Building off the audit event logs, integrated monitoring (5) allows the governance platform to notify when things don't seem right, or when resources are out of compliance (via tracing and active monitoring). Using techniques similar to the audit logging, performance and runtime metrics and other statistics can be captured during job runs to enable data quality and system-level observability.

Connecting the dots for true system-level observability requires (7) data asset lineage (at least from the catalog -> schema -> table), including the tracking of ownership (who owns what) and of where (the location and point in the end-to-end lineage) and how the data is accessed, transformed, and otherwise "used." Understanding the objective data quality for mission-critical tables and the general performance of active data pipelines provides a bird's-eye view over the complete behavior of the lakehouse. It is also worth mentioning that the same metrics used for auditing, performance, and observability can be reused to track total cost-of-ownership insights, which are especially useful when looking to reduce spend or deprecate tables.

We dedicate Chapter 14 to data sharing (via Delta Sharing), but without the integrated services discussed here (1, 2, 3, 4, 5), including data lineage (7), it becomes much more complicated to effectively share single-source-of-truth data in a cost-effective and governed way.

Last but not least, the icing on the cake of all our hard work is the ability to tie together "all we know about our data" to provide powerful data discovery (8), which is arguably one of the most widespread issues given the sprawl of data across a vast number of silos, platforms, and systems and services.

The core components of lakehouse governance include access controls, lakehouse data catalogs and metastores, elastic data management, audit logging, monitoring, data sharing, data lineage, and data discovery. It is common to see all of these components neatly bundled under a single catalog solution. For now we will discuss what each component is and then dive deeper into each facet:

Access controls (1)
: Access controls provide capabilities to secure and govern the data assets within a lakehouse through leakproof abstractions—in the case of Delta Lake table access, this means there is zero direct access to the underlying storage. Without the ability to identify a user or service, there would be no way to approve or deny access or to authorize permissions to create, read (view), write (insert), update (or upsert), or delete. It would be the Wild West.

Lakehouse data catalogs and metastores (2)
: This component enables capabilities to find and view data assets, including catalogs, databases (schemas), tables, views, volumes, and files, and to govern

permissions for access to and control of these resources. The catalog[1] provides critical metadata about each resource—defining where it resides (the location in cloud storage, for example), as well as the associated owner and specific data relating to the type of resource, such as the columns, constraints, and table properties for each table (like we explored in Chapter 5), and metadata specific to the database (`dbproperties`) containing a set of managed or external tables.

So access controls and data catalogs go hand in hand, as we can't have one without the other.

Elastic data management (3)
: The last core component of the lakehouse architecture is the data lake. We know by now that the Delta protocol aids in providing schema enforcement and evolution capabilities (as seen in Chapter 5), and that by having invariants on the table level, thanks to the Delta protocol, we reduce the complexity of managing data. The data lake provides elastic scalability to the databases and tables contained within our metastore and made available broadly via our data catalog. And as we will learn soon, identity and access management plays a key role in governing data assets securely.

Together, a strong foundational model can be constructed to power the lakehouse, paving the way for additional critical capabilities, including audit services and comprehensive monitoring of access and the generation of insights on data operations and actions:

Audit logging (4)
: Audit logging can be as simple as capturing changes in the behavior of the lakehouse—for example, a change to a role or policy affecting which identities can execute critical operations such as create or delete on highly controlled resources like catalogs, databases, and tables. Another common use case is logging when critical changes take place, such as an `ALTER TABLE` operation on a table (recording the table version of the operation), or when a table is *deleted*, *truncated*, or even *dropped*. Last, it is also wise to capture when a job fails due to *failed permissions*. The reason for capturing access failure is that it helps to surface which workflows (jobs, pipelines) are attempting to access data that may contain highly sensitive data (and therefore access is blocked for the right reasons); the same process can capture when permissions are missing (could have been revoked for other reasons), and this information can help get a broken job back on track more quickly.

1 The term *data catalog* can mean a metastore like Hive, or it can also encapsulate a full "enterprise" data catalog. This chapter caters to the engineering side of the house, and so we won't be discussing the integration of "data catalogs" for use by nonengineering personas in a typical enterprise.

Audit logs can be stored directly in Delta Lake tables to simplify their integration into more robust security-based workflows and to provide the data source to power active monitoring.

Monitoring (5)
: Capturing the behavior of the lakehouse through the lens of audit logs (for security purposes) and at the catalog, database, and table levels (for engineers, analysts, and scientists) simply provides a recording (timeseries) of metrics or events. Assuming that the audit logs are stored in Delta Lake, and that data asset changelogs are also stored in Delta Lake, these data sources can be used to create active monitoring solutions.

  Monitoring requires you to aggregate the metrics and transform them to generate key performance indicators (KPIs), and to convert events (audit events) into metrics (KPIs) to generate insights. Each KPI provides a measurement that can be used to understand trends within the lakehouse or on a specific data asset and is critical for sounding the alarm (via alerting or paging) or to providing a central communication channel for teams. A good place to begin when monitoring audit level events is with the access frequency to specific data assets. The same access data can also be used to surface when a data asset is popular, infrequently accessed, or never accessed for read or write.

Unified data quality metrics, access and permissions history, and system-wide event tracking come together to act like a flight recorder observing changes with respect to a data asset—stitching important historical moments in time together with the state of the many systems and services in the governance stack. Without proper monitoring and audit logging, advanced capabilities like read-only data sharing or zero-copy shares and data lineage recording simply wouldn't be as powerful:

Data sharing and zero-copy sharing (6)
: We were first introduced to the concept of data sharing in Chapter 9 and will spend the entirety of Chapter 14 looking at data sharing with the Delta Sharing Protocol. Data sharing is a complicated component of lakehouse governance, as it requires operational maturity to first establish a high-trust data ecosystem. When we share data with users and services outside of our control, it costs less and reduces the data management overhead *only if* the data can be read (in place) without requiring any export out of our lakehouse. The Delta Sharing protocol therefore requires the foundation (1, 2, 3, 4, and 5, and really 7 as well) to be in place, since the addition of managing shares and recipients is just an extension of the internal access management paradigm.

Data lineage (7)
: Data lineage can be active or passive. *Active lineage* is automatically generated during the runtime of a data application. This can be done by utilizing a framework that allows you to record the upstream sources through the use of

additional job-level metadata; that will automatically create and maintain the downstream resource metadata simply by successfully running the job. Unlike active lineage, *passive lineage* tells the story of what was and what could be. This means that the lineage is registered and recorded offline and isn't directly modified or synchronized when a job is run.

Each data application (pipeline, workflow, streaming, or batch) takes data from one or more sources (via reads), transforms the data, and sends it (via writes) to other locations (tables) inside or outside the lakehouse, or on to stream processors such as Apache Kafka or Pulsar. Using what we know about each data application, we can construct the directional lineage graph (DLAG) from each active application and use the lineage data to run downstream impact analysis when outages or data quality issues are identified in a source. We will cover data lineage more broadly in Chapter 13.

We capture data about the observable state changes, operations, and actions for our important data assets in order to create a history of what has occurred. This information is useful for managing compliance audits (GDPR, CCPA, and others), identifying risk, tracking data quality over time, and taking action if expectations diverge, and even for automating alerting to pinpoint data outages.

Last but not least is the addition of data discovery:

Data discovery (8)

Data discovery is used differently depending on the personas within the enterprise and is commonly split into *engineering-specific* data discovery and *business-based* data discovery.

The first set of capabilities is more of a schema repository for data assets (databases, tables, *queries, *dashboards, *monitors, and *alerts) inside the lakehouse that is targeted toward engineers, scientists, and analysts. The second is more of an organizational search engine for data that reaches across many data sources and suborganizations. The common persona for this second data discovery engine is targeted more toward business users who are looking for data that is not yet residing within the lakehouse.

Data discovery becomes an essential component to ensure that different personas can quickly identify the best starting point for their work—without the need to create a long series of meetings or complicated coordinated efforts. By reusing insights for access frequency (from auditing [4]) on specific tables, as well as a crowd-sourced usability score (1–5), a popularity score (or data NPS score) can also be defined to help surface data assets by usage and general usability.

As any data engineer can tell you, all sorts of issues can and will occur at runtime—for example, access to data assets can be revoked (for the right or wrong reasons), causing data pipelines to fail or become degraded. Tables can be deprecated, go into

read-only mode where they are no longer being updated, or even accidentally be deleted (without the proper governance checks and balances). Without a clear history of changes to permissions, table state, or established patterns for communicating state changes or degradations to data stakeholders, trust degrades.

Trust is easily broken without clear lines of communication. Data governance is one way to maintain a high-trust environment, with reliable tools and services that go beyond security and compliance to help connect disparate data teams working to solve complex problems.

This chapter skips over regional data governance and compliance regulations, as well as design patterns for managing cross-region data access. These topics are outside the scope of the book but remain a critical central tenet of any complete governance solution. However, the topics discussed here will support your work to achieve compliance. For those looking to dive deeper into the topic of data governance, please take a look at *The Enterprise Data Catalog* by Ole Olesen-Bagneux (O'Reilly).

The Emergence of Data Governance

Data governance is defined as an umbrella that brings together various principles and practices, as well as tools and workflows, to govern an organization's data assets throughout their complete life cycle. The life cycle of data encapsulates the full end-to-end journey from creation to deletion, including all transformations and any access and utilization of the data at any point in time along the way (within the data's existence).

Consider the life cycle of our data through the lens of a Delta table:

- The conduit for our data is the *table* itself.
- Each table provides a container that stores a bounded or unbounded set of data over time alongside a transaction log of the *who*, *what*, *where*, and *how* of each change made to the table.
- Tables don't just blink into and out of existence. Each table must first be created, rows must be inserted, read, modified, or deleted, and the table must also be deleted (dropped) to complete the full journey.

To expand the scope of table-level data life cycle management, the simple diagram in Figure 12-2 provides a lens into common steps, from data creation to archiving and ultimately to destruction.

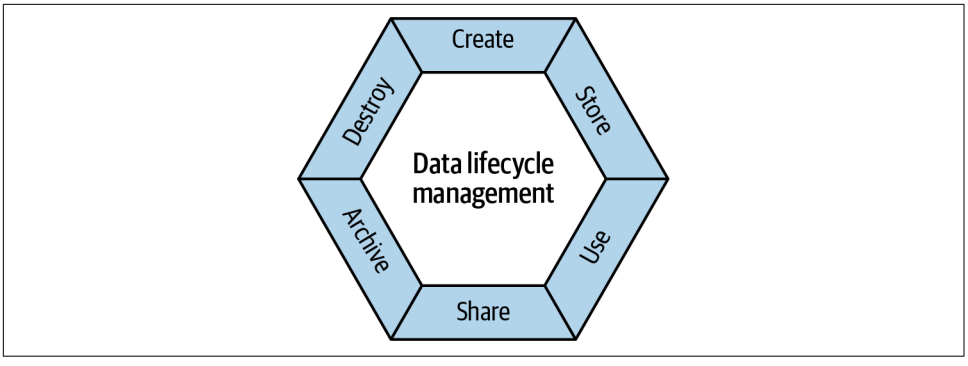

Figure 12-2. Data life cycle management

Similar to other common cycles, such as the software development cycle, the common data life cycle starts with (1) creation and continues to (2) storage, (3) usage, (4) sharing, (5) archiving, and ultimately (6) destruction. This life cycle encapsulates a complete history of actions and operations (a timeline) occurring at the resource level.

These observable moments in time are critical for the purposes of data governance, as well as for the maintenance and usability of the table from an engineering perspective. Each table is a *governable resource* referred to as a data asset.

It is important to consider the use of the term *asset* here. A data asset (*table*) is directly owned, managed, and governed by a person (or team) representing an organization. The organization in turn provides the funding to manage the data asset and to pay the responsible parties across engineering, product, security, privacy, and governance.

As a rule of thumb, data assets should be maintained only for as long as they are still providing value. *Data life cycle management* begins to make more sense when we think of data as existing only until it is no longer useful.

We learned about the medallion architecture for data quality in Chapter 9. This novel design pattern introduced the three-tiered approach for data refinement, from bronze to silver and into gold. This architecture plays a practical role when we're thinking about managing the life cycle of our data assets over time and when we're considering how long to retain data at a specific tier.

Aided by Figure 12-3, we can visualize the value of data assets as they are refined over time and across the logical data quality boundaries represented by bronze, silver, and gold.

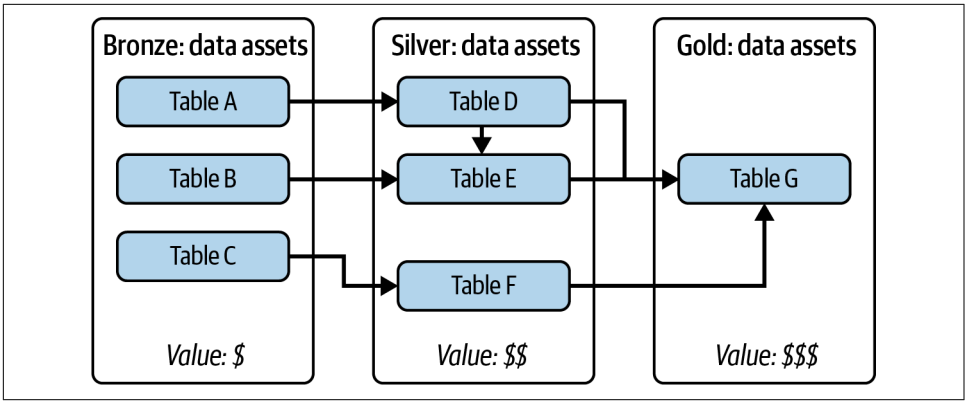

Figure 12-3. The value of our data assets increases as they are refined from bronze to silver and from silver to gold. The medallion architecture is a helpful framework when considering how long to retain data and, more specifically, which tables to retain at which point in the lineage (from bronze to gold).

Figure 12-3 shows the source tables and lineage of transformations for a curated *data product* named *(table G)*. Working backward from the gold data assets, we see that there is a decrease in the value of the tables as we retrace the lineage back through the silver tier (D–F), concluding with our bronze data assets (A–C). *Why is the single table worth more conceptually than the collection of the prior six tables?*

Simply put, the complexity to build, manage, monitor, and maintain the collection of data asset dependencies for table G represents a higher cost than that of the individual parts. Consider that the raw data represented by the bronze data assets (A–C) *is expected to survive only as long as necessary* in order to be accessed and further refined, joined with, or generally utilized by the direct downstream data consumers (D–F), and that the same expectations are in turn made of our silver-tier data assets by the gold tier—they must exist only as long as they are needed,[2] and they must provide a simplification and general increase in data quality the further down the lineage chain they go.

A helpful way of thinking about the end-to-end lineage is through the lens of data products.

2 Drawing a line between data value and data hoarding is difficult. If there is value yet to be discovered, then I would suggest keeping that data in bronze, or archiving it for a later point in time.

For data that isn't being refined through the traditional medallion architecture, for example, you may have Salesforce data being ingested into the lakehouse. This data is already fit for purpose and is of high quality, so it is justifiable to say that that data is ingested directly into your curated layer (gold). In most cases, as your data becomes more refined, it adds more value to the enterprise.

Data Products and Their Relationship to Data Assets

The term *data product* represents the *code*, *data*, and *metadata*, as well as the *logical infrastructure* required to build, produce, and manage a given curated data product. Figure 12-4 shows in detail the intersection between code, data, and the data about said data (metadata), as well as the infrastructure to run and serve up a data product.

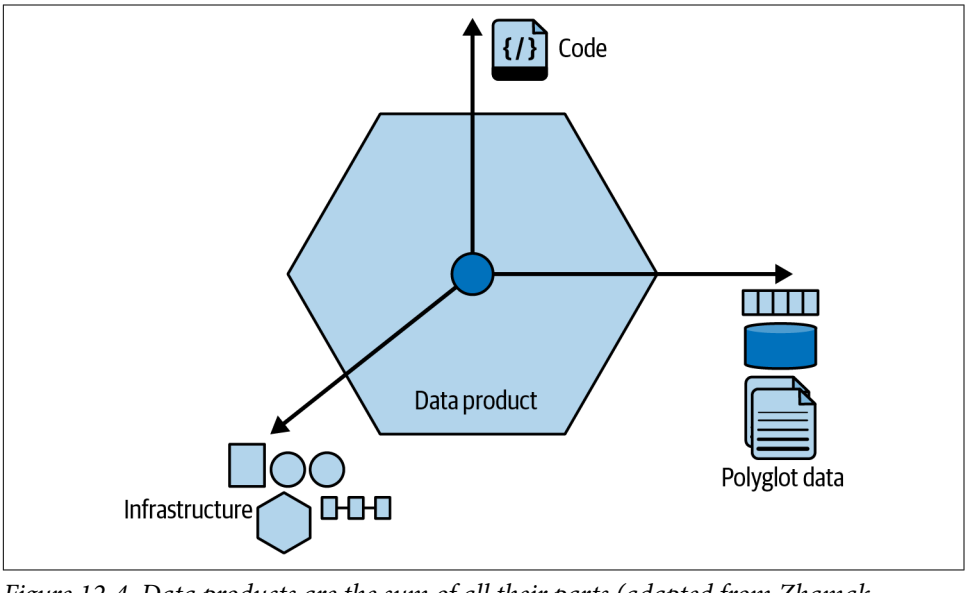

Figure 12-4. Data products are the sum of all their parts (adapted from Zhamak Dehghani)

Zhamak Dehghani introduced the novel idea of data products as part of her architectural paradigm the *data mesh*, where she proposed a rule that any curated data product must be purpose-built and capable of being used *as is* without requiring additional joins to other tables.[3] Essentially, the expense and effort of producing the data product should be paid in full on behalf of the consumers of the data product itself. This rule also helps tie together the simple idea that a data product is tied to a service, and that service is the production of useful, fit-for-purpose data. You can still

3 Zhamak Dehghani, *Data Mesh: Delivering Data-Driven Value at Scale* (O'Reilly).

join together the data from individual data products to create new data products, but the fact remains that additional joins *won't* need to be made by consumers of your data product for it to be usable for their given purpose.

Logically, it is also safe to assume that a data product can't exist without one or more data assets. Therefore, when we talk about data assets and data products, we are ultimately talking about data that is valuable enough to an organization that work went into designing, building, testing, releasing, monitoring, and maintaining the required applications and workflows to generate the set of valuable data assets encapsulating a specific data product. If this level of rigor and commitment to operations is ringing the traditional software project bell, that is correct.

Creating high-quality data requires engineers to follow the standard software development life cycle (SDLC). Essentially, this means designing for "no surprises" at runtime for the data product life cycle.

Data Products in the Lakehouse

Given the tendency for organizations to generate what feels like ever-increasing volumes of data through large data ingestion networks with increasingly complex dependency graphs, it is incredibly important that the lakehouse provides general capabilities for tracking the life cycle of highly valuable data assets—streaming or static.

This means being able to track the data assets' metadata, including upstream dependencies, as well as any downstream data asset dependencies. This is especially critical for downstream consumers, who must understand and react to changes in the volume of data and to modifications to the schema and structure of a given source or table, as well as other considerations and expectations in terms of the cadence of data being produced.

Maintaining High Trust

To maintain implicit trust in our data products, we must ensure that explicit and intentional additional metadata—including the union of data lineage, data quality, and end-to-end data observability—is available for our data products. For example, by being able to show that all processes involved in the production of a given mission-critical table utilize the same discipline, we can ensure our data consumers have the right information to feel confident, and this practice establishes a high-trust environment. You can also say that high trust is a consequence of following strict disciplines that come together to create a high-fidelity data product. More recently, data stewards and data product owners have come together to ensure that trust is part of the contractual agreement between the data producers and the data consumers of a given data product, thus providing a human touch in addition to metrics and monitoring.

If we take a step back to consider what tools, workflows (lakehouse orchestration), metadata, processes, architectural principles, and engineering best practices are required to manage the data contained by a Delta table representing a point in the lineage of a data journey from ingestion to deletion, across systems and services, users and their personas, data classification and access policies, and the curated *data products* representing data management at its finest, we quickly begin to realize the size and scale that is the umbrella of modern data governance.

Data Assets and Access

In the early days of traditional database management, there weren't large teams dedicated to governing how an organization managed the efficient collection, ingestion, transformation, cataloging, tagging, accessibility, and deprecation of data as seen with data governance organizations today; rather, the responsibility of managing access to a database was in the hands of the database administrator. This administrator was "in charge" of granting privileges to users, running expensive queries, and ensuring the database continued to operate.

The governance of which operations a user or group can execute is managed with privileges using the following SQL syntax groups: data control language, data definition language, and data manipulation language. We'll look at the data asset model next.

The Data Asset Model

The governance of a resource with respect to the lakehouse commonly describes the relationship between a policy and a governable object known as a *data asset*. In the simplest traditional sense, a data asset is a `TABLE` or `VIEW`, and a policy is a `GRANT` permission. The database, or schema, containing the table resource is also a data asset, as a policy grants access for a user, known as a *principal*, to execute an operation (show or select) on the data asset (database, table, or view). Before any principal can execute an action on resources, a data asset must first be created.

This data asset model is presented in Figure 12-5 and can pertain to any securable object that requires access and use controls through common SQL permissions.

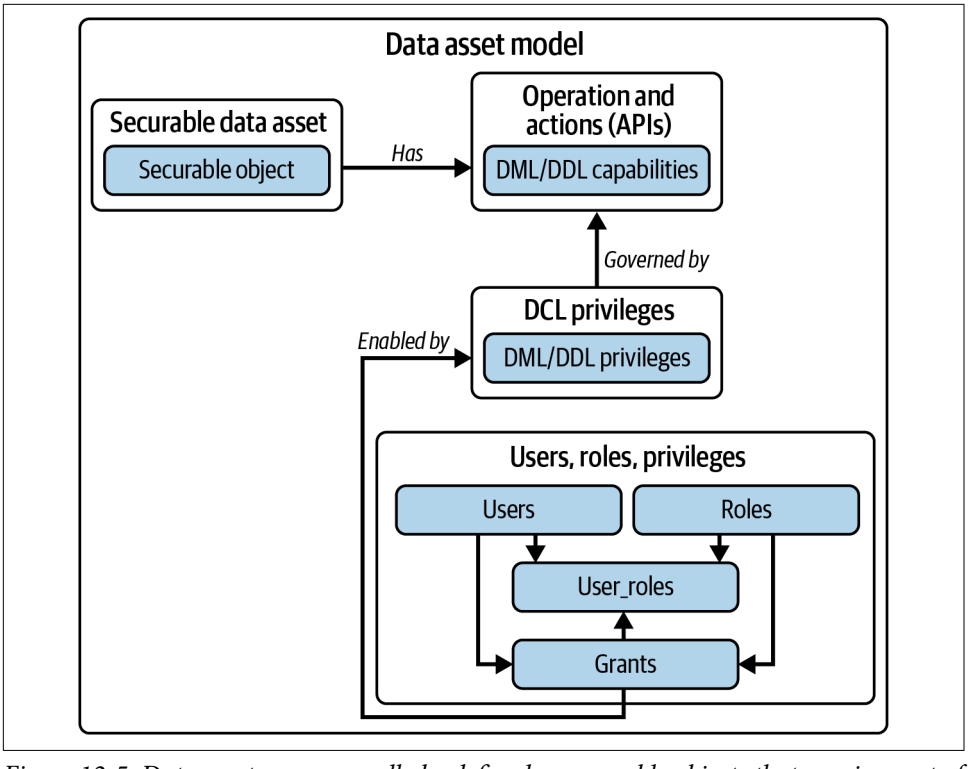

Figure 12-5. Data assets can generally be defined as securable objects that require a set of one or more permissions authorizing their access and general usage

The set of operations and actions that a principal can execute on a data asset is contained under the umbrella of *data definition language* (DDL), which contains `CREATE`, `ALTER`, `DROP` operations, and via *data manipulation language* (DML), which enables the `INSERT` and `UPDATE` actions, while the ability to execute one or more actions and operations is managed using *data control language* (DCL) by way of `GRANT` and `REVOKE` statements.

Nowadays, data assets have evolved to also encapsulate other resources that require access and use control (authorization) policies governing how they can be interacted with—for example, dashboards, queries (which in turn power dashboards), notebooks, machine learning models, and more.

Governing Data Access with SQL Grants

The governance of operations within SQL-like systems is handled through the grammar of DCL, DDL, and DML. Here is a quick refresher on these capabilities.

Data Control Language (DCL)

This special syntax is used for access management within SQL-like systems. Through the use of `GRANT` and `REVOKE` operations, a set of authorized actions (privileges) are associated with a set of `USER`s or `GROUP`s, enabling them to execute operations defined by DDL and DML, or removing one or more privileges that had previously been granted.

The syntax for `GRANT` permissions will vary depending on the flavor of the database, but they generally support ANSI-SQL standard syntax:

```
% GRANT priv_type [(column list)]
  ON [object_type]
  TO user_or_role, [user_or_role]
```

Controlling what actions a user or group can take isn't simply additive. In many cases, permissions are granted only for a finite amount of time before they are removed again. To fulfill the requirements of granting temporary permissions, the ability to remove permissions is enabled via the `REVOKE` syntax:

```
% REVOKE priv_type [(column list)]
  ON [object_type]
  TO user_or_role, [user_or_role]
```

Data Definition Language (DDL)

This syntax provides the following standard actions: `CREATE`, `ALTER`, `DROP`, `COMMENT`, and `RENAME`. We've used DDL in action directly as well as indirectly through the use of the Delta Scala, Python, and Rust companion libraries. In Chapter 5, we learned to create and alter tables, modify comments on columns, and even drop tables when we were through with them.

The `CREATE` syntax is used to define governable data assets. An example of the syntax for a standard SQL `CREATE` is shown next:

```
% CREATE [OR REPLACE] TABLE [IF NOT EXISTS] table_name (
    [column_name, type, ...]
  ) USING DELTA
  TBLPROPERTIES ('key'='value')
  CLUSTER BY (...)
```

The ALTER syntax is used to modify a governable data asset. The following examples show how to modify the properties of a table and how to add columns to a known table:

```
% ALTER TABLE table_name
  SET TBLPROPERTIES ('key'='value')

% ALTER TABLE table_name
  ADD COLUMNS (
    [column_name, type, ...],
  )
```

Data Manipulation Language (DML)

This syntax provides privileges to govern the operations a user or group can execute on a resource using the standard actions SELECT, INSERT, UPDATE, and DELETE:

```
% select [column,] from [table or inner select]
  [where,] [group by,] [having,] [order by], [limit]
```

Together DCL enables privileges to be assigned to users or groups that allow them to execute some or all of the actions governed by the resources created using DDL and the operations enabled by DML.

While the size and scale of data operations continues to grow across the globe, the paradigm of using simple GRANT and REVOKE privileges to control both access and authorization of data assets is still the simplest path toward adopting a unified governance strategy. Challenges arise almost immediately as we begin to consider interoperability with systems and services that simply don't speak SQL.

Unifying Governance Between Data Warehouses and Lakes

In the preceding section, we discovered that traditional data governance capabilities began with the addition of DCL syntax for SQL databases, which enabled the ability to allow or deny access to specific resources using GRANT and REVOKE statements.

Together the use of grants authorizes specific permissions associated with a user or role, enabling the execution of a set of actions on a secured resource (data asset). Governance for access using DCL works for traditional siloed databases (RDBMS) such as MySQL and Postgres as well as for most modern data warehouses via vendors like Databricks, AWS, and Snowflake.

There is a challenge, however, in using traditional SQL grants for governance of our lakehouse: *not all systems and services understand SQL*. To make matters worse, we don't have the ability to simply use one governance model to secure all data assets.

Consider the simple fact that the lakehouse still houses a traditional data lake just beneath the surface. This means we need to address the permissions and access model for the underlying data in order to provide a SQL-like interface to unify governance for the lakehouse.

Permissions Management

Just below the surface of the lakehouse lies the data lake. As we all know by now, the data lake is a data management paradigm that assists in the organization of raw data using primitives from the traditional filesystem. In most cases, cloud object stores are used, and at the root of these elastic systems are *buckets containing objects in a flat structure*.

Buckets encapsulate a resource root `"/"` representing a logical structure similar to the standard filesystem, but within a cloud object store. Figure 12-6 shows the breakdown of the bucket into its constituent parts. For example, just off the root we have top-level directories (paths and partitions) and their underlying files.

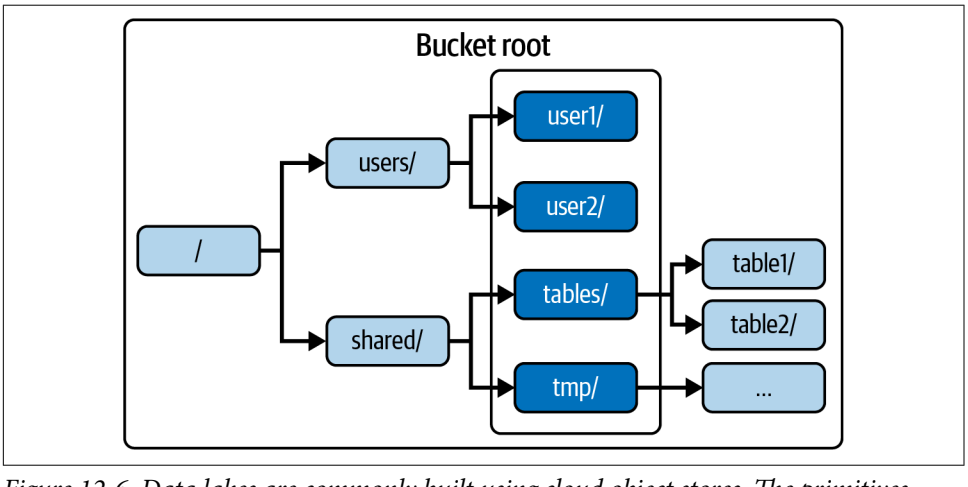

Figure 12-6. Data lakes are commonly built using cloud object stores. The primitives for these collections begin with the bucket, or root of the filesystem, and descend in an orderly fashion across directories and their subcollections of files or additional directories.

Each directory contains a collection of unstructured (raw) data—as commonly seen with log files, images, videos, or shareable assets such as configuration files (properties, YAML, JSON, etc.) and libraries (JARs, wheels, eggs)—as well as our structured but unprocessed data.

For structured data, it is advisable to use a well-known row-based format such as Apache Avro or Google's Protobuf, or a column-based format like Apache Parquet, which is simple to convert into Delta's table format using the Delta utilities,[4] as shown here:

```
% from delta.tables import *
  deltaTable = DeltaTable.convertToDelta(
                          spark,
                          "parquet.`<path-to-table>`")
```

In addition to all other types of unstructured and structured data, the data lake stores our managed (or unmanaged) Delta tables. So we have many possible kinds of files stored behind the scenes.

Understanding how to secure the underlying filesystem from unauthorized access is critical for lakehouse governance, and luckily, SQL-like permissions share a similar data management paradigm to that of the classic operating system (OS) filesystem permissions—access to files and directories is controlled using users, groups (akin to roles), and permissions granting *read*, *write*, and *execute* actions.

Filesystem Permissions

The OS running on our laptops and the OS running remotely on servers we've provisioned share similar access and delegation patterns. For example, it is the responsibility of the OS to oversee the distribution of finite resources (compute, RAM, storage) among many short- and long-lived processes (operations). Each process is itself the result of executing a command (action), and the execution is associated with a *user*, *group*, and *set of permissions*. Using this model, the OS is able to construct simple rules of governance.

Let's look at the `ls` command as a practical example:

```
% ls -lah /lakehouse/bronze/
```

The output of the command is a listing of filesystem resources (files, directories) as well as their metadata. The *metadata* includes the resource type (file or directory), the access mode (permissions), references (resources relying on this resource), ownership (user), and group association, as well as the file size, the last modified date, and the filename or directory name:

File type
: This is represented by a single character. Files are represented by a **–**, while directories are represented with *d*.

4 See "Convert a Parquet Table to a Delta Table" in the Delta Lake documentation.

Permissions
: These include read (*r*), write (*w*), and execute (*x*). Permissions are managed separately for the resource owner, a specific access group, and lastly anyone else (known as *others*).

References
: This tracks how many other resources link to a resource.

Owner
: Each resource has an owner. The owner is a known user in the OS. The owner of a resource has full control over how other users and processes interact through the assignment of group-level permissions.

Group
: Users are associated with one or more groups. Groups enable multiple users and processes to work together while restricting certain privileges. Groups within the context of the operating system are similar to roles within the context of the data warehouse. For each resource, a specific group can be granted permissions (outside of the owner of the resource), and for unknown group membership, default permissions can be applied as well.

When everything comes together, we can start to see the connection between file-system permissions and how they can apply to the governance of our lakehouse as well. Consider what the output of the following example tells us about the *ecomm_aggs_table*:

```
% ls -lh /lakehouse/bronze/ecomm_aggs_table/
drwxr-x---@ 338 dataeng eng_analysts   11K Oct 23 12:53 _delta_log
drwxr-x---@ 130 dataeng eng_analysts   4.1K Oct 23 12:34 date=2019-10-01
drwxr-x---@ 130 dataeng eng_analysts   4.1K Oct 23 12:34 date=2019-10-02
```

First off, the *_delta_log* directory informs us we are looking at a Delta table. It is owned by a user named *dataeng* who has full read-write-execute permissions (*rwx*). Additionally, the table is accessible for reading and execution by the *eng_analysts* group, but members of that group cannot modify the table since they are authorized for read-only access. For any other user in the OS, they would get an exception (not authorized) while attempting to interact with the files at this path.

A similar permissions model can be applied to our cloud object stores as well. The main difference is the way we identify users and manage groups.

Cloud Object Store Access Controls

The separation between storage and computation of the data lake ensures a physical boundary between the location of our data assets and the servers running our compute processes. If we dig further into the separation of concerns, we'll also discover that we are additionally cut off from the traditional OS-level user permissions model,

since the user (identity) bound to a local compute process is not directly known to the object store without the addition of a key (or token) signaling to the remote process that we are in fact allowed to execute a given action. This key helps to identify the request and authorize a simple rule set that will allow or deny the requested action in the form of a remote execution (read or write or delete).

In the absence of a shared operating system, we establish trust relationships between where we process data (compute) and where we store our data, utilizing identities. Identities help us to answer the following:

- What is the identity (user) of a given runtime process, and how does that apply to the traditional user permissions model?
- How can we *enable access* to one or more cloud-based resources?
- Once identified, in what ways can we *authorize* specific actions and operations to occur for a given user?

The paradigm shifts away from classic filesystem permissions (user, group, permission) and into a more flexible system called identity and access management, or IAM for short.

Identity and Access Management

If you heard a knock at the door, would you answer it? Would you let a stranger in? The whole reason why IAM exists is to ensure there is a *mutual trust-based relationship* between an unknown entity (who could be who they say they are) and your internal systems. So how do we identify a user, system, or service in a dynamic cloud-based world?

Identity

Each identity represents a user (human) or a service (API, pipeline job, task, etc.). Identities encapsulate both individual users as well as service principals, who are jokingly referred to as *headless users*, since they are not human but still represent a system doing things on behalf of a user. An identity acts like a passport, certifying the legitimacy of the user. In addition, the identity is used to connect the user to a set of permissions through the use of policies.

It is common to see access tokens issued for individual users, and for both long-lived tokens and certificates (certs) to be issued for service principals.

Authentication

While an identity might be legitimate, the whole point of authentication is to test to be absolutely certain. Most systems issue (generate) keys or tokens for only a specific period of time; this forces the identity to reauthenticate from time to time, proving

they are still legitimate. In the case of bad actors (hackers, spoofers) attempting to reuse a token they lifted for illegitimate purposes, a low time to live (TTL) on the token limits the potential impact of stolen identities. As a rule of thumb, the more secure the system, the lower the TTL for tokens.

Authorization

The identity and authentication mechanics come together to provide a guarantee that a user isn't simply an imposter. These two concepts are tightly coupled to the authorization process. Authorization is akin to `GRANT` permissions. We can assume that we know the identity of a user (since they have passed the test and proved that they are who they say they are), as they were able to gain entrance to the physical location of our resource (using a key, cert, or token to access data assets in the lakehouse). The authorization process is the bridge between the user and a set of policy files that describe what a user is allowed to do within a given system.

Access management

In a nutshell, access management is all about providing methods to control access to data and enforce security checks and balances, and it is the cornerstone of governance. Access controls provide a means of identifying what kinds of operations and actions can be executed on a given resource (data asset, file, directory, ML model) and provide capabilities to approve or deny based on policies.

The entire process of creating a user (identity), issuing credentials (tokens), and authenticating and authorizing access to resources is really no different than the `GRANT` mechanisms—the reverse being `REVOKE`, which would invalidate active credentials. No process is complete without the ability to also remove an identity, which completes the full-access life cycle.

IAM provides the missing capabilities enabling the implementation of `GRANT`-like permissions management for our lakehouse through the use of identities and access policies.

In the next section, we'll look at access policies and see how role-based access controls help simplify data access management through the use of personas (or actors), and we'll learn about creating and using policies as code.

Data Security

There are many pieces to the governance story, and in order to effectively scale a solution, there are important rules and ways of working that must be established up front—or carefully integrated into an existing solution.

For example, you might be familiar with the duck test: *if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.* This refers to our ability to

reason about something unfamiliar and to group it into a category of things "that are known to us." With respect to the various personas, or actors, operating within our lakehouse, we can use a modified duck test to create a limited number of roles that identify who has what level of access to which data assets as a first step on the path to more complex policy generation for authentication.

Role-based access controls

Role-based access controls (RBAC) are used to approve or deny system access and to authorize a subset of permissions on resources required to carry out the duties of specific personas using a role or roles within an organization.

For the lakehouse, consider the roles we play at our daily jobs (engineer, scientist, analyst, business functions), the team(s) and organization(s) we are part of, the logical dividing lines of our business (which can help establish data domains), the runtime environment for our systems, services, and data products (dev, stage, prod), and last, the classification of the data we are managing and accessing (all-access, restricted, sensitive, highly sensitive). The lines of what a role personifies can be a bit blurry at times, and for that reason, the R in RBAC can also denote *resource*.

Let's approach RBAC through a story. For the sake of the story, we all are employed at a global grocery chain called Complete Foods that sells local organic produce. Complete Foods sells products in physical stores, as well as online for delivery purposes. Each store operates in a specific geography, and the shopping trends will differ locally and regionally, as well as seasonally. This means that while all stores operate under the same corporate umbrella and share a majority of the same common products, regional inventory, vendor relationships, sales, and customers will differ based on where in the world the store is located.

However, the roles and responsibilities for employees requiring access to the lakehouse data will remain primarily the same, with level of access being based on need and reason (use case), as well as on required training on data privacy, governance, and sovereignty when it comes to accessing highly confidential data or when accessing customer data that is required for marketing and advertising campaigns.

Roles are not people. Each role can be assumed by a person or service. It is important to start simply and categorize the *who*, *what*, *where*, and *how*.

Establishing roles around personas

Understanding the who, what, where, how, and why is simplified when we abstract the roles associated with common personas within an organization. Let's explore some common dividing lines for personas within a typical organization:

Engineering role
: This can be applied to any developer role across hardware, software, security, platform, data, and machine learning, including headless users. The responsibilities include maintaining the systems for point of sale (in store and online), writing and maintaining mobile applications, defining event data, establishing data capture and ingestion networks, and handling personal data such as credit card numbers and users' home addresses, as well as learning from customer shopping habits to ensure that the right products are available in the right regions at the right time of year.

  Access patterns: All (read, read-write, admin)

  Role name: `role/developerRole`

Analyst role
: This includes business analysts or specialists. This role is responsible for working with business and engineering teams to ensure the right data is available to accurately capture critical business operations, and to assist in the decision-making process through the generation of insights, such as when to get pumpkin spice products back on the shelves, and what kinds of nondairy milk to continue to offer in what regions.

  Access patterns: Primarily read-only for data, with the ability to create and share queries, build dashboards, and analyze historical data or emerging trends

  Role name: `role/analystRole`

Scientist role
: This includes data and behavioral scientists. Responsibilities include working with the engineers and analysts to ensure the right data flows into the right places at the right time to power recommendations and other inference models.

  Access patterns: Primarily read-only, with the ability to create tables to power the training of models and to capture results for tests and experimentation

  Role name: `role/scientistRole`

Business role
: This includes manager, director, human resources, and even leadership. This role is responsible and accountable for building and maintaining the Complete Foods brand. There are local and global responsibilities, as well as regional store managers or buyers, and everyone will require access to sales numbers, forecasts, and subsets of data relating to employees or to concerns outside their line of business. Additionally, engineering leadership will require different access and capabilities than engineering managers and directors.

Access patterns: Mostly read-only; HR may need to create, modify, and delete employees

Role name: `role/businessRole`

The process for authorizing access to a given data asset (resource) in our lakehouse can be determined via a union of the following:

- The user's *role and responsibility* (*who* and *what*)
- The *resource location* (bucket and prefix) (*where*)
- The *environment where* they operate (dev, stage, prod)
- The *data classifications* (generally available, restricted, sensitive, etc.), which denote *what*
- The *operation* (*action*): read (view), modify (read, write), or admin (read, write, create, or delete) as the *how*

So remember: we always need to keep in mind the *who*, *what*, *where*, and *how*, as well as the *if*.

Think about it this way: *If we grant access to a given identity (who), then (what) operations are necessary to accomplish a given set of tasks (how), and in what environment (where) do they need user-level access versus headless user access? And last, what potential risks are involved in granting read-level versus read-write-level access?*

Additionally, aside from the considerations around whether access should be granted, the other question that must always be back of mind is whether the identity is allowed to view (read) all the data residing in the table. It is common to have data that is divided into groups based on the security and privacy considerations for the data access.

We will look at data classification patterns next.

Data classification

The following classifiers are a useful way to identify what kind of data is stored within a resource at a specific location in the data lake.

As a simple abstraction, let's think about data classification in terms of the stop-light pattern. A stop light signals to a driver to continue (green), slow down (yellow), or stop (red). As an analogy, when thinking about governing access to our data assets, the stop-light pattern provides a simple mental model to tag or label (identify) data that can be green, yellow, or red.

When in Doubt About Classifications

Every organization deals with different kinds of data. When in doubt, think about the damage to the company if a specific dataset (a table, raw data, etc.) was to leak to the public. Given the strict laws governing personal user data, some things will automatically come with a yellow or red classification. The guidelines and standards provided by GDPR, CCPA, and SOC2, for example, provide a compass to help you along your way. Each company is required to follow standards, and standards defined through policies make it easier to do the right thing. The more you work with different datasets, the easier it will be to intuit what is appropriate.

For example, access to data classified as "green" could be automated, assuming there are appropriate checks in place to ensure the resources are not leaking sensitive data. A practical example for "green" would be the earthquake and hazard data made generally available by the United States Geological Survey.

Access to data classified as "yellow" or "red" would require the grantee to consider who would have access, why they would need access and for how long, and how the access could benefit or harm the organization. When in doubt, always consider the *if. If we grant access to this data, do we trust the grantee(s) to do the right thing?*

Establishing rules and common ways of working can help to ensure that data is classified in a common way, reducing decision making to a scientific process:

General access
: This classification assumes the data is available to a general audience. For example, let's say Complete Foods believes it can sell more groceries by enabling services like Instacart, Uber Eats, and DoorDash to access our inventory data. By enabling open access—sign up, get a token, and hit the Delta sharing endpoint—we can ensure that any external organization can access specific tables associated with the general access role limited to read-only.

  Stop-light pattern: Green-level access

Restricted access
: This classification assumes data is read-only, with approval on a need-to-know (use) basis. Continuing the Complete Foods example from before, while external access to the inventory data (via the general-access classification) enables a mutually beneficial relationship to extend the reach of our grocery business and brand, there is data that represents our competitive advantage that must remain internal only, or restricted to external domains.

For example, let's say we have a price per product offered that is public (in store and via our partner services), but we also have an internal price representing the actual true cost to acquire a given product. In most cases, the margin (the delta between the cost to acquire a good and the price at the time of sale) isn't something we would like to advertise, as it represents our competitive advantage, as well as pricing negotiations that cost us very real money.

Stop-light pattern: Green-, yellow-, or red-level access

Sensitive access
: This classification applies to any sensitive data. Sensitive data would be damaging to the organization if leaked, but it doesn't contain critical information such as credit card numbers, Social Security numbers, payroll information, or medical or health information (which would cause compliance problems with HIPAA data). Sensitive data may contain personally identifiable information (PII) like users' first and last names, addresses, email addresses, birthdays, information about vendors (like the farms we purchase produce from), and other data relating to the operation of a business. Sensitive data may also contain information such as consumer behavior data, though without exposing a user's name or address or other PII. In the case in which daily aggregate data would be damaging if leaked, for example, if the data shows that the company's quarterly numbers are on a downward trend, even if the trend is represented as percentages versus actuals, this can still hurt the reputation of the company and would be a reason to tag the data as *highly sensitive access*.

  Stop-light pattern: Yellow- or red-level access

Highly sensitive access
: This classification applies to the most critically sensitive data available to an organization and to the user. This includes employee payroll information, company financial records, user credit card data as well as health-care data and home addresses, and more. Access to these data assets typically requires the completion of internal training, as well as a full audit trail related to access. Much of this data is traditionally reserved for human resources (HR) as well as payroll, and for specific actors within the business.

  Stop-light pattern: Red-level access

Now that we've identified the basic personas and roles related to data access (`developerRole`, `analystRole`, `scientistRole`, `businessRole`), as well as common classifiers for our data (general-access, restricted-access, sensitive-access, highly-sensitive-access), we can finish connecting the dots between IAM and data access policies and then finish up with a brief introduction to policy-as-code.

Using Prefix Patterns for Organizational Success

When it comes to S3 buckets and policies, one of the most useful things we can do is to take time up front to organize our data lake in order to simplify how we manage our Delta tables—which commonly involves setting up an S3 bucket, adding a warehouse directory, and hoping that teams do the right thing. Rather, the prework should include the setup of key patterns required for seamless runtime execution across environments, top-level catalogs, various databases (schemas), and their underlying tables, as well as dedicated space for data applications and their metadata, libraries, and configurations.

Let's look at the lakehouse structure in Example 12-1.

Example 12-1. Exploring the lakehouse namespace pattern

```
├── s3://com.common_foods.[dev|prod]
    └── common_foods
        ├── consumer
        │   ├── _apps
        │   │   └── clickstream
        │   │       ├── app.yaml
        │   │       └── v1.0.0
        │   │           ├── _checkpoints
        │   │           │   ├── commits
        │   │           │   ├── metadata
        │   │           │   ├── offsets
        │   │           │   └── state
        │   │           ├── config
        │   │           │   └── app.properties
        │   │           └── sources
        │   │               └── clickstream_app_v1.0.0.whl
        │   └── clickstream
        │       ├── _delta_log
        │       ├── event_date=2024-02-17
        │       └── event_date=2024-02-18
        ├── {table}
```

The lakehouse namespace pattern allows us to colocate our data applications alongside the physical Delta tables they produce. This reduces the number of policies required to manage the basics such as team-based access, line-of-business-level data management, and other concerns, like which environment to provide access to. When everything is done correctly, the development environment can act as a proving ground for new ideas, primed with mock data and built using anonymized production data (there is higher risk here, so remember the who, what, where, how, why, and if rules), and having two environments separated by a physical bucket makes it easier to follow the stop-light pattern, since dev and staging are traditionally all-access, while our production environment is almost always justifiably yellow- or red-level access, at least when it comes to personal data.

If you think back to the filesystem ownership pattern from earlier in the chapter, we have top-level ownership that defaults to admin over a given resource (file, application, directory), and then group-level access for approved identities becomes "read-only," while access for everyone else is simply blocked.

This pattern of default group-membership for engineers responsible for a data domain and the set of data applications powering the mission-critical data for a given domain should be part of the onboarding process for new team members once they have been trained and brought up to speed on the organization's ways of working.

From the lakehouse layout provided in Example 12-1, we see the data application and table resources for a clickstream data application underneath the consumer umbrella in the *common_foods* catalog. The directories contain the following:

Metadata (app.yaml)
: This can include resource configurations and other important application metadata, including the owning team, PagerDuty, or Slack channel information. Additionally, the *app.yaml* can include any runtime requirements in the form of CPU cores, RAM, min and max number of executors, access policies—you name it.

Source libraries (.whl, *.jar, *.py, etc.)*
: These libraries can be published directly to S3—or as an alternative, if you are working with containerized data applications, everything required for your application can be written to the container filesystem layer.

Configuration (app.properties, spark.conf)
: The application configuration can be supplied to your application using ConfigMaps for Kubernetes, as spark.conf or spark.properties for traditional Spark applications, or as any type of configuration that you support within your data applications. The important thing is that for each version (v1.0.0 in the example), all resources are self-contained. This pattern allows you to easily roll back to the "last" version if mistakes are made (we all make mistakes) without corrupting your checkpoints (from what was working).

Streaming checkpoints (_checkpoints)
: This collection contains the metadata for the stateful application (Structured Streaming or other). For example, if our upstream is another Delta table, the *_checkpoints* contain the last read version from Delta (reservoir version) that was processed, and the sink information, including the last "observed" commit version.

Table metadata and physical files
: The Delta table is included within the umbrella of the data product to minimize the number of policies, files, and roles needed to enable a team to operate within the lakehouse.

All application resources are located using a simple namespace pattern on the S3 prefix—*{catalog}/{database_or_schema}/_apps/{app_name}/**—with all versioned resources and assets contained within the semantic versioned release (v1.0.0). When we connect continuous integration and continuous delivery (CI/CD) with the GitHub repositories containing our data applications, it becomes simple to tie the version of the application alongside the Git tag of a Git release. This also enables automatic rollbacks in the case of failure by looking at the current release – 1.

Now let's move on to the actual Delta tables. The output tables of our data applications exist underneath the same relative path as the data application itself. The common ancestor of both the data application and the table is the database (or schema) contained within a specific catalog. This pattern might not always be possible, especially in the case where a data application is reading from multiple bronze tables to produce a silver-based output table.

For our application configuration, using Spark as an example, we can set the config property `spark.sql.warehouse.dir=s3://com.common_foods.prod/common_foods` to enable our application to read or write to tables contained under the *common_foods* catalog.

Data assets and policy-as-code

We can simplify the security and governance of our lakehouse using common access patterns. Take, for example, the introduction of Amazon S3 Access Grants: this abstraction simplifies the management of roles and the delegation of SQL-style `GRANT` permissions across traditional S3 buckets.

The following section explores using Amazon S3 Access Grants at a high level. This section assumes prior experience with Amazon S3 as well as with how policy management works.

Create an S3 bucket. The S3 bucket will act as a container encapsulating our production lakehouse. Using the Amazon CLI (shown in Example 12-2), we set up the bucket and call it *production.v1*.

Example 12-2. Setting up a bucket for our lakehouse

```
aws s3api create-bucket \
    --bucket com.dldgv2.production.v1 \
    --region us-west-1 \
    --create-bucket-configuration LocationConstraint=us-west-1
```

Once we have succeeded in setting up our bucket, the bucket location is returned. This means we have a unique ARN (Amazon Resource Name)—for example, *arn:aws:s3:::com.dldgv2.production.v1*.

Create an S3 Access Grants instance. An S3 Access Grants instance is a container for logically grouping one or more registered S3 locations and the grants that define who has what level of access to what for our S3 data for each location. There is one instance per AWS region within a single AWS account—the process to create the grant instance is shown in Example 12-3. This means that regional data access controls are honored even when global access is possible for S3 buckets.

Example 12-3. Creating an S3 Access Grants instance

```
% ACCOUNT_ID="123456789012" && \
     aws s3control create-access-grants-instance \
     --account-id $ACCOUNT_ID
```

Here are the results of creating the new grant instance:

```
    {
        "CreatedAt": "2024-01-15T22:54:18.587000+00:00",
        "AccessGrantsInstanceId": "default",
        "AccessGrantsInstanceArn": "arn:aws:s3:us-west-1:123456789012:access-grants/
    default"
    }
```

Now that we have set up an S3 bucket and the Access Grants instance (both in us-west-1), we can create an IAM role and trust policy to use for our S3 Access Grants.

Create the trust policy. A trust policy must be created to allow the AWS service (identified by the service access-grants.s3.amazon.com) permissions to generate temporary IAM credentials using the `GetDataAccess` action on an S3 resource. The trust policy is shown in Example 12-4.

Example 12-4. Create the trust-policy.json file

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "access-grants.s3.amazonaws.com"
            },
            "Action": [
                "sts:AssumeRole",
                "sts:SetSourceIdentity",
```

```
                "sts:SetContext"
            ]
        }
    ]
}
```

Now execute the following:

```
% aws iam create-role --role-name s3ag-location-role \
--assume-role-policy-document file://trust-policy.json
```

The final step to finish setting up the access grants is to create a policy enabling read and read-write capabilities on an S3 bucket prefix.

Create the S3 data access policy. The last step is simply to associate the generic read and write permissions on our S3 bucket:

```
% aws iam put-role-policy --role-name s3ag-location-role \
--policy-name s3ag-location-role --policy-document file://iam-policy.json
```

The *iam-policy.json* file is included in the book's GitHub materials for this chapter.

Now that we have established the S3 Access Grants, we can move on to simplifying how we manage read and read-write permissions, or even admin-level permissions, for resources in our lakehouse.

Applying policies at the role level

Next we will apply the principles of RBAC to our policies. This enables us to provide general-purpose rules that enforce access control to a set of resources. In this case, the resources are Delta Lake tables located within our S3 buckets.

Read. This will authorize read-only capabilities on a resource, or the ability to view metadata about a given data asset, including the table properties, ownership, lineage, and other related data. This capability is required to view the row-level data within a table, list the resources contained within a bucket prefix (filesystem path), or read table-level metadata. Example 12-5 shows how to use `SQL Grants` to enable `READ` for our `analystRole`.

Example 12-5. Applying an Amazon S3 Access Grants Read Policy

```
$ export ACCOUNT_ID="123456789012"

aws s3control create-access-grant \
--account-id $ACCOUNT_ID \
--access-grants-location-id default \
--access-grants-location-configuration S3SubPrefix="warehouse/gold/analysis/*" \
--permission READ \
--grantee GranteeType=IAM,GranteeIdentifier=arn:aws:iam::$ACCOUNT_ID:role/analystRole
```

This example shows a simplified method of granting permissions for Amazon S3 using access grants.

ReadWrite. In addition to the actions provided by *read*, the write capabilities add modify capabilities enabling the actor (identity) to insert (write) new data, update table metadata, and delete rows from a table. The simple policy is shown in Example 12-6.

Example 12-6. Applying an Amazon S3 Access Grants ReadWrite policy

```
$ export ACCOUNT_ID="123456789012"
export GRANT_ROLE="role/developerRole"

aws s3control create-access-grant \
  --account-id $ACCOUNT_ID \
  --access-grants-location-id default \
  --access-grants-location-configuration S3SubPrefix="warehouse/gold/analysis/*" \
  --permission READWRITE \
  --grantee GranteeType=IAM,GranteeIdentifier=arn:aws:iam::$ACCOUNT_ID:$GRANT_ROLE
```

Admin. In addition to the capabilities managed by *readwrite*, the admin role authorizes an actor to create—or delete—a data asset located at a specific location. For example, it is common to restrict destructive capabilities to only service principals; similarly, creating resources most often also means additional orchestration to manage and monitor a resource. Since headless users can act only on behalf of a user, this means they can only run workflows and commands and execute actions and operations that already exist. In other words, the service principal can trigger a specific action based on some external event, reducing the surface area of accidental "oops." It is best to use traditional IAM policies to control access to create and destroy lakehouse resource locations.

Limitations of RBAC. There are, of course, limitations when simply using roles alone to manage access; mainly what tends to happen is an explosion of roles. This can be considered "sprawl," and it is an unforeseen side effect of success. Let's be honest: if there are only four lines of business, and you have four supporting roles (developer, analyst, scientist, business), then you are looking at a max of $4 \times 4 \times n$ (with n being the number of tables within a line of business that require special rules to govern access) to handle the requirements of general governance across the company. What happens when you go from four lines of business to twenty? What about fifty? It is the what-ifs that define what to do next. If we are lucky and the company has taken off, and we've hired well and managed to maintain a robust set of engineering disciplines and practices, then we could technically begin to pivot into attribute-based access control (ABAC). This is also known as *tag-based policies* and can also live under the umbrella of *fine-grained access controls*.

Fine-Grained Access Controls for the Lakehouse

The solution to the problem of compounding complexity with coarse-grained access controls—or access based on allowing or denying read and write to data assets as a whole based on hierarchical roles—comes in the form of fine-grained access controls. There are many emerging techniques to provide fine-grained access controls using the notion of tags, or attributes.

For example, say we have a Delta Lake table that has twenty columns. This table could encapsulate orders for our customers. There is a high probability that the information about each order is important to many personas within the company, but access to the user information associated with the order could be out of compliance depending on the bylaws and rules governing access to customer data. Rather than the entire table being marked as "yellow" or "red" for its classification, the columns themselves can be tagged (using metadata) to denote whether the column can be read or whether it should be masked, or nullified, for general access.

The SQL in Example 12-7 shows how dynamic masking can be achieved within Databricks using Unity Catalog. To view the data stored in the user struct, the user or service principal querying the order data must be in the `consumer_privileged` account group, and in this case the user struct must also be tagged with the value of `pii`.

Example 12-7. Using dynamic views and tags for fine-grained access controls

```
-- SQL
CREATE VIEW consumer.prod.orders_redacted AS
SELECT
  order_id,
  region,
  items,
  amount,
  CASE
    WHEN has_tag_value('pii')
    AND is_account_group_member('consumer_privileged')
    THEN user
    ELSE named_struct(
      'user_id', sha1(user.user_id), 'email', null, 'age', null)
    AS user
  END
  FROM consumer.prod.orders
```

While the SQL shown in Example 12-7 provides us with a starting point to selectively redact data, the example utilizes the `has_tag_value` function alongside the `is_account_group_member` function, both of which are not available to the general public. Additionally, without the support of integrated generalized rule management, creating and maintaining dynamic views can become cumbersome over time. However, to end on a positive note, a simple solution to the problem can be to provide access to the physical table via views that explicitly redact any data marked as PII for the general public, while continuing to restrict direct access to the underlying physical table using the simpler coarse-grained access controls. This is a nice stopgap that can be achieved using open source alone.

Conclusion

The way we govern, secure, and store the precious assets inside our lakehouse can be complicated, complex, or simple; it all depends on size and scale (or the number of tables and other data assets) and at what point in time we realize the need for a more complete governance solution. No matter the point in the journey, start small—begin by creating separation between data catalogs at the bucket level to separate all-access data from highly sensitive data. Layer into your solution ways of synchronizing what people need from the data and what systems and services will need from that same data, and roll this into your strategy for who, what, and when.

In the next chapter, we will continue to look at metadata management, data flow, and lineage and round out what we started in this chapter.

CHAPTER 13

Metadata Management, Data Flow, and Lineage

In the preceding chapter, you were introduced to the foundational components required to build a successful lakehouse governance solution. These components included identity and access management, data catalogs, and metastores, as well as the physical cloud-based storage powering the lakehouse. We showed you how roles and personas aid in the generation of secure building blocks for layered security and privacy, and we concluded with a look at utilizing SQL-like permissions management to simplify access controls for the lakehouse. This chapter continues where the last one left off, tying together the components of metadata management alongside the dynamic flow of data, as captured through the lens of data lineage and observable data applications.

Metadata Management

Have you ever been lost in the woods, or been driving in a new place without GPS or even an old-school map? Being lost is something we all have in common, and the same feeling can be expressed by data teams who are just trying to get to a set of tables they know *should* exist. But where are those tables? Metadata management systems provide the missing components between being lost and having directions. In our case, the location we are trying to get to is a set of known tables within one or more data products that we can trust to provide us with the correct information to solve our data problem. The metastore and services built on top of this metadata, like any data discovery services, act as a compass to help us reach our waypoint or final destination. The metadata, which is our data about our data, is required to solve our problem and can provide assistance when we are trying to arrive at the correct data destination.

What Is Metadata Management?

Just as in data management, the life cycle of our metadata provides a way to keep track of the data assets we hold near and dear, as well as notes, descriptions, comments, and tags. The centralized metadata layer—a foundational component of our lakehouse data catalogs—provides a representation of an organization's information architecture. This includes the hierarchy represented by our catalog(s) and databases (schemas) and the tables and views contained therein. This basic hierarchy was presented in Chapter 12, when introducing the prefix patterns for organizational success. The role of the metadata layer is to provide the necessary descriptive data to produce a macro view across the entire lakehouse regarding the current state of all data assets, and to provide a compass pointing to those data assets available for use.

It is common to use the term *data catalog* or *metastore* when referring to the operational metadata layer. The terms *metastore* and *data catalog* are used interchangeably, as both terms describe a service that stores data about our data that can be accessed through APIs.

Data Catalogs

Depending on where you sit within your organization, you may find there are many interpretations of what a data catalog is. Essentially, in its most basic form, a data catalog is a tool that enables a user to locate the high-quality data they need to get their job done. At a minimum, the data catalog provides information about the components of a data product—the catalog, database (schema), tables, and views—along with a simple search component called the data discovery layer (or service).

> *The data catalog is used in the same way someone shopping at IKEA would use integrated search to locate something they want, be it a couch, table, or chair; this is very different from how someone would look through a paper catalog—there are expectations. For data, people have a general idea of what they need, and a good data catalog makes the journey simple.*
>
> —Andy Petrella

There are many different ways to solve the problem of looking things up, and what we are solving for and the definition of the problem should be actionable and based on real customer use cases.

For instance, we could create a manual list of all tables; the solution could be a simple shared spreadsheet—with the known limitation of the shared spreadsheet being the need to ensure that *someone* keeps the metadata up to date. This book is about solving problems, so the prior example is more an example of what *not* to do, but it might also be the simplest solution, depending on the size of your organization, and it ticks the boxes of enabling a user to search (filter) the spreadsheet (basic metastore) to narrow the set of tables and hopefully find (locate) what they need.

The problem with any process requiring manual human effort to maintain state is that without the right discipline, things will eventually and inevitably be out of sync just when you or someone else really needs them. This is the downside of offline or static data catalogs. They are, by definition, simply a promise of what *could be* rather than reflecting the true state of what *is*. Because manual synchronization doesn't scale, the trend in the industry has shifted toward automated cataloging and active data discovery.

Data Reliability, Stewards, and Permissions Management

The problem of maintaining the *who* (owners, producers, consumers), *what* (data product, data assets), *where* (location), *why* (can and should the consumers access and use this data?), *when* (is access limited to a specific start and end date? Is access required for batch or for streaming, or for both?), and *how* (is the data being used in part or completely? Is the data being copied or used only to materialize views?) is offloaded to the data stewards—or in some cases, to the data product owners.

The who, what, where, how, and why also have a role to play in regard to the reliability of the data product with respect to the data consumers. So you can say there is a reciprocal role to play in producing data as a product and ensuring that the consumers of a given data product abide by any associated rules and regulations for consumption of that data product.

The IAM capabilities provided by the lakehouse governance platform (as illustrated in Figure 12-1 in the previous chapter) should provide the data stewards with the ability to simplify access management for the data products that they oversee and are ultimately responsible for governing. This responsibility spreads a wide umbrella that covers the data producers, the life cycle of the data products themselves, and the contracts established for those data products.

In terms of contracts, it is typical to provide data product guarantees backed by a set of data-level objectives (DLOs)—no different than service-level objectives (SLOs)—that specify the minimum reliability that can be expected for a given data product. This helps standardize the way trust is defined across an organization. The DLOs are backed by a set of data-level indicators (DLIs)—likewise mirroring the traditional service-level indicators (SLIs)—which are the key performance metrics required to observe and confirm the reliability of a given data product.

For example, say we have a Delta Lake table that is generated by a streaming ingestion job. The DLO for the table specifies that all columns contained within the dataset will never be `NULL`. This guarantee can easily be enforced with the addition of simple constraints to our Delta Lake table, and due to the invariants of the Delta protocol, we can always meet the minimum requirements for the given DLO.

In practice, if the Delta Lake table is based on a streaming ingestion pipeline, then the DLO will typically also include the expected update or append frequency. We can produce statistics to articulate table freshness by measuring the delta between the streaming source timestamps and the available data within the table as our DLIs for each streaming microbatch. This measurement is commonly called the *table lag metric*, as it describes how far behind real time the ingestion pipeline is running, and therefore how stale or fresh our Delta Lake table is.

Outside of data-level guarantees, there are other obligations for the data producers, including table-level considerations like backward compatibility for table schemas, which provide a level of confidence and trust to the consumers of a given data product.

Ultimately, the data producers, stewards, and product owners hold the line to establish high trust for their data products. Providing data teams with high-trust data products can be achieved in part through the use of the metastore and catalog.

Why the Metastore Matters

It is nearly impossible to ignore the Hive Metastore when discussing the Lakehouse. This is because the Hive Metastore provides the capabilities for translating our file-based data lake tables into structures that can be queried like traditional SQL tables. Before Apache Spark SQL, the ability to query tables inside the data lake was achieved by using Hive SQL running MapReduce jobs inside Hadoop clusters. As Spark SQL became more widespread, the Hive Metastore continued to be maintained, but over time the industry no longer required a complete Hive distribution, and the Hive Metastore alone provided the missing pieces, enabling a Spark job to convert the Hive data into a Spark table object.

The Hive Metastore provides a set of basic features that can be utilized for data (database, table, and view) discovery, given the metastore itself resides in a traditional relational database (like Postgres or MySQL). This means that a user who has read access to the Hive Metastore can execute `SHOW` commands to list the databases, tables, views, and columns contained within, in order to discover what exists—or to query the resource metadata available through the `tblproperties`, `dbproperties`, or other system or discovery tables.

Because of the separations of concern between the physical metastore and our physical Delta Lake tables, IAM can provide filesystem management while `SQL grants` limit the surface area (which databases [schemas] and tables or columns a user can see within the `metastore`). Figure 13-1 shows the Hive access model as it relates to the metadata stored in the relational database (left) and the reference to the databases and tables located within our cloud object store or distributed filesystem (right).

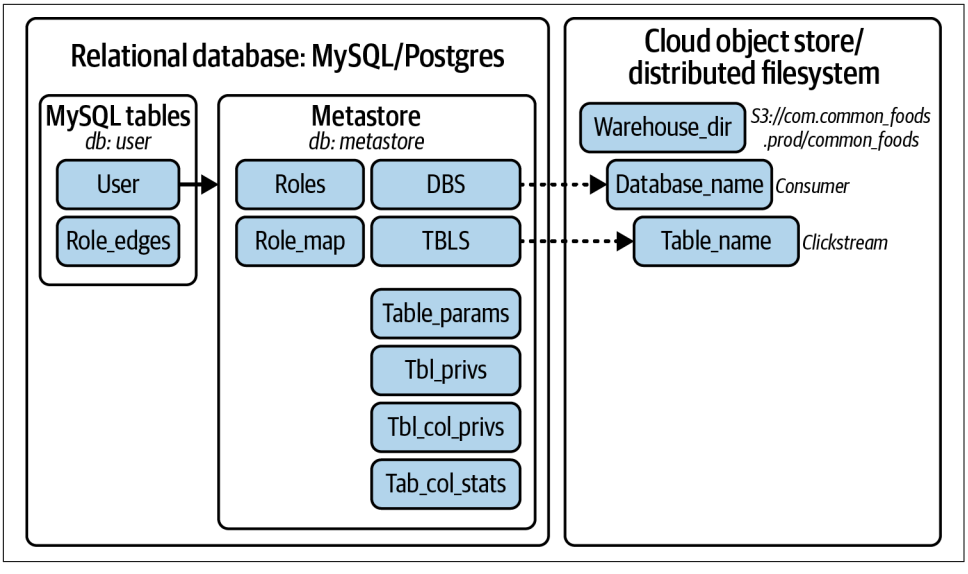

Figure 13-1. The Hive access model ensures a separation of concerns between access to database and table metadata and access to the physical files representing our Delta tables and located at a prefix within our storage layer

Figure 13-1 provides a high-level overview of the Hive Metastore. The metastore itself is a set of tables (>70) that enable the magic that provides us with a catalog of the where and what of our tables. However, the metastore is responsible only for storing the referential database and table data, including the *location* of these data assets with respect to their path on our cloud storage. The metadata also includes the table type, partitions, columns, and the schema.

While the basic information about the table is nice to have, we are missing a considerable amount of information that is really needed to operate our lakehouse at scale—not to mention, this is a book on Delta Lake and not on all table types or supported protocols. So we can safely ignore most of what the Hive Metastore provides, given that each Delta table contains a reference to its own metadata.

What the Hive Metastore provides to Delta for our lakehouse is the ability to identify the databases (schemas) and tables contained at a given cloud-storage prefix without requiring the object tree to be manually listed (which can be an expensive operation). Given that we have the Delta log (recording the table history), as well as the ability to fetch isolated snapshots of our tables (using time travel, or just for the current version of the table), we have limited use for the Hive Metastore outside of the general "listing" of catalogs, schemas, and the tables that reside within a known instance of the metastore.

In addition to the sparse capabilities mentioned above, there is one big limitation to address, especially if you are using the `delta-spark` library along with the Hive Metastore (or a variant such as AWS Glue, which is compatible with most of the Hive Metastore API). For any given data application, we can connect to only one catalog per session. This can be a bummer when we have requirements for joining tables contained across multiple catalogs. This limitation is due to setting the global `spark.sql.catalog.spark_catalog` as well as `spark.sql.warehouse.dir`. While this limiting factor can be worked around by creating copies of tables between different physical buckets (if we are using a bucket for each catalog), this reduces our ability to achieve a single source of data truth.

Unity Catalog

Unity Catalog is a universal catalog for data and AI. There are two versions of Unity Catalog at the time of writing—the internal proprietary version within Databricks and the open source software (OSS) version. The OSS version is interoperable with the Databricks version and provides the following key features: the metastore, a three-tiered namespace, governed assets, managed and unmanaged volumes, interoperability, and true system openness:

Metastore
: Unity Catalog utilizes a centralized metadata layer called the *metastore*. This provides the ability to catalog and share data assets across the lakehouse, within regions, and even across clouds. Additionally, the metastore provides a three-tiered namespace in which data can be organized.

Three-tiered namespace
: The namespace within Unity Catalog provides the following convention: `{catalog}.{database/schema}.{table}`. The namespace is a component of the metastore and enables us to organize our data and assets hierarchically.

  The hierarchy is used for more than simple organization; it enables our data applications to read and join across boundaries that traditionally required copying data between Hive tables due to limitations of the two-tiered Hive namespace. Enabling a single job to read from multiple catalogs makes it simple for our data applications to join data between many tables residing across many catalogs.

As an added bonus, we can also utilize fully qualified table names (between catalogs)—`spark.read.table('prod.consumer.clickstream')`, for example, which simplifies jobs that had previously relied on direct table paths to work around the limitations of the Hive Metastore.

Unified governance for data and AI
: Assets within Unity Catalog include catalogs, databases (schemas), tables, notebooks, workflows, queries, dashboards, filesystem volumes, ML models, and more. Using Unity Catalog's built-in governance and security—with strong authentication, secure credential vending, and asset-level access control—we can protect all our data and AI assets with a unified solution. This makes the solution to the complexities of providing filesystem-based access controls within a SQL-like system (covered in Chapter 12) much easier.

Managed and unmanaged UC volumes
: One of the exciting features of OSS Unity Catalog—as included in the 0.1 release—is the availability of managed and unmanaged UC volumes with S3 credential vending. This feature allows us to centrally manage access to S3 bucket locations storing unstructured or nontabular data. This includes raw images and binary data for machine learning, application configuration, and artifacts (JARs, wheels, eggs) for running data applications, as well as a landing zone data layer to act as a primary ingestion source for our lakehouse, where applicable.

Interoperability
: Unity Catalog supports Delta Lake, Apache Iceberg via UniForm, Parquet, CSV, JSON, and many other formats. It also implements the Iceberg REST Catalog APIs to interoperate with a broad ecosystem.

Openness
: Unity Catalog is Apache 2.0–licensed, including an OpenAPI specification, server, and clients. Adoption of open standards maximizes flexibility and customer choice by ensuring extensive interoperability across various engines, tools, and platforms.

Getting Started with Unity Catalog OSS

If you are interested in using the open source version of Unity Catalog, the project has provided a really good quickstart that will have you up and running in under five minutes (depending on your internet speed!). Just clone the repo and start the server:

```
% git clone git@github.com:unitycatalog/unitycatalog.git &&
  cd unitycatalog &&
  bin/start-uc-server
```

If things go well, you will be greeted by the following:

```
######################################################################
#  _    _       _ _            _____      _        _                 #
# | |  | |     (_) |          / ____|    | |      | |                #
# | |  | |_ __  _| |_ _   _  | |     __ _| |_ __ _| | ___   __ _  #
# | |  | | '_ \| | __| | | | | |    / _` | __/ _` | |/ _ \ / _` | #
# | |__| | | | | | |_| |_| | | |___| (_| | || (_| | | (_) | (_| | #
#  \____/|_| |_|_|\__|\__, |  \_____\__,_|\__\__,_|_|\___/ \__, | #
#                      __/ |                                 __/ | #
#                     |___/              v0.2.0-SNAPSHOT    |___/  #
######################################################################
```

There are a few example datasets to use in your exploration. Execute the following command to list out the available tables:

```
bin/uc table list --catalog unity --schema default
```

That is it for the quick introduction. Follow along with the quickstart docs online to learn to use the many features of Unity Catalog OSS available today.

Data Flow and Lineage

Data flows into our lakehouse in many different ways, and the way we capture the lineage of these data flows provides a view at the ingestion edge, or along the surface. On the surface, there is an understanding that tables must begin somewhere, and that the source of data powering a specific table today may and most likely *will* change at another point in time. The sources of data—outside the lakehouse—are impermanent. But the data product consumers shouldn't need to worry themselves with these internal data domain concerns.

For example, say we receive data from a third-party vendor every time an email or push notification is successfully sent. The data we ingest from each vendor is very specific to their APIs and internal data models and is also tied to whatever reliability contract we have established with that vendor at that point in time. The fact that we are using one vendor or another isn't the concern of data consumers, who are focused on insights into consumer behavior, or on increasing the open rate for emails, or on some conversion rate metric associated with the success of a marketing campaign.

It is the job of the data engineering team working under the consumer data domain (in the prior example) to transform the vendor-specific data into a common data format that eventually can be consumed by another team to produce insights within the external data domain (which is the gold layer of the medallion architecture). By providing common formats, we can transition from one vendor to another without interrupting the data flow into our mission-critical data assets (tables, reports, etc.).

So how does data lineage fit into this model?

Data Lineage

The purpose of data lineage is to record the movements, transformations, and refinements along a data journey, from the point of initial ingestion (data inception) within the lakehouse to the data's final destination—which can take the form of insights and other BI capabilities—or to provide a solid foundation for mission-critical ML models. Consider data lineage to be a sort of flight recorder, capturing important moments in time across our critical data applications—producing our data assets—with the purpose of being used to provide a measure of data quality, consistency, and overall compliance and to track the many data dependencies along that processing line.

Andy Petrella describes *lineage* as the intersection of "line" and "age," referring to the direct connection between data sources and how long they have shared a connection.[1]

The lineage of the many data sources and associated data applications comes together to provide an observable lens into the dependencies for our data products at runtime. In addition to helping with understanding the dependency graph, data lineage helps to ensure data teams understand the when, where, and why if any problems are experienced at runtime. Even with the best of intentions, things do inevitably go wrong, and flying blind is never a good look!

1 Andy Petrella, *Fundamentals of Data Observability: Implement Trustworthy End-to-End Data Solutions* (O'Reilly), 44.

Figure 13-2 supplies a visualization to aid in the discussion regarding data lineage. This diagram provides a simplistic view over the lineage, starting with data sources residing outside of the lakehouse (1) flowing into the internal data domain and utilizing a series of data applications (2) that produce a table or tables for each source (3). This data is then joined and further transformed by another data application before yielding the external data domain table (4).

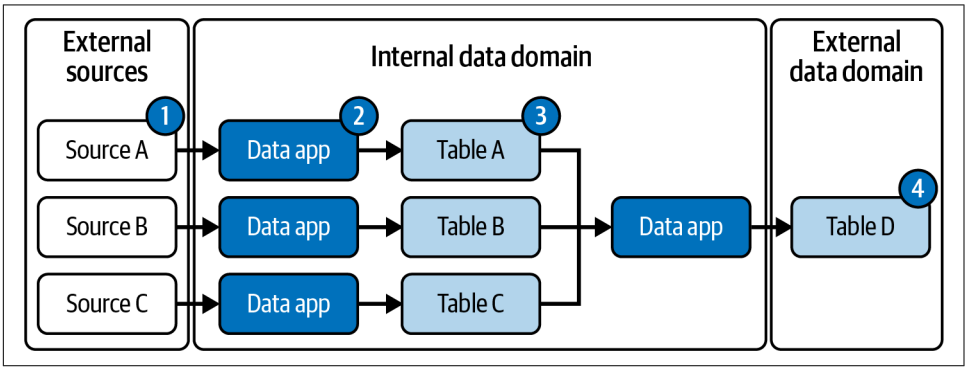

Figure 13-2. A starting point for data flow visualizations using data lineage

At the most rudimentary level, data lineage can be captured as a graph of sources to tables (or other data assets). However, this would ignore the fact that there are data applications (2) running to produce all tables other than the initial ingestion sources (1)—with respect to Figure 13-2. Therefore, we have both the concept of data lineage and that of data application lineage to consider.

Leaning on the data lineage to view the data flow allows us to quickly visualize "what changed" or to see "what is no longer behaving as expected," which can help to mitigate risk. To understand what changed, we need to go back to data application lineage (or workflow lineage).

Data application or workflow lineage

Data applications walk an interesting line with respect to complexity. At one extreme, a data application can be as simple as a SQL statement used to execute a transform, or as complex as a stateful aggregation application used to execute a complex operation like marketing funnel analysis. Regardless of the complexity, a good application exists in version control (like GitHub) and is associated with a release version (e.g., v1.0.0) and therefore has additional metadata that can be gathered to understand when things change. Furthermore, a data application requires resources to operate; this means there are compute resources associated with the runtime execution of each data application. Data applications add additional metadata to the data lineage

graph, including the runtime version of each data application, the option of cluster configuration, and the Git SHA for the "current" version of the app. This additional metadata can be used for data application observability and can play a key role in providing a wider view on the operational data lineage.

There are many common uses for data lineage. It provides catalog, database, table/view, and columnar-schema-based linkage between data applications to help us understand how tabular data is accessed and used across the lakehouse. This includes additional data asset types (where supported), which can help us understand which sources of data are used to train machine learning models, or what specific tables or views are used to construct operational dashboards.

Data lineage can help us to identify important transitional points within the medallion architecture and to understand what data layer (internal or external) within a data domain provides the right level of refinement to solve a data problem. It can help in resolving upstream and downstream dependencies of a specific table or view or in building frequency graphs for access, and it can be used for audit awareness and to understand all active data customers of a data asset.

It can be used to derive insights for lakehouse-wide access and audit level insights to power monitoring and provide answers for centralized data governance teams with respect to audits (can and "should" a given principal [user or group] execute an action [read, write] on a given data asset [file, table, dashboard, etc.]).

There are many areas in which it makes sense to reuse our lineage data. These include access and compliance monitoring; impact analysis, to quickly triage when things go wrong; data change management, to understand the impact of critical schema changes; and to provide communication to the active data consumers—for example, when there is a need to migrate from a v1 to a v2 data asset or product.

Use case: Automating data lineage using OpenLineage

OpenLineage is an open source framework for the collection and analysis of data lineage. It is extensible and has a growing community surrounding it. The design of the framework provides an *open standard* for lineage metadata designed to record metadata for a Job within a specific execution.

The diagram in Figure 13-3 shows the generic operating model, consisting of a Dataset, a Job, and a Run entity. For each core entity (Dataset, Job, and Run), there is an extension object identified by the Facet keyword. These extension objects encapsulate user-defined metadata enabling enrichment of entities.

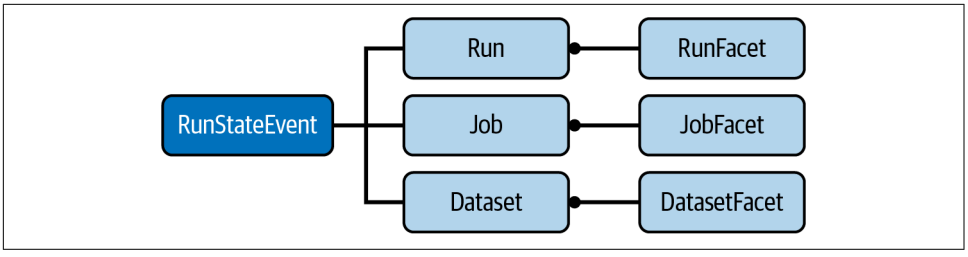

Figure 13-3. OpenLineage is built on top of simple entities encapsulating a Dataset, a Job, and a Run

Consider the fact that data doesn't simply exist in the lakehouse but requires a process (Job) to execute (Run) in order to ingest an initial table (Dataset) or to make modifications from one or more upstream tables (Datasets) in order to produce a new table or set of tables. This pattern and operating model essentially tracks the operational behavior of any data pipeline or simple data flow, as viewed through the lens of a data application (like we saw in Figure 13-2).

Getting started with OpenLineage

There are Java and Python client APIs available (at the time of writing). In the following examples, we'll be using the Python client APIs. If you would like to explore the full example, it is available in the book's GitHub content.

Example 13-1 showcases how to create the `OpenLineageClient` instance and sets the metadata to assign the data producer, the upstream dataset, the named job, and the namespaced run instance, as well as simple functions to emit the start and complete events.

Example 13-1. Setting up the OpenLineage client to send start and complete events

```
client = OpenLineageClient.from_environment()
producer = 'common_foods.consumer.clickstream'
job_name = 'consumer.clickstream.orders'

datasets = {
  'clickstream': Dataset(namespace='consumer', name='consumer.clickstream')
}

cs_job: Job = Job(namespace='consumer', name=job_name)
```

```
# create the Run instance
run: Run = Run(f"{job_name}:{str(uuid4())}")

def emit_start(client, run, job, producer):
  run_event = RunEvent(
    RunState.START, datetime.now().isoformat(), run, job, producer)
    client.emit(run_event)

def emit_complete(
  client, run, job, producer, inputs: List[Dataset], outputs: List[Dataset]):
  run_event = (RunEvent(
    RunState.COMPLETE,
    datetime.now().isoformat(),
    run, job, producer,
    inputs, outputs,
  )
  client.emit(run_event)

# insert your data pipeline code
app = DataApplication(config)

# before you start the application
emit_start(client, run, job, producer)

# start your data application
app.run()

# before exiting the process
(if app.status() == 'complete':
  emit_complete(client, run, job, producer, app.inputs, app.outputs)
 else:
  emit_failed(client, run, job, producer, app.exception())
)
```

The code in Example 13-1 requires manual effort to construct string names and naming conventions in order to identify the data producer and the datasets, and to handle the construction of the Dataset, Job, Run, and RunEvent identifiers. Over time, it is much easier to use standard libraries and runtime environment variables, or common configurations, to streamline the generation of these lineage objects and remove the requirements of manual engineering effort—this helps to mitigate the risk that jobs end up reusing names and breaking the lineage. Just like with the "what not to do" covered in "Data Catalogs" on page 298, problems will arise when we ignore automation or convention-based engineering.

Simplified Lineage with Decorators and Abstractions

If you are familiar with Python decorators, then an avenue to simplifying how data and data application lineage is recorded could be provided as a function wrapping the run or execute method of your data application. When writing your integrations, ensure you provide a way to capture "failures," since we can also use data lineage to observe the current state of data in flight—even if that means there is no data in flight due to a breakdown in the runtime of a specific data application. If you are writing Scala or Java applications, then provide a simple trait or abstract base class that can be used to provide consistent hooks into the data lineage architecture.

The pseudo code in Example 13-2 provides a decorator over the run method of a PySpark application. The only expectation is that there is a `run` method that takes a `DataFrame` and returns a `StreamingQuery` object. This allows for the lineage recorder to parse the `StreamingQuery` object to gather more details about the sources and sinks, and to record the structure of the data flowing out of the application through the use of the schema method of the `DataFrame`.

Example 13-2. Decorating the `run` method for simplified data lineage

```
% python
 _app: Application = Application.fromEnv()
 @lineage.record(
   app: Application = _app,
   git: _app.git,
 )
 def run(df: DataFrame) -> StreamingQuery:
   …
```

While the code in Example 13-2 is just a snippet, it provides enough information to facilitate the generation of the Dataset, Job, Run, and RunEvent objects needed to track lineage via OpenLineage.

The way that data flows through the lakehouse and between our Delta Lake tables by way of our data applications ultimately provides the building blocks to create high-trust data products in a dynamic way—just like water moving between streams, rivers, and deltas and into reservoirs. Just like in nature, there will always be ebbs and flows, and ultimately certain areas that used to provide many downstreams will eventually dry up—but with the end of any data product, or the deprecation of an older source of data truth, there will always be new sources and new ways of connecting the data dots.

This is the beauty of capturing data lineage: when it is done correctly, the information provides a real-time or last "active" state of the what, when, and how, using a narrow or wide lens. This additional lineage-based metadata can then be combined

with other Delta table metadata to provide invaluable information regarding the connectivity graphs as well as information to be used for monitoring, alerting, or data discovery. Together this information can answer questions like "When was the last update to a table?," "What data source can I use now that the old source is deprecated?," and more.

Data Sharing

What does it mean to share data or a data asset? In the simplest way, we provide the ability for a known identity (a stakeholder, customer, system, or service) to consume a collection of data by reading it directly from our single source of data truth. For our Delta tables, this means providing the capabilities to a known identity to read the Delta transaction log and generate a snapshot of the table so they can execute a table read.

Chapter 14 covers sharing with the Delta Sharing protocol; this is the simplest way to enable sharing within the lakehouse.

There are many reasons why we would want to make our data available to others—for example, we may be able to monetize our data to provide insights not available to other companies (as long as it abides by data use laws and isn't creepy), or we may need to provide data to our partners or suppliers, which is often the case in retail. And in the case of data that isn't exiting our company, sharing data between internal lines of business is critical to ensuring that everyone references the same sources of data truth.

Automating Data Life Cycles

Earlier in the chapter, we were introduced to the concept that data and data assets are expected to survive only as long as necessary. When it comes to the natural life cycle of data, sometimes we have a choice, and at other times we are bound by legal and regional requirements. Either way, data has an expiration date. Some data is more like milk—it needs to be used or it will spoil rather quickly—while at other times our data acts more like honey, crystalizing over time but easily returning to a perfectly healthy state with a minor amount of effort. *So how can we automate these data life cycles?*

Using table properties to manage data life cycles

We learned to apply properties to our Delta tables in Chapter 5. In the same way that the Delta protocol uses properties to control the utility-based functionality to ease the repetitive maintenance of our tables, we can utilize tables to unify the way we handle

repetitive actions such as honoring data retention policies. The following techniques can be used to build a strategy for automatic data life cycle management or to run automatic compliance checks within your lakehouse.

Add the retention policy to the Delta table

The example shown in Example 13-3 introduces how to use the `INTERVAL` type to create a simple way of deleting data from our Delta tables. Three new table properties will be introduced; the naming conventions used in the book can be adjusted to fit the prefix patterns established in any lakehouse.

Using the properties prefix `catalog.table.gov.retention.*` will provide a namespace for our retention-specific use case.

Example 13-3. Add the table properties

```
% spark.sql(f"""
ALTER TABLE delta.`{table_path}`
SET TBLPROPERTIES (
  'catalog.table.gov.retention.enabled'='true',
  'catalog.table.gov.retention.date_col'='event_date',
  'catalog.table.gov.retention.policy'='interval 28 days'
)
""")
```

Whenever we add new governance behavior to our lakehouse, it is good to provide a way of opting into or out of a given feature. In this case, the `catalog.table.gov.retention.enabled` boolean can turn the feature on or off. Additionally, if the default state is false unless the property exists on the table, then it is much easier to opt in and ignore anything else.

Next, the code shown in Example 13-4 introduces a function to convert the interval value (28 days) into a `Column` object containing an `IntervalType`.

Example 13-4. Convert from a `StringType` to an `IntervalType`

```
% python
  def convert_to_interval(interval: str):
    target = str.lower(interval).lstrip()
    target = if target.startswith("interval"):
      target.replace("interval", "").lstrip()
    else:
      target
    number, interval_type = re.split("\s+", target)
    amount = int(number)

    dt_interval = [None, None, None, None]
    if interval_type == "days":
```

```
        dt_interval[0] = lit(364 if amount > 365 else amount)
    elif interval_type == "hours":
        dt_interval[1] = lit(23 if amount > 24 else amount)
    elif interval_type == "mins":
        dt_interval[2] = lit(59 if amount > 60 else amount)
    elif interval_type == "secs":
        dt_interval[3] = lit(59 if amount > 60 else amount)
    else:
        raise RuntimeException(f"Unknown interval_type {interval_type}")

    return make_dt_interval(
        days=dt_interval[0],
        hours=dt_interval[1],
        mins=dt_interval[2],
        secs=dt_interval[3]
    )
```

The Python function from Example 13-4 can now be used to extract the `catalog.table.gov.retention.policy` rule in the form of an Interval from a Delta table. Next, we will use our new `convert_to_interval` function to take a Delta table and return the earliest date that is acceptable to retain. This can be used to automatically delete older data from the table, or even just to mark the table as out of compliance. The final flow is shown in Example 13-5.

Example 13-5. Ensuring compliance through standards

```
% python
table_path = "..."
dt = DeltaTable.forPath(spark, table_path)
props = dt.detail().first()['properties']
table_retention_enabled = bool(
  props.get('catalog.table.gov.retention.enabled', 'false'))
table_retention_policy = (props.get(
  'catalog.table.gov.retention.policy', 'interval 90 days'))

interval = convert_to_interval(table_retention_policy)

rules = (
  spark.sql("select current_timestamp() as now")
  .withColumn("retention_interval", interval)
  .withColumn("retain_after", to_date((col("now")-col("retention_interval"))))
)

rules.show(truncate=False)
```

We lean on the DeltaTable utility function to provide us with a simple means of getting to our table properties. From the table properties, we extract out the retention-related config. This includes the boolean (feature flag) that defaults to false, as well as the retention policy, which defaults to 90 days. Using the *interval* variable,

which is the `IntervalType` column, we can then take the *current time* (when we run this expression), along with the results from `convert_to_interval`, and subtract the *interval* and then cast it to a `DateType` in the *retain_after* column. When we take a look at the rules `DataFrame`, we will see the following:

```
+--------------------------+-----------------------------------+------------+
|now                       |retention_interval                 |retain_after|
+--------------------------+-----------------------------------+------------+
|2024-03-24 20:11:27.759222|INTERVAL '28 00:00:00' DAY TO SECOND|2024-02-25  |
+--------------------------+-----------------------------------+------------+
```

So, when we look back 28 days from March 24, we see that the date is February 25, due to the leap year.

The example starting in Example 13-3 and concluding in Example 13-5 shows a way to provide life cycle policy controls to our Delta tables. There are many places we can take this pattern, should we decide to extend outside of just data deletion, or we can choose simply to use this example to ensure we take control over how we delete older data. Remember the delete conditions presented in Chapter 6? You can use the column identity provided in the `catalog.table.gov.retention.date_col` to delete data older than the *retain_after* date.

Audit Logging

Audit is another critical component and important lens required for compliance within the lakehouse. Because each data asset has a specific set of rules (policies) and entitlements that must be enforced for compliance sake, we must therefore provide a simple way to query the access and permissions change log and general audit log of resources being created or removed from the lakehouse.

Thinking along the lines of what operations need to be recorded, we can use specific actions within the lakehouse like a flight recorder—similar to the recording of data as it flows to generate end-to-end lineage. Rather than tracking the journey in terms of the data life cycle and how the data flows through the data network making up the lakehouse, we are recording activity regarding the state changes for our data management.

In Chapter 12 we explained that audit logging can be as simple as capturing changes in the behavior of the lakehouse—for example, when there are changes to the roles or policies for critical operations on highly controlled resources like catalogs, databases, and tables.

Additionally, it is important to track operations for data in flight to provide a source of data (metrics) to help identify anomalies that can in turn help mitigate risks and identify threats or the potential for bad actors to take advantage of holes in security.

For example, say we want to understand what user or group has access (at any point in time) to any data asset (resource). Additionally, we would like to know which identity (user or group) performed a given operation (action), or the inverse, for any operation (action) performed to understand the resource, owners, and who "should" have been able to perform the given action.

To provide the security and governance personas with timely information and enable system-wide peace of mind, data must be collected and made available within the lakehouse to enable simplified audit event collection. Streamlining the audit trail is outside the scope of this book, however, considering that every data asset must have an accountable owner, and that each operation requires access controls that are handled via IAM permissions and role-based policies. We can start small by simply capturing the changes to IAM for resources owned by specific mission-critical data products.

This would provide a simple and humble beginning and enable streamlined audit capabilities to emerge using the collective metadata for tables and their lineage, and then building upon that with additional data about the frequency with which tables are accessed, refreshed, and deleted from, or even just to track what tables are out of compliance using the techniques introduced in Example 13-3 for automatic data life cycle management.

Monitoring and Alerting

It is essential for the success of our lakehouse to provide monitoring and alerting capabilities. These can be used solely for the purpose of data governance and security capabilities, or they can be extended to ensure each data product has proper operational observability, monitoring, and alerting capabilities as well.

General compliance monitoring

Returning to the use case for retention automation (Example 13-3), we discussed the fact that the retention duration could be used to check if a table was out of compliance. For example, say the governance organization required all table-based data assets to enable the `catalog.table.gov.retention.*` properties.

Aided by the data catalog, the governance engineers could easily set up a metadata read-only integration to check if the table owners have followed the rules and enabled retention policy configs to their tables. The scan could happen daily, recording which tables are out of general compliance, and could automatically use the `catalog.engineering.comms.[email|slack]` properties (introduced in Chapter 5) to send automated communications to the teams, or to escalate to the heads of the engineering organization. In this case, the alert isn't so much a PagerDuty alarm but could very well be integrated to page a team to be in compliance.

Data quality and pipeline degradations

We touched upon data quality when discussing the medallion architecture. For each table-based data asset (with a known data customer), if the pipeline fails, or if columns that once held important data go empty, this lets down the downstream consumers (the data customers). If there are table properties introduced to each Delta table to convey how often data is expected to land (the cadence of table refreshes, or freshness), then these can be used to automatically alert the data-producing team that things have gone wrong.

For a real scenario, the table properties introduced in Example 13-6 show four simple properties that provide a lot of powerful information.

Example 13-6. Declaring the intentions of each Delta table

```
% spark.sql(f"""
ALTER TABLE delta.`{table_path}`
SET TBLPROPERTIES (
  'catalog.table.deprecated'='false',
  'catalog.table.expectations.sla.refresh.frequency'='interval 1 hour',
  'catalog.table.expectations.checks.frequency'='interval 15 minutes',
  'catalog.table.expectations.checks.alert_after_num_failed'='3'
)
""")
```

Using the techniques introduced in Example 13-3 through Example 13-5, we can leverage a simple pattern to automatically run checks for a given table. The theory here is that unless a table is deprecated, there should be a known data service-level agreement (DSLA), or, at a minimum, specific DLOs and DLIs. With respect to our data assets (tables specifically), our downstream consumers tend to want to know the frequency with which data "becomes" available, or how often it is refreshed.

When making decisions based on when to use batch processing or microbatch processing, it usually comes down to the expectations of one or more upstream data sources. If nearly all sources usually refresh in under 15 minutes, but one source only updates daily, then if you need all data to provide specific data answers, you'll always be stuck in batch processing mode or be wasting money waiting on the laggard dataset. Making it easier to understand the average update frequency for a given table (without requiring meetings) can empower engineers and analysts to make decisions about whether streaming or batch processing makes the most sense to solve a problem.

Then when things go wrong, or when your pipelines stall due to "no new data" from your upstreams, you can check the DLOs for the laggard tables to understand what might have changed. Hopefully, if we've also incorporated data application lineage,

we can check to see how recently the data application that is powering the poorly behaving table was updated.

Leaning on the data lineage tables and some creative energy, a simple UI could also be built to provide up-to-date information about the data flow within your lakehouse and about what tables in the path are in compliance or are running slower than expected, or really any use case that can be automated to reduce meetings.

What Is Data Discovery?

Data discovery enables users to search (and explore) the data catalog to locate data assets (resources) within the lakehouse using a simple text-based interface. Behind the scenes, the discovery engine facilitates search by leaning on the metadata made available to it through the lakehouse data catalog and metastore. The same information required for metadata management, monitoring, and lineage comes together to enhance the search capabilities by providing a more robust search index.

Data discovery within the context of the lakehouse differs from that of traditional data catalog–based discovery, since the data that is present in the search index already resides within the lakehouse. It is common to see most sources of data within a large enterprise cataloged within a business data catalog; that capability is typically the starting point for business stakeholders to make informed decisions about what data to bring into the lakehouse.

For data discovery, a solution to the problem can be as simple as adding the table metadata (ownership and rules, as well as immediate upstream and downstream lineage) to an ElasticSearch index. If we wanted to layer in additional capabilities to the discovery engine—whether catalogs, databases/schemas, or other data asset types—we would only need to modify the types of metadata in our index and modify the search parameters to handle more complex discovery. Depending on the size and number of assets being maintained, the solution could be scaled accordingly, but for fewer than one million data assets, a simple ElasticSearch index would take us a very long way.

Considering what sorts of answers the customers of the lakehouse would be searching for can help inform what it means to be successful. In some cases, having validated "highly reliable" tables or "verified" owners is a useful step to reduce the number of tables matching the search criteria. As long as the process to get a specific tag or badge is a controlled process (meaning not just anyone can add their own tag), then the customers will trust that the process can't be gamed. If nothing else, think about how to balance complexity in terms of moving parts for the data discovery solution: How many sources of metadata need to be indexed, and how often? Is there a simple way to be notified when things change? Can we automate the process?

Having a good solution for data discovery can save countless hours and really raise the bar for a data organization. Just remember to balance speed and accuracy; a fast search result on bad data could waste company resources and lead to low trust in the lakehouse.

Conclusion

This chapter explored the value of metadata within the context of the lakehouse. Specifically, we looked at how metadata management acts as a critical component of the lakehouse platform and at how to utilize basic data asset information to capture more complex data flows through the use of data lineage. We spent time investigating how data lineage can be enhanced with data application lineage to enable context-aware insights, and we concluded with a brief overview of data discovery. In the next and final chapter, we will be looking at how data sharing with Delta Sharing completes the final component required for comprehensive lakehouse governance and security.

CHAPTER 14
Data Sharing with the Delta Sharing Protocol

Sharing is a natural part of life. We share as an avenue to communicate pride with respect to accomplishments or to relay information related to our other emotions, be they joy, anger, frustration, bliss—really, the full gamut of human expression. As kids, we learn to share toys, whether we'd like to or not, as the simple act of sharing introduces others to an experience they may otherwise be excluded from. As we mature, we share meals with friends and family as a token of our gratitude, or simply to come together and reunite. So sharing is very much a natural part of our world.

With respect to our Delta tables, we share the fruits of our labor—whether internally to our organization, or externally—for myriad reasons to reduce the level of effort for other data teams who require access to the valuable data contained within the tables. However, the process of sharing data is itself not always so cut-and-dried.

For example, it is still common for data teams to set up periodic jobs with the sole purpose of extracting (copying) tabular data from one source of truth—say, their foundational Delta tables—before transforming each batch of rows into a common intermediate format, like JSON, and then writing the transformed data (again) into an alternative cloud storage location (either internally or externally). In other cases, data teams rely on SFTP (SSH File Transfer Protocol) and even good old email to send data back and forth. We might ask ourselves where the problem lies—isn't it safe to copy data from point A to point B? Isn't that essentially what data engineering is? We'd be correct in asking these questions.

The problem commonly encountered is actually a complexity problem based on divergent sources of data truth. Rather than having a single table representing a foundational dataset, we now must manage the complexity between all copies and

deal with the expectations of all the teams represented by each downstream location to which we are actively exporting data.

Imagine that parts (partitions) of our table are exported to support 40 separate external locations, with each location representing a different cloud storage bucket, or prefix within a given bucket. Now, for each of our 40 separate locations, we include the added constraints of minimal permissions, and the sneaky problem of invalid (revoked) access permissions. What happens if we need to replace data in one or more partitions due to system failures or faults? Things tend to go wrong the more complex a system grows. Not to mention, there is a cost associated with reprocessing all the data again (for each downstream location), and this cost is included on both sides, represented by egress and ingress—when all along there has been an active single source of data truth represented by the original Delta table.

The problem described above is the issue with distributed synchronization—given that we can't assume that each export job will always succeed, we therefore must also *carry state* alongside each of our *simple* export jobs. So the simple act of periodic data export can easily become a complex and fragile process. Now for the good news: this chapter introduces the Delta Sharing Protocol, which is purpose-built to provide a secure and reliable way to share our Delta tables, regardless of where each table originates, and regardless of which cloud storage provider is used to store the table.

The Basics of Delta Sharing

The Delta Sharing Protocol provides an open solution to securely share live data from our lakehouse to any compute platform. Due to the open nature of the protocol, it is vendor and cloud agnostic, supporting the common cloud storage providers through the use of plug-ins without requiring one cloud over the other—or if we are running on prem, we can ditch the need for the cloud altogether.

In the chapter introduction, we were presented with the common problem of distributed synchronization, which introduces additional complexity, storage and compute costs, and complex state management to ensure that exported data is kept in sync with the originating Delta tables. A really impressive benefit when using Delta Sharing is that we remove the need to manage the complexity of managing exports altogether; rather, we need only concern ourselves with the creation of secure *shares*. Figure 14-1 shows this high-level concept. Each share provides the guardrails and access controls for our Delta tables and views, while removing the need to export data in the first place. Each share provides the *recipient* of the share with the ability to query any source-of-truth table or to view configuration by the share itself, with the bonus of being able to simply fetch table metadata (including the table properties and the table schema itself), and even to discover what tables are made available through the share itself.

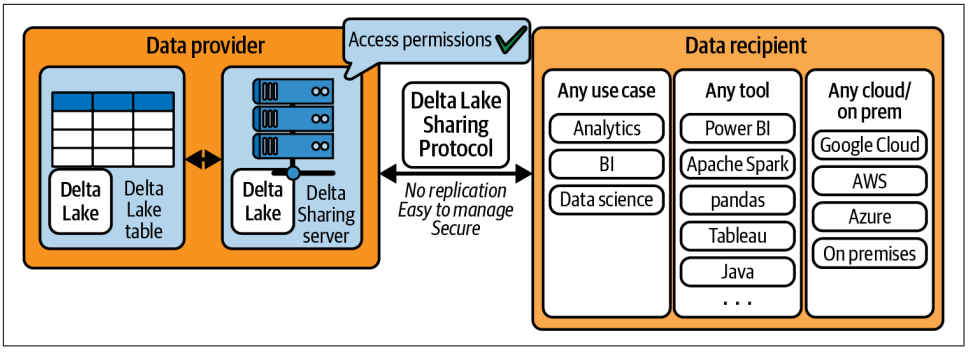

Figure 14-1. The relationship between the data provider and the data recipient

The relationship between the data provider and the data recipient can be thought of as being the same as the relationship between the data producer and the data consumer. On one side, the owner of the table or view is responsible for delegating a share. This share represents a presigned acknowledgment that the consumer of the data (the recipient) can access the Delta tables contained within the configuration of the respective share. Now let's look more closely at the notion of shares and recipients through the lens of data providers and recipients.

Data Providers

Data providers are responsible for managing access to their data products through the use of a share. A share represents a logical grouping of schemas, and of the tables or views accessible within each schema, to be shared with the recipients. Each recipient is an abstraction over an identity, known as a *principal*, which can act on behalf of a user, system, or service to provide read-only access to the tables or views allowed by a share (which we will go into in the next section).

Each share can be shared with one or more recipients, and each recipient can access all resources contained within a share. To put this information into perspective, an example share configuration is presented in Example 14-1. The share itself is configured in a similar way to an IAM-based policy file, providing the specific location of the tables or views that the recipient can access while reducing the complexity of managing cross-cloud (or on-prem) identity and access management (IAM). Lakehouse security and governance are covered in earlier chapters, if these concepts are new and a refresher is required.

Example 14-1. Configuring a share

```
version: 1
shares:
- name: "consumer_marketing_analysts_secure-read"
  schemas:
```

```
  - name: "consumer"
    tables:
    - name: "clickstream_hourly"
      location: "s3a://.../common_foods/consumer/clickstream_hourly"
      id: "eb6f82f5-a738-4bd8-943c-9cd8594b12ac"
```

Example 14-1 enables the recipient—in this case, the consumer marketing analysts—to access hourly clickstream data. The configuration itself can contain many different shares representing many different policies for many different recipients, and for each uniquely identified share, a collection of one or more schemas can be configured, with one or more tables or views per schema. This pattern enables us to simplify access controls through the use of logical groups. We will be looking into how this configuration is used later in the chapter when we explore the Delta Sharing server.

Data Recipients

The recipient of a share is a principal identified by a bearer token. While we go into much more detail regarding identity and access management in earlier chapters, it is worth pointing out that a principal represents a known identity, and the identity can be at the user level, or represent a logical group like a team or even an entire department or business unit, or be strictly headless—meaning it represents a system or service that is not human acting on behalf of a human (hence the headlessness).

With the Delta Sharing Protocol, there isn't a mechanism to support fine-grained access controls for individual users within a group (team, business unit, etc.). If you want to provide variable levels of access to individuals, you will need to provide each user (identity) their own recipient profile.

All of the information required to authenticate against the Delta Sharing server is packaged for the recipient in a simple profile file. Example 14-2 introduces us to the format of the profile, which is represented by a JSON object.

Example 14-2. The recipient profile

```
{
  "shareCredentialsVersion": 1,
  "endpoint": "https://commonfoods.io/delta-sharing/",
  "bearerToken": "<token>",
  "expirationTime": "2023-08-11T00:00:00.0Z"
}
```

The profile contains all the information necessary to authenticate with the Delta Sharing server from the `delta-sharing` client:

`shareCredentialsVersion`
: The file format version of the profile file. This version will be increased whenever non-forward-compatible changes are made to the profile format. When a client is running an unsupported profile file format version, it should show an error message instructing the user to upgrade to a newer version of their client.

`endpoint`
: The URL of the sharing server.

`bearerToken`
: The bearer token to access the server. This is just an opaque OAuth 2.0 token. The contents of the token can be as simple as a hash, or it can hold meaning, as with JWT tokens. It all depends on the authentication mechanism used and on whether we're using unstructured or structured tokens.

`expirationTime`
: The expiration time of the bearer token in ISO-8601 format. This field is optional, and if it is not provided, the bearer token can be seen as never expiring.

It is worth pointing out that while we can create long-lived or even perpetual tokens, it is a security antipattern and bad practice. Instead, always rotate your secrets (keep them secret, keep them safe!), and provide an API for external recipients to reauthenticate and retrieve their updated profile file and associated token. Being safe is much better than being sorry—especially when it comes to our Delta tables.

In the next section we will look at the Delta Sharing server. This service implements the Delta Sharing Protocol and offers a simple-to-use REST API powering the sharing service as well as the introspection API used by the Delta Sharing clients themselves.

Delta Sharing Server

The Delta Sharing Protocol provides a universal mechanism to create trust relationships between the data assets (schemas, tables, views, libraries, notebooks, AI models, dashboards, and more) owned by one identity and the one or many recipients of trust represented by a *share*. We can safely state that the promise of the sharing server is to act as both a bouncer—entrusted to accurately authenticate, authorize, and allow *recipients* access to a known share—and the authorized broker providing schema, table, and view metadata, as well as presigned access to the files making up a specific `Snapshot` of a given Delta table or view required to execute a table read.

We've briefly introduced the mechanics of the share (see Example 14-1) and the recipient (see Example 14-2), and we will now dive into the Delta Sharing REST APIs before discussing common strategies for managing the trust relationships encapsulated by the share and the recipients of a given share.

The Delta Sharing OSS project provides a reference implementation of the Delta Sharing server that can be used as you get started in your open source journey. Head on over to the Delta Sharing GitHub repository to get started.

Using the REST APIs

The REST APIs provide capabilities for a recipient to explore their share, view what schemas and tables, or views, they have access to, and even query the tables and views directly. All API requests must be signed with a bearer token, which is conveniently made accessible to the Delta Sharing clients through the recipient profile file. We will look at the API routes and view examples to help build a working model of the capabilities provided by the Delta Sharing server, which we will call the *Delta Sharing service* from here on out.

The REST APIs are intended to ensure that the Delta Sharing Protocol can be implemented easily by folks building Delta Sharing clients. If you are interested in using the Delta Sharing clients and want to skip the REST APIs section, then just move ahead to the section "Delta Sharing Clients" on page 332, as the rest of this section covers the REST API methods, all of which are encapsulated by most of the Delta Sharing clients.

Anatomy of the REST URI

The Delta Sharing service URI enables simplified scaling as well as routing using the concept of the sharing prefix:

```
% https://{endpoint}/{prefix}/{api-route}
```

It is common practice to enable load balancing and route redirection through the service URI. In this case, we can apply simple load-balancing requests using the `{prefix}` path. In addition to simple load balancing across the servers backing the deployment of our Delta Sharing service, we may also want to be intentional about how we route requests to different sharing instances based on their associated data domains.

For example, we could amend the prior example by adding more concrete use cases. Let's say we have established a set of four data domains—consumer, commercial,

analytics, and insights. Now we can use name-based routing via the sharing prefix to direct each request to the appropriate sharing endpoint, enabling each data domain to fulfill a specific share-based request:

```
% https://{endpoint}/<consumer|commercial|analytics|insights>/{api-route}
```

Consider when we first introduced the recipient profile files. Whereas we previously had a common route prefix named `delta-sharing` under the endpoint property of the recipient profile file, we can now be more consistent with respect to where the share lives within the distributed ecosystem:

```
{
  "shareCredentialsVersion": 1,
  "endpoint": "https://sharing.commonfoods.io/consumer/",
  "bearerToken": "<token>",
  "expirationTime": "2023-08-11T00:00:00.0Z"
}
```

Now the recipient profile file is specifically pointing to the `consumer` prefix. In the case where we need to redirect or modify the prefix again in the future, we can use simple DNS, or force the recipient to reauthenticate and receive a new profile pointing to the new location endpoint.

When we use the sharing service to distribute requests across logical data domains, we end up embracing the decentralized nature of how data is distributed across natural organizational boundaries. This also makes it easier to scale based on specific workloads, rather than needing to arbitrarily scale up to meet "any" demands.

Next, we'll move onto the actual API methods and see how a recipient can explore the capabilities associated with their unique share.

List Shares

REST APIs commonly provide a list resource—the request parameters are shown in Table 14-1. In this case, the resource provides the means to view the variable number of shares that have been configured and assigned to the recipient identified by the provided bearer token on the request. Running the code in Example 14-3, we see how simple it is to explore what data assets we have access to, beginning with the most basic concept of the Delta Sharing Protocol—the humble share.

Table 14-1. List shares API request parameters

HTTP request	Value
Method	GET
Header	Authorization: Bearer {token}
URL	{prefix}/shares

HTTP request	Value
Query parameters	`maxResults` (type: Int32, optional): The maximum number of results per page that should be returned. If the number of available results is larger than `maxResults`, the response will provide a `nextPageToken` that can be used to get the next page of results in subsequent list requests. The server may return fewer than `maxResults` items even if there are more available. The client should check `nextPageToken` in the response to determine if there are more available. Must be nonnegative. 0 will return no results, but `nextPageToken` may be populated. `pageToken` (type: String, optional): Specifies a page token to use. Set `pageToken` to the `nextPageToken` returned by a previous list request to get the next page of results. `nextPageToken` will not be returned in a response if there are no more results available.

Example 14-3. Using the Delta Sharing Protocol to list configured shares

```
% export DELTA_SHARING_URL="https://sharing.delta.io"
  export DELTA_SHARING_PREFIX="delta-sharing"
  export DELTA_SHARING_ENDPOINT="$DELTA_SHARING_URL/$DELTA_SHARING_PREFIX"
  export BEARER_TOKEN="faaie590d541265bcab1f2de9813274bf233"
  export REQUEST_URI="shares"
  export REQUEST_URL="$DELTA_SHARING_ENDPOINT/$REQUEST_URI"
  export QUERY_PARAMS="maxResults=10"

  curl -XGET \
    --header 'Authorization: Bearer $BEARER_TOKEN' \
    --url "$REQUEST_URL?$QUERY_PARAMS"
```

The response from the sharing service will provide us with a list of the one or many shares that have been configured for us, the recipient. The response to our request is as follows:

```
{
  "items":[
    {"name":"delta_sharing"}
  ]
}
```

The object returned is a collection identified by items, with a single item representing a share with the `name` of `delta_sharing`. The protocol also allows the share record to contain an `id` field:

```
% {
    "name": "<unique_share_name>",
    "id": "<uuid_or_hash>"
  }
```

If the optional `id` field is present, the value of the `id` must be *immutable* for the lifetime of the share.

Using the shares as a starting point, we can introspect what is available in a given share using the share introspection endpoint—in this case, we are going to see what the *delta_sharing* share entails.

Get Share

Each share can contain one or more schemas, and within each schema, one or more tables or views (or other data assets) can be configured. To view a share, we must first use the list shares API to understand what shares are available for us to view. Next, we just need to send our request to the API endpoint. Example 14-4 shows the full request, while Table 14-2 shows the API request parameters required to complete the request.

Table 14-2. Get share API request parameters

HTTP request	Value
Method	GET
Header	Authorization: Bearer {token}
URL	{prefix}/shares/{share}
URL parameters	**{share}**: The share name to query. It's case insensitive.

Example 14-4. Sending a request to the share endpoint

```
% ...
  export REQUEST_URI="shares/delta_sharing"
  export REQUEST_URL="$DELTA_SHARING_ENDPOINT/$REQUEST_URI"
  curl -XGET \
   --header 'Authorization: Bearer $BEARER_TOKEN' \
   --url "$REQUEST_URL"
```

The result of issuing the request to the get share endpoint isn't much different from the list shares endpoint:

```
    % {
        "share":{
          "name":"delta_sharing"
        }
      }
```

The only change from the list shares endpoint is the result is now a single object rather than the array of items. The results of this request are unique to the shares configured for a recipient.

Next we will look at how to introspect the schemas associated with the share itself.

List Schemas in Share

To view the configured schemas we are authorized to access, we use the list schemas endpoint. Example 14-5 shows the full request, while Table 14-3 provides the API request parameters required to complete the request.

Table 14-3. List schemas API request parameters

HTTP request	Value
Method	GET
Header	Authorization: Bearer {token}
URL	{prefix}/shares/{share}/schemas
URL parameters	**{share}**: The share name to query. It's case insensitive.
Query parameters	`maxResults` (type: Int32, optional): The maximum number of results per page that should be returned. If the number of available results is larger than `maxResults`, the response will provide a `nextPageToken` that can be used to get the next page of results in subsequent list requests. The server may return fewer than `maxResults` items, even if there are more available. The client should check `nextPageToken` in the response to determine if there are more available. Must be nonnegative. 0 will return no results, but `nextPageToken` may be populated. `pageToken` (type: String, optional): Specifies a page token to use. Set `pageToken` to the `nextPageToken` returned by a previous list request to get the next page of results. `nextPageToken` will not be returned in a response if there are no more results available.

Example 14-5. Sending a request to the list schemas endpoint

```
% ...
  export REQUEST_URI="shares/delta_sharing/schemas"
  export REQUEST_URL="$DELTA_SHARING_ENDPOINT/$REQUEST_URI"
  export QUERY_PARAMS="maxResults=10"

  curl -XGET \
   --header 'Content-Type: application/json' \
   --header 'Authorization: Bearer $BEARER_TOKEN' \
   --url "$REQUEST_URL?$QUERY_PARAMS"
```

As we observed with the list shares endpoint, the list schemas endpoint provides capabilities to paginate over an arbitrary number of schemas. While pagination may not be required in all cases, the way pagination works is the same for all list resources:

```
    % {
        "items":[
          {"name":"default","share":"delta_sharing"}
        ],
        "nextPageToken": "..."
      }
```

As we traverse the hierarchical tree from the share, now to the schemas, we are essentially unwrapping the exact same structure that represents our actual share itself. For context, look back at Example 14-1, where we learned to configure a share.

Next, we will learn to list the tables available underneath a specific schema, using the `default` schema returned from the request in Example 14-5.

List tables in schema

To view the configured tables of a given schema accessible by our recipient profile, we use the list tables endpoint. Example 14-6 shows the full request, while Table 14-4 provides the API request parameters required to complete the request.

Table 14-4. List tables API request parameters

HTTP request	Value
Method	GET
Header	Authorization: Bearer {token}
URL	{prefix}/shares/{share}/schemas/{schema}/tables
URL parameters	**{share}**: The share name to query. It's case insensitive. **{schema}**: The schema name to query, It's case insensitive.
Query parameters	`maxResults` (type: Int32, optional): The maximum number of results per page that should be returned. If the number of available results is larger than `maxResults`, the response will provide a `nextPageToken` that can be used to get the next page of results in subsequent list requests. The server may return fewer than `maxResults` items, even if there are more available. The client should check `nextPageToken` in the response to determine if there are more available. Must be nonnegative. 0 will return no results, but `nextPageToken` may be populated. `pageToken` (type: String, optional): Specifies a page token to use. Set `pageToken` to the `nextPageToken` returned by a previous list request to get the next page of results. `nextPageToken` will not be returned in a response if there are no more results available.

Example 14-6. Sending a request to the list tables endpoint

```
% ...
  export REQUEST_URI="shares/delta_sharing/schemas/default/tables"
  export REQUEST_URL="$DELTA_SHARING_ENDPOINT/$REQUEST_URI"
  export QUERY_PARAMS="maxResults=4"

  curl -XGET \
    --header 'Content-Type: application/json' \
    --header 'Authorization: Bearer $BEARER_TOKEN' \
    --url "$REQUEST_URL?$QUERY_PARAMS"
```

The response from the list tables request is shown next:

```
% {
    "items":[
     {"name":"COVID_19_NYT","schema":"default","share":"delta_sharing"},
     {"name":"boston-housing","schema":"default","share":"delta_sharing"},
     {"name":"flight-asa_2008","schema":"default","share":"delta_sharing"},
     {"name":"lending_club","schema":"default","share":"delta_sharing"}
    ],
    "nextPageToken":"CgE0Eg1kZWx0YV9zaGFyaW5nGgdkZWZhdWx0"
  }
```

We see that the service returned four tables in the result and is honoring the `maxResults` query parameter. Because the `nextPageToken` is included in the response object, we can now return this to the service in order to fetch the next set of tables, as we see in Example 14-7. If there were no more results, then the absence of the `nextPageToken` declares that we are at the end of the list.

Example 14-7. Continuing the list tables query with pagination

```
% ...
export QUERY_PARAMS="maxResults=4&nextPageToken=CgE0Eg1kZWx0YV9zaGFyaW5nGgdkZWZhdWx0"
curl \
  --request GET \
  --header 'Content-Type: application/json' \
  --header 'Authorization: Bearer $BEARER_TOKEN' \
  --url "$REQUEST_URL?$QUERY_PARAMS"
```

Given that the share is set up with a single schema (default), and underneath that schema there is a total of only seven tables—and because we are limiting the `maxResults` per request to just four tables—it takes us two requests to get the full list of tables:

```
% {
  "items":[
    {"name":"nyctaxi_2019","schema":"default","share":"delta_sharing"},
    {"name":"nyctaxi_2019_part","schema":"default","share":"delta_sharing"},
    {"name":"owid-covid-data","schema":"default","share":"delta_sharing"}
  ]
}
```

Now there is a better way of quickly viewing all tables available to us, without requiring us to first descend the hierarchical tree from the shares to the schemas of an individual share, and then again descend into one or more schemas per share to view the configured tables. We can simply use the next API to query all tables available to us.

List All Tables in Share

To quickly view all configured tables for our share, we use the list all tables endpoint. Example 14-8 shows the full request, while Table 14-5 provides the API request parameters required to complete the request.

Table 14-5. List all tables API request parameters

HTTP request	Value
Method	GET
Header	Authorization: Bearer {token}
URL	{prefix}/shares/{share}/all-tables
URL parameters	**{share}**: The share name to query. It's case insensitive.
Query parameters	`maxResults` (type: Int32, optional): The maximum number of results per page that should be returned. If the number of available results is larger than `maxResults`, the response will provide a `nextPageToken` that can be used to get the next page of results in subsequent list requests. The server may return fewer than `maxResults` items, even if there are more available. The client should check `nextPageToken` in the response to determine if there are more available. Must be nonnegative. 0 will return no results, but `nextPageToken` may be populated. `pageToken` (type: String, optional): Specifies a page token to use. Set `pageToken` to the `nextPageToken` returned by a previous list request to get the next page of results. `nextPageToken` will not be returned in a response if there are no more results available.

Example 14-8. Sending a request to the list all tables endpoint

```
% ...
  export REQUEST_URI="shares/delta_sharing/all-tables"
  export REQUEST_URL="$DELTA_SHARING_ENDPOINT/$REQUEST_URI"
  export QUERY_PARAMS="maxResults=10"

  curl -XGET \
    --header 'Authorization: Bearer $BEARER_TOKEN' \
    --url "$REQUEST_URL?$QUERY_PARAMS"
```

The response from the endpoint is as follows:

```
    % {
        "items":[
          {"name":"COVID_19_NYT","schema":"default","share":"delta_sharing"},
          {"name":"boston-housing","schema":"default","share":"delta_sharing"},
          {"name":"flight-asa_2008","schema":"default","share":"delta_sharing"},
          {"name":"lending_club","schema":"default","share":"delta_sharing"},
          {"name":"nyctaxi_2019","schema":"default","share":"delta_sharing"},
          {"name":"nyctaxi_2019_part","schema":"default","share":"delta_sharing"},
          {"name":"owid-covid-data","schema":"default","share":"delta_sharing"}
        ]
      }
```

Now that we've learned the basics of the Delta Sharing service, it is time to start having a little more fun and embrace using the Delta Sharing clients. The next set of APIs we will learn about are more easily viewed through the lens of the Delta Sharing clients, as they are purpose-built to speak the same language.

Delta Sharing Clients

The next set of APIs are available using the Delta Sharing Clients. We will learn to query the table version and metadata and then finish by learning to query the physical tables themselves. We will start off basic and learn to simply read rows from the table, and then we'll learn about the more advanced change data feed capabilities—assuming a given table is configured to track changes, which we can introspect by using the table properties and looking for the presence of `delta.enableChangeDataFeed=true`.

The source code for the following examples is located in the book's GitHub repository under */ch14/delta-sharing/*.

Delta Sharing with Apache Spark

The Delta Sharing clients for the Apache Spark ecosystem are familiar and simple to get started with. Due to the nature of the Spark ecosystem, the core libraries are all written for the JVM, and wrapping libraries are provided to interact with the PySpark ecosystem and convert our Spark SQL queries into catalyst expressions. This section will look at using the PySpark client, then the Spark Scala client, and finally, the Spark SQL extension for Delta Sharing.

PySpark client

Getting started with the PySpark client requires the `delta-sharing` Python package, which can be installed locally using `pip install delta-sharing`. In addition to the Python wrappers, if you want to be able to run a local `pytest`, you will also need to bring the necessary JARs to your local `SparkSession`. We will walk through the end-to-end use case now, starting with Example 14-9, where we create an instance of the `SharingClient` and generate the share URL, which encapsulates the profile file as well as the share, schema, and table we will be reading.

Example 14-9. Generating the share URL to use the Delta Sharing client

```
%
profile_path = ...
sharing_client = SharingClient(f"{profile_path}/open-datasets.share")
shares = sharing_client.list_shares()
first_share: Share = shares[0]

schemas = sharing_client.list_schemas(first_share)
first_schema: Schema = schemas[0]

tables = sharing_client.list_tables(first_schema)

lending_club: Table = tables[3]
```

The code from Example 14-9 instantiates the `SharingClient` by passing a reference to the location on the filesystem where we've stored our recipient profile file. We then fetch the list of available shares and for simplicity's sake take the first entity—which is a `Share` object—from the results list to use to fetch our schemas. We repeat this same pattern, taking the first `Schema` from the results list to fetch what tables are available to us. Consider this series of operations to just be a hierarchical traversal from the share to the schema.

Last, we retrieve the list of tables and take the `Table` object representing the `lending_club` remote Delta table (since we will be querying the lending club Delta table). The `Table` object provides us with everything we need to generate the `table_url`, which is required by the sharing client to query the remote table.

The function in Example 14-10 is from the book's source code, and it provides a simple way to generate the full `table_url` required for reading the remote Delta table.

Example 14-10. Generate the Delta Sharing table URL

```
% def table_url(self, table: Table) -> str:
  table_uri = f"#{table.share}.{table.schema}.{table.name}"
  return f"{self._profile.as_posix()}{table_uri}"
```

This method allows us to pass a `Table` object to an instance of our Sharing helper class, and the end result is the Delta Sharing table URL required to query the table. The URL is the concatenation of the path to the share profile file along with the *`<share>.<schema>.<table>`*. For example, executing the function from Example 14-10—`self.table_url(lending_club)`—yields the following `table_url`:

```
…/delta-sharing/profiles/open-datasets.share#delta_sharing.default.lending_club
```

Now, in order to read the remote table using the Delta Sharing client, we'll need to generate a `SparkSession` that includes the `delta-sharing-spark` JAR:

```
    spark: SparkSession = (
      SparkSession.builder
      .master("local[*]")
      .config("spark.jars.packages", "io.delta:delta-sharing-spark_2.12:3.1.0")
      .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
      .config(
        "spark.sql.catalog.spark_catalog",
        "org.apache.spark.sql.delta.catalog.DeltaCatalog"
      )
      .appName("delta_sharing_dldg")
      .getOrCreate()
    )
```

Armed with our `SparkSession` and the `table_url` from Example 14-10, we can now read from the remote Delta table using the new `deltaSharing` format on our DataFrameReader. The code in Example 14-11 shows us how to do that.

Example 14-11. Reading a remote Delta table using `deltaSharing`

```
df = ((
   spark.read
   .format("deltaSharing")
   .option("responseFormat", "parquet")
   .option("startingVersion", 1)
   .load(table_url)
  ).select(
    col("loan_amnt"),
    col("funded_amnt"),
    col("term"),
    col("grade"),
    col("home_ownership"),
    col("annual_inc"),
    col("loan_status")
))
```

Behind the scenes, the `delta-sharing` Python library and the underlying `delta-sharing-spark` Scala library work together to negotiate the network calls to the Delta Sharing service, utilizing the table version API (`startingVersion = 1`), which if implemented on the sharing service allows our remote procedure call to time travel to a specific version of the remote Delta table. We also are using the `responseFormat` option on the reader. The available options at the time of writing are either Parquet or Delta.

However, ignoring what is happening behind the scenes, the process is fairly transparent with respect to how we write our data applications. Given we can utilize the full set of DataFrame functions, there is no significant difference, except that we now

can directly query a remote Delta table with the benefits of cloud-agnostic IAM and the complications presented in Chapter 12.

Spark Scala client

We just looked at the PySpark client in more detail. To use the Scala-based Delta Sharing client, we need a similar but simplified process. Rather than requiring the PySpark bindings, we can focus solely on using the `io.delta:delta-sharing-spark_2.12:3.1.0` package.

The code in Example 14-12 shows a simple example using the Spark Scala client.

Example 14-12. Reading a remote Delta table using the Scala Delta Sharing extensions

```
% import org.apache.spark.sql.functions.{col}

val dldg_path = "/path/to/book/github/chapter/"
val profile_file_location = f"$dldg_path/delta-sharing/profiles/open-datasets.share"

val table_url = f"$profile_file_location#delta_sharing.default.owid-covid-data"

val df = spark.read
         .format("deltaSharing")
         .load(table_url)
         .select("iso_code", "location", "date")
         .where(col("iso_code").equalTo("USA"))
         .limit(100)
```

The same DataFrameReader options are available for the PySpark and Spark Scala clients. The only difference between this and the code in Example 14-11 is the addition of the `where` and `limit` clauses.

Depending on how the Delta Sharing service has been implemented, the `where` clause can be handled as a direct predicate pushdown, or the client can handle the filtering. Given that the Delta Sharing server is based on an open standard, the service implementation should be checked if we are experiencing less-than-ideal query times. The mechanism for handling modifications to the service response is through the use of hints. There are *jsonPredicateHints* as well as *limitHints*. These are all done using best effort and will evolve as the Delta Sharing Protocol does.

The list of available DataFrameReader options for the `delta Sharing` reader is presented in Table 14-6.

Spark SQL client

The Delta Sharing SQL extension provided by the `delta-sharing-spark` JAR enables us to easily access remote tables in a secure and efficient way. Assuming we have a `SparkSession` generated, we can use the SQL method `spark.sql(...)` to query the remote table:

```
% sql
CREATE TABLE lending_club USING deltaSharing
LOCATION '<profile-file-path>#delta_sharing.default.lending_club';

SELECT * FROM lending_club;
```

This opens up myriad ways to securely access external data that can also be mixed with secure local or internal tables. For example, say we are tasked with building some business intelligence reports that require access to data that is provided by an external business partner or vendor. We can safely join our internal data with their external data and remove the problems that can pop up due to copying and divergent sources of data truth.

Stream Processing with Delta Shares

We were introduced to stream processing with Delta in Chapter 7 and are now familiar with the notion of incremental processing of unbounded tables. The same processing paradigms are available to us with Delta Sharing as well. Behind the scenes, the sharing client utilizes the remote table version API, along with the checkpoints of the structured streaming app; when combined, they enable us to read the remote transaction log and actively stay in sync with our remote sources of data truth:

```
% val tablePath = "<profile-file-path>#<share-name>.<schema-name>.<table-name>"
val df = spark.readStream.format("deltaSharing")
  .option("startingVersion", "1")
  .option("skipChangeCommits", "true")
  .load(tablePath)
```

Since the remote shared table acts just like any of our other tables (or data sources), we can simply treat it the same way when we write our streaming data applications. This makes life so much simpler, since the only real differences are (a) the way in which we identify the table and (b) the way in which we authenticate; the rest of the APIs remain the same.

Now that we've seen how to read streaming tables using the Delta Sharing client, we can close out this section with the available options for the Delta Sharing clients. Table 14-6 provides a handy overview of the different configuration options.

Table 14-6. Configuration options for the Delta Sharing reader

Delta Sharing option	Data type	Description
`readChangeFeed`	Boolean	Enables the Delta Sharing client to read the change data feed.
`maxVersionsPerRpc`	String	When incrementally processing table changes (`readChangeFeed=true`) using `startingVersion` and `endingVersion`, this option provides a mechanism to control the volume of data read per remote procedure call.
`startingVersion`	Int	Supports *TimeTravel* on the remote shared table.
`endingVersion`	Int	Supports reading of bounded sets. For example, if you want to read from table version 1 to 10, you can set `startingVersion` to 1, and `endingVersion` to 10; in this way, you can meter the volume of data being read for a given operation.
`startingTimestamp`	Timestamp	Read the shared table from the closest transaction available to the provided `startingTimestamp`. The timestamp must be parsable as a TimestampType—for example, 2024-05-26 04:30:00.
`endingTimestamp`	Timestamp	Set the bounds for the table read to the closest transaction available to the provided `endingTimestamp`. The timestamp must be parsable as a TimestampType—for example, 2024-05-26 05:30:00.
`responseFormat`	String	Changes the format of the read operation. The supported options are `delta` and `parquet`. To handle reading from tables with *deletionVectors* or *columnMapping* support, the responseFormat must be "delta." This list will continue growing to support additional UniForm types in the future.
`maxFilesPerTrigger`	Int	How many new files to be considered in every microbatch. The default is 1,000. (streaming only)
`maxBytesPerTrigger`	String	How much data gets processed in each microbatch. This option sets a "soft max," meaning that a batch processes approximately this amount of data and may process more than the limit in order to make the streaming query move forward in cases when the smallest input unit is larger than this limit. If you use `Trigger.Once` for your streaming, this option is ignored. This is not set by default. (streaming only)
`ignoreChanges`	Boolean	Reprocess updates if files had to be rewritten in the source table due to a data changing operation such as `UPDATE`, `MERGE INTO`, `DELETE` (within partitions), or `OVERWRITE`. Unchanged rows may still be emitted; therefore your downstream consumers should be able to handle duplicates. Deletes are not propagated downstream. `ignoreChanges` subsumes `ignoreDeletes`. Thus if you use `ignoreChanges`, your stream will not be disrupted by either deletions or updates to the source table. (streaming only)
`ignoreDeletes`	Boolean	Ignore transactions that delete data at partition boundaries. (streaming only)
`skipChangeCommits`	Boolean	If set to `true`, transactions that delete or modify records on the source table are ignored. (streaming only)

Next, we will close this chapter with a listing of all the additional community-driven Delta Sharing connectors. These clients are lovingly built and shared to continue to extend the mission to bring Delta everywhere.

Delta Sharing Community Connectors

In addition to the common clients, there are even more connectors available from the community. Table 14-7 shows the connectors that are currently released (meaning they are ready for prime time).

Table 14-7. Additional Delta Sharing connectors[a]

Connector	Link	Status	Supported features

Power BI	Databricks owned	Released	`QueryTableVersion`
			`QueryTableMetadata`
			`QueryTableLatestSnapshot`

Node.js	goodwillpunning/nodejs-sharing-client	Released	`QueryTableVersion`
			`QueryTableMetadata`
			`QueryTableLatestSnapshot`

Java	databrickslabs/delta-sharing-java-connector	Released	`QueryTableVersion`
			`QueryTableMetadata`
			`QueryTableLatestSnapshot`

Arcuate	databrickslabs/arcuate	Released	`QueryTableVersion`
			`QueryTableMetadata`
			`QueryTableLatestSnapshot`

Rust	r3stl355/delta-sharing-rust-client	Released	`QueryTableVersion`
			`QueryTableMetadata`
			`QueryTableLatestSnapshot`

Go	/magpierre/delta-sharing/tree/golangdev/golang/delta_sharing_go	Released	`QueryTableVersion`
			`QueryTableMetadata`
			`QueryTableLatestSnapshot`

C++	/magpierre/delta-sharing/tree/cppdev/cpp/DeltaSharingClient	Released	`QueryTableVersion` `QueryTableLatestSnapshot`

[a] All Delta Sharing connectors are located on GitHub, except for the Databricks-owned connector.

Conclusion

This chapter introduced us to Delta Sharing and showed us how we can move beyond traditional data export workflows to reduce complexity with secure, trust-based data sharing from a single source of data truth. When we reduce data sharing complexity, we in turn remove the common headaches related to distributed synchronization and the problem with many sources of fragmented truth. As long as we abide by the appropriate best practices with respect to table-level backward compatibility (see Chapter 5 for details on schema evolution), then we can rest easy at night knowing that the tables and views we've worked so hard to produce can bring joy and delight to the recipients of our shares.

Index

A

B

C

D

E

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Authors

Denny Lee is a Unity Catalog, Apache Spark, and MLflow contributor, a Delta Lake maintainer, and a Principal Developer Advocate at Databricks. He is a hands-on distributed systems and data sciences engineer with extensive experience developing internet-scale data platforms and predictive analytics and AI systems. He has previously built enterprise DW/BI and big data systems at Microsoft, including Azure Cosmos DB, Project Isotope (HDInsight), and SQL Server. He was also the Senior Director of Data Sciences Engineering at SAP Concur. His current technical focuses include AI, distributed systems, Unity Catalog, Delta Lake, Apache Spark, deep learning, machine learning, and genomics.

Tristen Wentling is a Solutions Architect at Databricks, where he works with customers in the retail industry. Formerly a data scientist, he also has authored several blog posts covering topics such as best practices for production stream applications and building generative AI applications for ecommerce. Outside of technical work, Tristen spends a great deal of free time reading or heading to the beach. Tristen holds an MS in mathematics and a BS in applied mathematics.

Scott Haines is a Databricks Beacon and has been working with data, distributed systems, and real-time applications for over 15 years. His data journey began at Yahoo! and then took him to Twilio and more recently to Nike. He owns a consulting company named DataCircus and more recently wrote a book encapsulating his journey called *Modern Data Engineering with Apache Spark: A Hands-On Guide for Building Mission-Critical Streaming Applications* (Apress). He enjoys teaching people how to simplify data systems and data-intensive services and takes to the snow in the winter to pursue his love of snowboarding.

Prashanth Babu is a Databricks Certified Developer who helps guide design and implementation of customer use cases by building out reference architectures, best practices, frameworks, MVP, and prototypes, which enables customers to succeed in turning their data into value.

R. Tyler Croy helped create and still maintains the delta-rs project, which now helps thousands of organizations use Delta Lake from Rust, Python, and beyond. He also acts as a Databricks Beacon, helping teach others about Delta Lake and the Databricks platform. R. Tyler Croy contributed Chapter 6 of this book.

Colophon

The animal on the cover of *Delta Lake: The Definitive Guide* is an American pika (*Ochotona princeps*). Related to rabbits and hares, American pikas are small mammals that live in the mountains of western North America, from central British Columbia and Alberta in Canada to Oregon, Washington, Idaho, Montana, Wyom-

ing, Colorado, Utah, Nevada, California, and New Mexico in the United States. They are typically found at or above the tree line.

American pikas often live in talus fields or among piles of broken rock or boulders, where they forage for the vegetation that makes up their diet. They rely on existing spaces in the talus for their homes and do not dig burrows.

The American pika has been classified by the IUCN as being of least concern from a conservation standpoint. However, the population is reportedly decreasing, especially at lower elevations in the southwestern United States. American pikas are highly sensitive to high temperatures, have limited dispersal ability and low fecundity, and are vulnerable to decreases in snowpack. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from the Museum of Natural History. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.