14
Building Transactional Data Lakes

In the last few years, new technologies have emerged that have significantly enhanced the capabilities of traditional data lakes, enabling them to operate similarly to a data warehouse. These new technologies provide all the benefits of data lakes (such as low-cost object storage, and the ability to use serverless data processing services) while also making it much easier to update data in the data lake (amongst other benefits).

Traditional data lakes were built on the Apache Hive technology stack, which enables you to store data in various file formats (such as CSV, JSON, Parquet, and Avro). Hive enabled many tens of thousands of data lakes to be built on object storage, but over the years the limitations of Hive became more clear, as we will discuss in this chapter.

To overcome these limitations, a number of new table formats have been created by a number of different companies and open-source organizations. Keep reading to learn more about how these new table formats enable you to build a transactional data lake. The topics that we will cover in this chapter include:

- What does it mean for a data lake to be transactional?
- Deep dive into Delta Lake, Apache Hudi, and Apache Iceberg
- AWS service integrations for building transactional data lakes
- Hands-on – Working with Apache Iceberg tables in AWS

Before we get started, review the following *Technical requirements* section, which lists the prerequisites for performing the hands-on activity at the end of this chapter.

Technical requirements

In the last section of this chapter, we will go through a hands-on exercise that uses Amazon Glue to read data, and write the data out using the Apache Iceberg table format.

As with the other hands-on activities in this book, if you have access to an administrator user in your AWS account, you should have the permissions needed to complete these activities. If not, you will need to ensure that your user is granted access to create and run AWS Glue jobs, and to read and write data from Amazon S3.

You can find the SQL statements that we run in the hands-on activity section of this chapter in the GitHub repository for this book, using the following link: https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/tree/main/Chapter14.

What does it mean for a data lake to be transactional?

Transactional data lakes is a common way to refer to the abilities enabled by these new table formats, but what does that mean?

Let's start by looking at the definition of a database transaction in general, from Wikipedia (https://en.wikipedia.org/wiki/Database_transaction):

> *"A database transaction symbolizes a unit of work, performed within a database management system (or similar system) against a database, that is treated in a coherent and reliable way independent of other transactions."*

What this means is that you have the ability to update a database in a way that may potentially make multiple updates as part of the transaction, and you have the guarantee that all the individual updates will work and be applied consistently, or the whole transaction will fail. That means that if there are five updates as part of the transaction, and the third update fails, then the two previous updates that had been applied will be rolled back, and the last two updates will not be applied. Either everything in the transaction succeeds, or the database is returned to the state that it was in prior to the transaction being attempted.

This has been standard functionality in traditional databases and data warehouses pretty much since they were invented, but it was not easy to implement that same functionality when working with Hive-based data lakes (although Spark did have some functionality that attempted to emulate this). But there were also other limitations in Hive that were challenges for anyone building a data lake.

Limitations of Hive-based data lakes

Being unable to easily apply multiple updates in a transaction consistently was just one of the limitations of **Hive** that made data processing more complex, with another big limitation being the inability to update rows in a table without having to rewrite the entire table (or at least a partition of the table). Most data lakes are built on object storage, where data in the file (object) cannot be modified. If you needed to update a row (such as deleting a record), you would read the table, or at least the relevant partition of the table, then remove the row, and rewrite the table or partition.

Let's say you have a data lake that contains a customer table, partitioned by `state`, so that your files in the storage layer are laid out as follows:

```
/datalake/customer/state=new_york/parquetfile.1
/datalake/customer/state=new_york/parquetfile.2
/datalake/customer/state=new_york/parquetfile.3
/datalake/customer/state=new_jersey/parquetfile.1
/datalake/customer/state=new_jersey/parquetfile.2
/datalake/customer/state=new_jersey/parquetfile.3
```

If you have a customer from New York that requests that you delete all records that contain their information, and you want to honor that request, you would effectively need to do the following in your Spark code:

1. Read the records from the customer table for all customers in New York (such as `select * from customers where state = "new_york"`) into a temporary Spark dataframe
2. Create a new dataframe that excludes the customer information of the customer that you want to remove from your data lake
3. Write this new dataframe (that now excludes the specific customer) back to the *new_york* partition, overwriting all the existing files in that partition

However, for a large organization, it is possible that they may have many millions of customers in New York State (so they may have 300 Parquet files in the New York partition, and not just the 3 files shown in the above example). And with Hive, there is no built-in mechanism to find out which physical file in the data lake contains the record for a specific customer. As a result, you end up using a significant amount of compute power, and time, in order to just delete a single record from the data lake, as you need to replace all the files in the relevant partition.

In addition, if the update job is running, at some point it will delete the existing files from the partition, and then start writing out the new 300 files. If during this process, someone queries the data lake to list customers in New York, they will get inaccurate results, as only a subset of the 300 files may exist at the time they do the query. Even worse, if two jobs both attempt to update the table at the same time, there is a chance that both jobs fail at some point, and the underlying table, or partition, is left in an inconsistent state.

For a long time, customers that have built data lakes using Hive have struggled with these issues and had to invent complicated processes to try and ensure that their data stays consistent, and to ensure adequate performance. Three companies that built solutions to overcome some of these challenges ended up making their solutions available to the wider community.

Uber created a table format called Hudi, which was adopted as a top level project by Apache in 2020 and became Apache Hudi.

Netflix created a table format called Iceberg, and this was later donated to the Apache foundation and became a top-level Apache project in 2021.

Databricks (a company created by the original developers of Spark) created a table format called Delta Lake. They released an open-source version as a sub-project of the Linux Foundation, but also have a commercial version that is part of their Databricks Lakehouse Platform.

Each of these new table formats provide similar, although not identical, functionality, and we will examine each of these table formats in more detail in later sections of this chapter. First, we take a look at some of the specific benefits provided by these new table formats.

High-level benefits of open table formats

Each of the **open table formats** we will be examining provide the following functionality, although the way they implement this functionality may be different.

ACID transactions

The common properties that are enabled in transactions are often referred to by the acronym **ACID** (**Atomicity**, **Consistency**, **Isolation**, **Durability**):

- **Atomicity** refers to the fact that every update contained in a transaction is treated as a single unit. If any one of the updates fail, then the transaction as a whole fails, and any updates that had been applied are rolled back.
- **Consistency** refers to ensuring that reads and writes are always consistent. For example, if a transaction is running and performing a number of updates, someone querying the data must get a consistent result (i.e., not get a result that contains part of the updates only). This means any queries running against a table where updates are happening at the same time must either get results as the table was before the update started, or must get the results of the table as the table is after the update has been successfully completed and committed.

- **Isolation** refers to ensuring that transactions are isolated from each other. This means that if two updates are run at the same time, they must not interfere with each other. One of the transactions must first complete before the second transaction is run.
- **Durability** refers to ensuring that once an update has been applied, that it is permanent (well, at least until another update is applied to change that data). Once a transaction has been committed, all future reads must reflect the new state of the data.

Record level updates

Each of these new open table formats make it much easier and more performant to do updates (inserts, deletes, or changes) to a single row of a table. These open table formats handle the underlying complexities of working at the record level with very large datasets in a data lake. With some of the table formats, you are able to configure whether you want write or read performance to be optimized.

Schema evolution

Schema evolution refers to the ability to change the schema of the table without breaking the ability to query the table. This includes the ability to make changes such as adding columns, deleting columns, changing the column name, changing the order of columns, and even changing the type of a column (for example, changing an *integer* to a *long* data type).

Time travel

Time travel queries provide the ability to query a table as it was at some point in the past. Whenever an update is made to a table, the open table format effectively creates a new version of the table. This enables you to run a query and specify a timestamp, and the query result will be as the data was at the timestamp specified. Alternatively, you can choose to roll back a table to the state that it was in at a prior point in time.

An example of how this can be used is to recover from an incorrect change to a table. For example, if you have a new data pipeline job that performs some transformation, but you discover after it runs that there was some logic error with the job, you can return the table to the version that it was at prior to the job running. Ideally, though, you should have strong change control and testing processes, so this should never happen in a production environment. However, you can use this feature to roll back a table to a previous version in your development or testing environment if you discover a code error during development or testing.

While this is very useful, it does mean that the physical files stored in the data lake end up being larger than the actual dataset (since you have all versions of the data). It also means that if you delete data for governance reasons (such as deleting a customer record because they requested that you remove all their information), it may be possible to still access the data through time travel.

Therefore, each of the table formats provides a method to delete older versions, or snapshots, of a table.

And even though they are implemented differently, at a high level each of these table formats provides the above functionality by managing metadata related to the table, and by managing the layout of files in the object storage. Let's take a high-level look at how this works.

Overview of how open table formats work

For a long time, modern file formats (such as Parquet) have worked by collecting and storing metadata related to the data that they store. For example, with Parquet, different types of metadata is stored in the file, along with the actual data. This includes metadata about the file (such as number of columns and rows in the file) and column metadata (such as the data type stored in the column, as well as statistics such as the min and max values for that column).

Storing this type of metadata in the file enables a process that is reading data from the Parquet file to optimize query performance. For example, let's say we have a table that contains transaction information, and our table has around 20 million rows. The data is stored in Parquet format in a data lake, and is spread over 200 Parquet files. When a query runs and wants to report on transactions that were over 1,000 in value (stored in a column called *amount*), the query engine is able to use the metadata to avoid having to scan the data in every file. It does this by reading the metadata in each file, through which it can immediately identify if this specific file has data that needs to be read. If, for example, the metadata indicates that the minimum value for the *amount* column is 2.20 and the maximum value is 780.56, then the data in this file does not need to be scanned (as we are only querying for transactions with a value over 1,000). This can significantly improve query performance.

In a similar way, the new table formats we are looking at store metadata about the table, and the files that make up the table. Using this information enables them to further optimize performance and also overcome some of the limitations of the Hive format, which we discussed previously.

The exact metadata captured and tracked differs between the table formats, but each format has a way to track the current state of a table, as well as a history of how the table has evolved. Some of the formats also track key statistics for each file that makes up the table. For example, with **Apache Iceberg**, there are Manifest files that store metadata for each file that makes up a table, including column-level metrics and stats that can be used to optimize the query.

In this way, the table metadata will store some of the same data that is stored in a Parquet file, such as the min and max values for a column, the number of `null` values, etc. These details are stored for each file that makes up the table, enabling a query planner (an analytic engine that is querying the data) to identify the exact path of each file that has data relevant to a specific query, without even having to open the file to read the metadata.

With Hive, the analytic engine would have needed to list all files in an S3 partition, and then open each file to read the Parquet metadata, for example. A partition may have had hundreds, or even thousands, of files, and listing every file and then opening every file to read the file metadata could significantly slow down a query. By maintaining metadata on every file that makes up a table, the query engine just needs to read the table format metadata files, and from there it can determine exactly which files contain data that it needs to scan to fulfil the query. Reading the metadata file to determine which files to query can significantly increase performance over having to list all files in a partition, and then individually read the metadata from every file.

Another item that is common between the table formats is the approaches used to enable updates to a file. Again, these may be implemented slightly differently, but at a high level, there are two common approaches for updating tables, as we will see in this next section.

Approaches used by table formats for updating tables

There are two common approaches that are currently used by the different open table formats for applying updates to a table. All three formats support both the **copy-on-write (COW)** and

the **merge-on-read (MOR)** approaches.

Let's start with a detailed look at the COW approach.

COW approach to table updates

With the COW approach to updates and deletes, whenever a record is updated or deleted, a new copy of the underlying files that contain that record are created, with the updated data. At the same time, the metadata files are updated to reflect that there is a new version, or snapshot, of the data. These files contain metadata that points to the updated files so that when a query is run, the latest metadata files point to the latest data files. However, the metadata files for earlier versions/snapshots of the table still contain the pointers to the original file, with the older data. It is this mechanism that enables time travel.

Using COW does have a performance impact on updating records in a table (or deleting records from a table) because each of the underlying affected files need to be recreated with the new data. However, it does provide better *read* performance than the MOR approach (which we will discuss shortly).

Imagine that you have a user that purchases from your online eCommerce store every month for a full year, and that you have transaction data partitioned by month. Every month, you end up with approximately 300 Parquet files in that month's partition, and a subset of those files contain transactions for the specific user we mentioned earlier. If at some point you need to delete all records related to that user, with the COW approach, there will be a subset of files in every partition that will need to be rewritten. Even if just one of the 300 Parquet files in each partition needed to be updated, if the average file size was 1 GB, that would still require 12 GB of new data files to be written (since COW creates a new version of a file, rather than replacing the existing file).

COW is ideal for use cases where updates/deletes of records in a table are infrequent, and for where you want to optimize write performance, over read performance.

MOR approach to table updates

With the MOR approach to handling record updates and deletes, affected files are not rewritten when data inside those files is updated. Instead, if a record is either deleted or updated, a delete tracking metadata file is created to record the information about the deleted/updated record. For updated records, the newly updated record is also written to a new file. When a user queries the table, the query engine will know to ignore the records listed in the delete file, and will merge the original data with the file containing the updated data.

With this approach, writes are much faster (since a new version of the underlying Parquet files with changes do not need to be written when data is updated); however, reads are slower since the query engine needs to merge information from the deleted files and the files containing updated records with the original data whenever that table is queried.

Each of the table formats supports a process that you can run that will merge the updates and deletes into new copies of the Parquet files. Therefore with this approach, if you have a write-heavy workload you can use MOR tables to enable fast writes, and then in off-peak times you can run a process to create newly updated Parquet files to improve read performance.

Apache Hudi has supported the MOR approach since 2018, while Apache Iceberg introduced comprehensive MOR support with v2 of the Iceberg table format. Delta Lake traditionally only

supported COW, but in a July 2023 blog post on the *delta.io* website, they announced a new feature called *Deletion Vectors* that provides a MOR approach for Delta Lake tables.

Let's now do a deeper dive into each of the open table formats that we are looking at in this chapter, starting with Delta Lake.

Choosing between COW and MOR

As discussed above, there are pros and cons of each approach to updating tables. The following table summarizes the differences between the two approaches.

	Copy-On-Write (COW)	Merge-On-Read (MOR)
Action on record update/delete	New version of affected file is created containing newly updated records, or skipping deleted records	Deleted and updated rows are written to a deletion tracking file, and updated records are written to a new file
Action when table is read	The metadata for the table tracks the files that have the latest data, and only those files are read	The metadata for the table tracks the original file, the file tracking deleted records, and files containing new/updated data. The query engine needs to merge each of these files to query the data.
Optimized for reads or writes?	Optimized for table reads	Optimized for table writes

Figure 14.1: Comparison of COW and MOR

Let's now examine each of the open table formats that we are looking at in this chapter, starting with Delta Lake.

An overview of Delta Lake, Apache Hudi, and Apache Iceberg

The three table formats that we are reviewing in this book all provide similar functionality, as outlined above, but they also all have their own unique features and slightly different implementations. In this section, we are going to do a deep dive into each of the three open table formats.

Deep dive into Delta Lake

Let's start by looking at **Delta Lake**; however, we will not be covering the enhanced capabilities available as part of the paid Databricks offering. For example, Delta Live Tables provides ETL pipeline functionality, but is not open-sourced, so is not covered here.

Delta Lake has become a very popular table format, in large part as a result of Databricks having a very popular Lakehouse offering that incorporates Delta Lake. Databricks has made all Delta Lake API's open-source, including a number of performance optimization features that they initially built for their paying customers. Delta Lake also includes advanced features, such as Delta Sharing, an open protocol for secure data sharing across different organizations. There are also stand-alone readers and writers for Delta Lake, which enables clients written in popular languages such as Python, Ruby, and Rust to write data directly to Delta Lake tables without requiring a big data engine such as Apache Spark. However, while some of the other table formats support multiple file formats, Delta Lake only supports the Apache Parquet file format.

Delta Lake is built on an open-source standard called Delta Lake transaction log protocol. This standard specifies how data transactions should be recorded, and all implementations of Delta Lake must follow the Delta Lake transaction log protocol strictly. This is what enables one implementation of Delta Lake to be able to read and update files that were created by a different implementation.

There are various implementations of Delta Lake by open-source providers and commercial companies. For example, there is the delta-io implementation (`https://github.com/delta-io/delta/`), which is open-source, and there is an implementation by Microsoft called the Microsoft Fabric Lakehouse, as well as an implementation by Databricks called *Databricks Delta Lake*. Each of these implementations has its own additional functionality, but they should all be interoperable.

Advanced features available in Delta Lake

Let's take a look at some of the advanced features that are available in Delta Lake.

Delta Lake shallow clones

Some of the advanced functionality included in Delta Lake includes the ability to create a *shallow clone* of an existing table. You can create a shallow clone of any available version of a table, and when you make changes to the shallow clone version of the table, the original table is not impacted. Shallow clones reference the underlying files of the source table, but track any changes (updates, deletes, and inserts) separately. This means that you can test changes to a table using a shallow clone without impacting the original source table.

Use cases for shallow clones include making a clone of a production table, and then doing testing on the cloned table (such as changing the schema, or other major changes). Another use case is for machine learning state capture, where you create a copy of a table as it was at a specific point in time, using a shallow clone. You can then retest or retrain a model using a static version of the table. The changes to the shallow clone will have no impact on the original table, and changes to the original table do not reflect in the shallow clone.

Delta Lake Z Ordering

While we won't go into the details of how **Delta Lake Z Ordering** works under the hood, let's examine the basic principles of Z Ordering and how this helps improve query speed.

With Delta Lake, you can run a command to reorganize how the data is stored in files, and you can specify that you want to use Z Ordering to sort the data. While standard sorting of files allows you to optimize data sorting by one column, Z Ordering enables you to optimize data by multiple columns.

This is useful if you regularly run queries where you filter the data by a number of different columns. When you run the command to sort the data using Z Ordering, you can specify one or more columns to sort the data on. The Z Ordering algorithm will store the data in files in such a way as to optimize any queries that use those columns for filtering. Note, however, that the more columns you specify, the less effective the sorting is overall.

Therefore, if you mainly run queries where you filter by one specific column, there may not be much benefit gained from Z Ordering. However, if you regularly query your data and filter by two or three columns, then Z Ordering can be beneficial.

When you run a command to reorganize your data using Z Ordering, all the underlying Parquet files are rewritten with the objective of clustering data for the columns you specify in as few files as possible. So if you have 50 files, for example, and you regularly query the data on `column1` and `column2`, if you organize by those columns then queries will need to read fewer files in order to scan the needed data.

For a deeper dive into how Z Ordering works, see the following article from delta.io: https://delta.io/blog/2023-06-03-delta-lake-z-order/.

Delta Lake Change Data Feed (CDF)

With **Delta Lake Change Data Feed**, all the inserts, updates, and deletions to a table can be captured in an audit log. This is functionality that can be enabled at a table level, so you are able to decide on which tables you may want to enable this functionality.

There are two primary use cases for enabling Change Data Feed on a table. The first one is purely for a governance and audit perspective, where you want to be able to query all the changes that have been made to a table over time. Note, however, that the transaction log generated by this functionality does not include information on *who* made the change, but rather just records the changes made to a table.

The second use case enables you to optimize operations for performing updates on incremental downstream tables (such as an aggregation table). For more information on Change Data Feed functionality and an example of how this can be used to update downstream tables, see the delta.io blog post at https://delta.io/blog/2023-07-14-delta-lake-change-data-feed-cdf/

Let's now move on to our next table format, Apache Hudi, and do a deep dive into this table format.

Deep dive into Apache Hudi

Apache Hudi, originally developed at Uber for their massive data lake, has become a popular choice for building transactional data lakes and has broad support in modern analytic tools. Hudi is probably the oldest of the open table formats; it was developed at Uber in 2016, and its name is as an acronym for *Hadoop Updates, Deletes, and Incrementals*.

Let's look at some of the key concepts that make Hudi different to the other open table formats, and that will help you better understand how Hudi works under the hood.

Hudi Primary Keys

Hudi has a concept of a **primary key**, which is made up of a record key and the partition path that the record belongs to. Using this key enables Hudi to ensure that there are no duplicate records in the data lake (or at least that records are unique within a partition).

Hudi is highly customizable, and this includes the ability to choose from a number of different key generators. Here are a few of the built-in key generators:

- **SimpleKeyGenerator** is used to create a primary key where the record key consists of a single field (column) and the partition path is also based on a single field. For example, we may specify that the record key is based on the *customer_id* column, and the partition path is based on the *state* column.

- **ComplexKeyGenerator** is used to create a primary key where the record key and the partition path consist of one or more fields.
- **NonpartitionedKeyGenerator** is used for use cases when you do not need to partition your dataset. This effectively is used to set the partition path to be empty.

In addition to the standard key generators, you also have the ability to write your own key generator and use that.

File groups

The file layout in object storage for Hudi tables is based on a **partition path**, and within each partition, files are organized into **file groups**. Each file group contains several **file slices**, with each slice consisting of one or more base data files, which are produced at a certain commit/compaction event, along with a set of delta log files that record all the inserts/updates/deletes to the base file since the file was initially created.

Note that the delta log files are used for tables that are configured as MOR. When a table is configured as COW, updates are made directly to the base file at the time of writing.

Compaction

When working with Hudi MOR table types, you need to regularly run a **compaction** process that will update the base Parquet file with all the updates contained in the delta logs, in order to optimize read performance. With MOR tables, any updates to a table are contained in an Avro formatted delta log file (with Avro providing the best performance for writing out rows of data). When you read a table, the base Parquet file is merged with the delta log files in order to return the latest state of the table. However, merging the delta log Avro file with the Parquet file does add overhead to the query.

When you run the compaction process, all updates from the delta logs are merged into the Parquet file so that reads made directly after the compaction process completes only need to query the Parquet file. Of course, as new updates are made to the table, new delta logs are created, and therefore you need to run the compaction process on a regular basis.

Record level index

While each of the open table formats captures metadata about the files and the data they contain (such as column min and max values), Hudi is currently the only one that stores record-level metadata in an index file. With the record-level index, Hudi is able to immediately identify exactly which files contain data for a specific record, and therefore knows exactly which files need to be scanned to resolve a query, or which need to be rewritten when a specific record is updated or deleted (or which transaction log files needs to be updated in the case of MOR tables).

The index does this by recording the details of each record key, along with the relevant group/file ID that contains the data for that record. Again, with Hudi being very configurable, you can select from multiple different types of indexes, or even write your own custom index mechanism. For a more in-depth understanding of how Hudi indexes work, refer to the following Hudi documentation: `https://hudi.apache.org/docs/indexing`.

Let's now move on to a deep dive into the Apache Iceberg format, after which we'll have a look at the current state of support in AWS services for each of these formats.

Deep dive into Apache Iceberg

Apache Iceberg is the table format that appears to be getting the broadest support across vendors, and some industry experts feel it may end up becoming the most popular table format (although all three table formats have their benefits and supporters).

Much like the other two formats, Iceberg has its own metadata format to track the data files that make up a table and, more specifically, to track table snapshots (the state of a table at a specific point in time).

In the directory for an Iceberg table in S3 (for example, `dataeng-curated-zone-gse23/iceberg/streaming_films`) there are two directories. One is the **metadata** directory, and the other is the **data** directory. All the metadata is, as expected, stored in the **metadata** directory. Let's do a deep dive into the Apache Iceberg metadata files.

Iceberg Metadata file

The top level metadata file in the **metadata** directory is a file that begins with a sequence number (for example, the first file starts with `00000`) followed by a **Universally Unique Identifier** (**UUID**), and that ends with `metadata.json`. Whenever there is a new commit to the Iceberg table (a commit being an operation that ran against the table and changed it in some way), a new `metadata.json` file will be created, with an incrementing sequence number. So after the first update to a table, there will be a new file that starts with `00001`, followed by a unique UUID, and ending again with `metadata.json`.

We won't do a deep dive into everything that is tracked in this file, but the following are some of the metadata items stored in this file:

- The table format version (the current Iceberg version is V2)
- The location of the table as stored in Amazon S3
- The date and time of when the table was last updated (in Unix epoch format)
- The table schema (a list of all columns, with column name, type, and an ID number for the column)
- Details about how the table has been partitioned (such as which column the table is partitioned on)
- The properties of the table (such as that the files are stored in Parquet format and the S3 storage path for data)
- The UUID of the current snapshot (every time the table is updated, a new snapshot is created)

Statistics for each of the existing snapshots (such as the `append` or `delete` operation that was applied to the table, the snapshot timestamp, and the total size of the table and number of records contained in this snapshot)

For each snapshot, the `metadata.json` file also has a reference to the relevant metadata list file for that snapshot

When a query is run against the table, the query engine first queries the catalog (such as the AWS Glue catalog) to find out what the current `metadata.json` file is. It then reads the current `metadata.json` file and can then determine which manifest list file to query in order to read the relevant data. Let's now look at what is stored in the manifest list file.

The manifest list file

The manifest list file is another metadata file created and managed by Iceberg. Whenever an update is made to a table, a new snapshot is created, and a new manifest list file is created that

contains a list of the manifest files that are relevant to this snapshot. Manifest list files are stored in Avro format, and the filenames start with a prefix of *snap*.

When the table is queried, the query engine can read the relevant **manifest list file** for the snapshot that needs to be queried, and from the manifest list it can read a list of the **manifest files** that are needed to query the data in this snapshot.

By using a manifest list file that is associated with a specific snapshot, Iceberg is able to read only relevant manifest files, and doesn't need to open and read all manifest files.

Let's take a look at the last of the metadata files, the manifest files.

The manifest file

The manifest file tracks detailed metadata about a subset of the data files that make up a table. For larger tables, there will be multiple manifest files, which enables the query engine to read multiple files simultaneously through parallelism.

The manifest file contains valuable metadata about the data contained in a data file that the query engine can use in query planning to improve efficiency and performance. For example, for each data file, the manifest file contains information about which partition the data file stores data for, the number of records, the lower and upper bounds of columns, etc.

Putting it together

Apache Iceberg is able to read the metadata files (the metadata,json file, the metadata list file for a snapshot, and then the manifest files for that snapshot) and use this to determine exactly which data files need to be read in order to return results for a specific query. For large tables, this can significantly speed up query performance by minimizing the number of data files that need to be read for a specific query. The following figure illustrates the different metadata and data files:

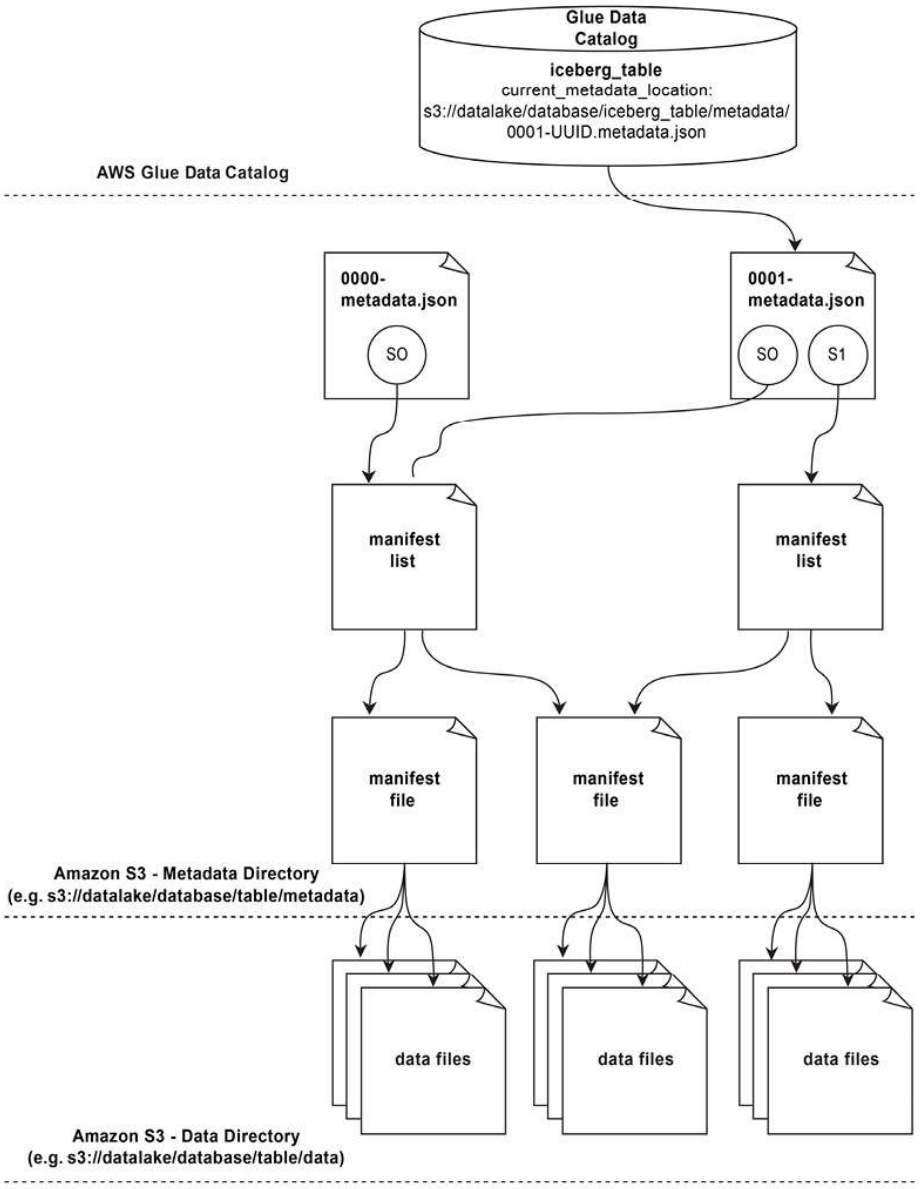

Figure 14.2: Apache Iceberg File Structure

In *Figure 14.2*, we see the following different levels of metadata and data for an Apache Iceberg table.

At the top of the diagram we have the **AWS Glue Data Catalog**, which stores the location of the latest `metadata.json` file for a table. When a query engine needs to query a table, it starts by querying the Glue Data Catalog for the location of the current `metadata.json` file.

In the next level down, we see two `metadata.json` files, which are written to the S3 **metadata** directory for the table. Every time there is a commit to a table, a new `metadata.json` file is generated containing details about the manifest list files for each snapshot. In the example illustrated in *Figure 14.2*, we can see that the latest snapshot (S1) points to a specific metadata list file. Note that the `metadata.json` and metadata list files are stored in the Amazon S3 metadata prefix for the table (such as `s3://datalake/database_name/table_name/metadata`).

We then see that the `metadata.list` file for a specific snapshot points to one or more manifest files. Multiple snapshot versions may point to the same manifest files.

Finally, we see the actual data files (multiple formats are supported, but most commonly these files are in Parquet format). The data files are stored in the Amazon S3 data prefix for the table (such as `s3://datalake/database_name/table_name/data`).

By capturing metadata for each snapshot, Iceberg enables time travel – the ability to query data as it was at a specific point in time by determining which snapshot was current at that point in time and then reading the relevant metadata and data files for that snapshot.

In the hands-on section of this chapter, we will get hands-on with creating an Apache Iceberg formatted table and have a closer look at the metadata files and how they change as we perform different operations on the table.

Because Iceberg works on the basis of snapshots, with every commit to the table creating a new snapshot, you can end up with a lot of metadata files, as well as delete files that are used to track data that has been deleted. Keeping this data is useful for time travel queries (where you query the table as it was as a previous point in time), but you generally do not want, or need, to keep all data from the point that the table was created. In order to remove some of the old data and metadata files that are not needed to query recent data, there are various maintenance tasks that can be applied to your Iceberg tables, as we discuss in the next section.

Maintenance tasks for Iceberg tables

There are two primary operations that can be used to clean up your Iceberg tables: optimizing the files that make up the table to improve read performance or deleting old data and metadata files that you may no longer need.

Let's take a look at how Amazon Athena implements Iceberg table maintenance.

Amazon Athena provides the `OPTIMIZE` statement, which can be used to compact the underlying data files that make up an Iceberg table. When you run the `OPTIMIZE` statement against an Iceberg table, Athena rewrites the underlying data files into a more optimized layout (this is also sometimes referred to as a **compaction** process).

One of the ways it does this is by rewriting underlying data files to exclude any records that were included in a delete file so that future reads do not need to merge the base data file and the delete file to return relevant results. Instead, the query engine can can just read the base file without having to merge any records from delete files.

The other optimization applied by the `OPTIMIZE` process is to rewrite files into an optimal size. Over time, as some smaller commits are applied against a table, you may end up with a significant number of small files, which are not efficient to read. By running the `OPTIMIZE` process, small files can be merged into bigger files to increase performance.

Vacuuming a table

The other table maintenance task supported by Amazon Athena is the `VACUUM` statement. When you run a `VACUUM` statement against an Iceberg table, Athena removes data and metadata files related to old snapshots that are no longer needed. There are various table properties that you can set to control how many snapshots to keep, such as `vacuum_max_snapshot_age_seconds`, and `vacuum_min_snapshots_to_keep`.

Keeping more snapshots enables you to perform time travel to older points in time, but it does mean that you use more storage space to store those additional snapshots (and this additional space may be a significant amount of space). Therefore, you need to balance how far in the

past you need to be able to run time-travel queries against how much storage space the old snapshots consume, and what the cost implications are. You should also consider any data governance requirements when deciding on how much old data to keep versus when data should be deleted.

At the time of writing, the default value for `vacuum_max_snapshot_age_seconds` was set to be 5 days, meaning that any snapshots older than 5 days would be deleted. This means that after running the `VACUUM` command using the default table setting, you would not be able to do time travel queries on data older than 5 days.

If you had a use case that required you to be able to query data for the last month, then you could update the value of this setting to be 31 days, and this would ensure that you could always query the status of the table as it was at any point in the last 31 days. If you had a table that was updated 3 times every day, this would mean that 93 snapshots would be kept.

For the `vacuum_min_snapshots_to_keep` setting, the default at the time of writing was `1`, meaning that at least one snapshot would be kept, irrespective of the snapshot age setting. If you set this value to `100`, and your table had 3 snapshots per day, then even if you had max snapshot age set to 31 days (covering 93 snapshots), additional snapshots would still be kept beyond the 31 days (at least 33 day's worth of snapshots would be kept). Effectively, the `vacuum_max_snapshot_age_seconds` property is ignored if it would result in keeping fewer snapshots than are set in the `vacuum_min_snapshots_to_keep` setting.

The previous two tasks are based on the way Amazon Athena implements the clean-up of Apache Iceberg tables. However, other tools that support the Apache Iceberg table format may implement these maintenance tasks differently. For details of the underlying specification for table maintenance operations, see the Apache Iceberg specification at **https://iceberg.apache.org/docs/1.2.0/maintenance/**.

When it comes to selecting which table format to use out of the three, it seems that Apache Iceberg currently has the most momentum, and is gaining broad support across many different analytic vendors. However, all three table formats have their own pros and cons, and AWS has a great blog post that compares the three different table formats in detail to help you make a decision about which table format may be right for your use case. See the blog post titled *Choosing an open table format for your transactional data lake on AWS* at **https://aws.amazon.com/blogs/big-data/choosing-an-open-table-format-for-your-transactional-data-lake-on-aws/**.

Now that we have a better understanding about the three table formats, let's take a look at the current state of support for these new transactional open table formats in different AWS services.

AWS service integrations for building transactional data lakes

AWS services constantly evolve as new services are introduced and existing services have new functionality added. This applies to the AWS analytic services as well, with many of these services introducing support for these new transactional open table formats over the last few years. In this section, we will look at the support for open table formats in various services, as at the time of publishing.

However, make sure to review the latest AWS documentation to understand the latest status of support across the services.

Open table format support in AWS Glue

AWS Glue has broad support for open table formats across the different components of the Glue service. In this section, we examine open table support in two of the key Glue components.

AWS Glue crawler support

As covered earlier in this book, the **AWS Glue crawler** is a component of the Glue service that can scan a data source (such as Amazon S3) and automatically register table information in the Glue Data Catalog. A common use case is when you have data in an Amazon S3 data lake and you want to automatically populate the catalog.

The AWS Glue crawler supports crawling data sources that are in Amazon S3 in Delta Lake, Apache Hudi, and Apache Iceberg format. When the crawler examines the objects in S3 that belong to these table formats, it is able to recognize the different metadata schemas, and successfully register the tables in the Glue Data Catalog, and correctly identify the open table format type.

For more information about current support for open table formats with the AWS Glue Crawler, see the AWS documentation at `https://docs.aws.amazon.com/glue/latest/dg/crawler-data-stores.html`.

AWS Glue ETL engine support

In November 2022, AWS announced support for the popular open table formats – Delta Lake, Apache Hudi, and Apache Iceberg—in AWS Glue for Apache Spark. With this announcement, AWS Glue for Apache Spark introduced native integration with these formats, meaning that users do not need to install a separate connector in order to work with these tables.

For more information on how to use AWS Glue for Apache Spark with these three table formats, see the AWS blog post titled *Introducing native support for Apache Hudi, Delta Lake, and Apache Iceberg on AWS Glue for Apache Spark* at `https://aws.amazon.com/blogs/big-data/part-1-getting-started-introducing-native-support-for-apache-hudi-delta-lake-and-apache-iceberg-on-aws-glue-for-apache-spark/`.

Open table support in AWS Lake Formation

AWS Lake Formation, which we discussed previously in this book, is a service that enables you to configure fine-grained permissions on data in an Amazon S3 data lake at the database and table level (and even down to the row, column, and cell level).

AWS Lake Formation also enables you to configure sharing of both S3-based tables and Redshift tables across AWS accounts.

The support for Lake Formation fine-grained access controls across different AWS services is complex, with different AWS services supporting different levels of integration with Lake Formation permissions. For example, Amazon EMR 6.9.0 and later supports Lake Formation column-level permissions on Apache Hudi tables, but at the time of writing does not support Lake Formation permissions on Apache Iceberg or Delta Lake tables. And while AWS Glue can read and write Iceberg tables that are controlled using IAM permissions, it does not support Iceberg (or the other table formats) when they are managed using Lake Formation permissions.

At the time of writing, the best Lake Formation support for open table formats can be found in the Amazon Athena and Amazon Redshift Spectrum services. With both of these services you can read data from tables that are controlled by Lake Formation permissions for Delta Lake, Apache Hudi, and Apache Iceberg formats.

For more information on the current state of Lake Formation permissions support for open table formats across various AWS services, see the AWS Lake Formation documentation titled *Working with other AWS services* at https://docs.aws.amazon.com/lake-formation/latest/dg/working-with-services.html. Underneath this section of the documentation there is detailed information for each of the compatible AWS services, including a discussion about the support for transactional table formats for each service.

Open table support in Amazon EMR

As of **Amazon EMR** release 6.9.0, all three of the open table formats that we have been discussing are now supported in EMR, without having to manually install additional libraries to the cluster.

However, depending on which packages you are using (such as Spark, Presto, Trino, or Flink) some additional configuration may be required, and there may be certain limitations. For details on how to configure EMR for each of the table formats, see the following documentation:

- Using Amazon EMR with Delta Lake: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-delta.html
- Using Amazon EMR with Apache Hive: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-hudi.html
- Using Amazon EMR with Apache Iceberg: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-iceberg.html

Note that the support in **Amazon EMR Serverless** is slightly different, and requires a different configuration in certain cases. However, all three table formats can be used with Amazon EMR Serverless. See the *Tutorials* section of the EMR Serverless documentation for information on the required configuration: https://docs.aws.amazon.com/emr/latest/EMR-Serverless-UserGuide/tutorials.html.

For a comprehensive guide to using Amazon Iceberg on Amazon EMR, see the AWS blog post titled *Build a high-performance, ACID compliant, evolving data lake using Apache Iceberg on Amazon EMR* at https://aws.amazon.com/blogs/big-data/build-a-high-performance-acid-compliant-evolving-data-lake-using-apache-iceberg-on-amazon-emr/.

Open table support in Amazon Redshift

Amazon Redshift Spectrum enables you to run queries against data in Amazon S3 and, optionally, also join that data with other data that has been loaded into your Redshift cluster. We discussed this earlier in the book, when we used an example of loading the most recent 12 months of frequently queried data into the Redshift cluster for optimal performance, and then being able to join that with 5 years of historical data that is in Amazon S3. Most queries would be querying recent data, so for the smaller percentage of queries that need to query the historical data, you can tolerate the slightly slower query performance of Redshift Spectrum.

In September 2020, AWS announced support for reading both Apache Hudi and Delta Lake formatted tables in an S3 data lake using Amazon Redshift Spectrum. However, for Apache Hudi,

at the time of writing, only COW-formatted Hudi tables were supported (Hudi MOR tables are not supported).

In July 2023, AWS announced preview support for querying Apache Iceberg formatted tables via Redshift Spectrum.

Note that Redshift Spectrum only supports limited write operations, such as insert into, and therefore you cannot update tables in any of the open table formats using Redshift Spectrum.

Open table support in Amazon Athena

Amazon Athena provides a serverless way to query data in a data lake without needing to provision or manage any infrastructure (as we have discussed previously in this book). Once a table has been added to the AWS Glue Data Catalog, Amazon Athena can query and update the data using standard SQL statements.

Amazon Athena provides strong support for open table formats, especially for the Apache Iceberg format. At the time of writing, Athena can be used to read data in all three of the table formats we have been discussing, and also supports insert, update, delete and maintenance operations for Apache Iceberg tables. In the hands-on section of this chapter, we will use Amazon Athena to create a new Iceberg table, and then perform various operations on the table using SQL commands.

When looking to implement an open table format in AWS, you have a wide variety of different AWS services that can be used, and of course you can use multiple different AWS services to work on the same table in a transactionally consistent way. For example, you can create an Iceberg table using Amazon Athena and have an ETL job that runs in AWS Glue to regularly load and update data in the table. You may then have a different team that uses an EMR cluster for their ETL, and they may read some of the data in your Iceberg table. Finally, a different team may have a visualization tool (such as Amazon QuickSight or Power BI) that connects to Redshift and uses Redshift Spectrum to visualize data contained in the Iceberg table.

As we discussed in this section, support for different features of the open table formats differs across services, so make sure to read the latest AWS documentation to understand any considerations or limitations when architecting solutions that will make use of one of the open table formats.

Let's now get hands-on to apply some of what we have learned about open table formats and how they work in AWS. In the next section, we will use Amazon Athena to create and work with an Apache Iceberg table.

Hands-on – Working with Apache Iceberg tables in AWS

As discussed in the previous section, Amazon Athena has strong support for the Apache Iceberg format, and as a serverless service, it is the quickest and simplest way to work with Apache Iceberg tables.

For the hands-on section of this chapter, we are going to use the Amazon Athena service to create an Apache Iceberg table, and then explore some of the features of Iceberg as we query and modify the table. To do this, we will create an Iceberg version of one of the tables we created earlier in this book.

Creating an Apache Iceberg table using Amazon Athena

To create our Apache Iceberg table, we will access the Athena console and then run DDL statements to specify the details of the table we want to create. At the time of writing, Amazon Athena supports the creation of Iceberg v2 tables. Remember to refer to the GitHub site for this book for a copy of the SQL statements used in this section (as mentioned at the start of this chapter):

1. Log into the **AWS Management Console** and use the top search bar to search for, and open, the **Athena** service.
2. Open a new **Query** tab and run the following statement to create a new database to hold our Iceberg tables:

```
create database curatedzonedb_iceberg;
```

3. Create a new version of our existing `streaming_films` table in Apache Iceberg format using the following statement. Make sure to change the S3 location specified in this statement to reflect the name of your S3 curated zone bucket:

```
CREATE TABLE curatedzonedb_iceberg.streaming_films_ib(
    timestamp string,
    eventtype string,
    film_id_streaming int,
    distributor string,
    platform string,
    state string,
    ingest_year string,
    ingest_month string,
    category_id bigint,
    category_name string,
    film_id bigint,
    title string,
    description string,
    release_year bigint,
    language_id bigint,
    original_language_id double,
    length bigint,
    rating string,
    special_features string
)
PARTITIONED BY (category_name)
LOCATION 's3://dataeng-curated-zone-gse23/iceberg/streaming_films/'
TBLPROPERTIES ('table_type' = 'ICEBERG', 'format' = 'parquet')
```

4. Open a new tab in your browser and navigate to the Amazon S3 console at **https://s3.console.aws.amazon.com/s3**, and then navigate to the location that you specified for your new Iceberg table (for example, `dataeng-curated-zone-gse23/iceberg/streaming_films/`).

You will notice that there is a metadata folder here, and in the folder there should be a single file ending in `metadata.json` . Download this file, open it with a text editor, and review the metadata that has been captured.

Having created our new Iceberg table, let's now populate the table with the data from the original table.

Adding data to our Iceberg table and running queries

In *Chapter 7*, *Transforming Data to Optimize for Analytics*, we created the `streaming_films` table by joining two other tables. We will now take the data from that table and write it into our Iceberg version of the table, before querying both the data and the metadata:

1. Go back to your browser tab where you had the **Athena console** open, open a new **Query** tab, and run the following to insert data from our `streaming_films` table into our new Iceberg formatted table:

```
insert into curatedzonedb_iceberg.streaming_films_ib
select *
from curatedzonedb.streaming_films
```

2. Go to your browser window where you had opened the **Amazon S3 console** and review the files in the `metadata` directory (note that you may need to click to refresh the list of files). Notice that we now have additional metadata files – the new `metadata.json` files as well as other files (for example, `.stats` files and `.avro` files). These files all contain different metadata that is used by Iceberg to track statistics, snapshots, and data files that make up the table, as we discussed in the *Deep dive into Apache Iceberg* section of this chapter.
3. In a new browser window, open up the **AWS Glue console** (https://console.aws.amazon.com/glue), navigate to the Glue **Data Catalog**, and open up the list of Glue databases. Select the `curatedzonedb_iceberg` database and then click on the `streaming_files_ib` table. Click on the **Advanced properties** tab to view some Iceberg-specific table properties.

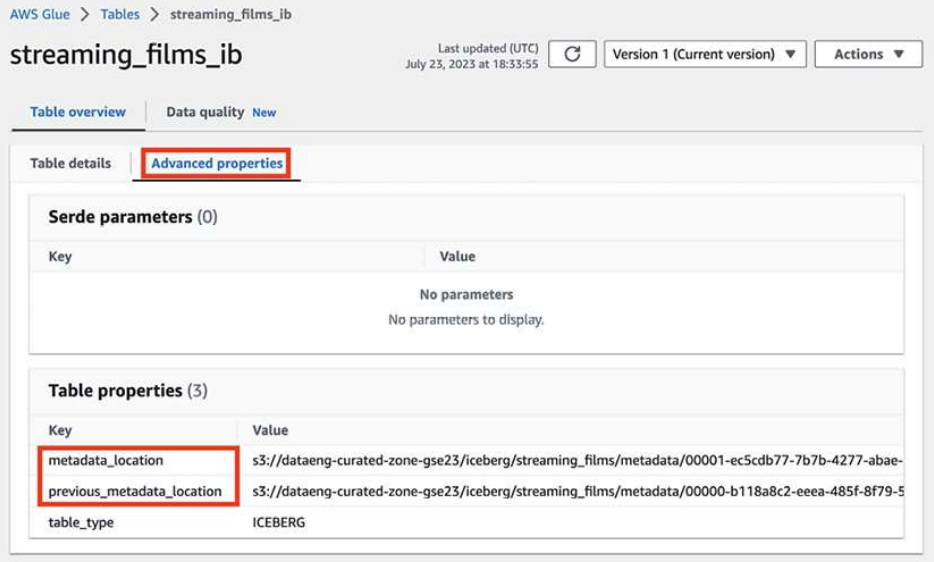

Figure 14.3: Glue advanced properties for the streaming_films_ib table

Note that the Glue Data Catalog is used to keep track of the location of the current (most recent) metadata file for a table, as well as the location of the previous version of the metadata file. Download the most recent version of the metadata file from Amazon S3 and compare it to the previous version.

4. Let's now query our new table using Athena. Go back to your browser tab where you have the Athena console open and run the following query:

```
select * from curatedzonedb_iceberg.streaming_films_ib limit 50;
```

5. We can also query the Iceberg metadata to view information on the manifests and data files that Iceberg uses to manage the table. Run the following queries (one at a time) and view the metadata results:

```
select * from "curatedzonedb_iceberg"."streaming_films_ib$manifests"
select * from "curatedzonedb_iceberg"."streaming_films_ib$files"
select * from "curatedzonedb_iceberg"."streaming_films_ib$partitions"
```

In the results of the `manifests` query, you can see information such as how many data files make up the current snapshot (in my case, it's 127) and the number of records (for my dataset, it is 8,550 records).

In the results of the `files` query, you can see details about each file that makes up the current snapshot, including items such as `record_count` and the file size. Note that there may be many small files that make up this dataset.

In the results of the `partitions` query, you can see details of how many partitions there are. We partitioned our table by film category, and in my dataset I have 16 different categories of films, so 16 partitions.

When we added data to the table, Iceberg registered a snapshot, which can be used to query the state of a table at a point in time. In the next section, we will modify the data in our table so that we can see how new snapshots are created, and how they can be queried.

Modifying data in our Iceberg table and running queries

So far, we have created a new Iceberg table and then added some data to the table, resulting in the creation of our first snapshot. Let's now delete one category of records from our table, and this change should result in the creation of a new snapshot:

1. Go back to your browser tab where you had the Athena console open and open a new **Query** tab and run the following to delete data that is in the *Documentary* category:

```
delete from curatedzonedb_iceberg.streaming_films_ib where category_name='Documentary'
```

2. Find your browser window with the S3 console and refresh the listing of files in the `meta-data` directory. Your listing of metadata files should look similar to the following:

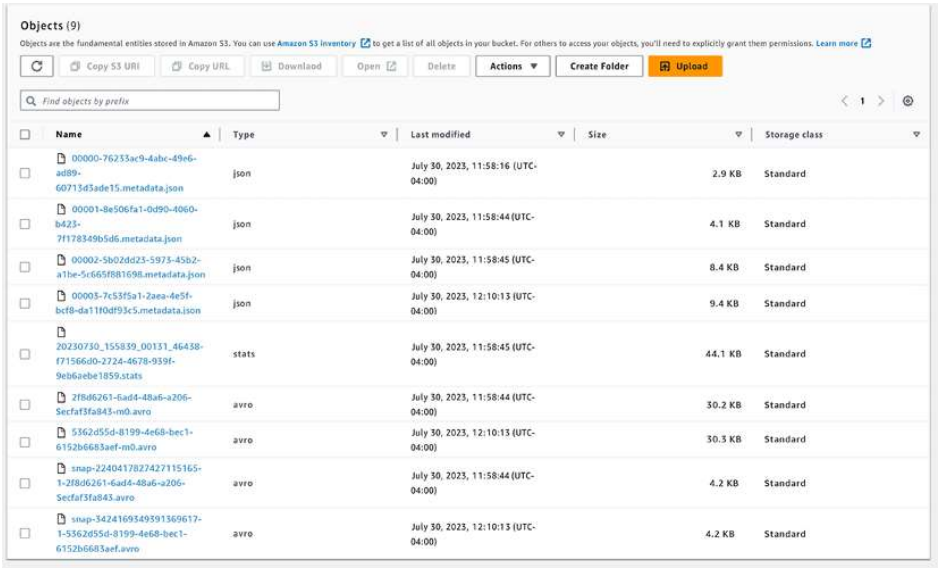

Figure 14.4: Listing of Iceberg metadata files

We now have four versions of the master `metadata.json` file. If you were to look at the table in the Glue Data Catalog, and examine **Advanced properties**, you will see that the `metadata_location` attribute now points to the file starting with `00003`, and the `previous_metadata_location` attribute points to the file starting with `00002`.

3. We now have two snapshots, and for each snapshot there is a manifest list file (Avro files starting with a UUID) and a manifest file (the Avro files starting with `snap`). And if you open the most recent `metadata.json` file, you will find that it lists the current snapshot ID (in the field `current-snapshot-id`) but has metadata for the current and previous snapshot/s. If you compare the **summary** section for the snapshots, you will see that it identifies the operation that created the new snapshot (`append` for the first snapshot and `delete` for the most current snapshot). It also lists the number of records and files that were added or deleted, partitions changed, total data files, total records, etc.
   Below is a screenshot of a portion of the most recent `metadata.json` file:

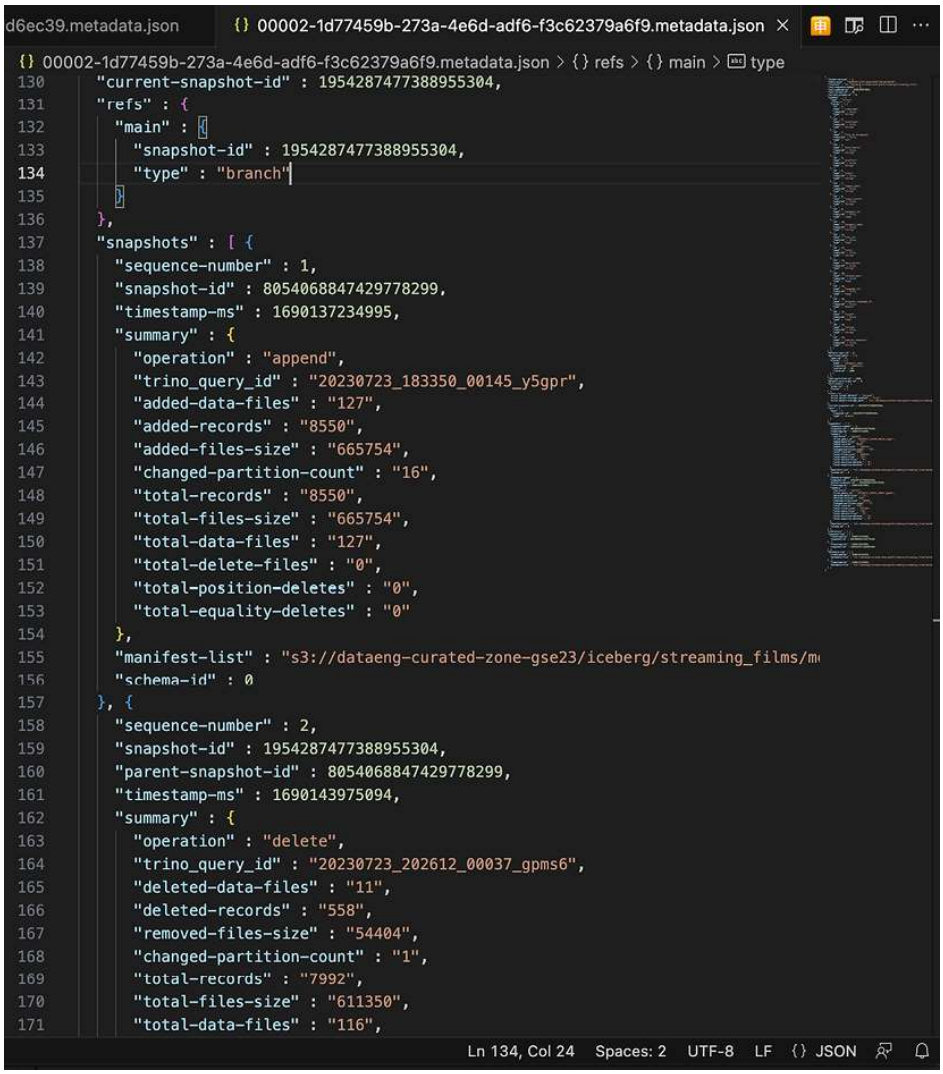

Figure 14.5: Most recent metadata.json file

The metadata file allows a query engine to identify which snapshot contains the latest state of data and to find the relevant manifest list files needed to query the latest state of data.

4. If you query the Iceberg metadata, such as by querying the partitions, you will see that there are now 15 partitions, where previously you had 16 partitions. Run the following query to list the Iceberg metadata for partitions, and note how the *Documentary* partition is no longer listed:

```
select * from "curatedzonedb_iceberg"."streaming_films_ib$partitions"
```

If we now query the manifest files, we will see that just the most recent manifest file is listed, and this contains data about the current snapshot:

```
select * from "curatedzonedb_iceberg"."streaming_films_ib$manifests"
```

5. Examine the results and notice that there are columns that provide information about the snapshot, such as the number of files, number of records, number of deleted files (which track data that has been deleted), number of deleted rows, etc.
6. Another special Iceberg query that we can run in Athena is a query that shows us all our **snapshots**. Use the following query statement to display a history of snapshots:

```
select * from "curatedzonedb_iceberg"."streaming_films_ib$history"
```

We should have two snapshots listed here. The first one is the original snapshot from when we added data to our table. The second snapshot (the one that shows a snapshot `parent_id`) is from when we deleted the `Documentary` category from our data.

7. Remember that with Iceberg, data is not physically deleted from files until you run table maintenance operations (as we discussed earlier in this chapter). This enables us to run a query that references a previous snapshot. Run the following **time travel** query, but **make sure to change the timestamp to a time prior to when you deleted the data** for the `Documentary` category. Note that the timestamp used in this query is specified in the UTC timezone, so make sure to specify a time **in the UTC timezone** for before you deleted the `Documentary` category. You can use the output of the previous command (the `history` query) to see the UTC timestamp for when the second snapshot was created, and then modify the query below to specify a timestamp between the first and second snapshots:

```
SELECT * FROM "curatedzonedb_iceberg"."streaming_films_ib" FOR TIMESTAMP AS OF TIMESTAMP '2023-07-23 19:
```

If you selected an appropriate time, you should find that you received results that showed all the movies in the `Documentary` category.

We can now move on to looking at the maintenance activities for Iceberg tables.

Iceberg table maintenance tasks

Over time, your Iceberg tables can end up using significantly more storage than the actual size of the current data. That is because with the snapshot approach, all data is kept forever … or at least until you run table maintenance tasks.

In the hands-on section of this chapter, we created a new table, inserted data into the table, and then deleted the data for a specific category. However, as we saw with the last query, we can specify a timestamp and query the table as it was prior to the deletion, demonstrating that all the data still exists.

In this section, we are going to run two table maintenance tasks. First, we will **optimize** the data layout by creating a new snapshot with the files reorganized in an optimized format. After that, we will run a **vacuum** command to delete older snapshots.

Before we do this, let's check on the size of our table in S3 as it currently is:

1. Open your browser window where you have the **Amazon S3** console.
2. Navigate to your `curated-zone` bucket, then to the folder for your `iceberg` database. For example, `dataeng-curated-zone-gse23/iceberg`.
3. Select the `streaming_films` prefix and then click on **Actions** / **Calculate total size**. Make a note of the number of files and size of data. For my dataset, I have 136 objects with a total size of 789 KB.

Optimizing the table layout

We firstly use the `OPTIMIZE` command to create a new snapshot of our data, but this time we ensure that the physical files in S3 are optimized. This covers items such as merging smaller

files into larger files and merging delete files (which contain information on data that has been deleted) into the underlying base files:

1. Open your browser window where you have the **Amazon Athena** console.
2. Before we run the optimization process against our Iceberg table, let's first look at the file metadata for our table to understand how our data is distributed across files. Execute the following statement in a query window. This lists the files that make up our current snapshot:

   ```
   select * from "curatedzonedb_iceberg"."streaming_films_ib$files"
   ```

   For my dataset, I have 116 files listed. Some of these files only have a single record, while others may have 100's of records.

3. In this step, we are going to optimize our table (sometimes also referred to as a compaction process) which involves merging small files into bigger files, and merging delete data into the underlying base files. Refer to the section earlier in this chapter titled *Maintenance tasks for Iceberg tables* for more details on how the `OPTIMIZE` statement works:

   ```
   OPTIMIZE curatedzonedb_iceberg.streaming_films_ib REWRITE DATA USING BIN_PACK
   ```

Note that Athena is not always consistent with where it allows database or table names to be quoted. For example, when querying metadata, such as by adding `$files` to the table name as we did in *Step 2*, you must have the table name in quotes. However, with the `OPTIMIZE` statement, if you put the database and table names in quotes, the query may not run and you will receive a confusing error indicating that `OPTIMIZE` is not a valid command.

Let's run the query to list file metadata for the current snapshot to determine whether our files are now more optimized:

```
select * from "curatedzonedb_iceberg"."streaming_films_ib$files"
```

For my dataset, I now have 15 files listed, one for each category of data. Recall that I started with 16 categories, but deleted the `Documentary` category, so I now have 15 categories, with all data for a category in a single file. This is a significant optimization from the 116 files that contained the data prior to this step. Each file is of course larger, with a higher record count (no more files with just 1 or 2 records in it), and this is optimized for querying.

We have now optimized the table so that when queries run against the most recent snapshot, the queries will be optimized for performance. However, all the data files for the previous snapshots are still there, as is the data for the deleted category. Go back to the S3 console and calculate total size for your `streaming_films` prefix. For my dataset, I now have 155 objects (compared to 136 prior to running the optimize) and total size is now 1 MB.

Reducing disk space by deleting snapshots

If we want to actually delete the data from storage (thereby reducing the size of data in S3), we can run a `VACUUM` operation, as we discussed earlier in this chapter. Let's take a look at how to run a `VACUUM` on our table in order to remove deleted data, and older snapshots:

1. Open your browser window where you have the **Amazon Athena** console.

2. Re-run the query to list the snapshot history for your table:

```
select * from "curatedzonedb_iceberg"."streaming_films_ib$history"
```

   We should now have three snapshots listed. One from when we inserted data into the table, a second one from when we deleted the `Documentary` category data, and a third snapshot resulting from running the `OPTIMIZE` command.

3. When we run the `VACUUM` command, we clean up old snapshots based on the value set in two table properties, as we discussed in the *Maintenance tasks for Iceberg tables* section earlier in this chapter. The defaults are to keep at least 1 snapshot and to keep at most 5 days worth of snapshots. If we want to delete the original snapshot that included the data on the `Documentary` category (which will reduce the size of data in S3) we should set the `vacuum_max_snapshot_age_seconds` property to have a much shorter duration. In the statement below, we change this value to be just 60 seconds, but because we have the default value of `vacuum_min_snapshots_to_keep` set to `1`, this will result in the oldest snapshots being deleted but will ensure that just the latest is kept.
4. Execute the following statement to modify the relevant table property:

```
ALTER TABLE curatedzonedb_iceberg.streaming_films_ib SET TBLPROPERTIES (
  'vacuum_max_snapshot_age_seconds'='60'
)
```

5. To confirm that this setting was successfully applied, run the following command to show the table properties:

```
SHOW TBLPROPERTIES curatedzonedb_iceberg.streaming_films_ib
```

Note that Athena is not always consistent with where it allows database or table names to be quoted. For example, when querying metadata, such as by adding `$history` to the table name as we did in *Step 2*, you must have the table name in quotes. However, with the `SHOW TBLPROPERTIES` statement, if you put the database and table names in quotes, the query may not run and you will receive an error indicating the command is invalid.

Let's now run the `VACUUM` command, which will result in only our most recent, optimized snapshot being kept:

```
VACUUM curatedzonedb_iceberg.streaming_films_ib
```

After running this command, we should only have one snapshot remaining (which you can confirm by running the `$history` query), and any files that were used only by the original snapshot (such as files containing data on the `Documentary` category) will have been deleted.

If we now re-run **Calculate total size** on the `streaming_files` prefix in Amazon S3, we should see fewer files, and a smaller total size. For my dataset, after running the `VACUUM` statement the number of files in the `streaming_films` prefix went down to 27 objects, and the total size down to 273 KB.

In the hands-on activity for this chapter, we used Amazon Athena to work with Apache Iceberg tables. We created a new table, inserted data, updated the table by deleting some data, and

then optimized and vacuumed the table. As we went through that process, we examined how this changed the underlying metadata that Apache Iceberg uses to track the table state.

Summary

In this chapter, we looked at how new open table formats are helping to solve some of the challenges experienced with traditional data lakes. This includes challenges around updating data at the record level, ensuring that users can consistently query a table even while it is being updated, managing changes to the underlying table schema, and more.

We did a deep dive into how three popular new table formats – Delta Lake, Apache Hive, and Apache Iceberg – use metadata to manage tables consistently and to provide advanced features such as the ability to query a table as it was at a point in the past (commonly referred to as time travel queries). We then examined how different AWS analytical services support different table formats, and even different features of those table formats.

Finally, we used the Amazon Athena service to get hands-on with working with Apache Iceberg, one of the most popular of the new table formats. After creating a new Apache Iceberg formatted table we did a number of operations on the table (such as inserting and deleting data), and also looked at how to perform table maintenance activities.

The new table formats we discussed in this chapter are having a big impact on the ability to treat a data lake more like a traditional data warehouse, and it is becoming increasingly popular to use these table formats when building new data lakes. However, there are other trends we have seen over the last few years that are having an impact on how organizations work across the multiple data lakes that may end up being created. And these trends are also leading to a new strategy where different teams in an organization own their own data lakes, rather than attempting to centralize all data. This new approach is often referred to as a data mesh approach, and we will do a deep dive into what a data mesh is in the next chapter.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/9s5mHNyECd