O'REILLY®

Apache Iceberg
The Definitive Guide

Data Lakehouse Functionality, Performance, and Scalability on the Data Lake

Early Release
Raw & Unedited

Compliments of

Tomer Shiran,
Jason Hughes,
Alex Merced &
Dipankar Mazumdar

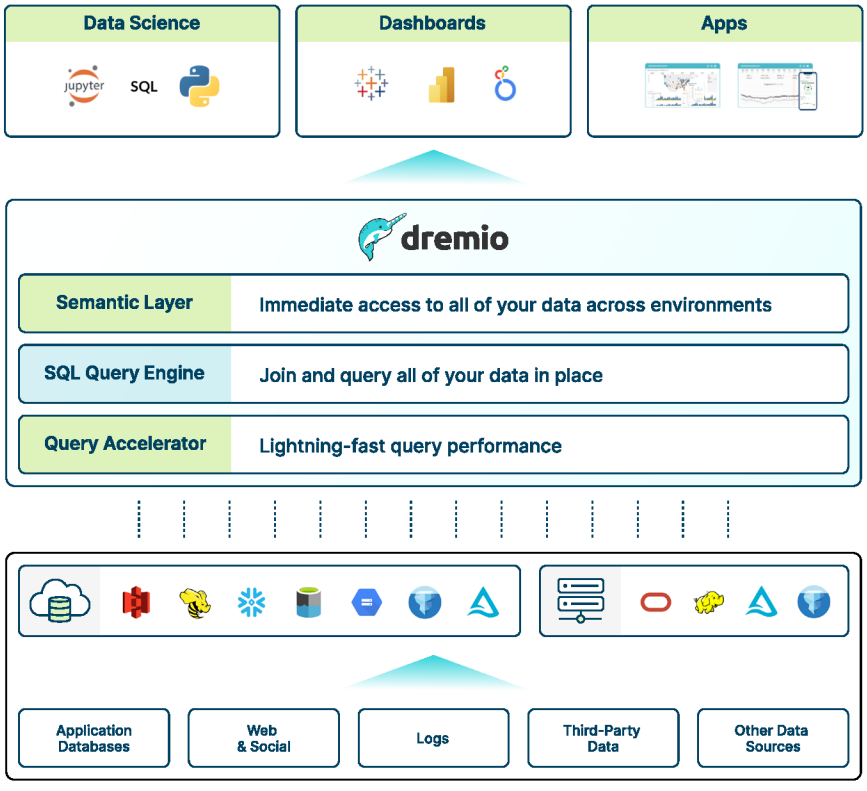
dremio

The Easy and Open Data Lakehouse

Self-service analytics with data warehouse functionality and data lake flexibility across all of your data

Dremio is an open data lakehouse, providing self-service SQL analytics, data warehouse performance and functionality, and data lake flexibility across all of your data.

We believe in the future of open data, co-creating Apache Arrow and directly contributing to Apache Iceberg, the next-generation table format for data.



Try it for free at Dremio.com/get-started

Apache Iceberg: The Definitive Guide

Data Lakehouse Functionality, Performance, and Scalability on the Data Lake

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Tomer Shiran, Jason Hughes, Alex Merced, and Dipankar Mazumdar

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Apache Iceberg: The Definitive Guide
by Tomer Shiran, Jason Hughes, Alex Merced, and Dipankar Mazumdar

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Gary O'Brien
Production Editor: Elizabeth Faerm
Copyeditor: TO COME
Proofreader: TO COME

Indexer: TO COME
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

February 2024: First Edition

Revision History for the Early Release
2023-02-27: First Release
2023-05-19: Second Release
2023-07-10: Third Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098148621* for release details.

This work is part of a collaboration between O'Reilly and Dremio. See our statement of editorial independence.

978-1-098-14862-1

[LSI]

Table of Contents

CHAPTER 1
What Is Apache Iceberg?

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

Data is a primary asset for organizations to make critical business decisions. Whether it is analyzing historical trends of the annual sales of a product or making predictions about future opportunities, data shapes the direction for organizations to make reliable choices. Further, in today's day and age, data isn't just nice-to-have, but a requirement for not only winning in the market, but even competing at all. With such a massive demand for information, there has been an enormous effort to accumulate data generated by the various systems within an organization to derive insights.

At the same time, the rate at which the various operational and analytical systems have been generating data has also skyrocketed. While more data has presented enterprises the opportunity to make better-informed decisions, there is also a dire need to have a platform that allows storing and analyzing all of this data so it can be used to build analytical products such as Business Intelligence (BI) reports and machine learning models to support decision making. This chapter will walk us through the history and evolution of data platforms from a practical point of view and present the benefits of a lakehouse architecture with open table formats like Apache Iceberg.

How Did We Get Here? A Brief History

In terms of storage and processing systems, relational databases (RDBMS) have long been a standard option for organizations to keep a record of all of their transactional data. For example, if you are a transportation company, you would like to maintain information about any new bookings made by a customer. This new booking would be a new *row* in a relational database system. Information like this can support the day-to-day operations of a business. RDBMS systems used for these purposes support a specific data processing category called Online Transaction Processing (OLTP). Examples of these OLTP-optimized RDBMS systems are PostgreSQL, MySQL, and Microsoft SQL Server. These OLTP systems are designed and optimized for interacting with one or a few rows at a time very quickly. However, for the example above, if you want to understand the *average profit* made on all of the new bookings for the last quarter, using the data stored in an OLTP-optimized RDBMS will lead to significant performance problems when your data gets large enough.

Now, imagine that your organization has a large number of operational systems. These systems generate a vast amount of data. Your analytics teams' goal is to build dashboards that rely on aggregations of the data from these different data sources (application databases). Unfortunately, OLTP systems are not designed to deal with such complex aggregate queries involving a large number of historical records. These workloads are known as Online Analytical Processing (OLAP) workloads. To address these limitations, a different kind of system optimized for OLAP workloads was needed.

Foundational Components of a System Designed for OLAP Workloads

A system designed for OLAP workloads is composed of a set of technological components that enable supporting modern-day analytical workloads.

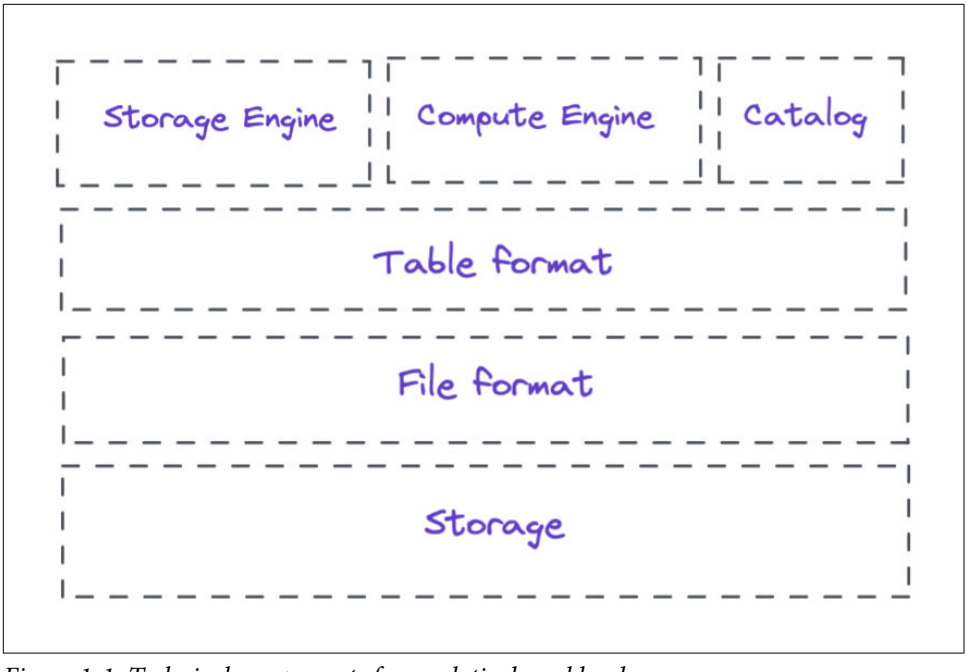


Figure 1-1. Technical components for analytical workloads

Storage

To analyze historical data coming in from a variety of sources, you need to have a system that allows you to store such huge amounts of data. Therefore, storage is the first component we would need in a system that can deal with analytical queries on large datasets. There are a few options for storage, such as a local file system on a direct-attached storage (DAS), a distributed file system on a set of nodes that you operate like Hadoop Distributed File System (HDFS), or object storage provided as a service by cloud providers like Amazon Simple Storage Service (S3).

Regarding the types of storage, you could use row-oriented databases or columnar. In recent years, columnar-oriented databases have seen a tremendous adoption rate as they have proved more efficient when dealing with vast volumes of data.

File Format

The file format is a component responsible for organizing the raw data in a particular format, which is then stored in a storage system. The choice of a file format impacts things such as compression of the files, data structure, and performance for a given workload.

File formats generally fall into three high-level categories: structured, semi-structured, and unstructured. In the structured and semi-structured categories, file

formats can be row-oriented or column-oriented (columnar). Row-oriented file formats store all columns of a given row together, while column-oriented file formats store all rows of a given column together. Two common examples of row-oriented file formats are comma-separated values (CSV) and Apache Avro. Examples of columnar file formats are Apache Parquet and Apache ORC.

Depending on the use cases, certain file formats can be more advantageous. For example, row-oriented file formats are generally better when dealing with a small number of records at a time. In comparison, columnar file formats are generally better if you are dealing with a sizable portion of records at a time.

Table Format

A table format is another critical component for a system that can support analytical workloads with aggregated queries on a vast volume of data. Table formats take the role of a metadata layer on top of the file formats described above and are responsible for specifying how the data files should be laid out on the storage.

Ultimately the goal of a table format is to abstract the complexity of the physical data structure and facilitate capabilities such as the ability to do data manipulation language (DML) operations (e.g., doing inserts, updates, deletes) and change a table's schema. Table formats also bring in the atomicity and consistency guarantees required for the safe execution of the DML operations on the data.

Storage Engine

A storage engine is the system responsible for actually doing the work of laying out the data in the form specified by the table format and keeping all the files & data structures up to date with the new data. Storage engines handle some of the critical tasks, such as physical optimization of the data, index maintenance, and getting rid of old data.

Catalog

When dealing with data from various sources and on a larger scale, it is important to identify the data you might need for your analysis quickly. A catalog's role is to tackle this problem by leveraging metadata to identify datasets. The catalog is the central location that engines and users can go to find out about the existence of a table and additional information about each table, such as table name, table schema, and where that table's data is stored on the storage system. Some catalogs are internal to a system and can only be directly interacted with via that system's engine, such as Postgres and Snowflake, while some catalogs are open for any system to use, such as Hive and Project Nessie.

Compute Engine

A compute engine is the final component needed in a system that can efficiently deal with a massive amount of data persisted in a storage system. A compute engine's role in such a system would be to run user workloads to process the data. Depending on the volume of data, computation load, and type of workload, you can utilize one or more compute engines to process the data. When dealing with a large dataset and/or heavy computational requirements, you might need to use a distributed compute engine in a processing paradigm called Massively Parallel Processing (MPP). A few examples of MPP-based compute engines are Apache Spark, Snowflake, and Dremio.

Bringing It All Together

Traditionally for OLAP workloads, these technical components have all been tightly coupled into a single system known as a data warehouse. Data warehouses allow organizations to store data coming in from a variety of sources and run analytical workloads on top of it. In the next section, we will discuss in detail the capabilities of a data warehouse, how the technical components are integrated, and the pros and cons of using such a system.

Data Warehouse

A data warehouse (DW) or OLAP database is a centralized repository that supports storing large volumes of data ingested from various sources such as operational systems, application databases, and logs.

Looking at the technical components described in the section above, this is how they get incorporated into a data warehouse. Figure 1-2 presents an architectural overview.

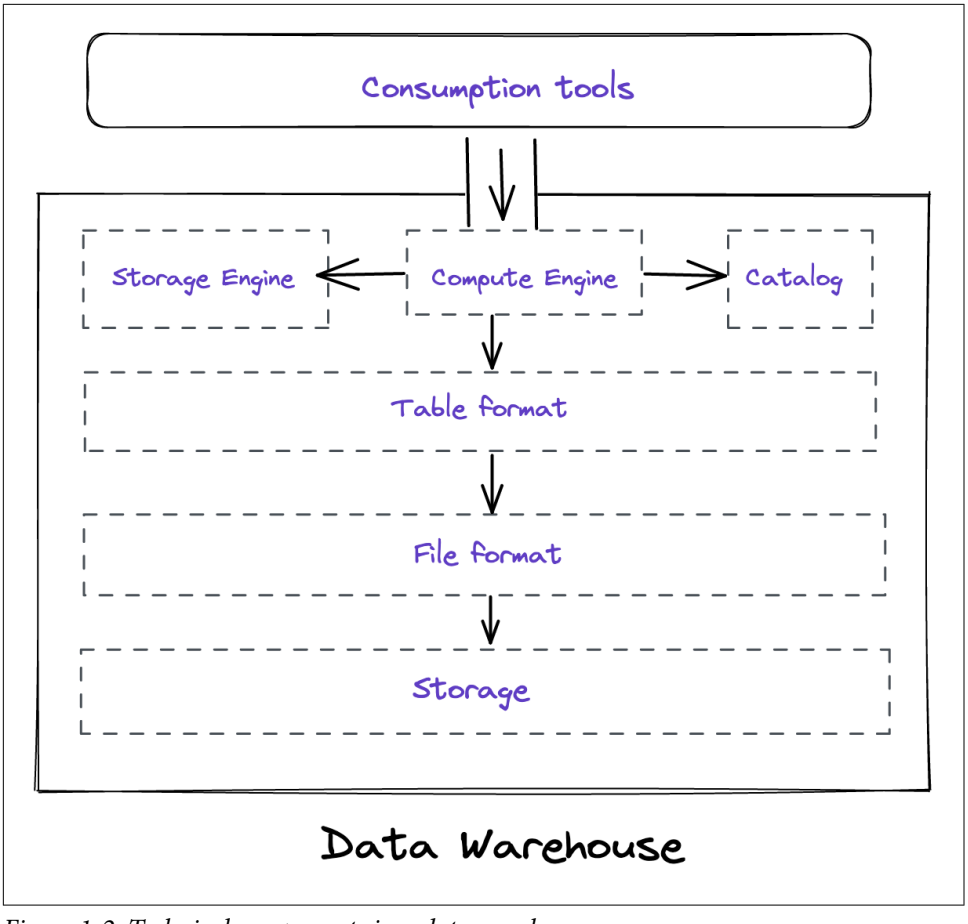


Figure 1-2. Technical components in a data warehouse

A data warehouse owns all the technical components in a single system. So, all the data stored in a DW system is stored on the DW's storage in the DW's proprietary file format in the DW's proprietary table format. This data is then managed exclusively by the DW's storage engine, registered in the DW's catalog, and can only be accessed by the user or analytical engines through the DW's compute engine.

Up until about 2015, the majority of DWs had the storage and compute components tightly coupled together on the same nodes, since most DWs were designed and run on-premises. However, this resulted in a lot of problems. Because datasets grew in volume faster and faster, as well as the number and intensity of workloads (i.e., compute tasks running on the warehouse), scaling became a big issue. Specifically, there was no way to independently increase the compute and storage resources depending on your tasks. If your storage needs grew faster than your compute needs,

it didn't matter – you still needed to pay for additional compute even though you didn't need it.

This led to the next generation of data warehouses being built with a big focus on the cloud. The next generation of data warehouses was built starting in around 2015 as cloud-native, allowing you to separate these two components and scale compute and storage resources as you would like for your tasks, as well as even shut down compute when you weren't using it and not lose your storage.

Pros and Cons of a Data Warehouse

While data warehouses, whether on-premises or cloud-based, make it easy for enterprises to make sense of all their historical data quickly, there are also certain areas where a warehouse still causes issues. We list the pros and cons of a data warehouse in Table 1-1.

Table 1-1. Pros and cons of a data warehouse

Pros	Cons
A data warehouse serves as the single source of truth as it allows storing & querying data from various sources.	Data in a warehouse is locked into a vendor-specific system that only the warehouse's compute engine can use, thereby locking the data.
Supports querying vast amounts of historical data enabling analytical workloads to run quickly.	Expensive in terms of both storage and computation. As the workload increases, the cost becomes hard to manage.
Provides effective data governance policies to ensure data is available, usable and aligned with the security policies.	Mainly supports structured data.
Organizes the data for you, ensuring it's optimized for querying	Organizations cannot run advanced analytical workloads such as machine learning natively in a data warehouse.
Ensures data written to a table conforms to the technical schema	

Data warehouses act as a centralized repository for organizations to store all their data coming in from a multitude of sources, allowing data consumers such as analysts and BI engineers to access data easily and quickly from one single source to start their analysis. In addition, the technological components powering data warehouses enable accessing vast volumes of data while supporting workloads such as business intelligence to run on top of it.

Although data warehouses have been elemental in the democratization of data and allowed businesses to derive historical insights from varied data sources, they are primarily limited to relational workloads. For example, if you go back to the transportation company example from earlier and say now, you want to derive insights into *how much total sales you will make in the next quarter.* In this case, you will need to build a forecasting model using historical data. However, you cannot achieve this capability natively with a data warehouse as the compute engine & the other technical components are not designed for machine learning-based tasks. So, the only

viable option is moving or exporting the data from the warehouse to other platforms supporting it. This means you will have data in multiple copies, which can lead to critical issues such as data drift, model decay, etc.

Another hindrance to running advanced analytical workloads on top of a data warehouse is that it only has support for structured data. But, the rapid generation and availability of other types of data, such as semi-structured and unstructured data (JSON, images, texts, etc.), have allowed machine learning models to bring out interesting insights. For our example, this could be understanding the *sentiments of all the new booking reviews* made in the last quarter. This ultimately impacts an organization's ability to make future-oriented decisions.

There are also specific design challenges in a data warehouse. If you go back to the diagram (Figure 1-2) above, you can see that all six technical components are tightly coupled in a data warehouse. Before you understand what that implies, an essential thing to observe is that both the file and the table formats are *internal* to a particular data warehouse. This design pattern leads to a *closed form* of data architecture. It means that the actual data is accessible only using the data warehouse's compute engine, which is specifically designed to interact with the warehouse's table and file formats. This type of architecture leaves organizations with a massive concern about locked-in data. With the increase in workloads and the vast volumes of data ingested to a warehouse over time, you are bound to that particular platform. And that means your analytical workloads, such as BI and any future tools you plan to onboard, have to run specifically on top of this particular data warehouse only. This also prevents you from migrating to another data platform that can cater specifically to your requirements.

Additionally, a significant cost factor is associated with storing data in a data warehouse and using the compute engines to process that data. This cost only increases with time as you increase the number of workloads in your environment, thereby invoking more compute resources. Other than the monetary costs, there are additional overheads, such as the need for engineering teams to build and manage numerous ETL (extract, transform, load) pipelines to move data from operational systems, delayed time-to-insight on the part of the data consumers, etc. These challenges have led organizations to seek alternative data platforms that allow data to be within their control and stored in open file formats, thereby allowing downstream applications such as BI and machine learning to run parallelly with much-reduced costs. It led to the emergence of Data Lakes.

Data Lake

While a data warehouse provides a mechanism for running analytics on structured data, it still had several issues that left a need for different solutions:

- A data warehouse could only store structured data
- Storage in a data warehouse is generally more expensive than on-prem Hadoop clusters or cloud object storage.

To address these issues the goal was to have an alternative storage solution that was cheaper and could store all our data. This is what's called the data lake.

Originally, you'd use a Hadoop to allow you to use a cluster of inexpensive computers to store large amounts of structured and unstructured data. Although it wasn't enough to just be able to store all this data. You'd want to run analytics on it too.

The Hadoop ecosystem included MapReduce, an analytics framework from which you'd write analytics jobs in Java and run them on the cluster. Many analysts are more comfortable writing SQL than Java, so Hive was created to convert SQL statements into MapReduce jobs.

To write SQL, a mechanism to distinguish which files in our storage are part of the dataset or table we want to run the SQL against was needed. This resulted in the birth of the Hive table format which recognized a directory and the files inside it as a table.

Over time. people moved away from using Hadoop clusters to using Cloud Object storage as it was easier to manage and cheaper to use. MapReduce also fell out of favor for other distributed query engines like Apache Spark, Presto, and Dremio. What did stick around was the Hive table format which became the standard in the space for recognizing files in your storage as singular tables on which you can run analytics.

A distinguishing feature of the data lake as compared to the data warehouse is the ability to leverage different compute engines for different workloads. This is important because there's never been a silver bullet of a compute engine that is best for every workload. This is just inherent to the nature of computing since there are always tradeoffs, and what you decide to tradeoff determines what a given system is good for and what it is not as well suited for.

Note that in data lakes, there isn't really any service that fulfills the needs of the storage engine function. Generally the compute engine decides how to write the data, then the data is usually never revisited and optimized, unless rewriting entire tables or partitions which is usually done on an ad-hoc basis. Refer to Figure 1-3 to see how the components of a data lake interact with one another.

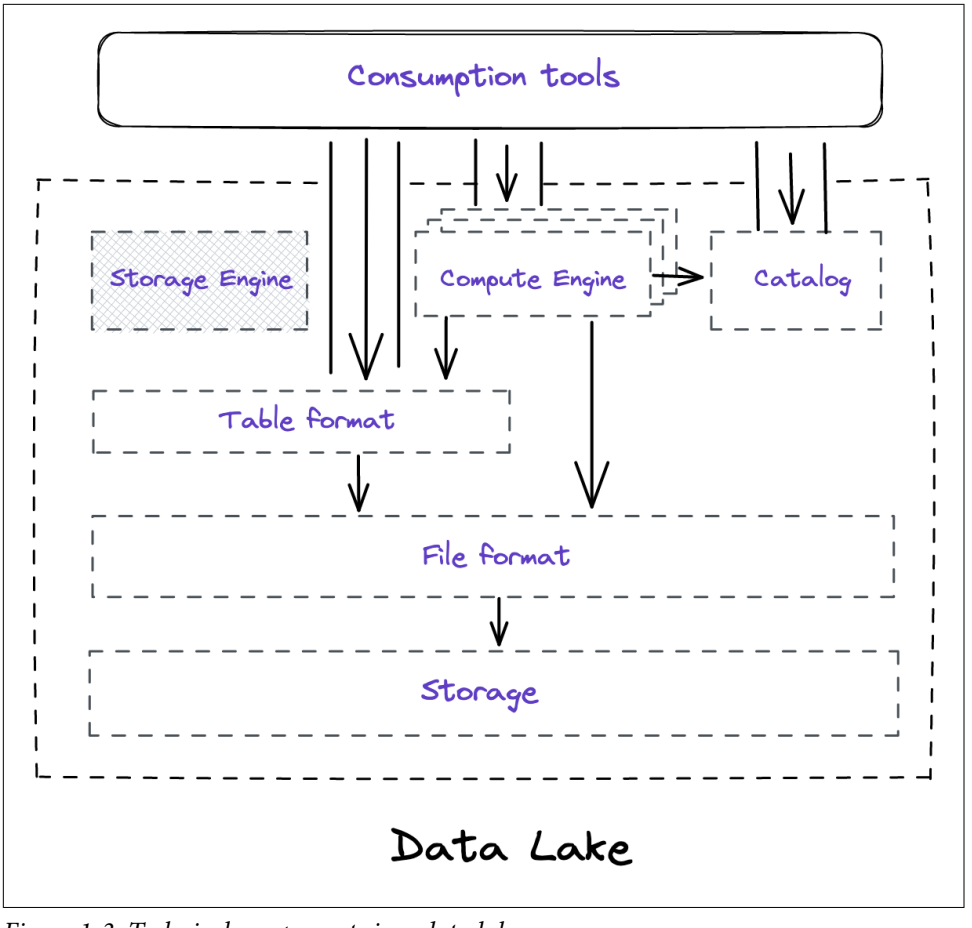


Figure 1-3. Technical components in a data lake

Pros and Cons of a Data Lake

No architectural pattern is perfect and that applies to data lakes. While data lakes have a lot of benefits like Lower Costs, the ability to store in open formats and handle unstructured data; data lakes also have several disadvantages such as performance issues, lack of ACID guarantees and lots of configuration. You can see a summary of these pros and cons in Table 1-2.

Pro: Lower Cost

The costs of storing data and executing queries on a data lake are much lower than in a data warehouse. This makes a data lake particularly useful for enabling analytics on data that isn't high enough priority to justify the cost of a data warehouse enabling a wider analytical reach.

Pro: Store Data in Open Formats
: In a data lake you can store the data in any file format you like unlike data warehouses where you have no say in how the data is stored, which would typically be a proprietary format built for that particular data warehouse. This allows you to have more control over the data and consume the data in a greater variety of tools that can support these open formats.

Pro: Handle Unstructured Data
: Data warehouses can't handle unstructured data, so if you wanted to run analytics on unstructured data the data lake was the only option.

Con: Performance
: Since each component of a data lake is decoupled, many of the optimizations that can exist in tightly coupled systems are absent. While they can be recreated, it requires a lot of effort and engineering performant to cobble the components (storage, file format, table format, engines) in a way to give you the comparable performance of a data warehouse. This made data lakes undesirable for high priority data analytics where performance and time mattered.

Con: Lots of Configuration
: As previously mentioned, creating a tighter coupling of your chosen components with the level of optimizations you'd expect from a data warehouse would require significant engineering. This would result in a need for lots of data engineers to configure all these tools, which can also be costly.

Table 1-2. Pros and cons of a data lake

Pros	Cons
• Lower Cost • Store Data in Open Formats • Handle unstructured data	• Performance • Lack of ACID Guarantees • Lots of Configurations

Should I Run Analytics on the Data Lake or Data Warehouse?

While Data Lakes provided a great place to land all your structured and unstructured data, there were still imperfections. After running ETL to land your data in your data lake you'd generally take one of two tracks when running analytics.

A Subset of Data goes to the Data Warehouse
: You'd set up an additional ETL pipeline to create a copy of a curated subset of data that is for high priority for analytics and store it in the warehouse to get the performance and flexibility of the data warehouse.

  This results in several issues:

- Additional costs in the compute for the additional ETL work and the cost for storing a copy of data you are already storing in a data warehouse where the storage costs are often greater.
- Additional copies of the data may be needed to populate data marts for different business lines and even more copies as analysts create physical copies of data subsets in the form of BI extracts to speed up dashboards. Leading to a web of data copies that are hard to govern, track and keep in sync.

You run analytics directly on the Data Lake
: You'd use query engines that support data lake workloads like Dremio, Presto, Apache Spark, Trino, Apache Impala and more to execute queries on the data lake. These engines are generally well suited for read-only workloads. However, due to the limitations of the Hive table format, they ran into complexity when trying to update the data safely from the data lake.

So the data lake and data warehouse each have their unique benefits and unique cons. It would be to our advantage to develop a new architecture that brings together all these benefits while minimizing all their faults, and that architecture is called a data lakehouse.

Enter Data Lakehouse

While using a data warehouse gave us performance and ease of use, analytics on data lakes gave us lower costs and reduced data drift from a complex web of data copies. The desire to thread the needle leads to great strides and innovation leading to what we now know as the data lakehouse.

What makes a data lakehouse truly unique are data lake table formats that eliminate all the previous issues with the Hive table format. You store the data in the same places you would with a data lake, you use the query engines you would use with a data lake, your data is stored in the same formats it would be on a data lake, what truly transforms your world from a "read only" data to a "center of my data world" data lakehouse is the table format (refer to Figure 1-4). Table formats enabled better consistency, performance and ACID guarantees when working with data directly on your data lake storage leading to several value propositions.

Fewer Copies, Less Drift
: With ACID guarantees and better performance you can now move workloads typically saved for the data warehouse like updates and other data manipulation. If you don't have to move your data to the lakehouse you can have a more streamlined architecture with fewer copies. Fewer copies mean less storage costs, less compute costs from moving data to a data warehouse, and better governance of your data to maintain compliance with regulations and internal controls.

Faster Queries, Fast Insights

The end goal is always to get business value from quality insights from our data, everything is else just steps to that end. If you can get faster queries that means you can get insights faster. Data Lakehouses enable faster performing queries by using optimizations at the query engine, table format and file format.

Mistakes Don't Have to Hurt

Data Lakehouse table formats enable the possibility to undo mistakes by using snapshot isolation, allowing you to revert the table back to prior snapshots. You can work with your data but not have to be up at night wondering if a mistake will lead to hours of auditing, repairing then backfilling.

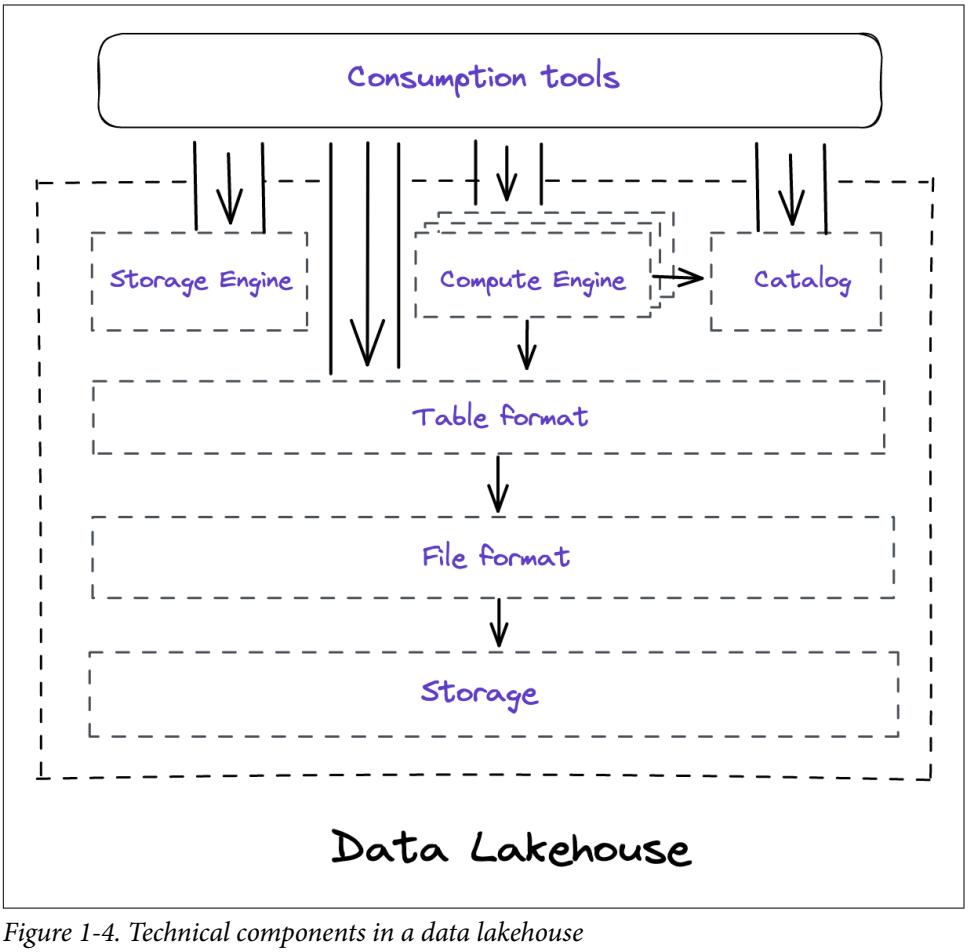


Figure 1-4. Technical components in a data lakehouse

Affordable Architecture is Business Value
: There are two ways to increase profits, increase revenue and lower costs, and data lakehouses not only help you get business insights to drive up revenue but can also help you lower costs. Reduce storage costs from avoiding duplication of your data, avoid additional compute costs from additional ETL work to move data and enjoy lowers prices for the storage and compute you are using relative to typical data warehouse rates.

Open Architecture, Peace of Mind
: Data Lakehouses are built on open formats such Apache Iceberg as a table format and Apache Parquet as a file format. Many tools can read and write to these formats which allows you avoid vendor lock-in which results in cost creep and prevents tool lock-out where you data sits in formats in which tools that could be great solutions can't access. Use open formats, you can rest easy that your data won't be siloed into a narrow set of tools.

The Key to the Puzzle
: So, with modern innovations from the open standards previously discussed, the best of all worlds can exist by operating strictly on the data lake, and this architectural pattern is the data lakehouse. The key component that makes all this possible is the table format that enables engines to have the guarantees and performance when working with your data that just didn't exist before, now let's get started with the Apache Iceberg table format.

What Is a Table Format?

A table format is a method of structuring a dataset's files to present them as a unified "table." From the user's perspective, it can be defined as the answer to the question "what data is in this table?"

This simple answer enables multiple individuals, teams, and tools to interact with the data in the table concurrently, whether they are reading from or writing to it. The main purpose of a table format is to provide an abstraction of the table to users and tools, making it easier for them to interact with the underlying data in an efficient manner.

Table formats have been around since the inception of Relational Database Management Systems (RDBMSs) such as System R, Multics, and Oracle, which first implemented Edgar Codd's relational model, although the term "table format" was not used at that time. In these systems, users could refer to a set of data as a table, and the database engine was responsible for managing the dataset's byte layout on disk in the form of files, while also handling complexities like transactions.

All interactions with the data in these RDBMSs, such as reading and writing, are managed by the database's storage engine. No other engine can interact with the files

directly without risking the system's corruption. The details of how the data is stored are abstracted away, and users take for granted that the platform knows where the data for a specific table is located and how to access it.

However, in today's big data world, relying on a single closed engine to manage all access to the underlying data is no longer practical, as traditional RDBMSs are no longer sufficient.

In a data lake, all your data is stored as files in some storage solution (e.g., Amazon S3, Azure's ADLS, Google's GCS), so a single table may be made of dozens, hundreds, or even thousands of individual files on that storage. When using SQL with our favorite analytical tools or writing ad hoc scripts in languages like Java, Scala, Python, and Rust, we wouldn't want to constantly define which of these files are in the table and which of them aren't. Not only would this be tedious but it would also likely lead to inconsistency across different uses of the data.

So the solution was to create a standard method of understanding "what data is in this table" for data lakes.

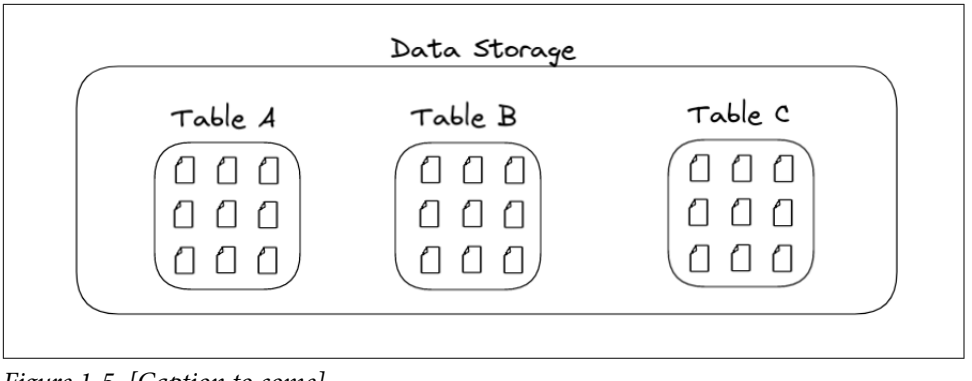


Figure 1-5. [Caption to come]

Early Data Lake Table Formats

When it came to the world of running analytics on Hadoop data lakes, the MapReduce framework was used which required users to write complex and tedious java jobs, which wasn't accessible to many analysts. Facebook, feeling the pain of this situation developed a framework called Hive in 2009. Hive provided a key benefit to make analytics on Hadoop much easier, the ability to write SQL instead of MapReduce jobs directly.

The Hive framework would take SQL statements and then convert them into MapReduce jobs that can be executed. In order to write SQL statements, there had to be a mechanism for understanding what data on your Hadoop storage represented

a unique table, and the Hive table format and the Hive Metastore for tracking these tables was born.

The Hive Table Format

The Hive table format took the approach of defining a table as any and all files within a specified directory, the partitions of those tables would be the subdirectories.

These directory paths defining the table are tracked by a service called the Hive metastore which query engines can access to know where to find the data applicable to their query.

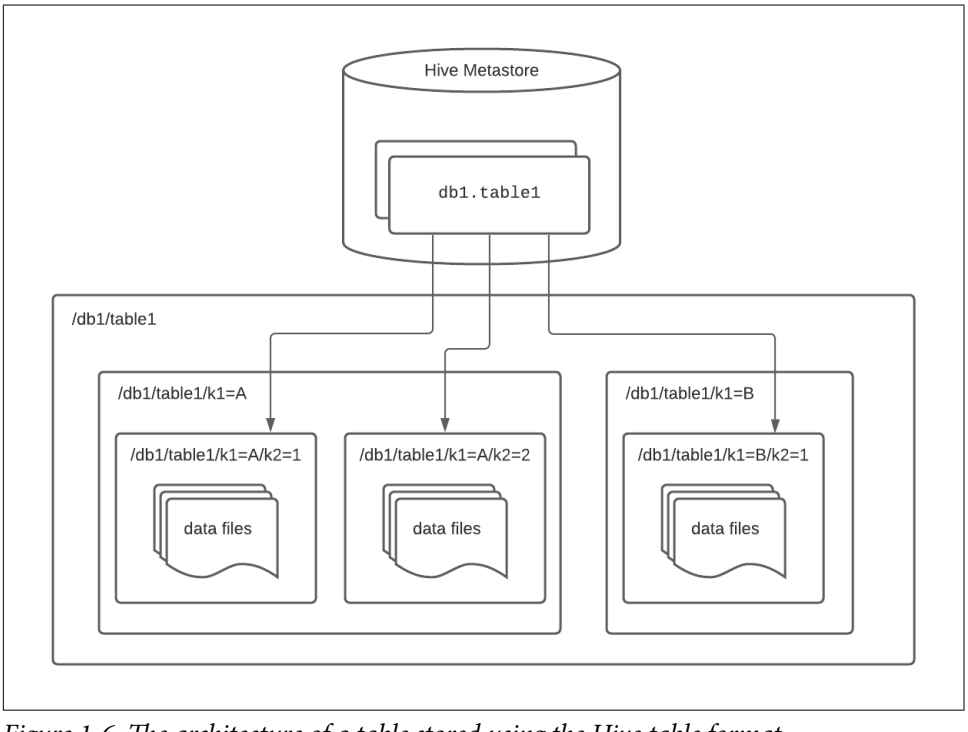


Figure 1-6. The architecture of a table stored using the Hive table format

The Hive table format had several benefits:

- It enabled more efficient query patterns than full table scans, so techniques like partitioning and bucketing made it possible to avoid scanning every file for faster queries
- It was file format agnostic so it allowed the data community overtime to develop better file formats like Apache Parquet and use them in their Hive tables and

did not require transformation prior to making the data available in a Hive table (e.g., Avro, CSV/TSV).

- Through atomic swaps of the listed directory in the hive metastore, you can make all or nothing (atomic) changes to an individual partition in the table.
- Over time this became the de facto standard working with most data tools and providing a uniform answer to "what data is in this table?".

While these benefits were significant there were also many limitations that become apparent as time passed:

- File level changes are inefficient since there was no mechanism to atomically swap a file in the same way the Hive Metastore could be used to swap a partition directory. You are essentially left making swaps at the partition level to update a single file atomically.
- While you could atomically swap a partition there wasn't a mechanism for atomically updating multiple partitions as one transaction. This opens up the possibility for end users seeing inconsistent data between transactions updating multiple partitions.
- There really aren't good mechanisms to enable concurrent simultaneous updates, especially with tools beyond Hive itself.
- An engine listing files and directories was time consuming and slowed down queries. Having to read and list files and directories that may not need scanning in the resulting query comes at a cost.
- Partition columns were often derived from other columns, such as deriving a month column from a timestamp. Partitioning only helped if you filtered by the partition column, and someone who has a filter on the timestamp column may not intuitively know to also filter on the derived month column leading to a full table scan since partitioning was not taken advantage of.
- Table statistics would be gathered through asynchronous jobs resulting in often state table statistics if any statistics were available at all, making it difficult for query engines to further optimize queries.
- Since object storage often throttles requests against the same prefix (think of an object storage prefix as analogous to a file directory), queries on tables with large numbers of files in a single partition (so all the files would be in one prefix) can have performance issues.

The larger the scale of the datasets and use cases, the more these problems would be amplified resulting in significant pain in need of a new solution, so newer table formats were created.

Modern Data Lake Table Formats

In seeking to address the limitations of the Hive table format, a new generation of table formats arose with different approaches in solving the problems with Hive.

Creators of modern table formats realized the flaw that leads to challenges with the Hive table format was that the definition of the table was based on the contents of directories, not on the individual data files. Modern table formats like Apache Iceberg, Apache Hudi, and Delta Lake all took this approach of defining tables as a canonical list of files, providing metadata for engines information on which *files* make up the table, not which *directories*. This more granular approach to defining "What is a table" unlocked the door to features like ACID (ATOMICITY, CONSISTENCY, ISOLATION, DURABILITY) Transactions, Time Travel and more.

What Problems Do Modern Table Formats Solve?

Modern table formats all aim to bring a core set of major benefits over the Hive table format:

- Modern table formats allowed for ACID transactions which are safe transactions that either complete in full or are canceled. In legacy formats like the Hive table format, many transactions could not have these guarantees.
- Enable safe transactions when there are multiple writers. If two or more writers write to a table, there is a mechanism to make sure the writer that completes their write second is aware and considers what the other writer(s) have done to keep the data consistent.
- Better collection of table statistics and metadata that can allow a query engine scanning the data to plan more efficiently.

While most modern table formats provide the above, the Apache Iceberg format provides these and solves many of the other problems with the Hive table format.

1.3.1 What Is Apache Iceberg?

Apache Iceberg is a table format created in 2017 at Netflix by Ryan Blue and Daniel Weeks that came out of the need to overcome challenges with performance, consistency and more with the Hive table format. In 2018, the project was open-sourced and donated to the Apache Software Foundation where many other organizations started getting involved with the project including Apple, Dremio, AWS, Tencent, LinkedIn, Stripe, and many more who have contributed to the project since.

How Did Apache Iceberg Come to Be?

Netflix in the creation of what became the Apache Iceberg format came to a conclusion that many of the problems with the Hive Format stemmed from one simple but fundamental flaw. That flaw is that each table is tracked as directories and subdirectories limiting the granularity that is necessary to provide consistency guarantees, better concurrency and more.

With this in mind netflix set out to create a new table format with several goals in mind:

Consistency
: If updates to a table occur over multiple transactions, it is possible for end users to experience inconsistency in the data they are viewing. An update to a table across multiple partitions should be done fast and atomically so data is consistent to end users. They either see the data before the update or after the update and nothing in between.

Performance
: With Hive's file/directory listing bottleneck, query planning would take excessively long to complete before actually executing the query. The table should provide metadata and avoid excessive file listing so not only can query planning can be quicker but the resulting plans can also be executed faster since they scan only the files necessary to satisfy the query.

Easy to Use
: To get the benefits of techniques like partitioning, end users should not have to be aware of the physical structure of the table. The table should be able to give users the benefits of partitioning based on naturally intuitive queries and not depend on filtering extra partition columns derived from a column they are already filtering by (like filtering by a month column when you've already filtered the timestamp it is derived from).

Evolvability
: Updating schemas of Hive tables could result in unsafe transactions and updating how a table is partitioned would result in a need to rewrite the entire table. A table should be able to evolve its schema and partitioning scheme safely and without rewriting the table.

Scalability
: All the above should be able to be accomplished at the petabyte scale of Netflix's data.

So they began creating the Iceberg format which focuses on defining tables as a canonical list of files instead of tracking a table as a list of directories and subdirectories.

The Apache Iceberg project is a specification, a standard of how metadata defining a data lakehouse table should be written across several files. To support the adoption of this standard Apache Iceberg has many support libraries to help individuals work with the format or for compute engines to implement support. Along with these libraries, the project has created implementations for open source compute engines like Apache Spark and Apache Flink.

Apache Iceberg aims for existing tools to embrace the standard and doesn't aim to create its own storage service, compute engine and running services in the core project in hopes existing options will support working with the standard.

The purpose of this approach is to let the ecosystem of existing data tools build out support for Apache Iceberg tables and let Iceberg become the standard for how engines can recognize and work with tables on the data lake. The goal is Apache Iceberg becomes so ubiquitous in the ecosystem that it becomes another implementation detail that many users don't have to think about, they just know they are working with tables and don't need to think about it beyond that regardless which tool they are using to interact with the table. This is already becoming a reality as many tools allow end users to work with Apache Iceberg tables so easily that they don't need to understand the underlying Iceberg format. Eventually with automated table optimization and ingestion tools, even more technical users like data engineers won't have to think as much about the underlying format.

1.3.3 Apache Iceberg's Architecture

Apache Iceberg tracks a table's partitioning, sorting, schema over time, and so much more by a tree of metadata (Refer to figure 1-7) that an engine can use to plan their queries at a fraction of the time it would take with a Hive table.

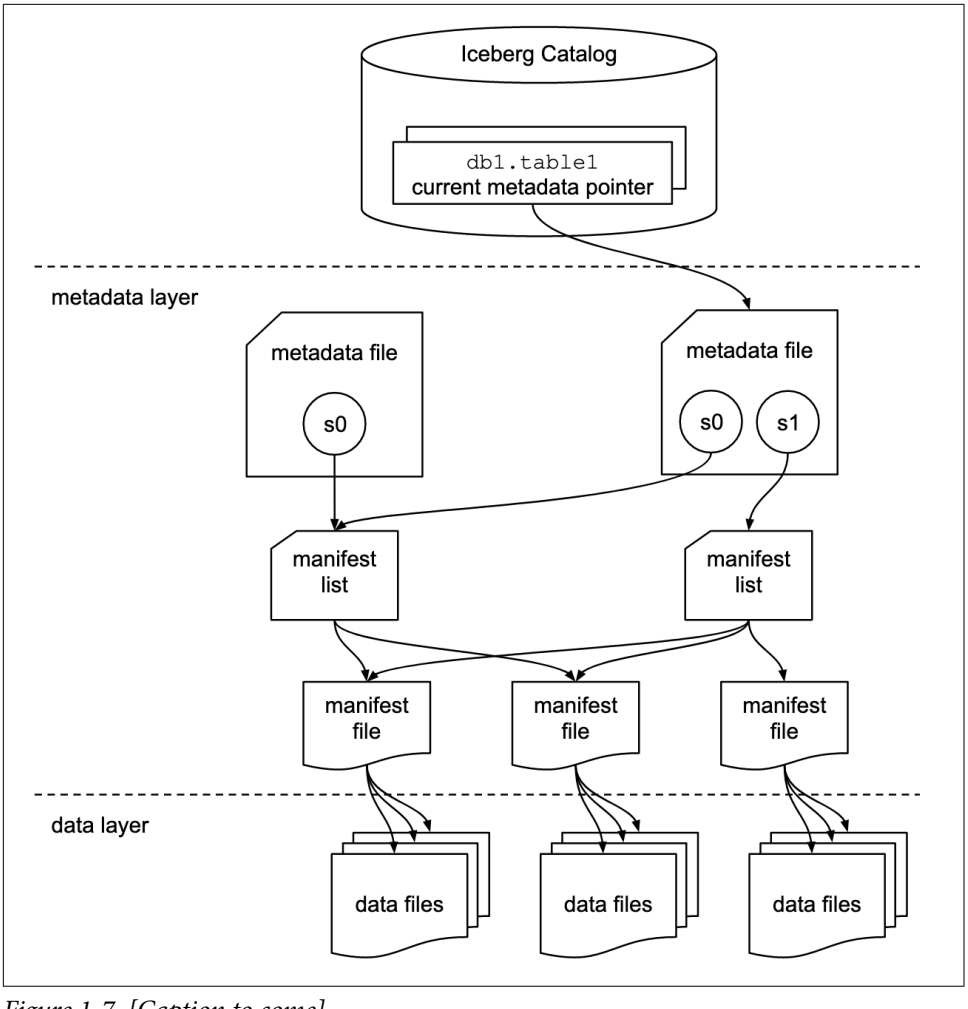


Figure 1-7. [Caption to come]

This metadata tree breaks down the metadata of the table into four components:

Manifest Files
: A list of data files, containing each data file's location/path and key metadata about those data files which allows for creating more efficient execution plans.

Manifest List
: A file that defines a single snapshot of the table as a list of manifest files along with stats on those manifests that allow for creating more efficient execution plans.

Metadata File
: A file that defines a table's structure including its schema, partitioning scheme, and a listing of snapshots.

Catalog
: This, like the Hive Metastore, tracks the table location, but instead of it containing a mapping of `table name -> set of directories`, it contains a mapping of `table name -> location of the table's most recent metadata file`. Several tools including a Hive metastore can be used as a catalog and we will be dedicating a whole chapter later on to this subject.

Each of these files will be covered in more depth in Chapter 3, *The Architecture of Apache Iceberg*.

Apache Iceberg Features

Apache Iceberg's unique architecture enables an ever growing number of features that go beyond just solving the challenges with Hive, but unlocking entirely new functionality for data lakes and data lakehouse workloads.

Below is a high level overview of key features of Apache Iceberg. We'll go into more depth on these features in later chapters.

ACID Transactions

Apache Iceberg uses techniques like optimistic concurrency control to enable ACID guarantees even when you have transactions being handled by multiple readers and writers. This way you can run transactions on your data lakehouse that either commit or fail and nothing in between. A pessimistic concurrency model to enable balancing locking considerations for a wider variety of use cases (e.g., ones in which there is a higher likelihood of update conflicts) is also coming in the future, at time of writing.

Concurrency guarantees are handled by the catalog as it is typically a mechanism that has built in ACID guarantees, This is what allows transactions on Iceberg tables to be atomic and provide correctness guarantees. If this didn't exist, two different systems could have conflicting updates resulting in data loss.

Partition Evolution

A big headache with data lakes prior to Apache Iceberg was dealing with the need to change the table's physical optimization. Too often, when your partitioning needs to change the only choice you have is to rewrite the entire table and at scale that can get very expensive. The alternative is to just live with the existing partitioning scheme and sacrifice the performance improvements a better partitioning scheme can provide.

With Apache Iceberg you can update how the table is partitioned at any time without the need to re-write the table and all of its data. Since partitioning has everything to do with the metadata, the operations needed to make this change to your table's structure are quick and cheap.

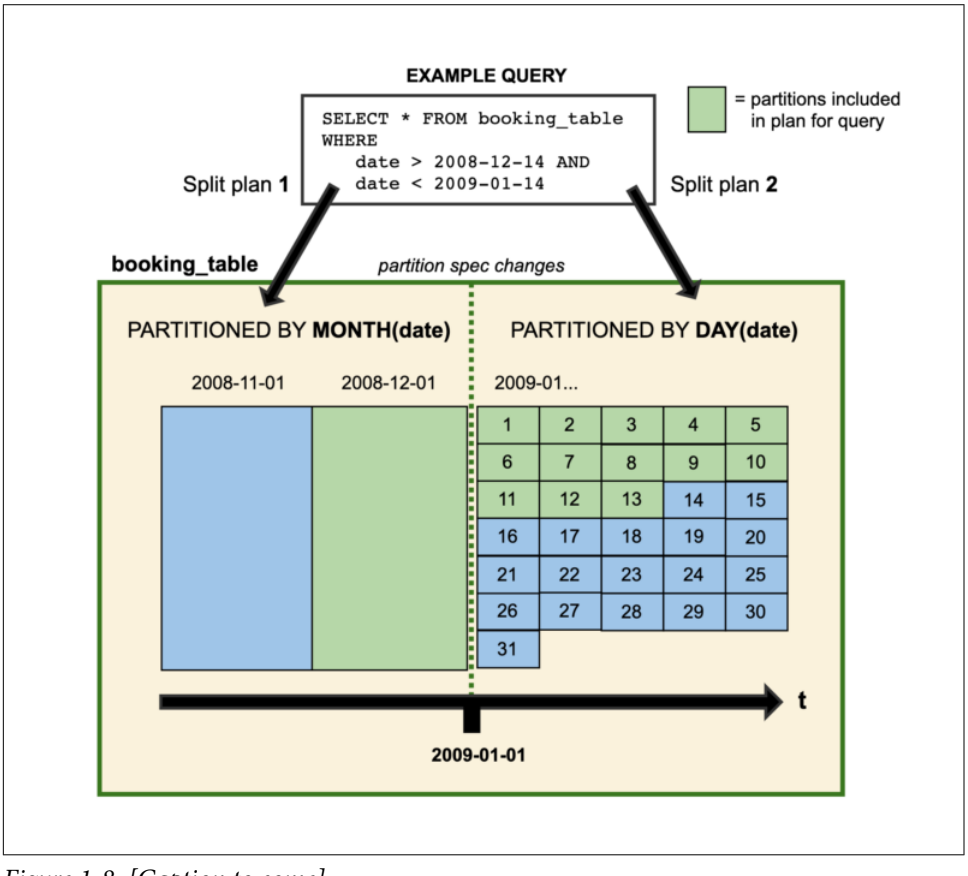


Figure 1-8. [Caption to come]

Hidden Partitioning

Sometimes users don't know how a table is physically partitioned, and frankly, they shouldn't have to care. Often a table is partitioned by some timestamp field and a user wants to query by that field (e.g., get average revenue by day for the last 90 days). However, to a user, the most intuitive way to do that is to include a filter of `event_timestamp >= DATE_SUB(CURRENT_DATE, INTERVAL 6 MONTH)`. However, this will result in a full table scan because the table is actually partitioned by separate fields called `event_year`, `event_month`, and `event_day` because partitioning on a timestamp results in tiny partitions since the values are at the second, millisecond, or lower granularity.

This problem is resolved with how Apache Iceberg's handles partitioning. Partitioning in Apache Iceberg comes in two parts, the column from which physical partitioning should be based on and an optional transform to that value including functions such as bucket, truncate, year, month, day and hour. The ability to apply a transform eliminates the need to create new columns just for partitioning. This results in more intuitive queries benefiting from partitioning as consumers will not need to add extra filter predicates to their queries on additional partitioning columns.

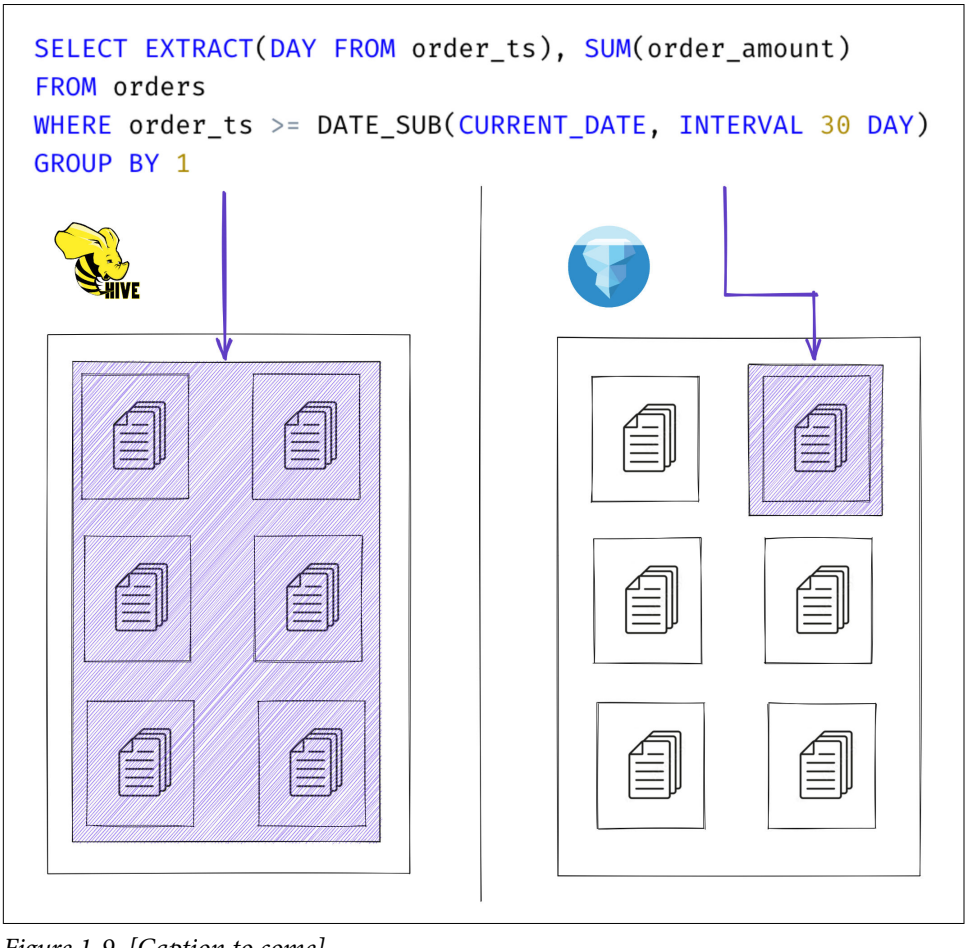


Figure 1-9. [Caption to come]

Row-Level Table Operations

You can optimize the table's row-level update patterns to take two forms: Copy-on-Write (COW) or Merge-on-Read. When using COW, for a change of any row in a given data file, the entire file is rewritten (with the row-level change made in the new file) even if a single record in it is updated. When using MOR, for any

row-level updates, only a new file that contains the changes to the affected row which is reconciled on reads is written. This gives flexibility to speed-up heavy update and delete workloads.

Time-Travel

Apache Iceberg provides immutable snapshots, so the information for the tables historical state is accessible allowing you to run queries on the state of the table at a given point in time in the past, or what's commonly known as time-travel. This can help you in situations such as doing end-of-quarter reporting without the need for duplicating the table's data to a separate location or for reproducing the output of a machine learning model as of a certain point in time.

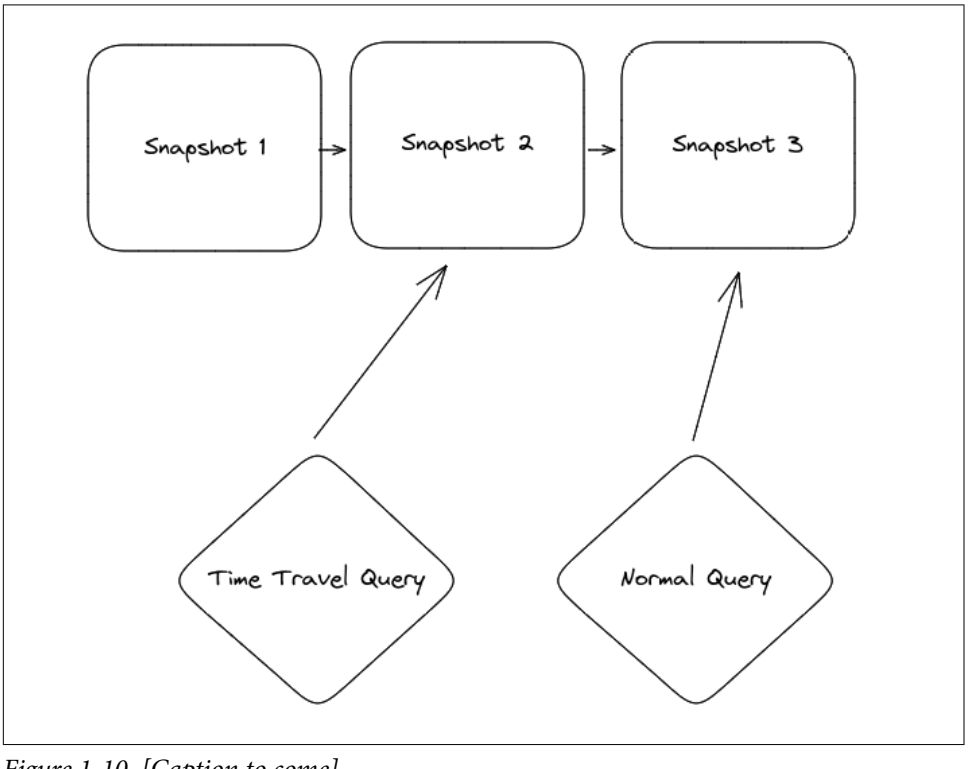


Figure 1-10. [Caption to come]

Version Rollback

Not only does Iceberg's Snapshot isolation allow you query the data as it is, but to also revert the tables current state to any of those previous snapshots. So undoing mistakes is as easy as rolling back.

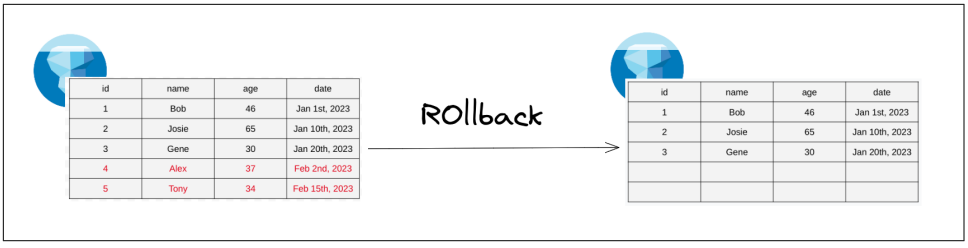


Figure 1-11. [Caption to come]

Schema Evolution

Tables change, whether that means adding/removing a column, renaming a column, or changing a column's data type. Regardless of how your table needs to evolve, Apache Iceberg gives you robust schema evolution features.

Conclusion

In this chapter we have learned that Apache Iceberg is a data lakehouse table format built to improve upon many of the areas that Hive tables lacked. By decoupling from relying on the physical structure of files along with its multi-level metadata tree, Iceberg is able to provide Hive transaction, ACID Guarantees, schema evolution, partition evolution and more to the data lakehouse. The Apache Iceberg project is able to do this by building a specification and supporting libraries that let existing data tools build support for the open table format.

In the next chapter, we'll go through a deep dive of Apache Iceberg's architecture that makes all of this possible.

References

[1] A. Thusoo et. al. Hive - A Warehousing Solution Over a Map-Reduce

Framework. Proc. VLDB Endow. 2009.

CHAPTER 2
The Architecture of Apache Iceberg

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

In this chapter, we'll go through the architecture and specification that enables Apache Iceberg to solve the Hive table format's problems and achieve the goals of the project by looking under the covers of an Iceberg table. We'll cover the different object types of an Apache Iceberg table and what each object provides and enables so you can understand what's happening under the hood when interacting with Apache Iceberg tables, as well as best architect your Apache Iceberg based lakehouse.

There are three different layers of an Apache Iceberg table: the catalog layer, the metadata layer, and the data layer. Figure 2-1 shows the different components that make up each layer of an Apache Iceberg table.

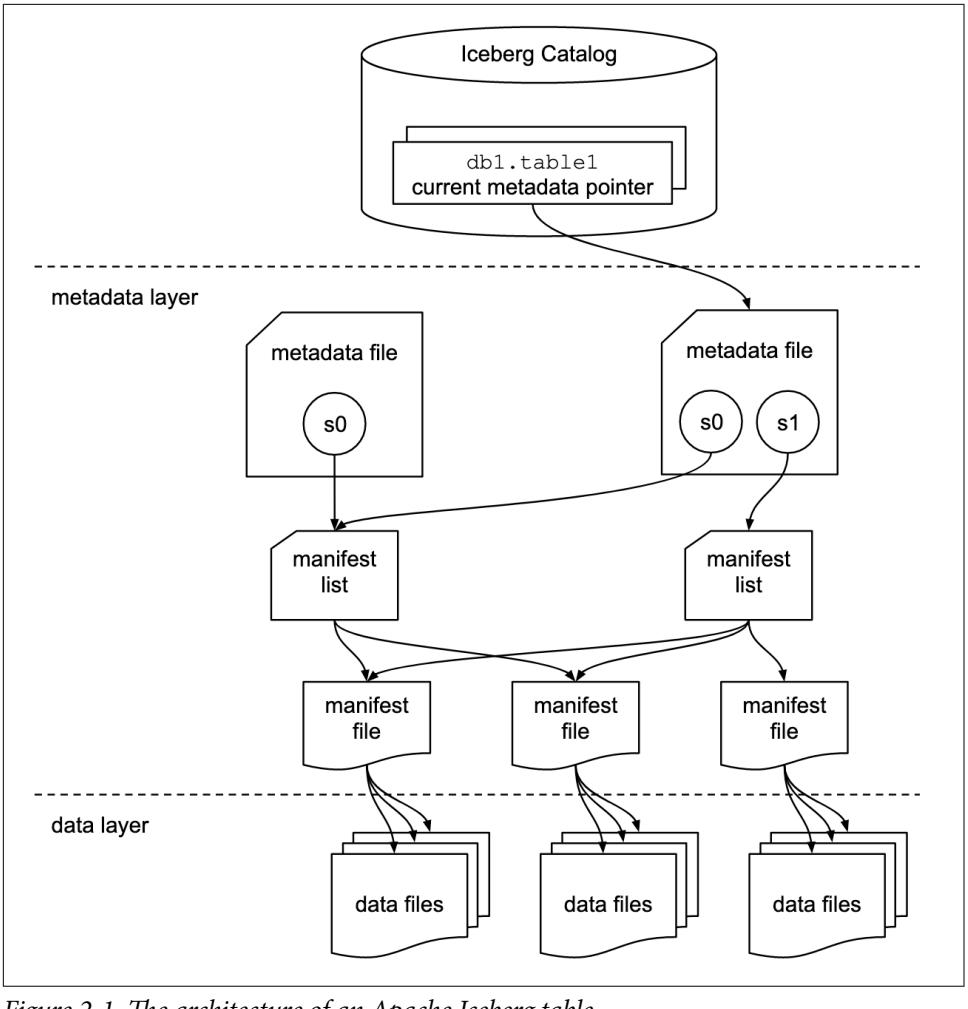


Figure 2-1. The architecture of an Apache Iceberg table

Now we'll go through each of these components in detail. Since it can be easier to understand concepts new to you by starting with a familiar one, we'll work from the bottom up, starting with the data layer.

Data Layer

The data layer of an Apache Iceberg table is what stores the actual data of the table and is primarily made up of the data files themselves, though also included are delete files and puffin files. The data layer is what provides the querying user with the data needed to provide a result. While there are some exceptions where structures in the metadata layer can provide a result (e.g., get me the max value for column X), most

commonly the data layer is involved in providing results to user queries. The data layer makes up the leaves of the tree structure of an Apache Iceberg table.

In real-world usage, the data layer is backed by a distributed file system (e.g., HDFS) or something that looks like a distributed file system, like object storage (e.g., Amazon S3, Azure Storage, Google Cloud Storage). This enables data lakehouse architectures to be built on and benefit from these extremely scalable and low-cost storage systems.

Data Files

Data files store the data itself. Apache Iceberg is file-format agnostic and currently supports Apache Parquet, Apache ORC, and Apache Avro. This is important because:

- Many organizations have data in multiple file formats because different groups are able to, or were able to, choose which file format they wanted to use on their own.
- It provides the flexibility to choose different formats depending on what is best suited for a given workload. For example, Parquet might be used for a table used for large-scale OLAP analytics, whereas Avro might be used for a table used for more low-latency streaming analytics.
- It future proofs organizations' choice of file format. If a new file format comes out in the future which is better suited for a set of workloads, that file format could be used in an Apache Iceberg table.

While Apache Iceberg is file-format agnostic, in the real-world the file format most commonly used is Apache Parquet. Parquet is most common because its columnar structure provides large performance gains for OLAP workloads over row-based file formats and it's become the de facto standard in the industry, meaning basically every engine and tool supports Parquet. Its columnar structure lays the foundation for performance features like the ability for a single file to be split multiple ways for increased parallelism, statistics for each of these split points, and increased compression which provides lower disk storage and higher read throughput.

In Figure 2-2, you can see how a given Parquet file has a set of rows ("Row group 0" in the figure) that are then broken down so all rows' values for a given column are stored together ("Column a" in the figure). All rows' values for a given column are further broken down into subsets of rows' values for this column which are called pages ("Page 0 in the figure). Each of these levels can be read independently by engines and tools, and therefore each can be read in parallel by a given engine or tool. In addition, Parquet stores statistics (e.g., minimum and maximum values for a given column for a given row group) that enables engines and tools to decide whether it needs to read all of the data or if it can prune row groups that don't fit the query.

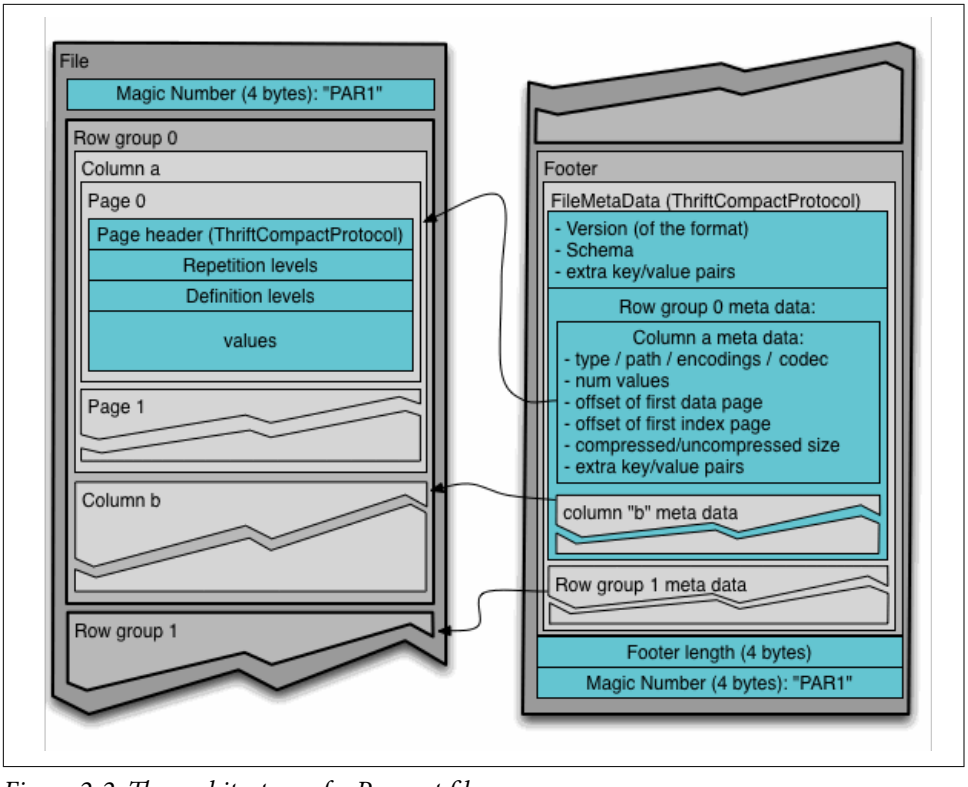


Figure 2-2. The architecture of a Parquet file

Delete Files

Delete files track which records in the dataset have been deleted. Because data lake storage is immutable, you can't update rows in a file in place - you need to write a new file. This new file can either be a copy of the old file with the changes reflected in a new copy of it (called copy-on-write) or it can be a new file that only has the changes written, which then engines reading the data coalesce (called merge-on-read). Delete files enable the merge-on-read (MOR) strategy for performing updates and deletes to Iceberg tables. I.e., delete files only apply to MOR tables - we'll go into more depth as to why in chapter 4 "Optimizing the Performance of Tables". Note that delete files are only supported in the Iceberg v2 format, which even at time of writing is widely adopted by almost every tool supporting Iceberg, but something to be aware of.

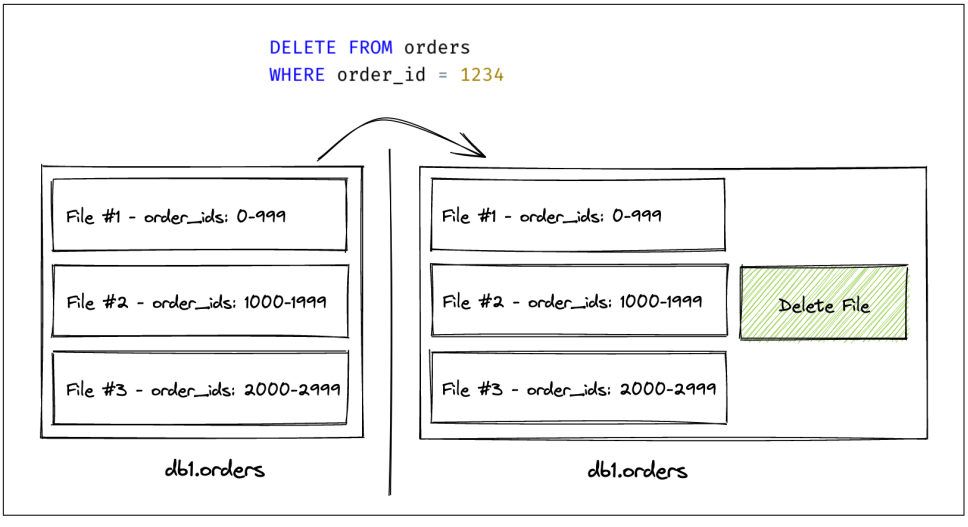


Figure 2-3. A diagram showing a MOR table before and after a DELETE is run on it

There are two ways to identify a given row that needs to be removed from the logical dataset when an engine reads the dataset: either identify the row by its exact position in the dataset or identify the row by the values of one or more fields of the row. Therefore, there are two types of delete files. The former is addressed by what are called positional delete files and and the latter is addressed by what are called equality delete files.

These two approaches have different pros and cons and therefore different situations where one is preferred over the other. We'll go into more depth into the considerations and in what situations is better to use which in Chapter 4 when copy-on-write vs merge-on-read is covered, but below is a high level description.

Positional Delete Files

Positional delete files denote what rows have been logically deleted, and therefore that the engine reading the data needs to remove from its representation of the table when it uses the table, by identifying the exact position in the table where the row is located. It does this by specifying the file path of the specific file that contains the row and the row number within that file.

Figure 2-4 shows deleting the row with the `order_id` of `1234`. Assuming the data in the file is sorted by `order_id` ascending, this row is in file #2 and is row #234 (note that the row referencing is zero-indexed, so row #0 in file #2 is `order_id = 1000`, and therefore row #234 in file #2 is `order_id = 1234`)

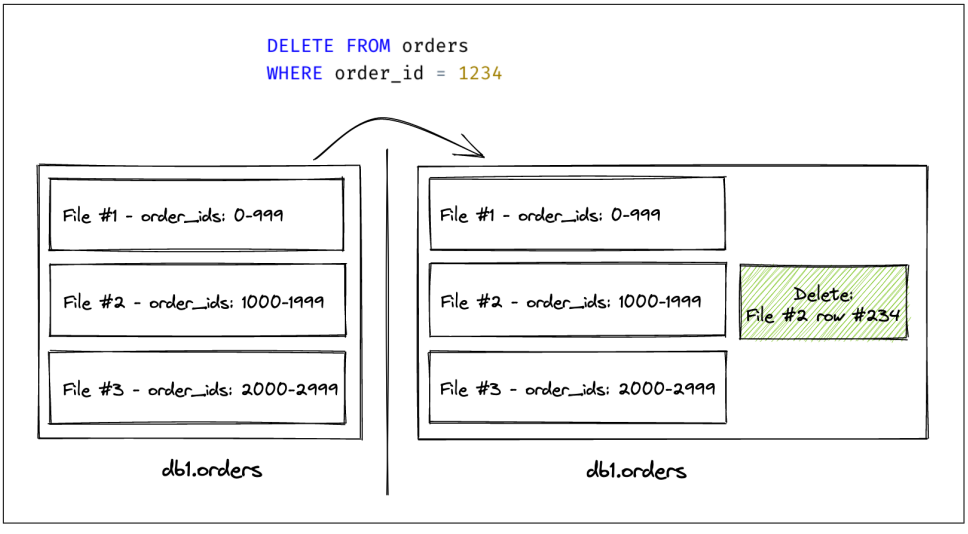


Figure 2-4. A diagram showing a MOR table configured for positional deletes before and after a DELETE is run on it

Equality Delete Files

Equality delete files denote what rows have been logically deleted, and therefore that the engine reading the data needs to remove from its representation of the table when it uses the table, by identifying the row by the values of one or more of the fields for the row. This is best done when there is a unique identifier for each row in the table (aka primary key) so a single field's value can uniquely identify a row (e.g., "delete the row where the row has a value for `order_id` of 1234"). However, multiple rows can also be deleted via this method too (e.g., "delete all rows where `interaction_customer_id` = 5678").

Figure 2-5 shows deleting the row with the `order_id` of `1234` using an equality delete file. An engine writes a delete file that says "delete any rows where order_id = 1234" which any engine reading it then adheres to. Note that in contrast to positional delete files, there is no reference to where these rows are located within the table.

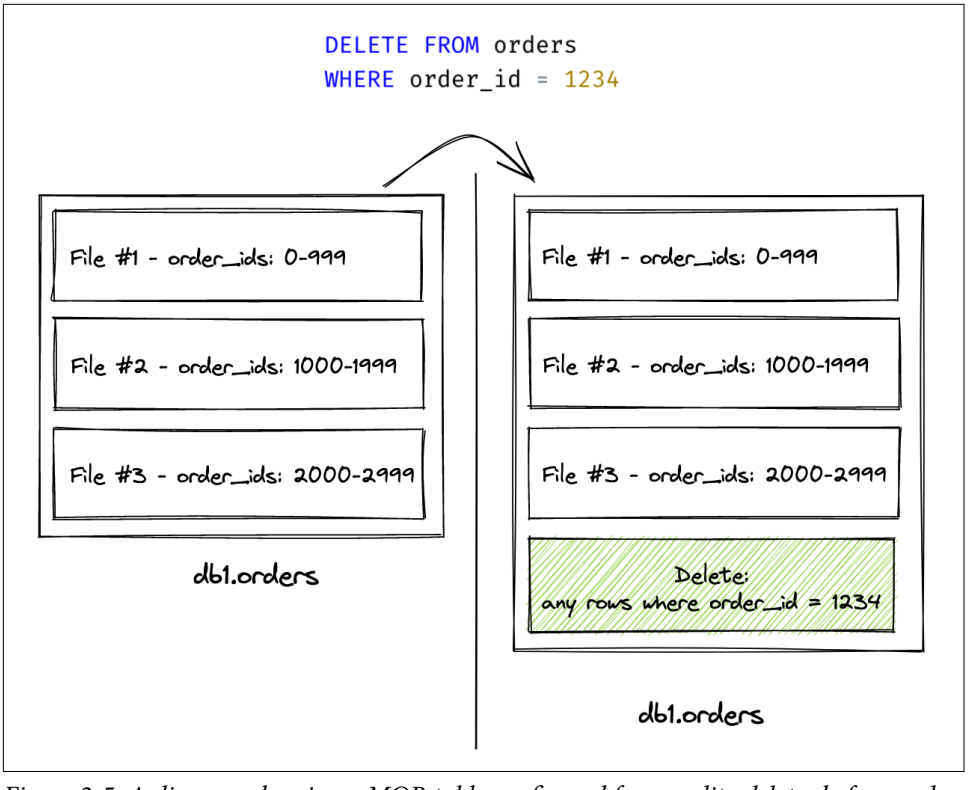


Figure 2-5. A diagram showing a MOR table configured for equality deletes before and after a DELETE is run on it

Puffin Files

While there are structures in data files and delete files to enhance the performance of interacting with the data in an Iceberg table, sometimes you need more advanced structures to enhance the performance of specific types of queries.

For example, what if you wanted to know how many unique people placed an order with you in the past 30 days? The statistics in the data files (nor the metadata files, as we'll see shortly) cover this kind of use case. Certainly you could use those statistics to improve performance some (e.g., pruning out only the data for the last 30 days), but you would still have to read every order in those 30 days and do aggregations in the engine, which can take too long depending on factors like the size of the data, resources allocated to the engine, and cardinality of the fields.

Enter the puffin file format. This file format stores statistics and indexes about the data in the table that improve the performance of an even broader range of queries than the statistics stored in the data files and metadata files.

The file contains sets of arbitrary byte sequences called 'blobs', along with the associated metadata required to analyze these blobs.

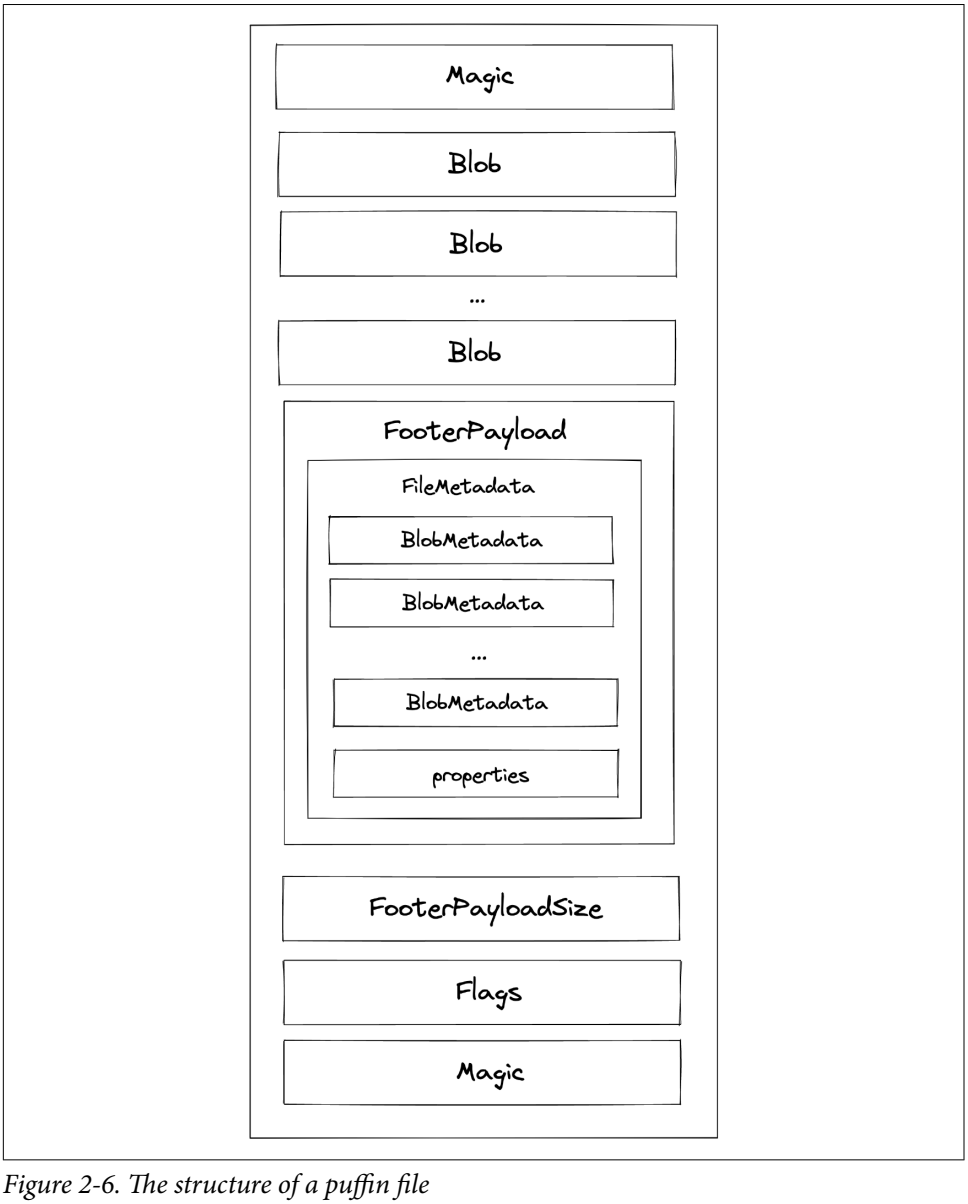


Figure 2-6. The structure of a puffin file

While this structure enables statistics and index structures of any type (e.g., bloom filters), currently the only type supported is the Theta sketch from the Apache DataSketches library. This structure enables computing the *approximate* number of

distinct values of a column for a given set of rows enabling the computation to be much faster and use much fewer resources, often orders of magnitude fewer.

Metadata Layer

The metadata layer is an integral part of an Iceberg table's architecture and contains all of the metadata files for an Iceberg table. It's a tree structure that tracks the data files and metadata about them as well as the operations that made them. This tree structure is made up of three file types, all of which are stored in data lake storage: manifest files, manifest lists, and metadata files. The metadata layer is essential for efficiently managing large datasets and enabling core features like time travel and schema evolution.

Manifest Files

Manifest files keep track of files in the data layer (i.e., data files, delete files, and puffin files) as well as additional details and statistics about each file. As mentioned in Chapter 1, the primary difference that allows Iceberg to address the problems of the Hive table format is tracking what data is in a table at the file level — manifest files are the files that do this tracking at the leaf-level of the metadata tree.

Note that while manifest files track data files as well as delete files, a separate set of manifest files are used for each of them (i.e., a single manifest file will contain only data files or delete files), though the manifest file schemas are identical.

Each manifest file keeps track of a subset of the data files. They contain information such as details about partition membership, record count, and lower and upper bounds of columns, that is used to improve efficiency and performance while reading the data from these data files. While some of these statistics are also stored in the data files themselves, a single manifest file stores these statistics for multiple data files, meaning the pruning done from the stats in a single manifest file greatly reduces the need to open many data files which can really hurt performance (even if just opening the footer of many data files, this still can take a long time). This process will be covered in depth in Chapter 3, Lifecycle of Read and Write Queries. These statistics are written by the engine/tool for each manifest's subset of data files during write operation.

Because these statistics are written in smaller batches by each engine for their subset of the data files written, it is much more lightweight to write these statistics compared to the Hive table format, where statistics are collected and stored as part of a long and expensive read job where the engine has to read an entire partition or entire table, compute the statistics for all of that data, then write the stats for that partition/table.

In practice, this means that the statistics collection jobs when using the Hive table format are not re-run very often (if at all), resulting in poorer query performance since engines do not have the information necessary to make informed and better decisions on how to execute a given query. As a result, Iceberg tables are much more likely to have up-to-date and accurate statistics, allowing engines to make better decisions when processing them, resulting in higher job performance.

Here is an example of the full contents of a manifest file:

```
{
        "status": 1,
        "snapshot_id":
        {
        "long": 8619686881304977000
        },
        "sequence_number": null,
        "file_sequence_number": null,
        "data_file":
        {
        "content": 0,
        "file_path": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders/data/created_ts_day=2023-03-21/00000-11-b5d3ab5d-1522-43e2-
bc61-0560f055bfa1-00001.parquet",
        "file_format": "PARQUET",
        "partition":
        {
                "created_ts_day":
                {
                "int": 19437
                }
        },
        "record_count": 1,
        "file_size_in_bytes": 1574,
        "column_sizes":
        {
                "array":
                [
                {
                        "key": 1,
                        "value": 48
                },
                {
                        "key": 2,
                        "value": 46
                },
                {
                        "key": 3,
                        "value": 47
                },
                {
                        "key": 4,
                        "value": 53
```

```
        },
        {
                "key": 5,
                "value": 57
        }
        ]
},
"value_counts":
{
        "array":
        [
        {
                "key": 1,
                "value": 1
        },
        {
                "key": 2,
                "value": 1
        },
        {
                "key": 3,
                "value": 1
        },
        {
                "key": 4,
                "value": 1
        },
        {
                "key": 5,
                "value": 1
        }
        ]
},
"null_value_counts":
{
        "array":
        [
        {
                "key": 1,
                "value": 0
        },
        {
                "key": 2,
                "value": 0
        },
        {
                "key": 3,
                "value": 0
        },
        {
                "key": 4,
                "value": 0
```

```
        },
        {
                "key": 5,
                "value": 0
        }
        ]
},
"nan_value_counts":
{
        "array":
        []
},
"lower_bounds":
{
        "array":
        [
        {
                "key": 1,
                "value": "å\u0001\u0000\u0000\u0000\u0000\u0000"
        },
        {
                "key": 2,
                "value": "ø\u0000\u0000\u0000\u0000\u0000\u0000\u0000"
        },
        {
                "key": 3,
                "value": "*<"
        },
        {
                "key": 4,
                "value": "placed"
        },
        {
                "key": 5,
                "value": "Èøje÷\u0005\u0000"
        }
        ]
},
"upper_bounds":
{
        "array":
        [
        {
                "key": 1,
                "value": "å\u0001\u0000\u0000\u0000\u0000\u0000"
        },
        {
                "key": 2,
                "value": "ø\u0000\u0000\u0000\u0000\u0000\u0000\u0000"
        },
        {
                "key": 3,
```

```
                        "value": "*<"
                },
                {
                        "key": 4,
                        "value": "placed"
                },
                {
                        "key": 5,
                        "value": "Èøje÷\u0005\u0000"
                }
                ]
        },
        "key_metadata": null,
        "split_offsets":
        {
                "array":
                [
                4
                ]
        },
        "equality_ids": null,
        "sort_order_id":
        {
                "int": 0
        }
        }
}
```


Manifest Lists

A manifest list is a snapshot of an Iceberg table that contains a list of all the manifest files, including the location, the partitions it belongs to and the upper and lower bounds for partition columns for the data file it tracks.

A manifest list contains an array of structs, each struct keeping track of a single manifest file. The struct's schema is detailed in Table 2-1, which has been adapted from the public Iceberg docs. Also note this is for v2 Iceberg tables.

Table 2-1. Schema of an Iceberg Manifest File

Always Present?	Field name	Data Type	Description
Yes	`manifest_path`	`string`	Location of the manifest file
Yes	`manifest_length`	`long`	Length of the manifest file in bytes
Yes	`partition_spec_id`	`int`	ID of a partition spec used to write the manifest; refers to an entry listed in `partition-specs` in the table's metadata file
Yes	`content`	`int with meaning: 0: data, 1: deletes`	The type of files tracked by the manifest, either data or delete files

Always Present?	Field name	Data Type	Description
Yes	`sequence_number`	`long`	The sequence number when the manifest was added to the table
Yes	`min_sequence_number`	`long`	The minimum data sequence number of all live data or delete files in the manifest
Yes	`added_snapshot_id`	`long`	ID of the snapshot where the manifest file was added
Yes	`added_files_count`	`int`	Number of entries in the manifest file that have `ADDED` (1) as the value for the `status` field
Yes	`existing_files_count`	`int`	Number of entries in the manifest file that have `EXISTING` (0) as the value for the `status` field
Yes	`deleted_files_count`	`int`	Number of entries in the manifest file that have `DELETED` (2) as the value for the `status` field
Yes	`added_rows_count`	`long`	Sum of the number of rows in all of files in the manifest that have `ADDED` as the value for the `status` field
Yes	`existing_rows_count`	`long`	Sum of the number of rows in all of files in the manifest that have `EXISTING` as the value for the `status` field
Yes	`deleted_rows_count`	`long`	Sum of the number of rows in all of files in the manifest that have `DELETED` as the value for the `status` field
No	`partitions`	`array<field_summary>` (see below)	A list of field summaries for each partition field in the spec. Each field in the list corresponds to a field in the manifest file's partition spec.
No	`key_metadata`	`binary`	Implementation-specific key metadata for encryption

As referenced in the second to the last row above, `field_summary` is a struct with the schema shown in Table 2-2.

Table 2-2. Schema of `field_summary`

Always Present?	Field name	Type	Description
Yes	`contains_null`	`boolean`	Whether the manifest contains at least one partition with a null value for the field
No	`contains_nan`	`boolean`	Whether the manifest contains at least one partition with a NaN value for the field
No	`lower_bound`	`bytes`	Lower bound for the non-null, non-NaN values in the partition field, or null if all values are null or NaN. The value is serialized to bytes
No	`upper_bound`	`bytes`	Upper bound for the non-null, non-NaN values in the partition field, or null if all values are null or NaN. The value is serialized to bytes

Here is an example of the full contents of a manifest list:

```
{
        "manifest_path": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders/metadata/a5969634-3cdb-4d66-9630-7849091c8556-m0.avro",
        "manifest_length": 7283,
        "partition_spec_id": 0,
        "content": 0,
        "sequence_number": 2,
        "min_sequence_number": 2,
        "added_snapshot_id": 8619686881304977000,
        "added_data_files_count": 2,
        "existing_data_files_count": 0,
        "deleted_data_files_count": 0,
        "added_rows_count": 2,
        "existing_rows_count": 0,
        "deleted_rows_count": 0,
        "partitions":
        {
        "array":
        [
                {
                "contains_null": false,
                "contains_nan":
                {
                        "boolean": false
                },
                "lower_bound":
                {
                        "bytes": "íK\u0000\u0000"
                },
                "upper_bound":
                {
                        "bytes": "íK\u0000\u0000"
                }
                }
        ]
        }
}
{
        "manifest_path": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders/metadata/624f7d12-62e7-4e38-8583-9e40bdc63335-m0.avro",
        "manifest_length": 7298,
        "partition_spec_id": 0,
        "content": 0,
        "sequence_number": 1,
        "min_sequence_number": 1,
        "added_snapshot_id": 8153845014111638000,
        "added_data_files_count": 2,
        "existing_data_files_count": 0,
        "deleted_data_files_count": 0,
        "added_rows_count": 2,
        "existing_rows_count": 0,
```

```
        "deleted_rows_count": 0,
        "partitions":
        {
        "array":
        [
                {
                "contains_null": false,
                "contains_nan":
                {
                        "boolean": false
                },
                "lower_bound":
                {
                        "bytes": "ëK\u0000\u0000"
                },
                "upper_bound":
                {
                        "bytes": "ìK\u0000\u0000"
                }
                }
        ]
        }
}
```

Metadata Files

Manifest lists are tracked by metadata files. Another aptly named file, metadata files store metadata about an Iceberg table at a certain point in time. This includes information about the table's schema, partition information, snapshots, and which snapshot is the current one.

Each time a change is made to an Iceberg table, a new metadata file is created and is registered as the latest version of the metadata file atomically via the catalog, which we'll cover in the next section. This ensures a linear history of the table commits and helps during scenarios such as concurrent writes, i.e., multiple engines writing data simultaneously. Also, this way, during read operations, engines will always see the latest version of the table.

The metadata file's schema is detailed in Table 2-3, which has been adapted from the public Iceberg docs.

Table 2-3. Metadata file schema

Always present?	Field name	Data Type	Description
Yes	`format-version`	`integer`	An integer version number for the format. Currently, this can be 1 or 2 based on the spec. Implementations must throw an exception if a table's version is higher than the supported version.

Always present?	Field name	Data Type	Description
Yes	`table-uuid`	`string`	A UUID that identifies the table, generated when the table is created. Implementations must throw an exception if a table's UUID does not match the expected UUID after refreshing metadata.
Yes	`location`	`string`	The table's base location. This is used by writers to determine where to store data files, manifest files, and table metadata files.
Yes	`last-sequence-number`	`64-bit signed integer`	The table's highest assigned sequence number, a monotonically increasing long that tracks the order of snapshots in a table.
Yes	`last-updated-ms`	`64-bit signed integer`	Timestamp in milliseconds from the unix epoch when the table was last updated. Each table metadata file should update this field just before writing.
Yes	`last-column-id`	`integer`	The highest assigned column ID for the table. This is used to ensure columns are always assigned an unused ID when evolving schemas.
Yes	`schemas`	`array`	A list of schemas, stored as objects with `schema-id`.
Yes	`current-schema-id`	`integer`	ID of the table's current schema.
Yes	`partition-specs`	`array`	A list of partition specs, stored as full partition spec objects.
Yes	`default-spec-id`	`integer`	ID of the "current" spec that writers should use by default.
Yes	`last-partition-id`	`integer`	The highest assigned partition field ID across all partition specs for the table. This is used to ensure partition fields are always assigned an unused ID when evolving specs.
No	`properties`	`map`	A string to string map of table properties. This is used to control settings that affect reading and writing and is not intended to be used for arbitrary metadata. For example, `commit.retry.num-retries` is used to control the number of commit retries.
No	`current-snapshot-id`	`64-bit signed integer`	ID of the current table snapshot; must be the same as the current ID of the `main` branch in `refs`.
No	`snapshots`	`array`	A list of valid snapshots. Valid snapshots are snapshots for which all data files exist in the file system. A data file must not be deleted from the file system until the last snapshot in which it was listed is garbage collected.
No	`snapshot-log`	`array`	A list of timestamp and snapshot ID pairs that encodes changes to the current snapshot for the table. Each time the `current-snapshot-id` is changed, a new entry should be added with the last-updated-ms and the new current-snapshot-id. When snapshots are expired from the list of valid snapshots, all entries before a snapshot that has expired should be removed.
No	`metadata-log`	`array`	A list of timestamp and metadata file location pairs that encodes changes to the previous metadata files for the table. Each time a new metadata file is created, a new entry of the previous metadata file location should be added to the list. Tables can be configured to remove oldest metadata log entries and keep a fixed-size log of the most recent entries after a commit.
Yes	`sort-orders`	`array`	A list of sort orders, stored as full sort order objects.

Always present?	Field name	Data Type	Description
Yes	`default-sort-order-id`	`integer`	Default sort order id of the table. Note that this could be used by writers, but is not used when reading because reads use the specs stored in manifest files.
No	`refs`	`map`	A map of snapshot references. The map keys are the unique snapshot reference names in the table, and the map values are snapshot reference objects. There is always a `main` branch reference pointing to the `current-snapshot-id` even if the `refs` map is null.
No	`statistics`	`array`	A list (optional) of table statistics.

Here is an example of the full contents of a metadata file:

```
{
        "format-version": 2,
        "table-uuid": "b39fb9ba-18ac-4c6d-8338-593c62265595",
        "location": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders",
        "last-sequence-number": 2,
        "last-updated-ms": 1679439331868,
        "last-column-id": 5,
        "current-schema-id": 0,
        "schemas":
        [
        {
                "type": "struct",
                "schema-id": 0,
                "fields":
                [
                {
                        "id": 1,
                        "name": "order_id",
                        "required": false,
                        "type": "long"
                },
                {
                        "id": 2,
                        "name": "customer_id",
                        "required": false,
                        "type": "long"
                },
                {
                        "id": 3,
                        "name": "order_amount",
                        "required": false,
                        "type": "decimal(10, 2)"
                },
                {
                        "id": 4,
                        "name": "status",
                        "required": false,
```

```
                "type": "string"
        },
        {
                "id": 5,
                "name": "created_ts",
                "required": false,
                "type": "timestamptz"
        }
        ]
}
],
"default-spec-id": 0,
"partition-specs":
[
{
        "spec-id": 0,
        "fields":
        [
        {
                "name": "created_ts_day",
                "transform": "day",
                "source-id": 5,
                "field-id": 1000
        }
        ]
}
],
"last-partition-id": 1000,
"default-sort-order-id": 0,
"sort-orders":
[
{
        "order-id": 0,
        "fields":
        []
}
],
"properties":
{
"owner": "ec2-user"
},
"current-snapshot-id": 8619686881304977663,
"refs":
{
"main":
{
        "snapshot-id": 8619686881304977663,
        "type": "branch"
}
},
"snapshots":
[
```

```
        {
                "sequence-number": 1,
                "snapshot-id": 8153845014111637777,
                "timestamp-ms": 1679439324767,
                "summary":
                {
                "operation": "append",
                "spark.app.id": "application_1679437452433_0003",
                "added-data-files": "2",
                "added-records": "2",
                "added-files-size": "3162",
                "changed-partition-count": "2",
                "total-records": "2",
                "total-files-size": "3162",
                "total-data-files": "2",
                "total-delete-files": "0",
                "total-position-deletes": "0",
                "total-equality-deletes": "0"
                },
                "manifest-list": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders/metadata/
snap-8153845014111637777-1-624f7d12-62e7-4e38-8583-9e40bdc63335.avro",
                "schema-id": 0
        },
        {
                "sequence-number": 2,
                "snapshot-id": 8619686881304977663,
                "parent-snapshot-id": 8153845014111637777,
                "timestamp-ms": 1679439331868,
                "summary":
                {
                "operation": "append",
                "spark.app.id": "application_1679437452433_0003",
                "added-data-files": "2",
                "added-records": "2",
                "added-files-size": "3148",
                "changed-partition-count": "1",
                "total-records": "4",
                "total-files-size": "6310",
                "total-data-files": "4",
                "total-delete-files": "0",
                "total-position-deletes": "0",
                "total-equality-deletes": "0"
                },
                "manifest-list": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders/metadata/snap-8619686881304977663-1-
a5969634-3cdb-4d66-9630-7849091c8556.avro",
                "schema-id": 0
        }
        ],
        "statistics":
        [],
```

```
        "snapshot-log":
        [
        {
                "timestamp-ms": 1679439324767,
                "snapshot-id": 8153845014111637777
        },
        {
                "timestamp-ms": 1679439331868,
                "snapshot-id": 8619686881304977663
        }
        ],
        "metadata-log":
        [
        {
                "timestamp-ms": 1679439283526,
                "metadata-file": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders/metadata/00000-c2f323ef-6571-4b86-
b6bd-97b6527072b1.metadata.json"
        },
        {
                "timestamp-ms": 1679439324767,
                "metadata-file": "s3://jason-dremio-product-us-west-2/iceberg-book/
iceberg_book.db/orders/metadata/00001-240007c3-5737-4898-
ac08-91bbf513a290.metadata.json"
        }
        ]
}
```

Catalog

Anyone reading from a table (let alone 10s, 100s, or 1,000s) needs to know where to go first — somewhere they can go to find out where to read/write data for a given table. The first step for anyone looking to interact with the table is to find the location of the metadata file that is the current metadata pointer.

This central place where you go to find the current location of the current metadata pointer is the Iceberg catalog.

The primary requirement for an Iceberg catalog is that it must support atomic operations for updating the current metadata pointer. This support for atomic operations is required so all readers and writers all see the same state of the table at a given point in time.

Within the catalog, there is a reference or pointer for each table to that table's current metadata file. For example, in Figure 2-1 there are 2 metadata files. The value for the table's current metadata pointer in the catalog is the location of the metadata file.

Because the only requirements for an Iceberg catalog is that it needs to store the current metadata pointer and provide atomic guarantees, there are many different

backends that can serve as an Iceberg catalog. Though different catalogs store the current metadata pointer differently. A few examples:

- With S3 as the catalog, there's a file called `version-hint.text` in the table's metadata folder whose contents is the version number of the current metadata file. Note anytime you use a distributed file system (or something that looks like one) to store the current metadata pointer, the catalog used is actually called the "hadoop" catalog.
- With Hive metastore as the catalog, the table entry in the Hive metastore has a table property called `location` which stores the location of the current metadata file.
- With Nessie as the catalog, the table entry in Nessie has a table property called `metadataLocation` that stores the location of the current metadata file for the table.

In the above examples of the manifest files, manifest lists, and metadata files, we were using the AWS glue catalog. Leveraging Iceberg metadata about the table, we can see what the catalog is saying the current metadata file is. Running the following query gives us the details about the current state of the table, most notably the current metadata file location:

```
SELECT *
FROM my_catalog.iceberg_book.orders.metadata_log_entries
ORDER BY timestamp DESC
LIMIT 1
```

Timestamp	Metadata File	Latest Snapshot ID	Latest Schema ID	Latest Sequence Number
2023-03-21 22:55:31.868	s3://jason-dremio-product-us-west-2/iceberg-book/iceberg_book.db/orders/metadata/00002-509f0747-4dc4-4965-b354-ce5fb747c2f5.metadata.json	8619686881304977663	0	2

So, if we want to go read data from this table that's using the glue catalog, we know we then need to go retrieve the metadata file at the path `s3://jason-dremio-product-us-west-2/iceberg-book/iceberg_book.db/orders/metadata/00002-509f0747-4dc4-4965-b354-ce5fb747c2f5.metadata.json`.

Conclusion

In this chapter we've gone through the architecture and format of Apache Iceberg tables that enable them to solve the Hive table format's problems and achieve capabilities such as ACID transactions on the data lake. The file types and structures we've gone through are leveraged by engines and tools to read and write data efficiently, as well as achieve more advanced capabilities such as time travel and schema evolution.

In the next chapter, we'll go through the lifecycle of queries run in these engines and tools to see exactly how these file types and structures are leveraged.

CHAPTER 3
Lifecycle of Write and Read Queries

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

The Apache Iceberg table format provides high-performance queries during reads and writes, allowing running OLAP analytical workloads directly on the data lake. What facilitates this superior performance is the way the various components of the Iceberg table format are designed. It is, therefore, critical to understand the structure of these components so query engines can effectively use them for faster query planning and execution. We discussed these architectural components in detail in Chapter 2. At a high level, all these components can be segregated into three different layers, as presented in Figure 3-1. Let us quickly understand how a query engine interacts with these components for reads and writes.

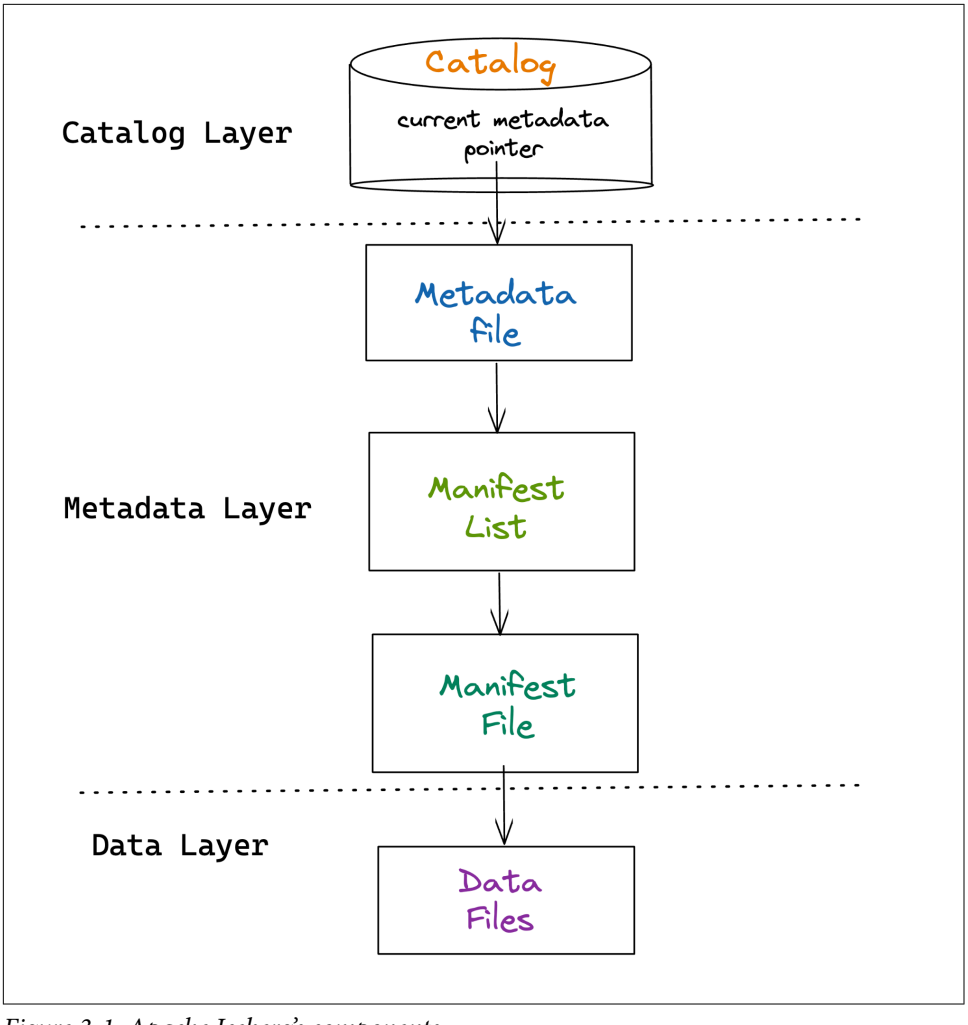


Figure 3-1. Apache Iceberg's components

Catalog Layer
: As we learned in Chapter 3, a catalog holds the references to the current metadata pointer, i.e., the latest metadata file. Irrespective of whether you are doing a read or write operation, the catalog is the first component that a query engine interacts with. In the case of reads, the engine reaches out to the catalog to learn about the current state of the table, and for writes, the catalog is used to adhere to the schema defined and to know about the table's partitioning scheme.

Metadata Layer
: The metadata layer in Apache Iceberg consists of three different components - metadata file, manifest list, and manifest file. Each time a query engine writes

something to an Iceberg table, a new metadata file is created atomically and is defined as the latest version of the metadata file. This ensures a linear history of the table commits and helps during scenarios such as concurrent writes, i.e., multiple engines writing data simultaneously. Also, this way, during read operations, engines will always see the latest version of the table. Query engines interact with the manifest lists to get information about partition specification summary that help them to skip the non-required manifest files for faster performance. Finally information from the manifest files such as upper and lower bounds for a specific column, null value counts, partition-specific data, etc. are used by the engine for file pruning.

Data Layer
: Query engines filter through the metadata files described above to read the data files required by a particular query efficiently. On the write side, data files get written on the file storage, and the related metadata files are created and updated accordingly.

In the following sections, you will learn the lifecycle of the various write and read operations in Apache Iceberg and how each operation interacts with the components described above to bring the best query performance. Note that, throughout this chapter, we will present queries using Spark SQL and Dremio Sonar as the compute engines.

Write Queries in Apache Iceberg

The write process in Apache Iceberg involves a series of steps that enable query engines to efficiently insert and update data. When a write query is initiated, it is sent to the engine for parsing. The catalog is then consulted to ensure consistency and integrity in data and to write data as per the defined partition strategies. The metadata and data files are then written based on the query. Finally, the catalog file is updated to reflect the latest metadata, enabling subsequent read operations to access the most up-to-date version of the data.

Create Table

Let us first create an Iceberg table and understand the process underneath. Here is an example query to create a table called `orders` with four columns. While the syntax to do so in Spark and Dremio Sonar are very similar, they're split out separately in the rest of this chapter so the code can be run directly in each system so you can follow along. These code samples are also provided in a public Github repo at xxx. This table is partitioned at the hour granularity of the `order_ts` field. Note how you don't have to add an explicit column for partitioning with Iceberg tables. This feature is called *hidden partitioning*, as discussed in Chapter 1.

```
## Spark SQL
CREATE TABLE orders (
   order_id BIGINT,
   customer_id BIGINT,
   order_amount DECIMAL(10, 2),
   order_ts TIMESTAMP
)
USING iceberg
PARTITIONED BY (HOUR(order_ts))
## Dremio Sonar
CREATE TABLE orders (
  order_id BIGINT,
  customer_id BIGINT,
  order_amount DECIMAL(10, 2),
  order_ts TIMESTAMP
)
PARTITION BY (HOUR(order_ts))
```

Send Query to the Engine

First, the query is sent to the query engine to parse it on their end. Then, since it is a CREATE statement, the engine will start creating and defining the table.

Write Metadata File

At this point, the engine starts creating a metadata file `v1.metadata.json` in the data lake file system to store information about the table. A generic form of the URL of the path looks something like this - `s3://path/to/warehouse/db1/table1/metadata/v1.metadata.json`. Based on the information on the table path, i.e. `/path/to/warehouse/db1/table1`, the engine writes the metadata file. It then defines the schema of the table orders by specifying the columns, data types, etc. and stores it in the metadata file. And finally it assigns a unique identifier to the table, i.e. the `table-uuid`. Once the query executes successfully, the metadata file `v1.metadata.json` is written to the data lake file storage.

```
s3://datalake/db1/orders/metadata/v1.metadata.json
```

If you inspect the metadata file, you will see the schema of the defined table along with the partition specification as seen below.

```
"schema" : {
    "type" : "struct",
    "schema-id" : 0,
    "fields" : [ {
      "id" : 1,
      "name" : "order_id",
      "required" : false,
      "type" : "long"
    }, {
      "id" : 2,
```

```
      "name" : "customer_id",
      "required" : false,
      "type" : "long"
    }, {
      "id" : 3,
      "name" : "order_amount",
      "required" : false,
      "type" : "decimal(10, 2)"
    }, {
      "id" : 4,
      "name" : "order_ts",
      "required" : false,
      "type" : "timestamptz"
    }
},
 "partition-spec" : [ {
    "name" : "order_ts_hour",
    "transform" : "hour",
    "source-id" : 4,
    "field-id" : 1000
  } ]
```

This is the current state of the table, i.e. you have created a table but it is an empty table with no records. In Iceberg terms, this is called a *snapshot* (Refer to Chapter 2 for details).

An important thing to note here is that since, to this point, you haven't inserted any records, there is no actual data in the table, so there are no data files in your data lake. Therefore, the snapshot doesn't point to any manifest list; hence, there are no manifest files.

Update the Catalog File to Commit Changes

Finally, the engine updates the current metadata pointer to point to the `v1.metadata.json` file in the catalog file `version-hint.text`, as this is the present state of our table.

Note that the name of the catalog file `version-hint.text` is specific to the catalog choice. For this demonstration, we have leveraged the file system-based Hadoop catalog. In Chapter 6 we'll compare and contrast the different Iceberg catalog choices you have.

Figure 3-2 illustrates the Iceberg components hierarchy after the table is created.

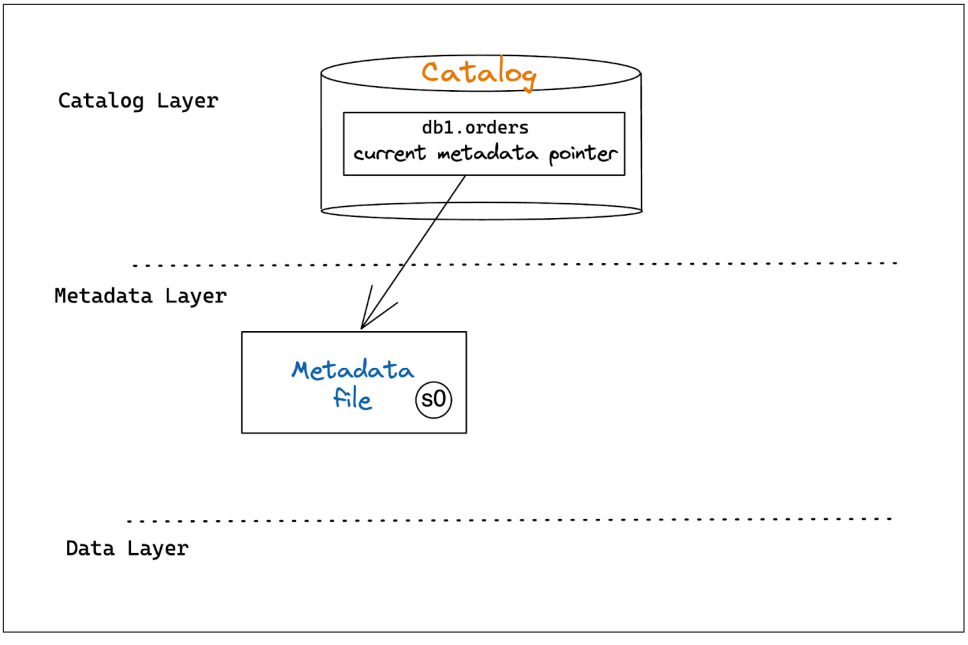


Figure 3-2. Iceberg component's hierarchy after executing CREATE

Insert Query

Now, let us insert some records into the table and understand how things work underneath. We have created a table called orders with 4 columns. For this demonstration, we will input the following values into the table: `order_id: 123`, `customer_id: 456`, `order_amount: 36.17`, `order_ts: 2023-03-07 08:10:23`. Here is the query.

```
## Spark SQL/Dremio Sonar
INSERT INTO orders VALUES (
   123,
   456,
   36.17,
   '2023-03-07 08:10:23'
)
```

Send Query to the Engine

The query is sent to the query engine to parse it. Since this is an INSERT statement, the engine needs information about the table such as its schema to start with query planning.

Check the Catalog

First, the query engine makes a request of the catalog to determine the location of the current metadata file and then reads it. Because we are using the Hadoop catalog, the engine will read the `/orders/metadata/version-hint.txt` file and see that the contents of the file is a single integer 1. Because of this and leveraging logic from the catalog implementation, the engine knows the current metadata file location is located at `/orders/metadata/v1.metadata.json` , the file our previous CREATE TABLE operation created. So, the engine will read this file. Although the engine's motivation, in this case, is inserting new data files, it still interacts with the catalog primarily for two reasons.

- The engine needs to understand the current schema of the table to adhere to it.
- Learn about the partitioning scheme to organize data accordingly while writing.

Write Data & Metadata Files

After the engine learns about the table schema and the partitioning scheme, it starts writing the new data files and the related metadata files. Here's what happens in this process.

The engine first writes the records as a Parquet data file (Parquet is default but this can be changed) based on the hourly-defined partitioning scheme of the table. Additionally, if a sort order is defined for the table, records will be sorted before being written into the data file. This is what it might look like in the file system.

```
s3://datalake/db1/orders/data/order_ts_hour=2023-03-07-08/0_0_0.parquet
```

After writing the data file, the engine creates a manifest file. This manifest file is given information about the path of the actual data file the engine created. In addition, the engine also writes statistical information, such as upper and lower bounds of a column, null value counts, etc., in the manifest file, which are highly beneficial for the query engine to prune files and provide the best performance. The engine computes this information while processing the data it's going to write, so this is a relatively lightweight operation, at least compared to a process starting from scratch having to compute the statistics. The manifest file is written as an .avro file in the storage system.

```
s3://datalake/db1/orders/metadata/62acb3d7-e992-4cbc-8e41-58809fcacb3e.avro
```

Here is a manifest file's content. Please note that this is not the full content of the metadata file. This is an excerpt with some of the key information pertaining to our topic.

```
{
   "data_file" : {
   "file_path" :
```

```
“s3://datalake/db1/orders/data/order_ts_hour=2023-03-07-08/0_0_0.parquet”,

      "file_format" : "PARQUET",
      "block_size_in_bytes" : 67108864,
      "null_value_counts" : [],
      "lower_bounds" : {
            "array": [{
            "key": 1,
            "value": 123
                 }],
      }
        "upper_bounds" : {
            "array": [{
            "key": 1,
            "value": 123
                 }],
      },
   }
}
```

Next, the engine creates a manifest list to keep track of the manifest file. If existing manifest files are associated with this snapshot, those will also be added to this new manifest list. The engine writes this file to the data lake with information such as the manifest file's path, the number of data files/rows added or deleted, and statistics about partitions, such as the lower and upper bounds of the partition columns. Again, the engine already has all of this information, so it's a lightweight operation to have these statistics. This information helps read queries exclude any non-required manifest files, facilitating faster queries.

```
s3://datalake/db1/orders/metadata/
snap-8333017788700497002-1-4010cc03-5585-458c-9fdc-188de318c3e6.avro
```

Here's a snippet of the content of a manifest list.

```
{
 "manifest_path":
"s3://datalake/db1/orders/metadata/62acb3d7-e992-4cbc-8e41-58809fcacb3e.avro",
 "manifest_length": 6152,
 "added_snapshot_id": 8333017788700497002,
 "added_data_files_count": 1,
 "added_rows_count": 1,
 "deleted_rows_count": 0,
 "partitions": {
        "array": [ {
            "contains_null": false,
            "lower_bound": {
                "bytes": "¹Ô\\u0006\\u0000"
            },
            "upper_bound": {
                "bytes": "¹Ô\\u0006\\u0000"
            }
```

```
        } ]
    }
}
```

Finally, the engine creates a new metadata file, `v2.metadata.json`, with a new snapshot, s1, by considering the existing metadata file `v1.metadata.json` (previously current) while keeping track of the previous snapshot, s0. This new metadata file includes information about the manifest list created by the engine with details such as the manifest list file path, snapshot id, summary of operation, etc. Also, the engine makes a reference that this manifest list (or snapshot) is now the current one.

```
s3://datalake/db1/orders/metadata/v2.metadata.json
```

Here's what the content of this new metadata file looks like.

```
 "current-snapshot-id" : 8333017788700497002,
 "refs" : {
 "main" :  {
 "snapshot-id" : 8333017788700497002,
      "type" : "branch"
      }
},
"snapshots" : [ {
    "snapshot-id" : 8333017788700497002,
    "summary" : {
      "operation" : "append",
      "added-data-files" : "1",
      "added-records" : "1",
},
    "manifest-list" : "s3://datalake/db1/orders/metadata/
snap-8333017788700497002-1-4010cc03-5585-458c-9fdc-188de318c3e6.avro",
  } ],
```

Update the Catalog File to Commit Changes

Now, the engine goes to the catalog again to ensure no other snapshots were committed while this INSERT operation was being run. By doing this validation, Iceberg guarantees no interference in operations in a scenario where multiple writers write data concurrently. Iceberg makes sure that the one writing the data first will get committed first, and any conflicted write operation will go back to the previous steps and re-attempt until the write is successful or fails.

In the end, the engine atomically updates the catalog to refer to the new metadata `v2.metadata.json`, which now becomes the current metadata file.

A visual representation of what the Iceberg component hierarchy looks like at this stage is presented in Figure 3-3.

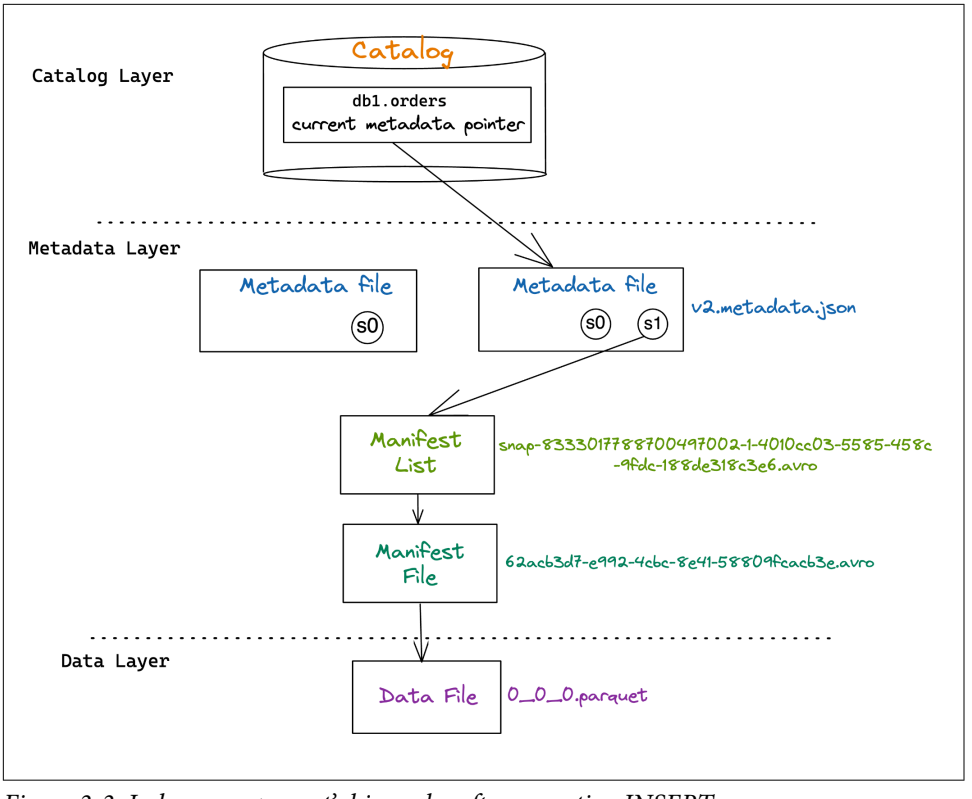


Figure 3-3. Iceberg component's hierarchy after executing INSERT

Merge Query

For our next write operation, we will do an UPSERT/MERGE INTO. Such queries are usually run when you want to update an existing row if a specific value exists in the table, and if not, you just insert the new row. Here is the query.

```
## Spark SQL
MERGE INTO orders o
USING (SELECT * FROM orders_staging) s
ON o.order_id = s.order_id
WHEN MATCHED THEN UPDATE SET order_amount = s.order_amount
WHEN NOT MATCHED THEN INSERT *;
## Dremio Sonar
MERGE INTO orders o
USING (SELECT * FROM orders_staging) s
ON o.order_id = s.order_id
WHEN MATCHED THEN UPDATE SET order_amount = s.order_amount
WHEN NOT MATCHED THEN INSERT (order_id, customer_id, order_amount, order_ts) VALUES
(s.order_id, s.customer_id, s.order_amount, s.order_ts)
```

So, for our example, let's say there is a stage table, `orders_staging`, which consists of two records - one that has an update for the existing `order_id` (`order_id=123`) and another that is an entirely new order. We want to keep the orders table updated with the latest details for each order, and therefore we will update the `order_amount` if the `order_id` already exists in the destination table (`orders`). If not, we will just insert the new record.

Send Query to the Engine

The query is first parsed by the query engine. In this case, since two tables are involved (stage & destination), the engine needs the data for both tables to start with the query planning.

Check the Catalog

Similar to the INSERT operation (discussed in the previous section), the query engine first makes a request to the catalog to determine the current metadata file location and then it reads it. Because the catalog used for this exercise is Hadoop, the engine will read the `/orders/metadata/version-hint.txt file` and retrieve the content of the file, which is the integer 2. After getting this information and using the catalog logic, the engine learns that the current metadata file location is `/orders/metadata/v2.metadata.json`. This is the file that our previous INSERT operation generated. So, the engine will read this file. It will then look at the current schema of the table so the write operations can adhere to it. Finally, the engine will learn how data files are organized based on the partitioning strategy and start writing the new data files.

Write Data & Metadata Files

First, the query engine will read and load data in memory from both the `orders_staging` and orders table to determine the matching records. Note that we will go over the READ process in detail in the next section. The engine will traverse through each record in both tables based on the `order_id` field and find out the records that match.

One important thing to note here is that as the engine is determining the matches, what gets tracked in memory will be based on the two strategies defined by the Iceberg table properties: *copy-on-write* or *merge-on-read*.

While we'll go into more depth on these two strategies in Chapter 5, in short, with the copy-on-write strategy, whenever the Iceberg table is updated, any associated data files with the relevant records will be rewritten as a new data file. However, with merge-on-read, the data files will not be rewritten; instead new *delete* files will be generated to keep a track of the changes.

In our case, we'll use the copy-on-write strategy. So, the data file `0_0_0.parquet` which contains the record with `order_id = 123` from the orders table will be read into memory. Then the `order_amount` field for this `order_id` will be updated with the new `order_amount` from the `order_staging` table in the in-memory copy of this data. Finally, these modified details are then written to a new parquet data file.

```
s3://datalake/db1/orders/data/order_ts_hour=2023-03-07-08/0_0_1.parquet
```

Note that we just had one record in the `orders` table in this specific example. However, even if there were other records in this table that didn't match the condition specified in the query, the engine would still make a copy of all these records and only the matching rows would have been updated. This is due to the write strategy, copy-on-write. You will learn more about these writing strategies in Chapter 5.

Now, the record in the `order_staging` table that didn't match the condition will be treated as a regular INSERT and will be written as a new data file in a different partition as the `hour(order_ts)` value is different for this one.

```
s3://datalake/db1/orders/data/order_ts_hour=2023-01-27-10/0_0_0.parquet
```

After writing the data files, the engine creates a new manifest file that holds a reference to the file path of these two data files. Additionally, various statistics about these data files such as the lower and upper bounds of a column, value counts, etc. are included in the manifest file.

```
s3://datalake/db1/orders/metadata/faf71ac0-3aee-4910-9080-c2e688148066.avro
```

Here is a snippet of what the manifest file looks like.

```
{
      "data_file" : {
      "file_path" :
"s3://datalake/db1/orders/data/order_ts_hour=2023-01-27-10/0_0_0.parquet",
      "file_format" : "PARQUET",
      "block_size_in_bytes" : 67108864,
      "null_value_counts" : [],
      "lower_bounds" : {
            "array": [{
            "key": 1,
            "value": 125
                }],
       }
       "upper_bounds" : {
             "array": [{
             "key": 1,
             "value": 125
                  }],
       }
    },
       "data_file" : {
```

```
            "file_path" :
        "s3://datalake/db1/orders/data/order_ts_hour=2023-03-07-08/0_0_1.parquet",
            "file_format" : "PARQUET",
            "block_size_in_bytes" : 67108864,
            "null_value_counts" : [],
            "lower_bounds" : {
                  "array": [{
                  "key": 1,
                  "value": 123
                       }],
             }
             "upper_bounds" : {
                   "array": [{
                   "key": 2,
                   "value": 200
                        }],
                  }
             }
      }
```

The engine then generates a new manifest list that points to the manifest file created in the previous step. It also tracks any existing manifest files and writes the manifest list to the data lake.

```
s3://datalake/db1/orders/metadata/snap-5139476312242609518-1-e22ff753-2738-4d7d-
a810-d65dcc1abe63.avro
```

Upon inspecting the manifest list, you can also see things such as partition statistics, number of added and deleted files, etc.

```
            {
             "manifest_path":
             "s3://datalake/db1/orders/metadata/faf71ac0-3aee-4910-9080-
c2e688148066.avro",
             "manifest_length": 6196,
             "added_snapshot_id": 5139476312242609518,
             "added_data_files_count": 2,
             "added_rows_count": 2,
             "partitions": {
                    "array": [ {
                        "contains_null": false,
                        "lower_bound": {
                            "bytes": "¹Ô\\u0006\\u0000"
                        },
                        "upper_bound": {
                            "bytes": "¹Ô\\u0006\\u0000"
                        }
                    } ]
                }
            }
            {
            "manifest_path":
```

```
              "s3://datalake/db1/orders/metadata/e22ff753-2738-4d7d-a810-
d65dcc1abe63-m0.avro",
              "manifest_length": 6162,
              "added_snapshot_id": 5139476312242609518,
              "added_data_files_count": 0,
              "added_rows_count": 0,
              . . . . . .
              }
```

After that, the engine goes on to create a new metadata file `v3.metadata.json` with a new snapshot s2 based on the previously current metadata file `v2.metadata.json` and the snapshots included as part of that, i.e. `s0` and `s1`.

```
s3://datalake/db1/orders/metadata/v3.metadata.json
```

The content of the manifest file looks something like this.

```
              "current-snapshot-id" : 5139476312242609518,
              "refs" : {
              "main" :  {
              "snapshot-id" : 5139476312242609518,
                    "type" : "branch"
                    }
              },
              "snapshots" : [ {
                  "snapshot-id" : 5139476312242609518,
                  "summary" : {
                    "operation" : "overwrite",
                    "added-data-files" : "2",
                    "deleted-data-files" : "1",
                    "added-records" : "2",
              },
              "manifest-list" : s3://datalake/db1/orders/metadata/
snap-5139476312242609518-1-e22ff753-2738-4d7d-a810-d65dcc1abe63.avro",
               } ],
```

Update the Catalog File to Commit Changes

Finally, the engine runs a check at this point to ensure there are no write conflicts and then updates the catalog with the value of the latest metadata file which is `v3.metadata.json`.

Visually, the Iceberg components' would look like Figure 3-4 at this stage of the UPSERT operation.

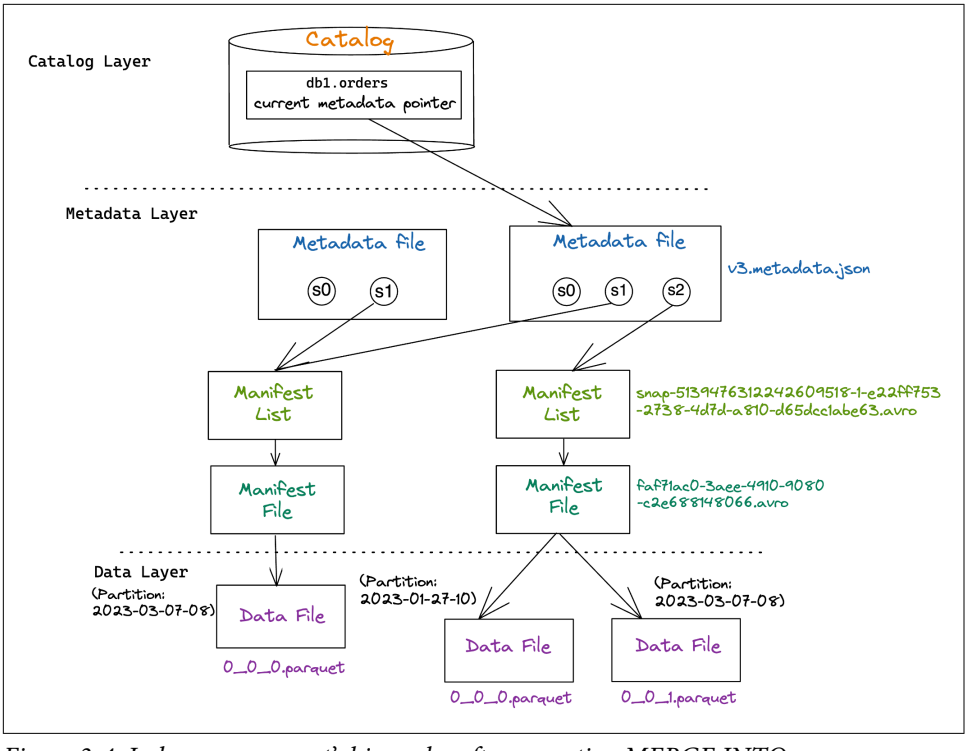


Figure 3-4. Iceberg component's hierarchy after executing MERGE INTO

Read Queries in Apache Iceberg

Reading data from Apache Iceberg tables follows a well-defined sequence of actions, seamlessly allowing queries to be transformed into actionable insights. When a read query is initiated, it is sent to the query engine first. The engine leverages the catalog to retrieve the latest metadata file location, which contains critical information about the table's schema and other metadata files such as manifest list that ultimately leads to the actual data files. Statistical information about columns are used in this process to limit the number of files being read, which helps improve query performance.

SELECT Query

In this section we will understand how the various components of Apache Iceberg work together when a READ query is executed. Here is the query that we will run.

```
## Spark SQL/Dremio Sonar
SELECT *
FROM orders
WHERE order_ts BETWEEN '2023-01-01' AND '2023-01-31'
```

Send Query to the Engine

The query is first sent to the engine that parses it. At this stage, the engine will start planning the query based on the metadata files.

Check the Catalog

The query engine reaches out to the Iceberg catalog, which in this case is a Hadoop file system. It requests the catalog for the current *metadata file* path for the orders table and then reads it. As discussed in the previous two sections, the engine will read the `/orders/metadata/version-hint.txt` file as we are using a Hadoop catalog here. The content inside this file is a single integer, i.e. `3`. Based on this information and the logic implemented for catalog implementation, the engine knows that the current metadata file location is `/orders/metadata/v3.metadata.json`. This is the file that our previous MERGE INTO operation generated.

Get Information from the Metadata File

The engine then opens and reads the metadata file `v3.metadata.json` to get information about a couple of things.

- First, it determines the schema of the table to prepare its internal memory structures for reading the data.

```
"schema" : {
    "type" : "struct",
    "schema-id" : 0,
    "fields" : [ {
      "id" : 1,
      "name" : "order_id",
      "required" : false,
      "type" : "long"
    }, {
      "id" : 2,
      "name" : "customer_id",
      "required" : false,
      "type" : "long"
    }, {
      "id" : 3,
      "name" : "order_amount",
      "required" : false,
      "type" : "decimal(10, 2)"
    }, {
      "id" : 4,
      "name" : "order_ts",
      "required" : false,
      "type" : "timestamptz"
    } ]
  },
```

- Then it learns about the table's partitioning scheme to understand how the data is organized. This can later be leveraged by the query engine to skip non-relevant data files.

```
"partition-spec" : [ {
    "name" : "order_ts_hour",
    "transform" : "hour",
    "source-id" : 4,
    "field-id" : 1000
  } ]
```

One of the most important pieces of information that the engine retrieves from the metadata file is the `current-snapshot-id`. This is what signifies the current state of the table. Based on the `current-snapshot-id`, the engine will locate the *manifest list* file path from the snapshots array so it can traverse further and scan the relevant files.

```
"current-snapshot-id" : 5139476312242609518,
  "refs" : {
    "main" : {
      "snapshot-id" : 5139476312242609518,
      "type" : "branch"
    }
  },
"snapshots" : [{
    "snapshot-id" : 7327164675870333694,
    "manifest-list" : "s3://datalake/db1/orders/metadata/snap-7327164675870333694-1-
f5e79df9-7027-4d0c-a39b-7a3091741d6f.avro",
    "schema-id" : 0
  },
{
    "snapshot-id" : 5139476312242609518,
    "parent-snapshot-id" : 8333017788700497002,
    "manifest-list" : "s3://datalake/db1/orders/metadata/snap-5139476312242609518-1-
e22ff753-2738-4d7d-a810-d65dcc1abe63.avro",
  },
{
    "snapshot-id" : 8333017788700497002,
    "parent-snapshot-id" : 7327164675870333694
    "manifest-list" : "s3://datalake/db1/orders/metadata/
snap-8333017788700497002-1-4010cc03-5585-458c-9fdc-188de318c3e6.avro",
    "schema-id" : 0
  }
]
```

Get Information from the Manifest List

After getting the location of the manifest list file path from the metadata file, the query engine reads this file `snap-5139476312242609518-1-e22ff753-2738-4d7d-a810-d65dcc1abe63.avro` to derive further details.

The most critical piece of information that the engine gets from this file is the *manifest file* path location for each snapshot within that manifest list. The engine needs this information to get the relevant data files needed for a specific query.

The manifest list also contains critical information on partitions, such as the `partition-spec-id`. This tells the engine about the specific partition scheme used to write a particular snapshot. As of now, the value of this field is 0 (seen in the JSON below), which implies that this is the only partition that has been defined for the table.

There are also other partition-specific statistics, such as the lower and upper bounds of the partition columns for a manifest. This information is specifically beneficial when the engine determines which manifest files to skip for better file pruning.

Other details such as total number of data files added/deleted, number of rows added/deleted, etc. for each snapshot are also found in this file.

```
{
"manifest_path" : "s3://datalake/db1/orders/metadata/faf71ac0-3aee-4910-9080-
c2e688148066.avro",
"partition_spec_id" : 0,
"added_snapshot_id" : 5139476312242609518,
"added_data_files_count" : 2,
"partitions" : [
    {
       "contains_null" : false,
       "contains_nan" : false,
       "lower_bound" : "ShkHAA==",
       "upper_bound" : "8BwHAA=="
     }
 ],}
```

Get information from the Manifest Files

The engine then opens the manifest file `faf71ac0-3aee-4910-9080-c2e688148066.avro` that wasn't pruned (i.e., relevant to the query). It reads the file to get the details as presented below.

First, the query engine scans each and every entry of the data files that are part of this manifest file. It compares the partition values that each of these data files belongs to, to the value requested in the query. Here are the two partition values for our example.

```
"partition" : {
      "order_ts_hour" : 2023-01-27-10
        },
"partition" : {
      "order_ts_hour" : 2023-03-07-08
        },
```

Now for our query, the request is to get all the order details between '2023-01-01' and '2023-01-31'. Therefore, the engine will ignore the second partition, 2023-03-07-08 as it doesn't match the filter value range. Since the filter values match the first partition value, the engine will check for all the records in this partition, starting from 2023-01-01 00:00:00 to 23:59:59 and 2023-01-31 00:00:00 to 23:59:59.

Based on the partition value, the engine looks for the corresponding data file, which in this case is the 0_0_0.parquet file. The engine also gathers other statistical information such as the lower and upper bounds of each column, null value counts, etc. to skip any non-relevant files. Note that the engine doesn't need to use the statistics in this case as there is just one single data file in the partition that satisfies the query.

```
              "data_file" : {
                    "file_path" :
                    "s3://datalake/db1/orders/data/
order_ts_hour=2023-01-27-10/0_0_0.parquet",
                    "partition" : {
                          "order_ts_hour" : 2023-01-27-10
                      },
                    "lower_bounds" : [{
                               "key" : 1,
                               "value" : "fQAAAAAAAAA="
                          },
                          {
                               "key" : 2,
                               "value" : "QQEAAAAAAAA="
                          }],
                     "upper_bounds" : [{
                               "key" : 1,
                               "value" : "fQAAAAAAAAA="
                           },
                           {
                               "key" : 2,
                               "value" : "QQEAAAAAAAA="
                           }],
                },
              "data_file" : {
                    "file_path" :
                    "s3://datalake/db1/orders/data/
order_ts_hour=2023-03-07-08/0_0_1.parquet"
                    "partition" : {
                          "order_ts_hour" : 2023-03-07-08
                      },
                  },
```

Data and file optimization techniques such as partitioning and metrics-based filtering (upper/lower bound of columns) that is available by default in Apache Iceberg allows the engine to avoid full table scans as seen with this example, thereby facilitating significant performance guarantees. Finally, the record is returned back to the user.

	order_id	customer_id	order_amount	order_ts
1	125	321	20.50	2023-01-27 10:30:05 +00:00

Figure 3-5. [Figure caption to come]

Visually, this entire process of READ looks like below.

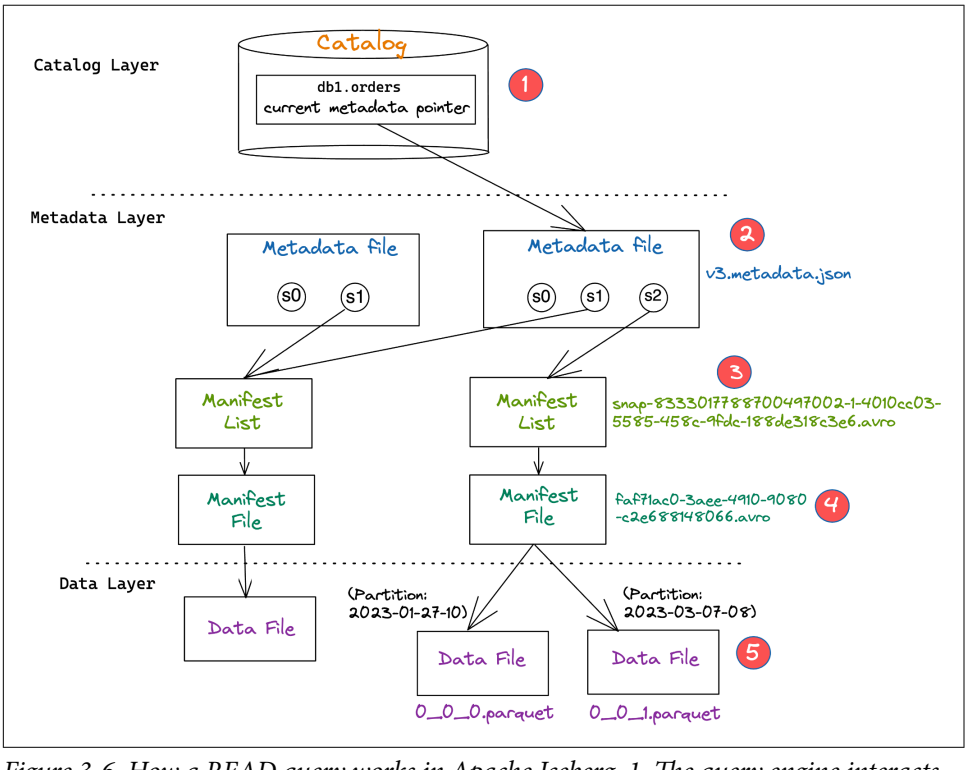


Figure 3-6. How a READ query works in Apache Iceberg. 1. The query engine interacts with the catalog to get the current metadata file (v3.metadata.json). 2. It then gets the current-snapshot-id (S2 in this case) & manifest list location for that snapshot 3. The manifest file path is then retrieved from the manifest list 4. The engine determines the data file path based on the partition filter (2023-03-07-08) from the manifest file 5. Required data file is then returned to the user.

Time-travel Query

An important capability in the world of databases and data warehouses is the ability to go back in time to a particular state of the table in order to query historical data, i.e. data that has been changed or deleted. Apache Iceberg brings a similar 'time

travel' capability to a data lakehouse architecture. This can be particularly useful for scenarios such as analyzing your organization's previous quarters' data, restoring accidentally deleted rows, or reproducing analysis results. Apache Iceberg provides two ways to run time travel queries:

- Using a timestamp
- Using a snapshot ID

In this section, you will learn how to run time travel queries for an Apache Iceberg table.

For the purpose of this demonstration, let's say we needed to travel back to the state before we executed the MERGE INTO query, i.e. when we just ran our INSERT statement. So, given those assumptions, the first thing we need to understand is the history of the Iceberg table. One of the best things about Apache Iceberg is that it allows you to analyze various table-specific metadata information via system tables called metadata tables. To analyze our order table's history, we will query the history metadata table. Here is the query.

```
## Spark SQL
SELECT * FROM catalog.db.orders.history;
## Dremio Sonar
SELECT * FROM TABLE (table_history('orders'))
```

This gives us a list of all the transactions that have happened in this table.

made_current_at	snapshot_id	parent_id	is_current_ancestor
2023-03-06 21:28:35.360	7327164675870333694	*null*	true
2023-03-07 20:45:08.914	8333017788700497002	7327164675870333694	true
2023-03-09 19:58:40.448	5139476312242609518	8333017788700497002	true

Figure 3-7. [Figure caption to come]

To summarize the history table:

- The first snapshot with id `7327164675870333694` was generated after we ran the CREATE statement
- The second snapshot, `8333017788700497002` was created after we inserted a new record using the INSERT statement
- And finally, our MERGE INTO query created the third snapshot, `5139476312242609518`

Since our requirement is to time travel to the state prior to the final transaction (i.e. MERGE), the timestamp or the snapshot id we will be targeting is the second one. This is the query that we will run.

```
## Spark SQL
SELECT * FROM orders
TIMESTAMP AS OF '2023-03-07 20:45:08.914'
## Dremio Sonar
SELECT * FROM orders
AT TIMESTAMP '2023-03-07 20:45:08.914'
```

If we want to use the snapshot ID to time travel, the query would look like this.

```
## Spark SQL
SELECT *
FROM orders
VERSION AS OF 8333017788700497002
## Dremio Sonar
SELECT *
FROM orders
AT SNAPSHOT 8333017788700497002
```

Now, let's quickly understand what happens underneath with the Iceberg components when the time travel query is run and how the relevant data file is returned back to the user.

Send Query to Engine

As with any SELECT statement, the query is first sent to the engine, which parses it. The engine will leverage the table metadata to start planning the query.

Check the Catalog

In this step, the query engine requests the catalog to know the location of the current metadata file and reads it. Since we have leveraged the Hadoop catalog in this exercise, the engine will read the content of the `/orders/metadata/version-hint.txt` file, which is the integer 3. With this information and following the catalog's implementation logic, the engine determines that the location of the current metadata file is `/orders/metadata/v3.metadata.json`. The engine will finally read this file to understand the table schema and things such as partitioning strategy.

Get Information from the Metadata File

Next, the engine reads the metadata file to get the following information.

The current metadata file keeps track of all the snapshots generated for our Iceberg table unless the snapshots were intentionally expired as part of the metadata maintenance strategies (more about it in Chapter 4). From the available list of snapshots, the engine will determine the particular snapshot specified in the time travel query based on either the timestamp value or the snapshot id.

For our specific example, the timestamp is `2023-03-07 20:45:08.914` and the snapshot id is `8333017788700497002`. Based on this information, the corresponding

snapshot is as presented below. Note that the timestamp value is in millisecond UNIX epoch format in the metadata file (`1678221908914`), which, when converted to date-time format, is March 7, `2023 8:45:08.914`. This value matches our time travel query.

```
       {
    "snapshot-id" : 7327164675870333694,
    "timestamp-ms" : 1678138115360,
    "summary" : {
      "operation" : "append",
      "total-records" : "0",
      "total-files-size" : "0",
     },
    "manifest-list" : "s3://datalake/db1/orders/metadata/snap-7327164675870333694-1-
f5e79df9-7027-4d0c-a39b-7a3091741d6f.avro",
    "schema-id" : 0
  },
{
    "snapshot-id" : 8333017788700497002,
    "parent-snapshot-id" : 7327164675870333694,
    "timestamp-ms" : 1678221908914,
    "summary" : {
      "operation" : "append",
      "added-data-files" : "1",
      "added-records" : "1",
      "total-records" : "1",
    },
    "manifest-list" : "s3://datalake/db1/orders/metadata/
snap-8333017788700497002-1-4010cc03-5585-458c-9fdc-188de318c3e6.avro",}
```

The engine also learns about the table's schema and the partitioning scheme so it can use it later for file pruning.

And finally, it gets the location of the corresponding manifest list path for that particular snapshot, i.e. `s3://datalake/db1/orders/metadata/snap-8333017788700497002-1-4010cc03-5585-458c-9fdc-188de318c3e6.avro`

Get Information from the Manifest List

Based on the manifest list path, the engine opens and reads the `snap-8333017788700497002-1-4010cc03-5585-458c-9fdc-188de318c3e6.avro` file.

Here is the content of this manifest list.

```
{    "manifest_path" : "s3://datalake/db1/orders/metadata/62acb3d7-
e992-4cbc-8e41-58809fcacb3e.avro",
    "manifest_length" : 6152,
    "partition_spec_id" : 0,
    "added_snapshot_id" : 8333017788700497000,
    "added_data_files_count" : 1,
```

```
    "existing_data_files_count" : 0,
    "deleted_data_files_count" : 0,
    "partitions" : [
        {
            "contains_null" : false,
            "contains_nan" : false,
            "lower_bound" : "8BwHAA==",
            "upper_bound" : "8BwHAA=="
        }
    ],
    "added_rows_count" : 1,
    "deleted_rows_count" : 0
}
```

The engine derives couple of important information from the manifest list, such as:

- The manifest file path location which holds references to the actual data files. In this case it is, `s3://datalake/db1/orders/metadata/62acb3d7-e992-4cbc-8e41-58809fcacb3e.avro`
- Additionally, information about the number of data files added/deleted, statistical information about partitions, etc. are also gathered.

Get Information from the Manifest File

Finally, the engine reads the manifest file `62acb3d7-e992-4cbc-8e41-58809fcacb3e.avro` and gets details about the following.

The most important information in the manifest file is the *data file path*, which contains the records for a query. Usually, the engine would look at each of the data file's entries in the manifest file and, depending on the partition value the file belongs to, it will determine the relevant data file. However, in this case, just one data file belongs to a single partition. So, that would be the one for our 'time travel' query.

```
              "data_file" : {
                "file_path" : "s3://datalake/db1/orders/data/
order_ts_hour=2023-03-07-08/0_0_0.parquet",
                "file_format" : "PARQUET",
                "partition" : {
                    "order_ts_hour" : 2023-03-07-08
                },
                 "lower_bounds" : [
                    {
                        "key" : 1,
                        "value" : "ewAAAAAAAAA="
                    } ],
                  "upper_bounds" : [
                    {
                         "key" : 1,
                         "value" : "ewAAAAAAAAA="
```

```
            } ],
        }
```

Other than the data file path location, the engine also gathers statistical information on the columns as discussed in the previous sections.

In the end, the engine reads the data file `0_0_0.parquet` and the below output is returned to the user.

	order_id	customer_id	order_amount	order_ts
1	123	456	36.17	2023-03-07 08:10:23 +00:00

Figure 3-8. [Figure caption to come]

This is the record we inserted into the table before running the MERGE INTO query.

Figure 3-9 provides a visual summary of the process.

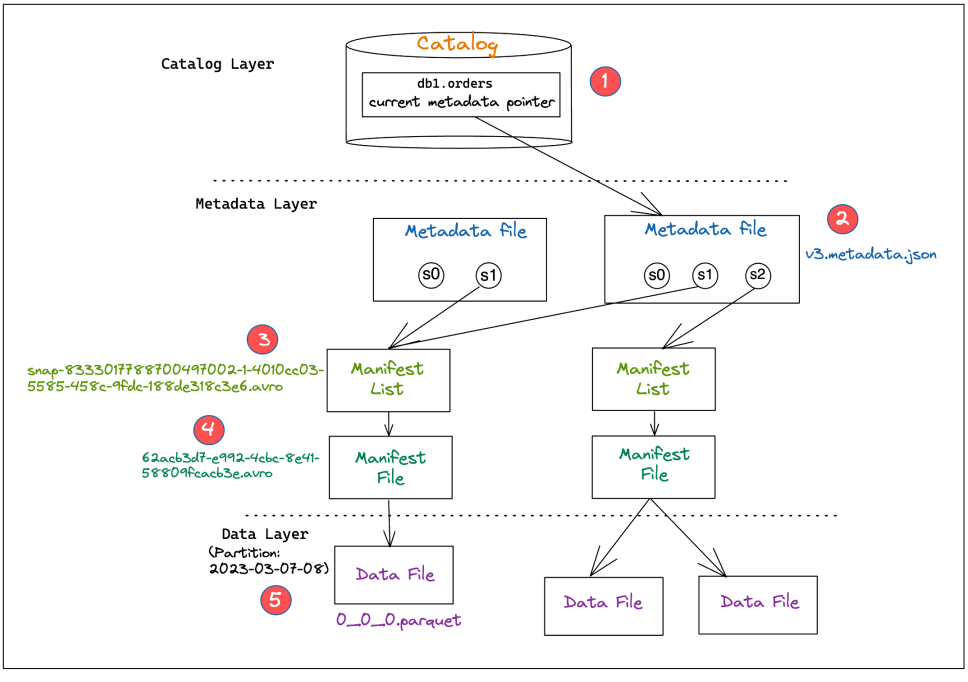


Figure 3-9. How the TIME TRAVEL query works in Iceberg. 1. The query engine interacts with the catalog to get the current metadata file (v3.metadata.json). 2. It then selects the snapshot (S1 in this case) based on either the timestamp or version ID supplied in the time travel query & gets the manifest list location for that snapshot 3. The manifest file path is then retrieved from the manifest list 4. The engine determines the data file path based on the partition filter (2023-03-07-08) from the manifest file 5. Required data file is then returned to the user.

Conclusion

We discussed the internal working mechanism of various read and write queries such as creating tables, inserting and updating records in this chapter to understand how different architectural components of Apache Iceberg are leveraged by compute engines.

In the next chapter, we will go through the available out-of-the-box optimization techniques in Apache Iceberg to ensure high performance when reading and writing data to tables.

CHAPTER 4

Optimizing the Performance of Iceberg Tables

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

As you saw in Chapter 3, Apache Iceberg tables provides a layer of metadata that allows the query engine to create smarter query plans for better performance. This metadata, though, is only the beginning of how you can optimize the performance of your data.

There are several levers for optimization we can tweak:

- The number of data files that hold our data
- How the data is sorted in those files
- The partitioning of the table
- How we handle the writing of row-level updates
- The metrics we collect

- External factors like how we store the data, process the data, etc.

In this chapter we'll explore all these topics, where slowdown can occur and how to avoid them and accelerate our tables even further.

Compaction

Every procedure that any process has to do comes at a cost in terms of time. Stated differently, the more steps to doing something, the longer it will take. So when you are querying your Apache Iceberg tables, each file needs to be opened, scanned and when done, closed. The more files you have to scan for a query the more of a cost these file operations will put on your query. This problem is magnified in the world of streaming or "real time" data where data is ingested as it is created, creating lots of files with only a few records in each.

In contrast, batch ingestion where you may ingest a whole day or week's worth of records in one job allows you to more efficiently plan how to write the data to more well organized files. Although either way it is possible to run into the "small files problem" where too many small files have an impact on the speed and performance of your scans because you're doing more file operations, have a lot more metadata to read (there is metadata on each file), and have to delete more files when doing cleanup and maintenance operations. See the two scenarios in Diagram 4-1 below.

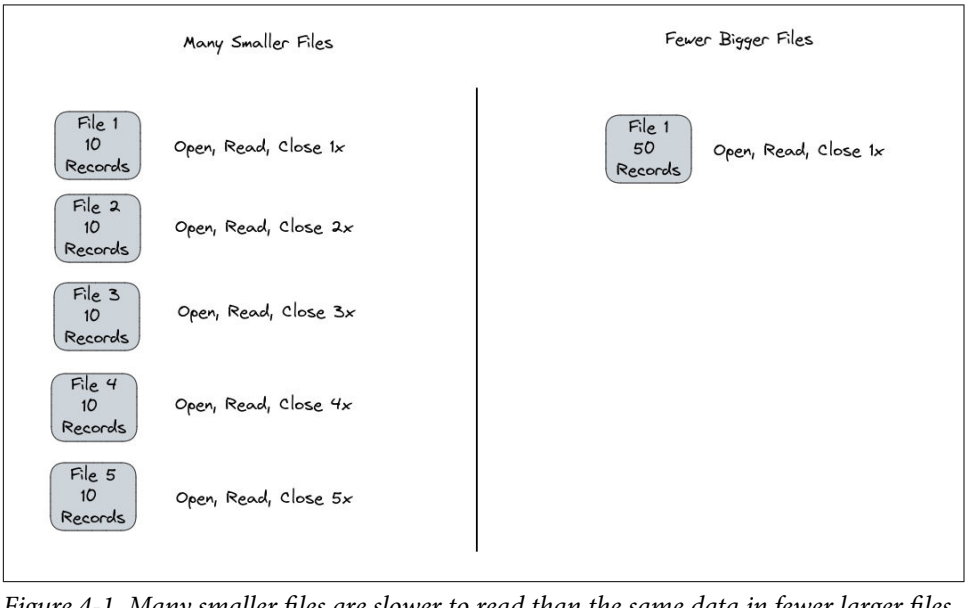


Figure 4-1. Many smaller files are slower to read than the same data in fewer larger files.

Essentially when it comes to reading data there are fixed costs we can't avoid and variable costs we can avoid using different strategies. Fixed costs include reading the particular data relevant to our query–we can't avoid having to read the data to process it. Although, variable costs would include the file operations to access that data. So using many of the strategies we discuss throughout this chapter we can reduce those variable costs as much as possible. After using these strategies you'll be using only the necessary compute to get your job done cheaper and faster (getting the job done faster has the benefit of being able to terminate compute clusters earlier reducing their costs.)

The solution to this problem is to periodically take the data in all these small files and write that same data into fewer larger files. This process is called compaction as you are compacting the many into few. You can see compaction illustrated in diagram 4-2.

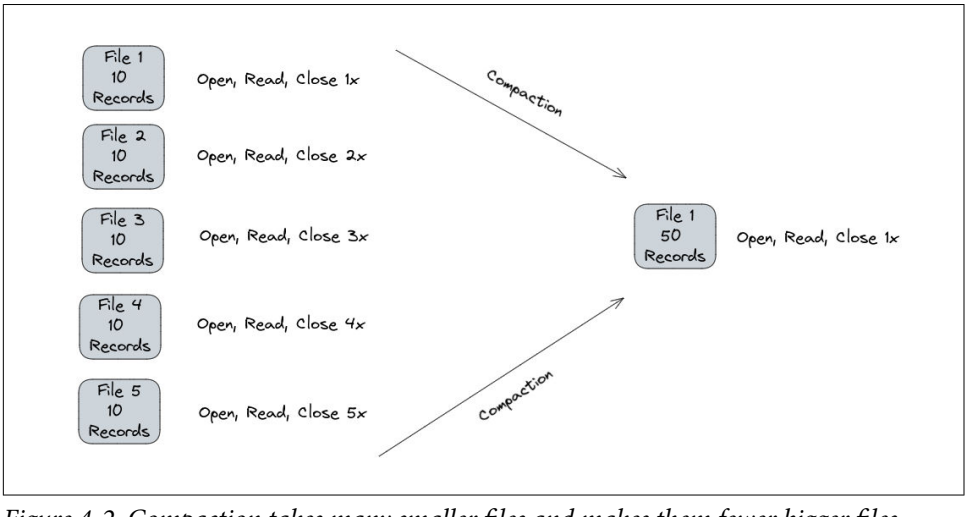


Figure 4-2. Compaction takes many smaller files and makes them fewer bigger files.

Hands on with Compaction

You may be thinking that while the solution sounds simple, this will involve you having to write some extensive code in Java or Python. Fortunately,Apache Iceberg's actions package (the Actions package is particularly for Spark, but other engines can create their maintenance operation implementation)includes several maintenance procedures. This package is used from within Apache Spark either by writing SparkSQL as shown through most of this chapter or by writing imperative codelike the exhibit below.

```
Table table =  catalog.loadTable("myTable");
SparkActions
    .get()
    .rewriteDataFiles(table)
```

```
    .option("rewrite-job-order", "files-desc")
    .execute();
```

In this snippet, we have initiated a new instance of our table and then triggered `rewriteDataFiles`, which is the Spark action for compaction. Compaction is a process of rewriting many smaller files into fewer larger files. The builder pattern used by SparkActions allows us to chain methods together to fine-grain tune the compaction job to express not only that we want compaction to be done but how we want that compaction to be done.

There are several methods we can chain between the call to rewriteDataFiles and the execute method which begins the job such as:

binPack
: This would set the compaction strategy to binPack (discussed later) which is the default and doesn't need to be explicitly supplied.

Sort
: Changes the compaction strategy to sorts the data rewritten by one of more fields in a priority order, further discussed in section x.x.x.

zOrder
: Changes the compaction strategy to zOrder sort the data based on multiple fields with equal weighting, further discussed in section x.x.x.

filter
: You can pass an expression used to limit which files are rewritten

option
: Changes a single option

options
: Takes a map of several option configurations

There are several possible options we can pass to configure the job, here are a few important ones.

target-file-size-bytes
: This will set the intended size of the output files. By default, this will use the write.target.file-size-bytes property of the table which defaults to 512mb.

File Size and Row Group Size

For Apache Parquet files there is row group size and file size. Row Group size is the size of one group of rows in a file which can have multiple groups. So by default, a table's default configuration would allow for 128mb row groups and 512mb file size (four row groups per file). You'll always want to make sure these two settings are

aligned (row size evenly divides to into file size). Less row groups results in a smaller file size as there is less groups to have group metadata written for, while more row groups improves predicate push down cause the row group metadata can have more fine-grained ranges making it possible for the query engine to eliminate reading more row groups that don't contain data relevant to the current query..

Another example, you may want to increase the file size to 1 GB per file but keep row groups to 128mb (8 row groups per file), that way there are less files to open and close. Although, if the type of queries you're running often require reading most of the data then you'd prefer less row groups since predicate pushdown will not speed up getting all the data.

Row group file and file size can both be set as table properties (write.parquet.row-group-size-bytes and write.target-file-size-bytes respectively) but the file size can be set for individual compaction jobs using the options settings.

max-concurrent-file-group-rewrites
: The ceiling for the number of file groups to written simultaneously

What Are File Groups

The engine as it plans the new files to be written in the compaction job, will begin grouping these files into "file groups" which will be written in parallel. In our compaction jobs we can configure options on how big these file groups can be and how many should be written simultaneously to help prevent memory issues.

max-file-group-size-bytes
: The max size of a file group, not of one single file. This setting should be used when dealing with partitions larger than available memory to the worker writing a particular file group so it can split that partition into multiple file groups to be written asynchronously.

partial-progress-enabled:
: Allows commits to occur while file groups are compacted, so for long running compaction this can allow concurrent queries to benefit from already compacted files.

partial-progress-max-commits
: If partial progress is enabled, this setting sets the maximum number of commits allowed to complete the job.

Partial Progress

Partial progress allows new snapshots to be created as file groups are completed, this allows queries coming in to benefit from the already compacted files as others are completed. Keep in mind, more snapshots meet more metadata files taking up storage in your table location but if you want your readers to benefit from a compaction job sooner than later this can be a useful feature.If you want to balance out the cost of additional snapshots with the benefits of partial progress you adjust the max commits to limit the number of total commits a single compaction job will make.

rewrite-job-order

What order to write file groups, which can matter when using partial progress to make sure the higher priority file groups are committed sooner than later. Values for this can be based on order of groups based on byte size or number of files in a group (bytes-asc, bytes-desc, files-asc, files-desc, none).

```
Table table = catalog.loadTable("myTable");
SparkActions
    .get()
    .rewriteDataFiles(table)
    .sort()
    .filter(Expressions.and(
    Expressions.greaterThanOrEqual("date", "2023-01-01"),
    Expressions.lessThanOrEqual("date", "2023-01-31")))
    .option("rewrite-job-order", "files-desc")
    .execute();
```

In this example:

- We use the sort strategy which by default will use any sort order stated in the tables properties
- We added a filter to only rewrite the data from January, notice the filter needs to be passed an expression created with Apache Iceberg's internal expression building interface.
- We set the rewrite-job-order to rewrite larger groups of files first, so a file rewritten from 5 files will be written before a file rewritten from 2 files.

Apache Iceberg Expressions

The Expressions library is made to make it easier to create expressions around Apache Iceberg's metadata structures. The library provides APIs to build and manipulate these expressions, which can then be used to filter data in tables and read operations.

The library provides support for a range of operations, including:

Logical Operators:
: These include AND, OR, and NOT, which are used to combine or invert other expressions.

Comparison Operators:
: These include equals, not equals, less than, less than or equal to, greater than, and greater than or equal to. These can be used to compare values.

In and NotInExpressions:
: These are used to test if a value is in a set of values.

IsNull and NotNull Expressions:
: These are used to test if a value is null or not.

AlwaysTrue and AlwaysFalse:
: These are special expressions that are always true or always false.

The expressions library is important in Iceberg as it allows efficient pruning of data during reads, ensuring only the relevant pieces of data are accessed. This improves the performance of data retrieval, particularly on large datasets.

For example, when reading data from a table, an application can use the expressions library to construct a filter expression. Iceberg then uses this expression to read only the files that might contain matching rows, significantly reducing the amount of data that needs to be read. Similarly, when reading a specific file, Iceberg uses the filter to read only the row groups that might contain matching rows.

In addition to these, Iceberg's expressions are also used in manifest files to summarize the data in each data file, which allows Iceberg to skip files that do not contain rows that could match a filter. This mechanism is essential for Iceberg's scalable metadata architecture.

While this is all fine and good, this can be more easily done using the Spark SQL extensions which include call procedures that can be called using the following syntax from Spark SQL.

```
-- using positional arguments
CALL catalog.system.procedure(arg1, arg2, arg3)

-- using named arguments
CALL catalog.system.procedure(argkey1 => argval1, argkey2 => argval2)
```

To use the rewriteDataFiles procedure in this syntax it would look like:

```
-- Rewrite Data Files CALL Procedure in SparkSQL
CALL catalog.system.rewrite_data_files(
  table => 'musicians',
  strategy => 'binpack',
  where => 'genre = "rock"',
  options => map(
    'rewrite-job-order','bytes-asc',
    'target-file-size-bytes','1073741824', -- 1GB
    'max-file-group-size-bytes','10737418240' -- 10GB
  )
)
```

In this scenario we may have been streaming some data into our musicians table and noticed that a lot of small files were generated for rock bands, so instead of running compaction on the whole table which can be time consuming we target just the data that is the problem. We also tell Spark to prioritize file groups that are larger in bytes and keep files around 1 gigabyte each with each file group around 10 gigabytes. You can see what the result of these settings would be in diagram 4-3.

Also notice the use of double quotes in our `where` filter, because we had to use single quotes around the filter we use double quotes in the string even if normally SQL would use single quotes for "rock".The `where` option essentially is equivalent of the filter method mentioned earlier, without it the whole table would be possible rewritten.

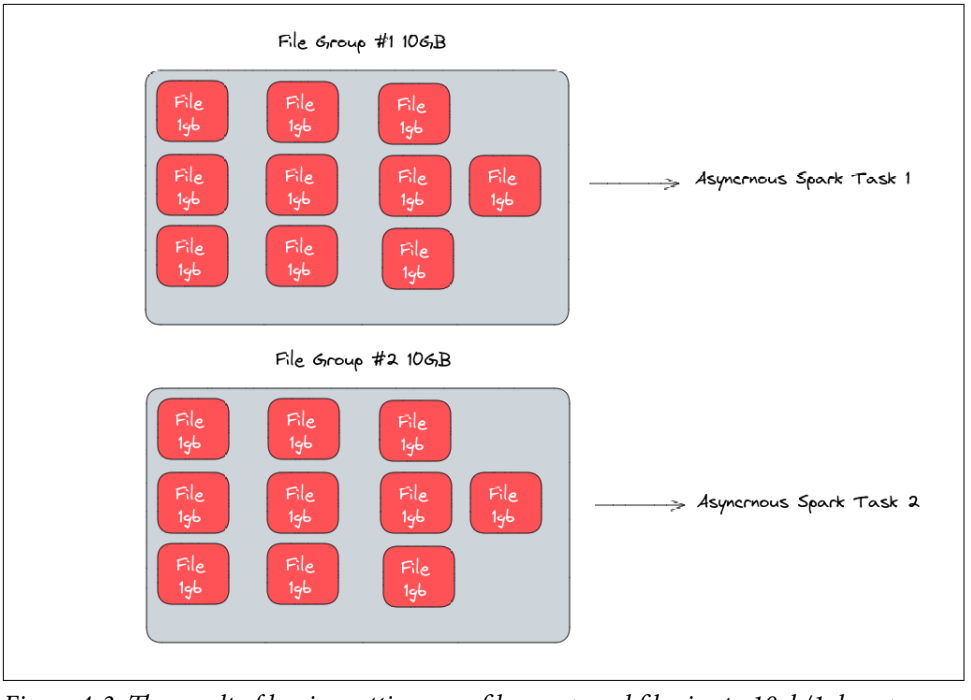


Figure 4-3. The result of having setting max file group and file size to 10gb/1gb respectively.

Other engines can implement their own custom compaction tools, for example Dremio has its own Iceberg table management feature via its OPTIMIZE command which is a unique implementation but follows many of the API's from the RewriteDataFiles action.

```
OPTIMIZE TABLE catalog.MyTable
```

The above command would achieve your basic bin_pack compaction compacting all the files into fewer more optimal files. But like the rewriteDataFiles procedure in Spark we can get more granular. Like in the following examples.

Compacting only a particular partition:

```
OPTIMIZE TABLE catalog.MyTable
   FOR PARTITIONS sales_year IN (2022, 2023) AND sales_month IN ('JAN', 'FEB',
'MAR')
```

Compacting with particular file size parameters:

```
 OPTIMIZE TABLE catalog.MyTable
   REWRITE DATA (MIN_FILE_SIZE_MB=100, MAX_FILE_SIZE_MB=1000, TAR-
GET_FILE_SIZE_MB=512)
```

Rewriting only the manifests:

```
OPTIMIZE TABLE catalog.MyTable
   REWRITE MANIFESTS
```

So there are different engines you can use to run compaction whether it's the Spark or Dremio to achieve compaction of your Apache Iceberg tables.

Compaction Strategies

As mentioned earlier, there are several compaction strategies that you can use when using the rewriteDataFiles procedure:

binPack
: Pure compaction, write small files to large files regardless of record order.

Sort
: Run compaction, sort the records by one or more fields (sorts by each field specified sequentially).

zOrder
: Run a z-order sort as you do the compaction (equal weighting of fields to be sorted by)

Standard sorting and zOrder sorting will be covered later in the book since, but in the below table you'll find a summary of these strategies before we start with an explanation of the binpack strategy.

Table 4-1. Pros and Cons of Compaction Strategies

Strategy	What it does	Pros	Cons
bin_pack	Combines files only, no global sorting (Will do local sorting within tasks)	Fastest compaction jobs	Data is not clustered
sort	Sort by one or more fields sequentially prior to allocating tasks (sort by field a, then within that sort by field b)	Data clustered by often queried fields can lead to much faster read times.	Longer compaction jobs vs bin_pack
zorder	Sort by multiple fields equally weighted prior to allocating tasks. (X and Y values in this range are in one grouping those in another range in another grouping)	If queries often rely on filters on multiple fields, this can even further improve read times.	Longer running compaction jobs vs. bin_pack

The binpack strategy is essentially pure compaction with no other considerations to how the data is organized beyond the size of files. Of the three strategies binpack is the fastest as it can just write the contents of the smaller files to a larger file of your target size while either of the sort strategies must first sort the data before it can allocate file groups for writing. This is particularly useful when you have streaming

data and need compaction to run at a speed that meets your SLAs (Service Level Agreements).

If an Apache Iceberg table has a sort order set within its settings, even if you use binpack, this sort order will be used for sorting data within a single task (local sort). Using the sort and z-order strategies will sort the data before the query engine allocates the records into different tasks optimizing the clustering of data across tasks.

If you were ingesting streaming data you may need to run a quick compaction on data that is ingested after every hour. You could do something like this:

```
CALL catalog.system.rewrite_data_files(
  table => 'streamingtable',
  strategy => 'binpack',
  where => 'created_at between "2023-01-26 09:00:00" and "2023-01-26 09:59:59" ',
  options => map(
    'rewrite-job-order','bytes-asc',
    'target-file-size-bytes','1073741824',
    'max-file-group-size-bytes','10737418240',
    'partial-progress-enabled', 'true'
  )
)
```

In this compaction job:

- you use the binpack strategy which is faster to keep up with your streaming SLA
- It only compacts data ingested between 9 and 10, this could be whatever the last hour is
- Partial Progress commits are enabled so as file groups are written they are immediately committed so the reader begins seeing performance improvements immediately
- Since the compaction covers only previously written data and any streaming writes should only add new data files there should be no conflicts between any concurrent writes.

Using a faster strategy on a limited scope of data can make your compaction jobs much faster. Of course, you could probably compact the data even more if you allowed compaction of data beyond the hour but you have to balance out the need to run the compaction job quickly with the need for optimization. You may have an additional compaction job for the day's worth of data overnight and a compaction of the week over the weekend to keep optimizing in intervals that continue to optimize while interfering as little with other operations as possible.

Automating Compaction

It would be a little tricky to meet all your SLAs if you have to manually run these compaction jobs, so looking into how to automate these processes could be a real benefit. Here are a couple of approaches you can take to automate these jobs.

- You can use an orchestration tool like Airflow, Dagster, Prefect, Argo or Luigi to send the proper SQL to an engine like Spark or Dremio after an ingestion job completes or at a certain time or periodic interval.
- You can use serverless functions to trigger the job after data lands in cloud object storage.
- You can set up Cron jobs to run at specific times to run the appropriate jobs.

These approaches require you to script out and deploy these services manually. However, there is also a class of managed Apache Iceberg catalog services that include automated table maintenance which includes compaction. Examples of these kinds of services include Dremio Arctic and Tabular.

Sorting

Before we get into the details of sort compaction strategy let's understand sorting as it relates to optimizing a table.

Why Sort

Sorting or "clustering" your data has a very particular benefit when it comes to your queries; it helps limit the number of files that need to be scanned to get the data needed for a query. Sorting the data allows data with similar values to be concentrated in fewer files allowing for more efficient query planning.

For example, suppose you have a dataset that represents all the players in NFL teams across 100 parquet files that isn't sorted in any particular way. If you did a query just for players on the Detroit Lions, even if a file of 100 records only has one record of a Detroit Lions player then that file must be added to my query plan and be scanned, So it is possible you may need to scan up to 53 files (Max number of players that can be on an NFL team). If you sorted the data alphabetically by team name, then all the Detroit Lions players should be in about 4 files (100 files divided by 32 NFL teams equals 3.125) which would probably include a handful players from the Green Bay Packers and the Denver Broncos. So, by having the data sorted you've reduced the number of files you have to scan from possibly 53 to 4, which as we discussed in the compaction section, greatly improves the performance of the query. Refer to diagram 4-4 to see the benefits in scanning sorted datasets.

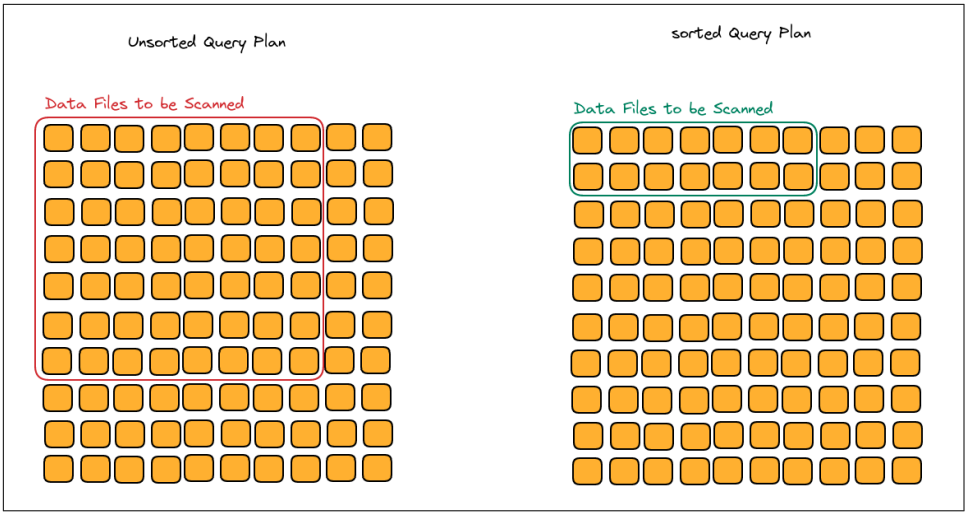


Figure 4-4. Sorted datasets result in scanning less data files

Sorted data can be quite useful if the how the data is sorted leans into typical query patterns like in this example where we may regularly query the NFL data based on a particular team. Sorting data in Apache Iceberg can happen at many different points, so we want to make sure we leverage all of these points.

Sorting When Creating a Table

There are two main ways to create a table, either with a standard CREATE TABLE statement:

```
-- Spark Syntax
CREATE TABLE catalog.nfl_players (
    id bigint ,
    player_name varchar,
        team varchar,
        num_of_touchdowns int,
        num_of_yards int,
      player_position varchar,
      player_number int,
)

-- Dremio Syntax
CREATE TABLE catalog.nfl_players (
    id bigint ,
    player_name varchar,
        team varchar,
        num_of_touchdowns int,
        num_of_yards int,
      player_position varchar,
```

```
      player_number int,
)
```

The other method by using a CREATE TABLE AS (CTAS) statement:

```
-- Spark SQL & Dremio Syntax
CREATE TABLE catalog.nfl_players
        AS (SELECT * FROM non_iceberg_teams_table);
```

There are a couple things you may want to do:

- After creating the table set the sort order of the table, which any engine that supports the property will use to sort the data before writing and will also be the default sort field when using the sort compaction strategy.
  ```
  ALTER TABLE catalog.nfl_teams WRITE ORDERED BY team;
  ```

- If doing a CTAS sort the data in your AS query.
  ```
  CREATE TABLE catalog.nfl_teams
          AS (SELECT * FROM non_iceberg_teams_table ORDER BY team);

  ALTER TABLE catalog.nfl_teams WRITE ORDERED BY team;
  ```

The ALTER TABLE statement sets a global sort order that will be used for all future writes by engines that honor the sort order. You could also specify it per INSERT like so:

```
INSERT INTO catalog.nfl_teams
        SELECT *
        FROM staging_table
                ORDER BY team
```

This will ensure the data is sorted as you write it, but it isn't perfect. Going back to our previous example, if our NFL dataset was updated each year for changes in the team's roster, you may end up having many files splitting Lions and Packers players from multiple writes, since we now need to write the new Lions players this year–but data files are immutable, so you write at least one new file with Lions players. This is where the sort compaction strategy comes into play.

The sort compaction strategy will sort the data across all the files targeted by the job. So, for example, if you wanted to rewrite the entire dataset with all players sorted by team globally, you can run the following statement.

```
CALL catalog.system.rewrite_data_files(
  table => 'nfl_teams',
  strategy => 'sort',
  sort_order => 'team ASC NULLS LAST'
)
```

Let's breakdown the string we passed for our sort order:

- 'team' we will sort the data by the team field
- 'ASC' the data will be sorted in ascending order (DESC for descending order)
- 'NULLS LAST' will put all players with a null value at the end of the sort after the Washington Commanders (NULLS FIRST if we wanted them before the Arizona Cardinals)

We can see the result of the sort in diagram 4-5.

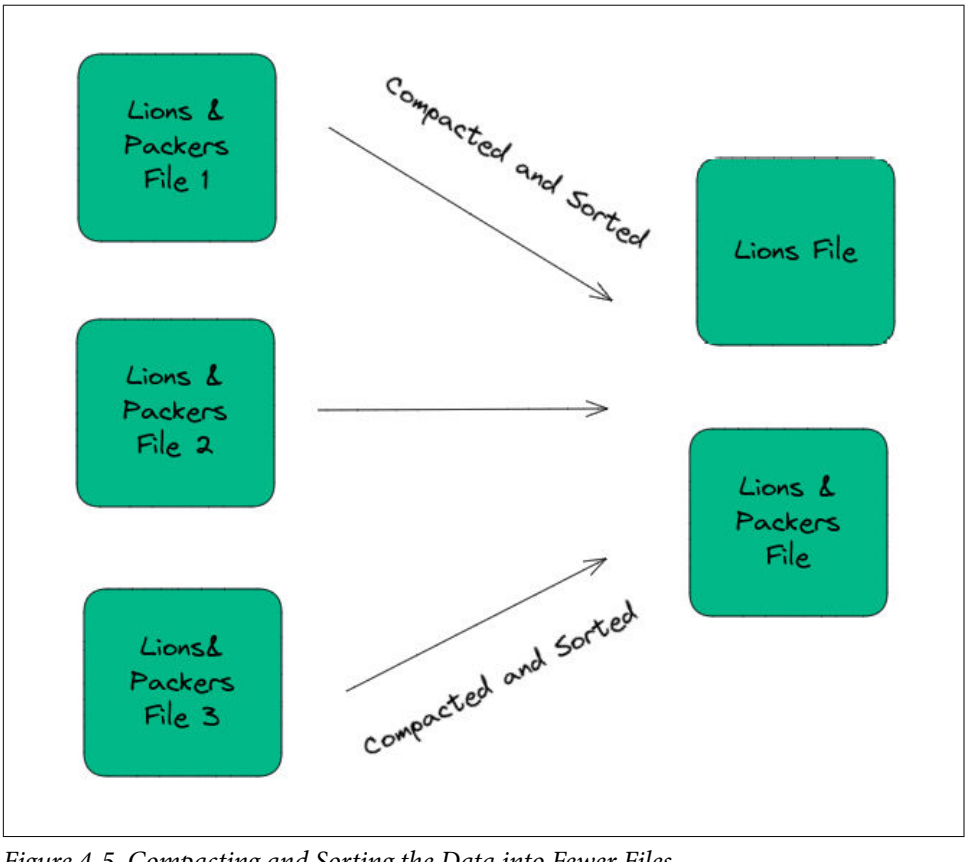


Figure 4-5. Compacting and Sorting the Data into Fewer Files

We can sort by additional fields, for example, we may want the data sorted by team, but then within each team we want it sorted alphabetically by name, we can achieve this by running a job with these parameters.

```
CALL catalog.system.rewrite_data_files(
  table => 'nfl_teams',
  strategy => 'sort',
```

```
  sort_order => 'team ASC NULLS LAST, name ASC NULLS FIRST'
)
```

So sorting by team will have the highest weight followed by name. You'll probably see players in this order in the file where the Lions roster ends and the Packers roster begins as seen in diagram 4-6.

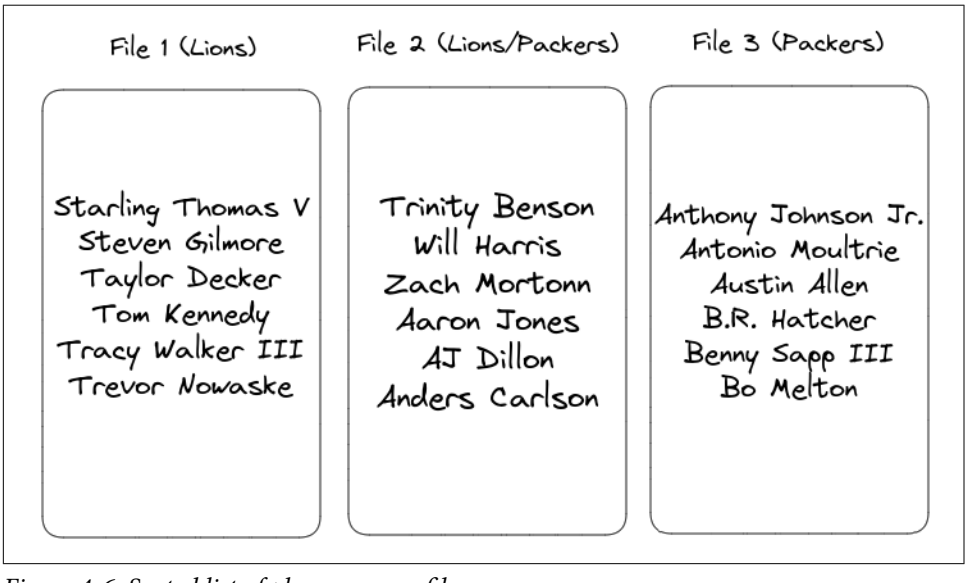


Figure 4-6. Sorted list of players across files.

If end users regularly asked questions like "Who are all the players in the Lions whose name starts with A" this dual sort will accelerate the query even further. Although, if we asked, "Who are all the players in the NFL whose name starts with A" this wouldn't be as helpful as all the "A" players are stretched across more files than if we had just sorted by name alone. This is where z-ordering can be useful.

Bottom line, to get the best advantage of sorting you need to understand the types of questions your end users are asking so you can have the data sorted to lean into their questions effectively.

Z-Order

There are times when multiple fields are equally a priority when querying a table, this is where a z-order sort may be quite helpful. With a z-order sort you are sorting the data by multiple data points which allows engines a greater ability to reduce the files scanned in the final query plan. Let's imagine you're trying to locate item Z in a 4 by 4 grid.

[Title to come]

Z-order sorting, also known as Morton order or a space filling curve, is a method of transforming multi-dimensional data into one dimension while preserving the locality of the data points. This method is often used in computer graphics and spatial databases to optimize queries and reduce the time required to access related data points in multiple dimensions.

Z-order ranges are determined by interleaving the binary representations of the coordinates of the data points. For example, if we have a 2D point (3, 4) with binary representations (11, 100), the Z-order value will be 011000 in binary or 24 in decimal.

When we interleave the binary representations for a Z-order value, we need to make sure they have the same number of digits. So, we'll pad the shorter binary value with zeros on the left to match the length of the longer one. In this case, we pad (11) to become (011).

So 3 is 011, and 4 is 100.

Now we interleave them:

The first bit comes from the first bit of the binary representation of 3: 0

The second bit comes from the first bit of the binary representation of 4: 01

The third bit comes from the second bit of the binary representation of 3: 011

The fourth bit comes from the second bit of the binary representation of 4: 0110

The fifth bit comes from the third bit of the binary representation of 4: 01100

The sixth bit is 0, as there is no corresponding bit in the binary representation of 3: 011000

So, the Z-order value in binary is 011000, which is 24 in decimal.

Z-order sorting is advantageous when dealing with range queries in multi-dimensional data. For instance, in a database with geographic data, queries for data points within a certain area can be optimized using Z-order sorting. It's also beneficial in quadtree or octree structures, used in 2D and 3D graphics respectively, as it aids in efficient traversal and rendering of objects.

However, Z-order is not ideal for all circumstances. In databases, if the queries are not range-based or do not involve multiple dimensions, single dimension sorting would be more efficient. Also, Z-order can lose effectiveness as you increase the number of dimensions. Usually two or three dimensions is an ideal use case for z-order. However, for high-dimensional data (for instance, in the range of 10 or more dimensions), other methods such as KD-trees, R-trees or other space partitioning or indexing strategies might be more effective.

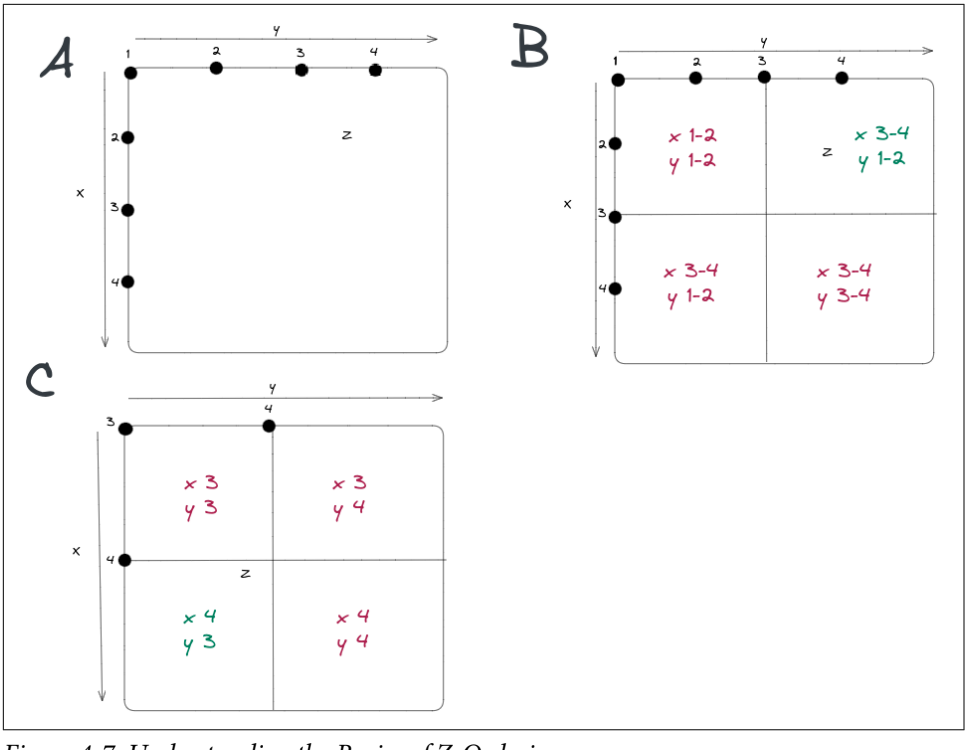


Figure 4-7. Understanding the Basics of Z-Ordering.

Referring to diagram 4-7 (A) we have a value (z) located and we want to narrow the area we want to search. We can narrow down our search by breaking down the field into four quadrants based on ranges of X & Y values as demonstrated by diagram 4-7 (B).

So if we know what quadrant we are looking for we can eliminate ¾ quadrants from our search, we can then take that quadrant and break it down even further and apply another z-order sort to the data in the quadrant like in figure 4-7 (C).

Since there are multiple factors our search is based on (X & Y), we could eliminate 75% of searchable area by taking this approach. So, the idea is we sort and cluster our data in the data files in a similar way. For example, let's say we have a dataset of all people involved in a medical cohort study, and we are trying to organize outcomes in the cohort by age and height; zordering the data may be quite worthwhile. See this in action in diagram 4-8.

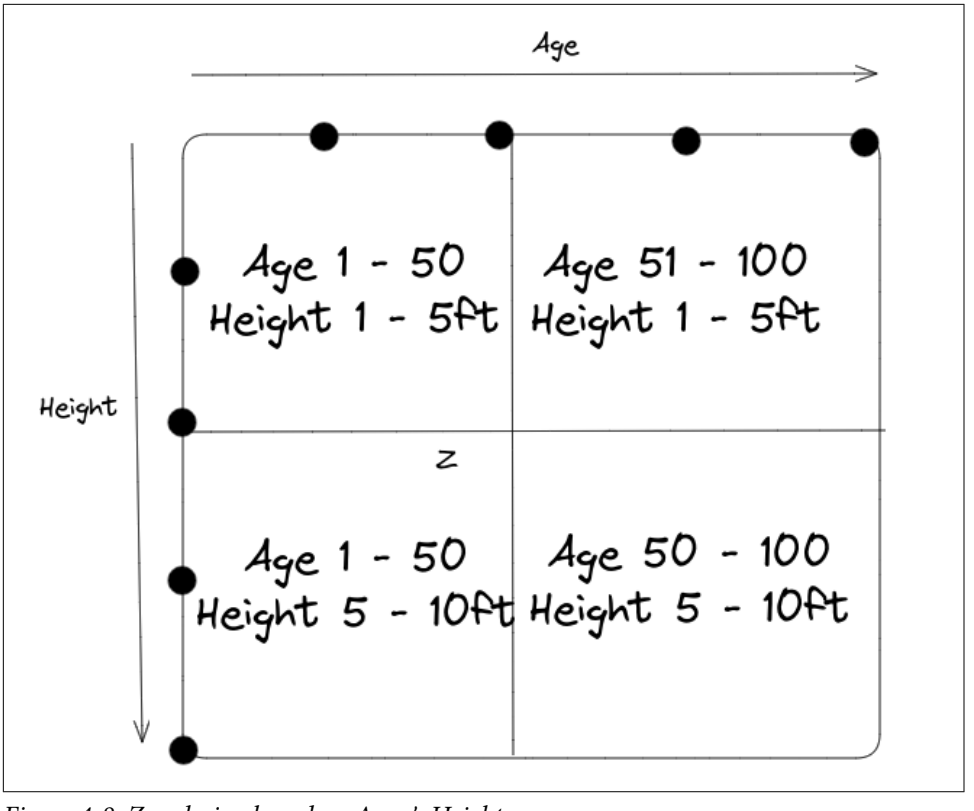


Figure 4-8. Z-ordering based on Age & Height.

Data that falls into a particular quadrant will be in the same data files, which can really slim down files to scan as we try to run analytics on different age/height groups. If you are searching people with a height of 6ft and age of 60, you could immediately eliminate the data files that have data that belongs in the other 3 quadrants.

This works because data files will fall into four categories:

- A: File with records where Age 1-50 and Height 1-5
- B: File with records where Age 51-100 and Height 1-5
- C: File with records where Age 1-50 and Height 5-10
- D: File with records where Age 51-100 and Height 5-10

If the engine knows we are searching for someone aged 60 with a height of 6, as it uses the Apache Iceberg metadata to plan the query all the data files in categories A,B, and C will be eliminated and never scanned.

Achieving this would involve running a compaction job like this:

```
CALL catalog.system.rewrite_data_files(
  table => 'people',
  strategy => 'sort',
  sort_order => 'zorder(age,height)'
)
```

Using the sort & z-order compaction strategies now only allows us to reduce the number of files our data exists in but make sure the order of the data in those files enable even more efficient query planning.

Partitioning

Sorting works great but there are a few issues with:

- The data becomes unsorted as new data is ingested, and new data is somewhat scattered until the next compaction job can cluster the data across multiple jobs. This happens because any new data is added to a new file that may be sorted among those records but not in the context of all previous records.
- A file can still have data for multiple values of the sorted field in the same data file, which isn't ideal if queries typically only need data with a particular value. Referring to our earlier example, we still had files with data of Lions and Packers players in the same file. If only looking for Lions players, scanning those Packers records are inefficient.

If you know a particular field is pivotal to how the data is accessed, you may want to go beyond sorting into partitioning. When a table is partitioned, instead of just sorting the order based on a field it will write records with distinct values of the target field into their own data files.

For example, in politics, you'll likely often query voter data based on a voter's party affiliation, making this a good partition field. This would mean all voters in the "Blue" party will be listed in distinct files from those in "Red, "Yellow" and "Green" parties. If you were to query for voters in the "Yellow" party then none of the data files you scan would include anyone from any other parties. You can see this illustrated in diagram 4-9.

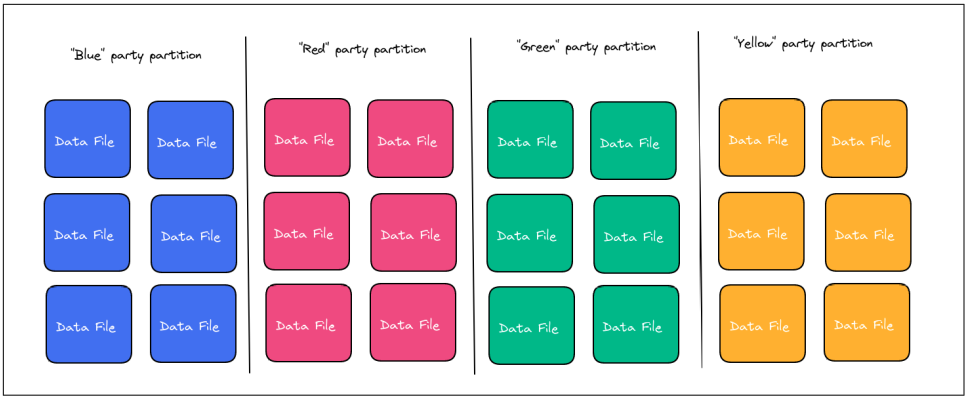


Figure 4-9. Partitioning and group data into groups of files.

Traditionally, partitioning a table based on derived values of a particular field required creating an additional field that had to be maintained separately and required users to have knowledge of that separate field when querying. For example:

- Partitioning by day, month, year on a timestamp column required you to create an additional column based on the timestamp expressing the year, month, or day in isolation.
- Partitioning by the first letter of a text value would require you to create an additional column that only had that letter
- Partitioning into buckets (a set number of divisions to evenly distribute records into based on a hash function) you'd have to create an additional column that stated which bucket the record belonged in.

You'd then set the partitioning at table creation to be based on that derived fields, and the files would be organized into subdirectories based on their partition.

```
--Spark SQL
CREATE TABLE MyHiveTable (...) PARTITIONED BY month;
```

We'd have to manually transform the value every time we inserted records.

```
INSERT INTO MyTable (SELECT MONTH(time) AS month, ... FROM data_source);
```

When querying the table, the engine had no awareness of the relationship between the original field and the derived field. This would mean that the following query would benefit from partitioning:

```
SELECT * FROM MYTABLE WHERE time BETWEEN '2022-07-01 00:00:00' AND '2022-07-31
00:00:00' AND month = 7;
```

However, users often aren't aware of this workaround column (and shouldn't have to be). Meaning most of the time, users would issue something like the following query which would result in a full table scan, making the query take much longer to complete and consume much more resources.

```
SELECT * FROM MYTABLE WHERE time BETWEEN '2022-07-01 00:00:00' AND '2022-07-31
00:00:00';
```

The latter query is more intuitive for a business user or data analyst using the data as they may not be as aware of the internal engineering of the table, resulting in many accidental full table scans. This is where Iceberg's hidden partitioning capability comes in.

Hidden Partitioning

Apache Iceberg handles partitioning quite differently, addressing many of these pain points when optimizing your tables with partitioning. One resulting feature of this approach is called *Hidden partitioning*.

It starts with how Apache Iceberg tracks partitioning. Instead of tracking it by relying on how files are physically laid out, Iceberg tracks the range of partition values at the snapshot and manifest level, allowing for many levels of new flexibility.

- Instead of having to generate additional columns to partition based on transform values you can use built in *transforms* which engines and tools can apply when planning queries from the metadata.
- Since you don't need an additional column when using these transforms, you store less in your data files.
- Since the metadata allows the engine to be aware of the transform on the original column, you can filter solely on the original column and get the benefit of partitioning.

That means if you create a table partitioned by month like:

```
CREATE TABLE catalog.MyTable (...) PARTITIONED BY months(time) USING iceberg;
```

The following query would benefit from partitioning:

```
SELECT * FROM MYTABLE WHERE time BETWEEN '2022-07-01 00:00:00' AND '2022-07-31
00:00:00';
```

As you may have seen in the prior CREATE TABLE statement, transforms are used by applying them like a function on the target column being transformed. There are several transforms available when planning your partitioning.

1. *years*

2. *months*

3. *days*

4. *Hours*

   The YEAR, MONTH, DAY and HOUR transforms work on a timestamp column. Keep in mind, if you specify MONTH, it'll track the month and year of the timestamp and if you use DAY it'll track the year, month and day of the timestamp so there is no need to use multiple transforms for more granular partitioning.

5. *Truncate*

   The TRUNCATE transform partitions the table based on the truncated value of a column, for example if you wanted to partition a table based on the first letter of a persons name you could create a table like so:

   ```
   CREATE TABLE catalog.MyTable (...) PARTITIONED BY truncate(name, 1)
   USING iceberg;
   ```

6. *bucket*

   The BUCKET transform is perfect for partitioning based on a field with high cardinality (lots of unique values). The BUCKET transform will use a HASH function to distribute the records across a specified number of buckets. So, for example, maybe you want to partition voter data based on zip codes, but there are so many possible zip codes. It would result in too many partitions with small data files, so you could run something like the following:

   ```
   CREATE TABLE catalog.voters (...) PARTITIONED BY bucket(24, zip) USING
   iceberg;
   ```

   Any bucket will have several zip codes included, but at least if you look for a particular zip code you are not doing a full table scan, just a scan of the bucket that includes the zip code you're searching for. So, with Apache Iceberg's hidden partitioning we have a more expressive way to express common partitioning patterns. Taking advantage of them requires no additional thought from the end user than to filter by the fields they'd naturally filter by.

Partition Evolution

Another challenge with traditional partitioning is that since it relied on the physical structure of the files being laid out into subdirectories, changing how the table was partitioned required you to rewrite the entire table. This becomes an inevitable problem as data and query patterns evolve, necessitating a rethinking of how we partition and sort the data.

Apache Iceberg solves this problem with its metadata-tracked partitioning as well, because the metadata not only tracks partition values but historical partition schemes allowing for the evolution of the partition scheme. So if the data in two different files were written based on two different partition schemes, the Iceberg metadata would make the engine aware so it can create a plan with partition scheme A separately from that with partition scheme B creating an overall scan plan at the end.

For example, let's say you have a table of membership records partitioned by the year someone registered as a member.

```
CREATE TABLE catalog.members (...) PARTITIONED BY years(registration_ts) USING ice-
berg;
```

Then, several years later the pace of membership growth made it worthwhile to start breaking them down by month. You could alter the table to adjust the partitioning like so:

```
ALTER TABLE catalog.members ADD PARTITION FIELD months(registration_ts)
```

The neat thing about Apache Iceberg's date-related partition transforms is that if you evolve to something granular, there is no need to remove the less granular partitioning rule. Although, if using bucket or truncate and you decide you no longer wanted to partition the table by a particular field you can do so like this:

```
ALTER TABLE catalog.members DROP PARTITION FIELD bucket(24, id);
```

Although you should be careful about dropping partition fields as it can affect the results of queries to the metadata tables (metadata tables discussed in X.X.X).

When a partitioning scheme is updated, it only applies to new data written to the table going forward so there is no need to rewrite the existing data, but also keep in mind any data rewritten by the rewriteDataFiles procedure will be rewritten using the new partitioning scheme, so if you want to keep older data in the old scheme make sure to use the proper filters in your compaction jobs to not rewrite it.

Other Partitioning Considerations

If you migrate an Hive table using the migrate procedure (discussed in X.X.X) it may currently be partitioned on a derived column, but you want to express to Apache Iceberg that it should use an Iceberg transform instead, there is a REPLACE PARTITION command.

```
ALTER TABLE catalog.members REPLACE PARTITION FIELD registration_day WITH days(reg-
istration_ts) AS day_of_registration;
```

This will not alter any data files but allow the metadata to track the partition values using Iceberg transforms.

Optimizing tables can be done many ways such as using partitioning to write data with unique values to unique files, sorting the data in those files, and then making sure to compact those files into fewer larger files will keep your table performance nice and crisp. Although it's not always about general use optimization, there are particular use cases like row-level updates and deletes we can optimize for as well using Copy-on-write and Merge-on-read.

Copy-on-Write vs Merge-on-read

Another consideration when it comes to the speed of your workloads is how you handle row level updates. When you are adding new data, it just gets added to a new data file but when you want to update pre-existing rows to either update or delete them there are some considerations:

- In data lakes, and therefore in Apache Iceberg, data files are immutable meaning they can't be changed. This provides lots of benefits, such as the ability to achieve snapshot isolation. (Since files old snapshots refer to will have consistent data)
- If you're updating 10 rows, there is no guarantee they are in the same file, so you may have to rewrite 10 files for the new snapshot and every row of data in them to update 10 rows.

In order to deal with this consideration there are three approaches to dealing with row-level updates covered in detail throughout this section and summarized in the chart below.

Table 4-2. [Table title to come]

Update Style	Read Speed	Write Speed	Best Practices
Copy-On-Write	Fastest Reads	Slowest Updates/Deletes	
Merge-On-Read (Position Deletes)	Fast Reads	Fast Updates/Deletes	Use regular compaction to minimize read costs
Merge-On-Read (Equality Deletes)	Slow Reads	Fastest Updates/Deletes	Use Frequent compaction to minimize read costs

Copy-on-Write

The default approach is referred to as Copy-on-Write (COW). In this approach if even a single row in a data file is updated or deleted, that data file is rewritten and the new file takes its place in the new snapshot. See this exemplified in diagram 4-10.

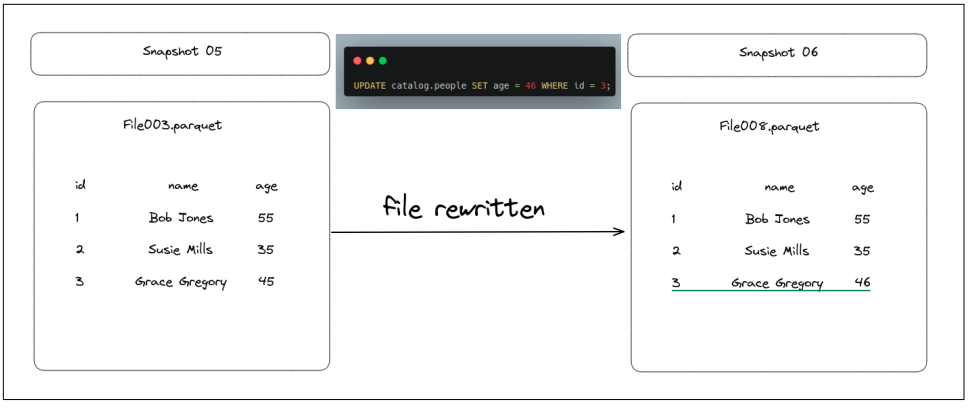


Figure 4-10. The results of using Copy-on-Write for updating a single row..

This is ideal if you're optimizing for reads because read queries can just read the data without having to reconcile any deleted or updated files. Although, if your workloads consist of very regular row-level updates, rewriting entire data files for those updates may slow down your updates beyond what your SLAs allow.

Table 4-3. [Table caption to come]

PROS	CONS
Faster Reads	Slower Row Level Updates/Deletes

Merge-on-Read

The slternative to Copy-on-Write is Merge-on-Read, where instead of rewriting an entire data file, you capture the records to be updated in the existing file in a delete file, which tracks which records should be ignored.

If you are deleting a record:

- The record is listed in a delete file
- When a reader reads the table they will reconcile the data file with the delete file

If you are updating a record:

- The record to be updated is tracked in a delete file
- A new data file is created with only the updated record
- When a reader reads the table they will ignore the old version of the record because of the delete file and use the new version in the new data file.

See this in action in diagram 4-11.

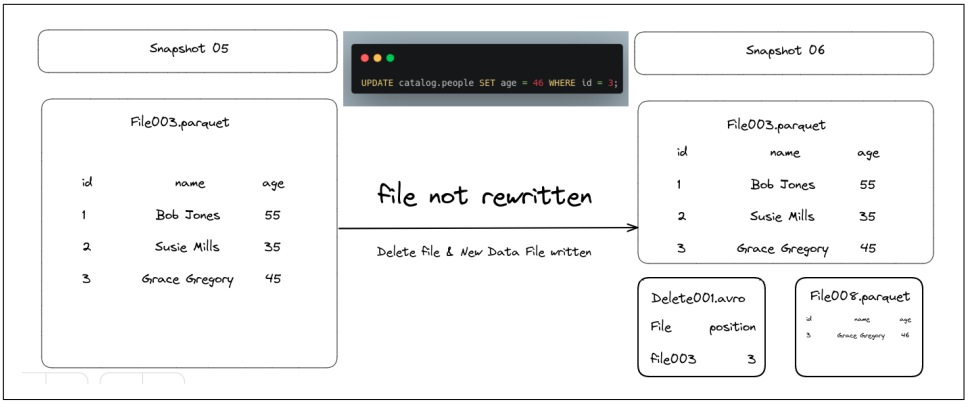


Figure 4-11. The results of using Merge-on-Read for updating a single row.

This avoids the need to rewrite unchanged records to new files just because they exist in a data file with a record to be updated, speeding up the write transaction. But it comes at the cost of slower reads as reads have to read the delete files to know which records to ignore in the proper data files.

To minimize the cost to reads you'll want to run regular compaction jobs, and to keep those compaction jobs running efficiently, let's take advantage of some of the properties we learned before:

- Use a fitler/where clause to only run compaction on the files ingested in last time frame (hour, day).
- Use partial progress mode to make commits as file groups are rewritten so readers can start seeing marginal improvements sooner than later.

Using these techniques, you can speed up the write side of heavy update workloads while minimizing the costs to read performance.

Table 4-4. [Table caption to come]

PROS	CONS
Faster Row/Level Updates	Slower Reads due to reconciling of delete files

Types of Delete Files

When doing Merge-on-Read writes, delete files enable us to track which records need to be ignored in existing data files for future reads. Let's use an analogy to understand the high-level of what these two types of delete files do for us.

When you've got a ton of data and you want to kick out a specific row, you've got a couple of choices. First, you can find it by where it's sitting in the dataset, kind of like

finding your friend in a movie theater by their seat number. Or, you can look for it based on what it's made of – like picking out your friend in a crowd because they're wearing a bright red hat.

So, when you want to delete stuff, you've got two types of files you can use. If you're going for the first option, you're gonna use what they call "positional delete files". But if you're all about the second method, you're gonna need "equality delete files".

Now, these two methods each have their own strengths and weaknesses. That means depending on the situation, you might wanna pick one over the other. It's all about what works best for you!

So now let's explore the two types of delete files–position deltes and equality deletes.

Position deletes track what rows in which files should be ignored, the table below is example of how this data is laid out in a position delete file.

Table 4-5. [Table caption to come]

Row to Delete (Position Deletes)	
File Path	**Position**
001.parquet	0
001.parquet	5
006.parquet	5

So when reading the specified files it'll skip the row at the specified position. This requires a lot less of a cost at read time since it has a pretty specific point at which it must skip a row. However, this has write time costs since the writer of the delete file will need to know the position of the deleted record which requires it to read the file with the deleted records to identify those positions.

Equality deletes instead specify values that, if a record matches, should be ignored. The table below shows how the data in a equality delete file may be laid out.

Table 4-6. [Table caption to come]

Rows to Delete (Equality Deletes)	
Team	**State**
Yellow	NY
Green	MA

This requires no write time costs since you don't need to open and read files to track the targeted values, but it has much greater read time costs. The read time costs exist because there is no information where records with matching values exist, so when reading the data there has to be a comparison with every record that could possibly contain a matching record. Equality deletes are great if you need the most write speed

possible, but aggressive compaction should be planned to reconcile those equality deletes to reduce the impact on your reads.

Configuring COW or MOR

Whether you a table is configured to handle row-level updates via COW or MOR depends on two things:

- The table properties
- Whether the engine you write to Apache Iceberg with supports merge-on-read writes

The table properties that determine whether to a particular transactions is handled via COW or MOR are the following:

write.delete.mode
: Approach to use for delete transactions

write.update.mode
: Approach to use for update transactions

Write.merge.mode
: Approach to use for merge transactions

Keep in mind for this and all Apache Iceberg table properties, while many are part of the specification, it is still on the specific compute engine to honor the specification. You may run into different behavior despite your properties based on the engine, so read up on which table properties are honored by engines you use for particular jobs that use those properties. Query engine developers will have every intention of honoring all Apache Iceberg table properties, but it does require implementations for the specific engines architecture. Over time engines should have all these properties honored so you get the same behavior across all engines.

Since the Apache Spark support for Apache Iceberg is handled from within the Apache Iceberg project all these properties are honored from within Spark and can be set at the creation of a table in Spark like so:

```
CREATE TABLE catalog.people (
    id int,
    first_name string,
    last_name string
) TBLPROPERTIES (
    'write.delete.mode'='copy-on-write',
    'write.update.mode'='merge-on-read',
    'write.merge.mode'='merge-on-read'
) USING iceberg;
```

This property can also be set after the table is created using an ALTER TABLE statement:

```
ALTER TABLE catalog.people SET TBLPROPERTIES (
    'write.delete.mode'='merge-on-read',
    'write.update.mode'='copy-on-write',
    'write.merge.mode'='copy-on-write'
);
```

It's assimple as that. But, remember when working with non-Apache spark Engines:

- Table properties may or may not be honored, it's up to the engine to implement support
- When using Merge-on-Read, make sure the engines you use to query your data can read delete files.

Other Considerations

Beyond your data files and how they are organized, there are many levers for improving performance. We will discuss many of them in the following sections.

Metrics Collection

As seen in Chapter 3, the manifest for each group of data files is tracking metrics for each field in the table to help with min/max filtering and other optimizations. The types of column level metrics that are tracked include.

- Counts of values, null-values, distinct values
- Upper and Lower bound values

If you have very wide tables (tables with lots of fields, e.g., 100+) the amount of metrics being tracked can start to become a burden on reading your metadata (too much of a good thing).

Fortunately, using Apache Iceberg's table properties, you can fine tune which column has its metrics tracked and which columns won't. This way, you can track metrics on columns that are often filtered by and not capture metrics on ones that aren't, so their metric data doesn't bloat your metadata.

You can tailor the level of metrics collection for the columns you want (don't need to specify all of them) using table properties like so:

```
ALTER TABLE catalog.db.students SET TBLPROPERTIES (
    'write.metadata.metrics.column.col1'='none',
    'write.metadata.metrics.column.col2'='full',
    'write.metadata.metrics.column.col3'='counts',
```

```
    'write.metadata.metrics.column.col4'='truncate(16)',
);
```

As you can see you can set how the metrics are collected for each individual column to several potential values:

None
: Don't collected any metrics

Counts
: Only collect counts (values, distinct values, null values)

Truncate(XX)
: Counts and Truncate the value to a certain number of characters and base the upper/lower bounds on that.

Full
: Counts and Upper/Lower bounds based on the full value.

You don't need to set this explicitly for every column as, by default, iIceberg sets this to truncate(16).

Rewriting Manifests

Sometimes the issue isn't your data files, they are a good sized with well sorted data. But, they've been written across several snapshots, so an individual manifest could be listing more of these files. While manifests are more lightweight, more manifests still means more file operations, there is a separate rewriteManifests procedure to rewrite ONLY the manifest files so you have a smaller total number of manifest files, and those manifest files list a large number of data files.

```
CALL catalog.system.rewrite_manifests('MyTable')
```

If you run into any memory issues while running this operation, you can turn off Spark caching by passing a second argument. If you are rewriting lots of manifests and they are being cached by spark it could result in issues with individual executor nodes, this can also be solved by turning off Spark caching.

```
CALL catalog.system.rewrite_manifests('MyTable', false)
```

When it would be a good to run this operation is a matter of when your data file sizes are optimal, but the number of manifest files aren't. For example, if you have 5gb of data in one partition split among 10 data files, but these files are listed within five manifest files, you don't need to rewrite the data files, but you can probably consolidate listing the ten files in one manifest.

Optimizing Storage

As you make updates to the table or run compaction jobs, new files are created, but old files aren't being deleted since those files are associated with historical snapshots of the table. To prevent storing a bunch of unneeded data you should periodically expire snapshots. Keep in mind, you cannot time travel to an expired snapshot. During expiration any files associated with the expired snapshots found not to be associated with any valid snapshots will get deleted.

You can expire snapshots that were created on or before a particular timestamp:

```
CALL catalog.system.expire_snapshots('MyTable', TIMESTAMP '2023-02-01
00:00:00.000', 100)
```

The second argument is a minimum number of snapshots to retain (by default it will retain the last 5 days of snapshots), so it will only expire snapshots that are on or before the timestamp, but if the snapshot falls within the 100 most recent snapshots it will not expire.

You can also expire particular snapshot IDs:

```
CALL catalog.system.expire_snapshots(table => 'MyTable', snapshot_ids => ARRAY(53))
```

In this example, a snapshot with the particular ID of 53 is expired. We can look up the snapshotID by either opening up the metadata.json and examining its contents or we can use the metadata tables which are detailed in Chapter 6. You may have a snapshot where you expose sensitive data by accidence and what to just expire that single snapshot to clean up the datafiles created in that transaction, this would give you that flexibility.

There are six arguments that can be passed to the expire_snapshots procedure:

table
: The table to run the operation on

older_than
: Expire all snapshots on or before this timestamp

retain_last
: Minimum number of snapshots to retain

snapshot_ids
: Specific snapshot ids to expire

max_concurrent _deletes
: Number of threads to use for deleting files

stream_results
: Deletion files will be sent to Spark driver by RDD partition when true, useful for avoiding OOM (Out of Memory) issues when deleting large files.

Another consideration when optimizing storage are what is called an Orphan Files, these are files and artifacts that accumulate in the table's data directory but are not tracked in the metadata tree because they were written by failed jobs. These files will not be cleaned up by expiring snapshots, so a special procedure should sporadically be run to deal with this. This procedure will look at every file in your table's default location and assess whether it relates to active snapshots. This can be an intensive process (which is why you should only do it sporadically). To delete orphan files run a command like the following:

```
CALL catalog.system.remove_orphan_files(table => 'MyTable')
```

The arguments you can pass to the removeOrphanFiles procedure are the following:

table
: the table to operate on

older_than
: only delete files created on or before this timestamp

location
: Where to look for orphan files, defaults to tables default location.

dry_run
: Boolean if true, won't delete files just return a list of what would be deleted.

max_concurrent_deletes
: List the max number of threads for deleting files.

While for most tables, the data will be located in its default location there are times you may add external files via the addFiles procedure (covered in X.X.X) and later may want to clean artifacts in these directories. This is where the location argument comes in.

Write Distribution Mode

This requires an understanding of how MPP (Massively Parallel Processing) systems handle writing files. These systems distribute the work across several nodes, each doing a job or task. The write distribution is how the records to be written are distributed across these tasks. If no specific write distribution mode is set, data will be distributed arbitrarily. The first X number of records will go to the first task, the next X number to the next task.

Each task is processed separately, so that task will create at least one file for each partition it has at least one record for. So if you have ten records that belong in partition A distributed across ten tasks, you will end up with ten files in that partition with one record each, which isn't ideal.

It would be better if all the records for that partition were allocated to the same tasks so they can be written to the same file. This is where the write distribution comes in–how the data is distributed among tasks. There are three options.

None
: No special distribution, this is the fastest during write time and ideal for pre-sorted data.

Hash
: The data is hash distributed by partition key

Range
: The data is range distributed by partition key or sort order

In a hash distribution, the value of each record is put through a hash function and grouped together based on the result. Multiple values may end up in the same grouping based on the hash function. For example, if you have the values 1, 2, 3, 4, 5 and 6 in your data you may get a hash distribution of data with 1 and 4 in task A, 2 and 5 in task B then 3 and 6 in task C. You'll still write the least amount of files needed for all your partitions but less sequential writing.

In a range distribution, the data is sorted and distributed, so you'd likely have values 1 and 2 in task A, 3 and 4 in task B, and 5 and 6 in task C. This sorting will be done by the partition value or by the SortOrder if the table has one. So if a SortOrder is specified data will be not just grouped into tasks by partition value but also by value of the SortOrder field. This is ideal for data that can benefit from clustering on certain fields. Although, sorting the data for distribution sequentially has more overhead than throwing them in a hash function and distributing them based on the output.

There is also a write distribution property to specify the behavior for deletes, updates and merges.

```
ALTER TABLE catalog.MyTable SET TBLPROPERTIES (
    'write.distribution-mode'='hash',
    'write.delete.distribution-mode'='none',
    'write.update.distribution-mode'='range',
    'write.merge.distribution-mode'='hash',
);
```

In a situation where you are regularly updating many rows but rarely deleting rows you may want to have different distribution modes as a different distribution mode may be more advantageous depending on your query patterns. Bottom line,

if delete, merge or update transactions occur infrequently you are better off with copy-on-write for that type of transaction so you have less to compact during future compaction.

Object Storage Considerations

Object storage is a unique take on storing data. Instead of keeping files in a neat folder structure like a traditional file system, object storage tosses everything into what we call 'buckets'. Each file becomes an 'object' and gets a bunch of metadata tagged along with it. This metadata tells us all sorts of stuff about the file.

Now, when you want to grab a file from object storage, you're not clicking through folders. Instead, you're using APIs. Just like you'd use a GET or PUT request to interact with a website, you're doing the same here to access your data. For example, you'd use a GET request to ask for a file, the system checks the metadata to find the file, and voila, you've got your data.

This API-first approach helps the system juggle your data, like making copies in different places or dealing with loads of requests at the same time. Object storage, which most cloud vendors provide, is ideal for data lakes and data lakehouses, but has one potential bottleneck.

Because of the architecture and how object stores handle parallelism, there is often limits on how many requests can go to files under the same "prefix", so if I wanted to access /prefix1/fileA.txt and /prefix1/fileB.txt, even though they are different files accessing both count towards the limit on prefix1. This becomes a problem in partitions with lots of files as queries can result in many requests to these partitions then run into throttling slowing down the query.

Running compaction to limit the number of files in a partition can help, but Apache Iceberg is uniquely suited for this scenario since it doesn't rely on how its files are physically laid out, meaning it can write files in the same partition across many prefixes.

You can enable this in your table properties like so:

```
ALTER TABLE catalog.MyTable SET TBLPROPERTIES (
    'write.object-storage.enabled'= true
);
```

This will distribute files in the same partition across many prefixes, including a hash to avoid potential throttling.

So, instead of:

```
s3://bucket/database/table/field=value1/datafile1.parquet
s3://bucket/database/table/field=value1/datafile2.parquet
s3://bucket/database/table/field=value1/datafile3.parquet
```

```
s3://bucket/database/table/field=value1/datafile4.parquet
s3://bucket/database/table/field=value2/datafile5.parquet
s3://bucket/database/table/field=value2/datafile6.parquet
s3://bucket/database/table/field=value2/datafile7.parquet
s3://bucket/database/table/field=value2/datafile8.parquet
```

You'll get this:

```
s3://bucket/4809098/database/table/field=value1/datafile1.parquet
s3://bucket/5840329/database/table/field=value1/datafile2.parquet
s3://bucket/2342344/database/table/field=value1/datafile3.parquet
s3://bucket/423423/database/table/field=value1/datafile4.parquet
s3://bucket/234234/database/table/field=value2/datafile5.parquet
s3://bucket/543454/database/table/field=value2/datafile6.parquet
s3://bucket/876456/database/table/field=value2/datafile7.parquet
s3://bucket/133654/database/table/field=value2/datafile8.parquet
```

With the hash in the file path, each file in the same partition is now treated as if it were under a different prefix avoiding throttling.

Data File Bloom Filters

A bloom filter is a way of knowing whether a value possibly exists in a data set. Imagine a line-up of bits (those 0s and 1s in binary code) all set to a length we decide. Now, when we add data to our dataset, we run each value through a process called a hash function. This function spits out a spot on our bit lineup, and we flip that bit from 0 to 1. This flipped bit is like a flag that says, "Hey, a value that hashes to this spot might be in the dataset."

For example, let's say we feed 1000 records through a bloom filter that has 10 bits. When it's done, our bloom filter might look like this:

```
[0,1,1,0,0,1,1,1,1,0]
```

Now, let's say we want to find a certain value, let's call it X. We put X through the same hash function, and it points us to spot number 3 on our bit lineup. According to our bloom filter, there's a '1' in that 3rd spot. This means there's a chance our value X could be in the dataset because a value hashed to this spot before. So, we go ahead and check the dataset to see if X is really there.

Now let's say you're looking for a different value, we'll call it Y. When we run Y through our hash function, it points us to the 4th spot on our bit lineup. But our bloom filter has a '0' there, which means no value hashed to this spot. So, we can confidently say that Y is definitely not in our dataset, and we can save time by not digging through the data.

Bloom filters are handy because they can help us avoid unnecessary data scans. If we want to make them more precise, we can add more hash functions and bits. But

remember, the more we add, the bigger our bloom filter gets, and it needs more space. As with most things in life, it's a balancing act. Everything is a trade off.

We can enable the writing of bloom filters for a particular column in our parquet files (this can also be done for ORC files) via our table properties:

```
ALTER TABLE catalog.MyTable SET TBLPROPERTIES (
    'write.parquet.bloom-filter-enabled.column.col1'= true,
    'write.parquet.bloom-filter-max-bytes'= 1048576
);
```

Then engines querying your data may take advantage of these bloom filters to help make reading the data files even faster by skipping data files where bloom filters clearly indicate data needed doesn't exist.

About the Authors

Tomer Shiran is the Founder and Chief Product Officer of Dremio, an open data lakehouse platform that enables companies to run analytics in the cloud without the cost, complexity and lock-in of data warehouses. As the company's founding CEO, Tomer built a world-class organization that has raised over $400M and now serves hundreds of the world's largest enterprises, including 3 of the Fortune 5. Prior to Dremio, Tomer was the 4th employee and VP Product of MapR, a Big Data analytics pioneer. He also held numerous product management and engineering roles at Microsoft and IBM Research, founded several websites that have served millions of users and hundreds of thousands of paying customers, and is a successful author and presenter on a wide range of industry topics. He holds an MS in Computer Engineering from Carnegie Mellon University and a BS in Computer Science from Technion - Israel Institute of Technology.

Jason Hughes is the Director of Technical Advocacy at Dremio. Previously at Dremio, he's been a Product Director, Technical Director and a Senior Solutions Architect. He's been working in technology and data for over a decade, including roles as tech lead for the field at Dremio, the pre-sales and post-sales lead for Presto and QueryGrid for the Americas at Teradata, and leading the development, deployment, and management of a custom CRM system for multiple auto dealerships. He is passionate about making customers and individuals successful and self-sufficient. When he's not working, he's usually taking his dog to the dog park, playing hockey, or cooking (when he feels like it). He lives in San Diego, California.

Alex Merced is a developer advocate for Dremio and has worked as a developer and instructor for companies like GenEd Systems, Crossfield Digital, CampusGuard and General Assembly. Alex is passionate about technology and has put out tech content on outlets such as blogs, videos and his podcasts Datanation and Web Dev 101. Alex Merced has contributed a variety of libraries in the Javascript and Python worlds including SencilloDB, CoquitoJS, dremio-simple-query and more.

Dipankar Mazumdar is currently a Data Eng/Science Advocate at Dremio where his primary focus is advocating data practitioners on Dremio's open lakehouse platform and various open-sourced projects, such as Apache Iceberg. Dipankar is also interested in Visual Analytics research, and his latest work was on "Explainability of ensemble models" using multidimensional projection techniques.