increases maintenance overhead but also makes the system harder to audit and reason about. It's also important to be deliberate about where business logic resides.

Embedding complex transformations inside masking functions can blur responsibilities; these rules typically belong in upstream transformation pipelines, not in security enforcement. Equally critical is

ensuring that the **masking and filtering functions themselves are properly secured**. If they're altered or bypassed, the entire protection model can be weakened. Finally, organizations should be cautious of **over-masking**, which removes more information than necessary and can inadvertently block legitimate analytical tasks that depend on those fields.

5. Implement row filtering and column masking

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/5-implement-row-filtering-column-masking>

Implement row filtering and column masking

Completed



When working with sensitive information in a data lakehouse, you need more than just role-based permissions. Unity Catalog allows you to enforce **fine-grained access control** so that different users can query the same tables but see only the data they're allowed to. Two approaches are commonly used, as explained in the previous unit:

- **Row and Column Security** – restricts which rows and columns a user can see at the table level.
- **Dynamic Views** – encapsulates filtering and masking logic in a view definition that adapts automatically to user context.

Both strategies depend on group membership checks, such as `is_account_group_member()`, to determine whether a user should see protected values.

Row and column security

Row and column security lets you define rules directly on a table. Two operations are available: **column masking** and **row filtering**.

Column masking

Column masking ensures that sensitive values are hidden unless a user belongs to an authorized group. For example, the `c_phone` column in a **customers** table can be masked with a function:

```
CREATE OR REPLACE FUNCTION phone_mask(c_phone STRING)
  RETURN CASE WHEN is_account_group_member('metastore_admins')
    THEN c_phone
    ELSE 'REDACTED PHONE NUMBER'
  END;
```

The masking function is then applied to the column:

```
ALTER TABLE customers
  ALTER COLUMN c_phone
  SET MASK phone_mask;
```

From this point forward, queries against the table return actual phone numbers only to `metastore_admins`. Other users always see `REDACTED PHONE NUMBER`.

Row filtering

Row filtering removes rows from query results for users who aren't authorized. You define a function that returns `true` only when a row should be visible. For example:

```
CREATE OR REPLACE FUNCTION nation_filter(c_nationkey INT)
  RETURN IF(is_account_group_member('admin'), true, c_nationkey = 21);
```

When applied as a row filter:

```
ALTER TABLE customers
  SET ROW FILTER nation_filter ON (c_nationkey);
```

This ensures that non-admins only see rows where `c_nationkey = 21`. Admin users continue to see all rows.

Together, column masking and row filtering provide a way to enforce restrictions directly on the base table.

Dynamic views

An alternative to modifying a table is to create a **dynamic view**. Views allow you to express row and column restrictions in SQL, and every user querying the view automatically sees the appropriate subset of the data.

Consider a **customers_new** table with sensitive information. A dynamic view can redact phone numbers and limit rows in one definition:

```

CREATE OR REPLACE VIEW vw_customers AS
SELECT
  c_custkey,
  c_name,
  c_address,
  c_nationkey,
  CASE
    WHEN is_account_group_member('admins') THEN c_phone
    ELSE 'REDACTED PHONE NUMBER'
  END as c_phone,
  c_acctbal,
  c_mktsegment,
  c_comment
FROM customers_new
WHERE
  CASE WHEN
    is_account_group_member('admins') THEN TRUE
    ELSE c_nationkey = 21
  END;

```

When users query this view, the rules apply automatically:

- Phone numbers are shown only for members of the `admins` group.
- Non-admins see only rows with `c_nationkey = 21`.

Access to the view can then be granted with standard SQL permissions:

```

GRANT SELECT ON VIEW vw_customers TO `account users`;

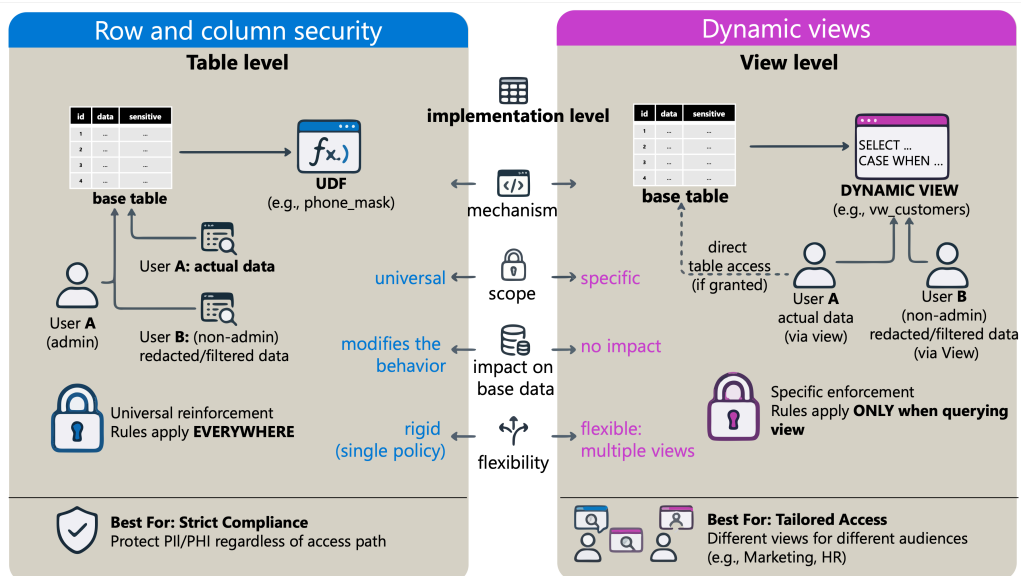
```

The underlying table remains untouched, while the view enforces security logic.

Choosing between the two approaches

Both approaches achieve fine-grained security, but they differ in scope:

- **Row and Column Security** is embedded at the table level and guarantees that restrictions apply everywhere the table is used.
- **Dynamic Views** provide flexibility: you can create multiple views with different rules for different audiences, while keeping the base table unrestricted.



In practice, you use table-level controls when you need strict enforcement, and dynamic views when you want adaptable, shareable abstractions.

6. Access Azure Key Vault secrets

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/6-access-azure-key-vault>

Access Azure Key Vault secrets

Completed

When you work with Azure Databricks, you often need to connect to external data sources, APIs, or services that require authentication credentials. Storing passwords, connection strings, or API keys directly in notebooks creates security risks and makes credential management difficult. Azure Key Vault integration solves this challenge by providing a secure, centralized way to manage secrets in your Databricks workflows.

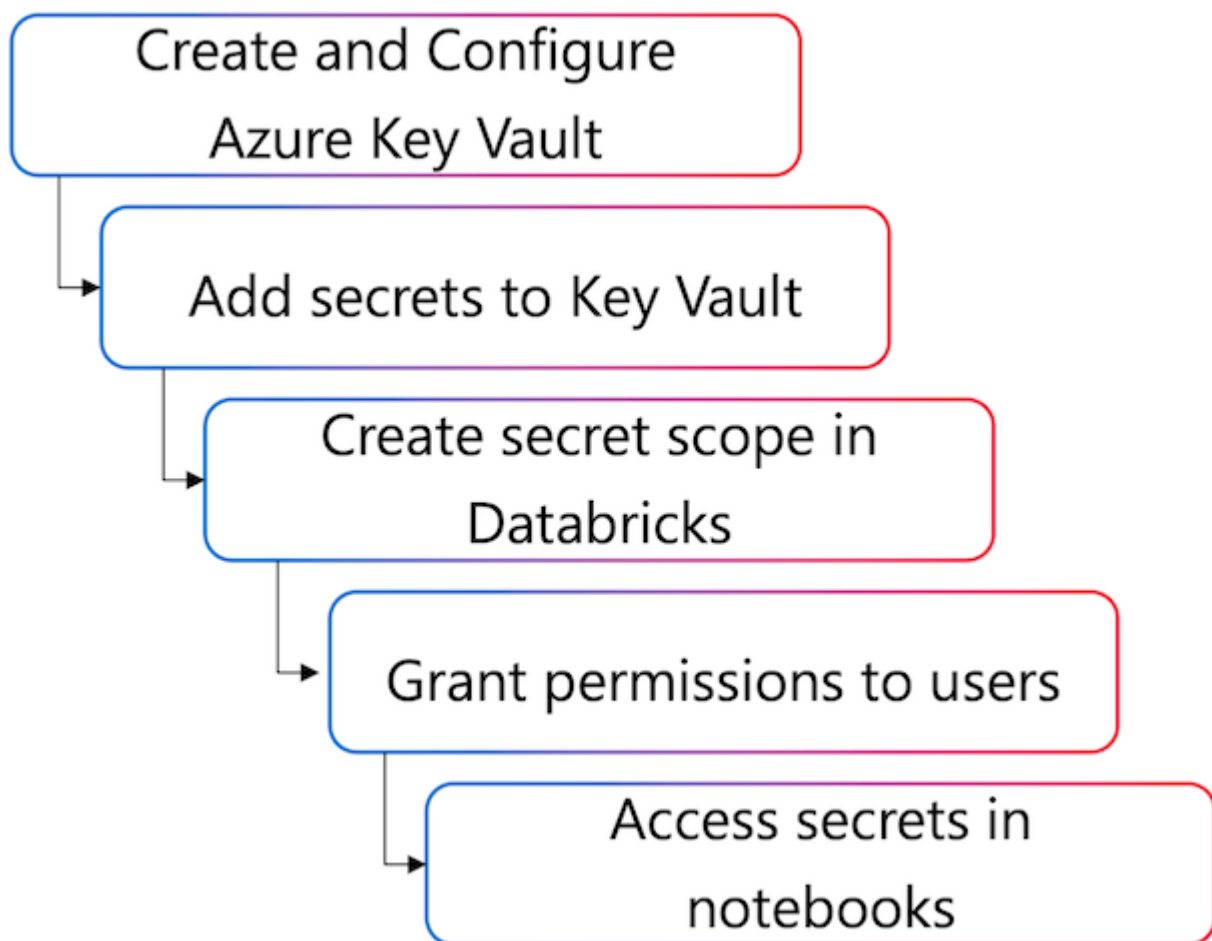


As a data engineer working in Azure, you typically use Azure Key Vault rather than Databricks-backed secret scopes. This approach aligns with enterprise security practices and provides better integration with other Azure services. You can reference secrets stored in Azure Key Vault from your notebooks, jobs, and cluster configurations without exposing sensitive values.

Understanding secret scopes in Azure Databricks

Azure Databricks uses secret scopes to organize and control access to secrets. A secret scope is a named collection of secrets that you reference in your code. Each scope can contain multiple secrets, and you assign permissions at the scope level.

The following diagram shows the overall workflow for using Azure Key Vault-backed secret scopes in Databricks:



Azure Databricks supports two types of secret scopes:

- **Databricks-backed scopes** store secrets in an encrypted database managed by Azure Databricks. You create and manage these secrets using the Databricks CLI or API.
- **Azure Key Vault-backed scopes** provide a read-only interface to secrets stored in an Azure Key Vault instance. You manage the actual secrets in Azure Key Vault, while the scope in Azure

Databricks provides controlled access.

For production workloads, Azure Key Vault-backed scopes offer several advantages. They integrate with your organization's existing Azure security infrastructure. They support network restrictions that Databricks-backed scopes cannot. They eliminate the need to synchronize secrets across multiple systems. When you update a secret in Key Vault, all Azure Databricks workspaces that reference that scope automatically use the updated value after you restart your clusters.

Important

Permissions on Azure Key Vault-backed scopes apply to all secrets in the linked Key Vault instance. If you need to restrict access to specific secrets, create separate Key Vault instances and link them to different secret scopes.

Creating an Azure Key Vault-backed secret scope

Before you create an Azure Key Vault-backed scope, you need an Azure Key Vault instance configured correctly. Your Key Vault must use the **Vault access policy** permission model rather than Azure role-based access control. Under the Key Vault's networking settings, you must allow trusted Microsoft services to bypass the firewall, even if you restrict access to specific virtual networks.

You create the secret scope using a special URL in your Azure Databricks workspace. Navigate to `https://<databricks-instance>#secrets/createScope`, replacing `<databricks-instance>` with your workspace URL. This URL is case-sensitive—the **S** in `createScope` must be uppercase.

When you create the scope, you provide a descriptive name and specify who can manage it. The **Manage Principal** setting determines whether only the creator or all workspace users have **MANAGE** permission on the scope. **MANAGE** permission allows reading secrets, writing new secrets, and controlling who else can access the scope. Restricting management to the creator requires the Premium plan.

[Home](#) / [Create Secret Scope](#)

Create Secret Scope

CancelCreate

A store for secrets that is identified by a name and backed by a specific store type. [Learn more](#)

Scope Name

Manage Principal

Creator



Azure Key Vault

DNS Name

`https://xxx.vault.azure.net/`

Resource ID

`/subscriptions/xxxxxx/...`

You also need two pieces of information from your Azure Key Vault: the **DNS Name** (for example, `https://databrickskv.vault.azure.net/`) and the **Resource ID** (for example, `/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/resourcegroups/databricks-rg/providers/Microsoft.KeyVault/vaults/databrickskv`). You can find both values in the **Properties** section of your Key Vault in the Azure portal.

When you create the scope, Databricks automatically grants the Azure Databricks service principal the **Get** and **List** permissions on your Key Vault. This allows the service to retrieve secret values on behalf of authorized users. Your account must have the Key Vault Contributor, Contributor, or Owner role to complete this configuration.

Note

After creating the scope, you manage the actual secrets in Azure Key Vault, not in Databricks. You use the Azure portal or Azure CLI to add, update, or delete secrets in the Key Vault instance.

Retrieving secrets in notebooks and jobs

Once you create a secret scope, you access secrets in your notebooks using the `dbutils.secrets.get()` function. This function takes two parameters: the scope name and the secret key name. The function returns the secret value, which Azure Databricks automatically redacts in notebook outputs.

```
username = dbutils.secrets.get(scope="jdbc", key="username")
password = dbutils.secrets.get(scope="jdbc", key="password")

df = (spark.read
      .format("jdbc")
      .option("url", "jdbc:sqlserver://server.database.windows.net:1433")
      .option("dbtable", "customers")
      .option("user", username)
      .option("password", password)
      .load()
    )
```

In this example, you retrieve database credentials from a secret scope named `jdbc`. The `username` and `password` variables contain the actual secret values, but if you print these variables or display them in notebook output, Azure Databricks replaces the values with `[REDACTED]`.

This redaction provides protection against accidental exposure, but it doesn't prevent intentional access by authorized users. Any user with permission to run code on the cluster can access secret values programmatically. You control access to secrets by managing permissions on the secret scope and the cluster.

You can also list the secrets available in a scope to discover what keys exist:

```
dbutils.secrets.list("jdbc")
```

This returns metadata about the secrets, including their key names, but not the actual values. This helps you verify that the secrets you need exist in the scope before you reference them.

Tip

Name your secrets descriptively so that their purpose is clear when you list them. For example, use `database-username` instead of `user1`, or `storage-account-key` instead of `key`.

Using the `secret()` function in SQL

In addition to `dbutils.secrets.get()`, you can retrieve secrets directly in SQL queries using the `secret()` function. This function is available in Databricks SQL and Spark SQL, making it convenient when working with SQL notebooks or queries.

```
SELECT secret('jdbc', 'username') AS db_username;
```

You can also use the `secret()` function in SQL statements that configure connections or perform operations:

```
CREATE TABLE customer_data
USING jdbc
OPTIONS (
  url "jdbc:sqlserver://server.database.windows.net:1433",
  dbtable "customers",
  user secret('jdbc', 'username'),
  password secret('jdbc', 'password')
);
```

The `secret()` function provides the same redaction behavior as `dbutils.secrets.get()` —secret values are replaced with `[REDACTED]` in query outputs. For cases where you want to return `NULL` instead of raising an error when a secret doesn't exist, use the `try_secret()` function.

Using secrets in Spark configurations

Sometimes you need to reference secrets in cluster configurations rather than in notebook code. This is useful when you want to set Spark properties or environment variables that multiple notebooks on the cluster can access.

You reference a secret in a **Spark configuration property** using double curly braces:

```
spark.storage.key {{secrets/storage/account-key}}
```

This syntax tells Databricks to retrieve the secret named `account-key` from the scope named `storage` and assign it to the `spark.storage.key` configuration property. You configure this in the **Advanced Options** section when you create or edit a cluster.

In your notebook, you retrieve the value using:

```
storage_key = spark.conf.get("spark.storage.key")
```

Reference secrets in environment variables

You can also reference secrets in environment variables. This is particularly useful for init scripts that run before notebooks execute:

```
STORAGE_KEY={{secrets/storage/account-key}}
```

In an init script, you access this environment variable like any other:

```
if [ -n "$$STORAGE_KEY" ]; then
    echo "Configuring storage with key"
fi
```

Keep in mind that any user with CAN ATTACH TO permissions on the cluster can read environment variables and Spark configuration properties. This means they can access secrets even if they don't have direct permission to read from the secret scope. Use this approach only when the secrets need to be available to all cluster users.

When you update a secret in Azure Key Vault, you must restart your cluster for Azure Databricks to fetch the new value. The cluster caches secret values when it starts, and it doesn't automatically detect changes in Key Vault.

Managing secret scope permissions

Access to secrets is controlled through permissions on the secret scope. When you create a scope, you receive MANAGE permission by default. This allows you to read secrets, add new secrets (in Databricks-backed scopes), and control who else can access the scope.

You assign permissions using the Databricks CLI. Three permission levels exist: **READ** allows retrieving secret values, **WRITE** allows adding or updating secrets (only for Databricks-backed scopes, not Key Vault-backed scopes), and **MANAGE** allows reading secrets and controlling permissions.

Important

For Azure Key Vault-backed scopes, you can only manage who has access to the scope through Databricks ACLs. You cannot add or update secrets through Databricks—secrets must be managed directly in Azure Key Vault. The WRITE permission is not applicable to Key Vault-backed scopes.

To grant the READ permission to a group:

```
databricks secrets put-acl jdbc data-engineers READ
```

This command grants the `data-engineers` group permission to retrieve secrets from the `jdbc` scope. You can grant permissions to individual users by using their email addresses or to service principals using their application IDs.

You can view all permissions on a scope:

```
databricks secrets list-acls jdbc
```

This shows which principals have access and their permission levels. You can also check the permissions for a specific principal:

```
databricks secrets get-acl jdbc data-engineers
```

To remove a principal's access to a scope:

```
databricks secrets delete-acl jdbc data-engineers
```

Note

The ability to restrict MANAGE permission to only the creator (rather than all workspace users) requires the Premium plan. On the Standard plan, you must grant MANAGE permission to all workspace users when creating an Azure Key Vault-backed secret scope.

With these capabilities, you can securely access Azure Key Vault secrets in your Databricks workflows while maintaining control over who can use those secrets. This integration strengthens your security posture and aligns with enterprise governance requirements.

7. Authenticate data access with service principals

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/7-authenticate-data-access-service-principals>

Authenticate data access with service principals

Completed

As a data engineer working with Azure Databricks, you often need to access data stored in Azure Data Lake Storage or other cloud storage services. Service principals provide a secure, automated way to authenticate these data access requests without using personal credentials.

Understanding service principals for data access

A service principal is an identity created for applications and services to access Azure resources. Think of it as an automated user account specifically designed for applications rather than humans.

When your Databricks notebooks or jobs need to read data from Azure storage, they can authenticate using a service principal. This approach offers several advantages over using personal credentials. The authentication happens automatically without manual login, works consistently across different users and environments, and allows precise control over which storage locations the principal can access.

A service principal consists of three key components you'll work with:

- **Application (client) ID:** A unique identifier for the service principal
- **Directory (tenant) ID:** The Microsoft Entra ID tenant where the service principal exists
- **Client secret:** A password-like credential that proves the identity of the service principal

Note

While service principals remain a valid authentication method, Microsoft recommends using **managed identities** for Unity Catalog storage credentials when possible. Managed identities eliminate the need to manage and rotate secrets and can access storage accounts protected by network rules. However, service principals are still useful in scenarios where managed identities aren't available or when you need more explicit control over credentials.

We will cover managed identities in the next unit.

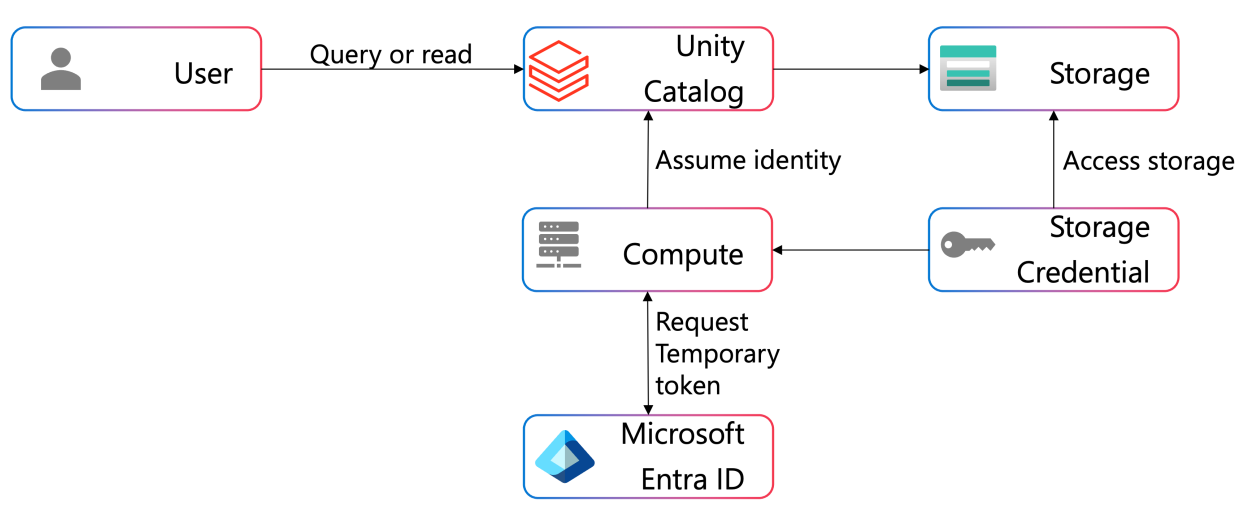
Using service principals with Unity Catalog

Unity Catalog provides a centralized way to manage data access using service principals through storage credentials and external locations.

A **storage credential** in Unity Catalog encapsulates a service principal's authentication details. Instead of embedding credentials directly in your code, you reference the storage credential by name. Unity Catalog handles the authentication behind the scenes, generating temporary tokens that allow your queries to access the underlying storage.

Here's how the workflow operates in practice:

When you query a table or read from an external location governed by Unity Catalog, the system checks which storage credential is associated with that location. Unity Catalog assumes the service principal identity, requests a temporary access token from Microsoft Entra ID, and provides that token to your compute resources. Your cluster or SQL warehouse then uses the token to access the storage directly.



This approach offers important benefits. Your sensitive credentials never appear in notebook code or job configurations. Unity Catalog audit logs record which principal accessed which data and when. You can grant and revoke access to storage locations through Unity Catalog permissions without changing any code.

Important

As a data engineer, you typically won't create the storage credential itself—that's an administrative task. You'll work with existing storage credentials that have been configured by your workspace administrators or metastore admins. Your role focuses on using these credentials through external locations and tables.

To access data through a Unity Catalog-managed external location, you simply query the table or reference the external location path. Unity Catalog handles the authentication automatically:

```
-- Query a table in an external location secured by a service principal
SELECT * FROM catalog_name.schema_name.external_table;
```

No explicit credential configuration is needed in your code. The table's metadata includes the external location reference, which points to a storage credential containing the service principal details.

Using service principals in notebooks

You might encounter scenarios where you need to configure service principal authentication directly in your notebook code. This pattern is common when working with storage that isn't registered in Unity Catalog or when using legacy configurations.

To authenticate to Azure Data Lake Storage using a service principal, you configure Spark session properties. The service principal's client secret should always come from a secure secret store, never hardcoded in your notebook.

Here's how you retrieve a secret and configure Spark to use service principal authentication:

```
# Retrieve the client secret from Databricks secrets
service_credential = dbutils.secrets.get(scope="storage-secrets", key="sp-client-secret")

# Configure Spark to authenticate using the service principal
spark.conf.set("fs.azure.account.auth.type.<storage-account>.dfs.core.windows.net",
               "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider")
spark.conf.set("fs.azure.account.oauth.provider.type.<storage-account>.dfs.core.windows.net",
               "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider")
spark.conf.set("fs.azure.account.oauth2.client.id.<storage-account>.dfs.core.windows.net",
               "<application-id>")
spark.conf.set("fs.azure.account.oauth2.client.secret.<storage-account>.dfs.core.windows.net",
               service_credential)
spark.conf.set("fs.azure.account.oauth2.client.endpoint.<storage-account>.dfs.core.windows.net",
               "https://login.microsoftonline.com/<directory-id>/oauth2/token")
```

With these configurations in place, you can read data using the standard ABFS (Azure Blob File System) protocol:

```
# Read data from ADLS Gen2 using the service principal for authentication
df = spark.read.parquet("abfss://<container>@<storage-account>.dfs.core.windows.net")
```

The authentication happens transparently. Spark uses the OAuth 2.0 client credentials flow to obtain an access token from Microsoft Entra ID, then presents that token when accessing the storage account.

Security considerations and best practices

When working with service principals for data access, several security practices help protect your credentials and data.

Never hardcode secrets in notebooks or code. Client secrets are sensitive credentials that grant access to your data. Always retrieve them from a secure secret store like Azure Key Vault or Databricks secrets. This practice prevents accidental exposure if you share notebooks or commit code to version control.

Understand that secrets require rotation. Unlike managed identities, service principal client secrets have expiration dates and must be rotated periodically. As a data engineer, you should be aware that if a secret expires, your data access will fail. Work with your administrators to understand the rotation schedule and know who to contact if authentication suddenly stops working.

Apply the principle of least privilege. Each service principal should have access only to the specific storage locations and data it needs. If a service principal is compromised, limited permissions reduce the potential impact. When requesting access, specify exactly which storage accounts and containers you need rather than requesting broad permissions.

As you move forward in your work with Unity Catalog, you'll discover that managed identities offer a more streamlined approach to authenticating data access. In the next unit, we'll explore how

managed identities eliminate many of the challenges associated with service principals while providing enhanced security and simplified operations.

8. Authenticate resource access with managed identities

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/8-authenticate-resource-access-managed-identities>

Authenticate resource access with managed identities

Completed

Managed identities provide a secure, automated way to authenticate your Unity Catalog workspaces to storage resources without managing credentials in your code or configuration files.



As a data engineer, you use managed identities to create **storage credentials** in Unity Catalog. These credentials form the foundation for accessing external data through external tables or reading files directly from cloud storage. Unlike traditional authentication methods that require you to create, store, and rotate secrets, managed identities eliminate credential management overhead while providing enterprise-grade security.

Understanding managed identities for Unity Catalog

Managed identities are Azure resources that provide an identity for your applications when connecting to services that support Microsoft Entra ID authentication. Azure manages the lifecycle of these identities automatically, removing the burden of credential rotation and secret management from your team.

With Unity Catalog, you use managed identities through an **Azure Databricks access connector**. This first-party Azure resource acts as a bridge between your managed identity and your Azure Databricks account. The access connector can contain either a **system-assigned managed identity**

that Azure creates automatically, or one or more **user-assigned managed identities** that you create and manage separately.

Two primary use cases exist for managed identities in Unity Catalog. First, you use them to connect to the metastore's root storage account where Unity Catalog stores managed tables. Second, you use them to access external storage accounts for reading files or creating external tables. Both scenarios use the same configuration approach but serve different purposes in your data architecture.

Note

System-assigned managed identities are tied to the lifecycle of the access connector. If you delete the access connector, Azure automatically deletes the system-assigned managed identity. User-assigned managed identities persist independently and can be attached to multiple access connectors.

Creating storage credentials with managed identities

Storage credentials in Unity Catalog encapsulate the authentication information needed to access cloud storage. When you create a storage credential using a managed identity, you establish a secure connection pathway that Unity Catalog can use to access data on behalf of authorized users.

The process begins in the Azure portal where you create an access connector for Azure Databricks. You select the same Azure region as your storage account to minimize latency and avoid cross-region data transfer costs. During creation, you choose whether to enable a system-assigned managed identity or attach user-assigned managed identities. The resource receives a unique identifier in the format `/subscriptions/{subscription-id}/resourceGroups/{resource-group}/providers/Microsoft.Databricks/accessConnectors/{connector-name}`.

Create an Access Connector for Azure Databricks ...

Basics Tags Managed Identity Review + create

System assigned managed identity

Enable system assigned identity to grant the resource access to other existing resources.

Status ⓘ

☐ Off

☒ On

User assigned managed identity

Add user assigned identities to grant the resource access to other existing resources.

 Add

 Remove

Name	↑↓	Resource group	↑↓	Subscription	↑↓
No user assigned managed identities assigned to this resource. Select 'Add' to add more.					

After creating the access connector, you grant the managed identity permissions on your Azure Data Lake Storage account. The **Storage Blob Data Contributor** role provides read and write access to blob data, which is typically what you need for Unity Catalog operations. You assign this role at either the storage account level for broad access or at the container level for more granular control.

For optimal performance, you should also grant the **Storage Queue Data Contributor** role to enable file event notifications. This configuration allows Azure Databricks to subscribe to change notifications from your storage account, making file processing operations more efficient. When files are added or modified in your storage containers, Unity Catalog receives immediate notifications rather than scanning for changes.

With the Azure configuration complete, you create the storage credential in Unity Catalog. In Catalog Explorer, you navigate to the Credentials tab and create a new storage credential. You select **Azure Managed Identity** as the credential type and provide the access connector's resource ID. If you're using a user-assigned managed identity, you also include its resource ID. The storage credential receives a unique name that you reference when creating external locations.

Create a new credential ✕

A storage credential represents an authentication and authorization mechanism for accessing data stored on your cloud tenant. [Learn more](#)

☒ Storage Credential ☐ Service Credential ⓘ

Credential Type*

Azure Managed Identity

Credential name*

my-credential

Access connector ID [Learn more](#) *

/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e/resourceGroups/databricks-rg-da1

User assigned managed identity ID (optional)

Comment

> Advanced Options

Cancel

Create

Important

To create storage credentials, you need the **CREATE STORAGE CREDENTIAL** privilege on the Unity Catalog metastore. Account admins and metastore admins have this privilege by default. Without this privilege, you can't create new storage credentials but can still use existing ones that you have permissions to access.

Accessing external storage through storage credentials

Storage credentials work together with **external locations** to provide governed access to your cloud storage. An external location combines a storage credential with a specific cloud storage path, creating a managed access point that you can grant permissions on.

When you create an external location, you specify a path in your Azure Data Lake Storage account using the `abfss://` protocol. For example, `abfss://data@mystorageaccount.dfs.core.windows.net/raw/` points to the raw folder in the data container. You associate this path with your storage credential, which provides the authentication mechanism. Users who have privileges on the external location can read from or write to that path, depending on their specific permissions.

This architecture provides several benefits for data engineering teams. You can create multiple external locations that use the same storage credential but point to different paths, allowing you to organize access to different datasets or environments. You can grant different sets of users access to different external locations, even when they all use the same underlying storage account. You can mark storage credentials as read-only, which prevents creation of external locations that allow write access.

External locations also support **workspace binding**, which restricts access to specific workspaces in your metastore. This feature is valuable when you use workspaces to separate production and development environments. An external location pointing to production data can be bound to only the production workspace, ensuring that development workspaces can't accidentally access or modify production data. You can also bind storage credentials themselves to specific workspaces, controlling which workspaces can use those credentials to create new external locations.

```
# Create an external table using a managed identity-backed external location
CREATE EXTERNAL TABLE sales_data
USING DELTA
LOCATION 'abfss://data@mystorageaccount.dfs.core.windows.net/sales/';

# Query the external table
SELECT * FROM sales_data WHERE sale_date >= '2024-01-01';
```

When your Unity Catalog workspace accesses data through a storage credential, the managed identity authenticates to Azure Data Lake Storage. The workspace doesn't need to know the actual credentials—it simply presents the managed identity. Azure verifies the identity and checks whether the assigned roles allow the requested operation. This happens transparently in the background, requiring no additional code or configuration in your notebooks.

Tip

When you work with network-restricted storage accounts, ensure your workspace's virtual network can reach the storage account. Managed identities handle authentication, but network connectivity must already be established through private endpoints or service endpoints.

9. Exercise - Secure Unity Catalog Objects

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/9-exercise-secure-unity-catalog-objects>

Exercise - Secure Unity Catalog Objects

Completed

Now it's your chance to secure Unity Catalog objects. In this lab, you secure Unity Catalog objects in Azure Databricks by granting fine-grained access control to a Databricks group, applying row filters to restrict customer data by region, and masking PII email addresses using column mask functions. You also create an Azure Key Vault-backed secret scope and retrieve secrets securely inside a notebook, so sensitive credentials are never exposed in code.

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](#)

10. Module assessment

<https://learn.microsoft.com/en-us/training/modules/secure-unity-catalog-objects/10-knowledge-check>

Module assessment

Completed

11. Summary

Summary

Completed

Security in **Unity Catalog** extends beyond simple permission checks to encompass the entire lifecycle of data access. Throughout this module, you explored how Unity Catalog enforces security at multiple layers—from validating permissions and issuing **scoped tokens** during query execution, to controlling table and schema access through **inherited and explicit grants**, to protecting sensitive data through **fine-grained row and column filtering**.

You learned practical approaches for managing credentials and authentication. **Azure Key Vault** integration lets you retrieve secrets securely without exposing sensitive values in your code. **Service principals** provide programmatic access to storage resources with explicit credential control. **Managed identities** eliminate credential management overhead entirely while enabling access to network-protected storage accounts. Each authentication method serves specific scenarios, and choosing the right approach depends on your security requirements and operational constraints.

The access control strategies you encountered—from hierarchical permissions that simplify administration to fine-grained masking functions that protect individual columns—form a comprehensive security framework. **Row and Column Security** offers centralized enforcement with minimal object sprawl, while **dynamic views** provide flexibility when you need to compose data from multiple sources. Understanding when to apply each pattern helps you build solutions that balance governance with usability.

As you implement these security practices in your own environments, start with clear requirements for who should access what data. Design your permission model to scale with your organization, using groups and inherited permissions where appropriate. Test your row and column filtering logic thoroughly to ensure users see exactly what they should. Regularly review your storage credentials and secret scope permissions to maintain a strong security posture. Unity Catalog provides the tools—your role is to apply them thoughtfully to protect your data assets while enabling productive analytics workflows.