O'REILLY®
Technical Guide

Optimizing Your Apache Iceberg Lakehouse

Improving Performance & Scalability

Early Release
RAW & UNEDITED

Lester Martin

Starburst
Rocket fuel for AI

Starburst Icehouse Architecture

A single data foundation for AI & Enterprise Intelligence.

Powered by Iceberg

Built on Trino

Unified for faster insights across your business

Learn more at starburst.io

Optimizing Your Apache Iceberg Lakehouse

Improving Performance and Scalability

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

Lester Martin

O'REILLY®

Optimizing Your Apache Iceberg Lakehouse
by Lester Martin

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

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
Cover Designer: TO COME
Cover Illustrator: TO COME
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

November 2026: First Edition

Revision History for the Early Release
2026-04-22: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341673946* for release details.

979-8-341-67392-2

[LSI]

Table of Contents

Brief Table of Contents (*Not Yet Final*)

CHAPTER 1

Exploring the Apache Iceberg Architecture

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's 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'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *gobrien@oreilly.com*.

It's time to dig deeper into the architecture underlying Apache Iceberg. Like all data lakehouse table formats, Iceberg is built on the model of collocating many large files with the same file format and logical structure in a repository, accessed as if they were a traditional RDBMS table. Unlike RDBMS technologies, data lakehouses clearly separate storage from compute. A repository full of data files and scalable processing capacity is not enough for lakehouses; we need metadata to complete the picture.

That metadata is persisted as files on the data lake repository alongside the data files, which are used when querying an Iceberg table. This chapter will explore the fundamental architecture of how that metadata is represented, why it is valuable, how it needs to be

regularly maintained, and how it enables interoperability among multiple compute engines.

Metadata and Iceberg

Rich metadata is the key to Iceberg's efficient use of object stores, allowing committable transactions, and the benefits of time-travel and roll backs that come with the versioning approach used for data modifications. This metadata is stored alongside the data files themselves; under the root location of an Iceberg table as conceptually visualized in Example 1-1.

Example 1-1. Data and metadata directory locations.

```
///my_schema/my_table ❶
                    /metadata ❷
                             /md_file_A
                             /md_file_B
                    /data ❸
                             /file_1
                             /file_2
                             /file_3
```

❶ Base location of the Iceberg table on the data lake repository.

❷ Subfolder where the metadata layer's files are stored.

❸ Subfolder containing the data files that make up the contents of the table.

This section will detail the various types of metadata files used by Iceberg, introduce catalogs, and showcase how performance is enhanced by the ability to leverage metadata before retrieving any data files.

Why Metadata is Key

Compute engines like Trino must gather some amount of metadata to start understanding the design and content of the table. Some of this metadata information is simply the location of the data files that make up a table and what kind of file format is being used. This answers the "where is it" and "what is it" questions and is part of what is referred to as *structural metadata*. Additional structural metadata includes the schema of the table itself, the column names,

and their data types. This gives us the "what does it look like" information. Armed with the answers to these three important questions, compute engines can then start executing SQL operations against an Iceberg table.

Structural metadata is critical to core functionality, but there is additional metadata information that goes beyond this as well. *Statistical metadata* provides the opportunity to improve performance and scalability. At the lowest level, much of this statistical, or summary, information is stored as metadata within the data files themselves. This is especially true when using an analytics-oriented columnar file format such as Apache Parquet.

For example, Figure 1-1 simplifies a table made up of three Parquet data files. Parquet files contain metadata to include information such as the number of rows in a particular file and the number of null values for each column.

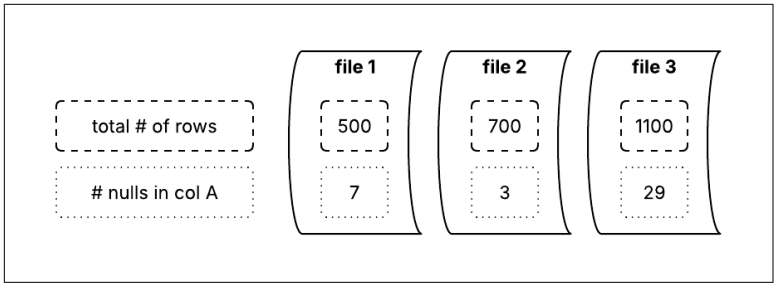

Figure 1-1. Statistical metadata stored in Parquet files.

This file-level metadata, which is stored as footer information in the data file, is available when querying to find out how many rows are present across the entire table.

```
SELECT COUNT(*) FROM my_table
-- returns 2300
```

The same is true when determining how many rows have a null value for a particular column.

```
SELECT COUNT(*) FROM my_table WHERE my_col IS null;
-- returns 39
```

Compute engines can quickly access this metadata from each file, internally calculate the final answer, and then provide results to the query submitter. These are simple examples, and definitely not all

query results can be determined by only looking at the metadata, but this information is:

- Critical to simply querying at all (structural metadata)
- Impactful to performance (statistical metadata)

> Reference Apache Parquet Metadata for more information on the metadata stored in this file type.

The metadata stored in the data files is useful to compute engines once they have determined they need to actually read records from a particular file. The Iceberg metadata files themselves are leveraged to help compute engines decide which files should be read at all. We will revisit the importance of this and functionally how it occurs once we understand the various types of metadata files.

The Relationship Between Catalogs and Metadata

The top of Figure 1-2 introduces an additional component within the architecture called a *catalog*. The catalog is a process that executes outside of the data lake. Its primary function is to link a logical table name to the metadata information existing on the data lake. With awareness of the metadata layer's files, compute engines can then determine which data files need to be retrieved and which to ignore. The metadata layer's files are stored in the `./metadata` folder as also explained in Example 1-1. The various types of files that exist in the metadata layer are detailed in this section.

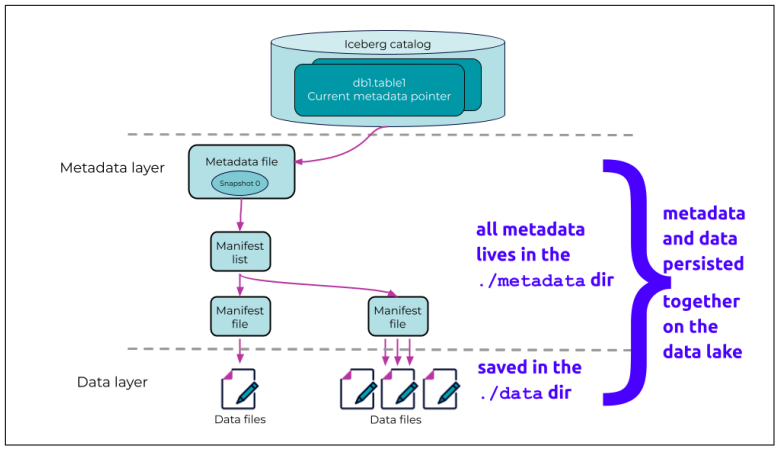

Figure 1-2. Iceberg architecture overview.

In addition to the specific location of metadata files, Iceberg catalogs maintain the organizational details of how tables are logically grouped together. This includes maintaining a list of schemas, or databases, each of which have their own list of tables. Operations such as creating & dropping schemas and tables are persisted in the catalog.

In comparison, classical RDBMSs also have their own catalogs. For example, an Oracle server maintains a list of schemas, which then have tables. PostgreSQL does the same thing, but calls that collection of tables a schema. The catalogs in these systems can only list the schemas physically present in each instance of the RDBMS services. Oracle instance A might show 10 schemas, while Oracle instance B could show 20. Schemas in instance A are unique from instance B; even if they share the same name.

Conversely, an Iceberg catalog could support many more schemas as it is not tied to a specific type, or physical instance, of a compute engine. As mentioned before, each table within a unique schema could be physically stored in a completely different data lake location, too. This allows the catalog to be more universal across the tables being represented and the compute engines needing access to the tables.

There are multiple implementations of Iceberg catalogs, including open-source and proprietary options. Some well-known and widely-used catalogs include Apache Hive Metastore (HMS), AWS Glue,

and implementations of the Iceberg REST Catalog API Specification, such as Apache Polaris.

> An Iceberg table should never be registered to more than one catalog. The smallest problem will be that compute engines will not always be getting hold of the most current metadata references. The worst problem is a condition where multiple, independent, versions of metadata are created and maintained.

The compute engine you use to access Iceberg tables will support a list of integrated catalogs that you can leverage. For example, Starburst Galaxy supports HMS, AWS Glue, Iceberg REST, Apache Polaris, a proprietary catalog, and several others. In fact, the Trino engine that Starburst Galaxy is built upon allows multiple catalog types to be selected and multiple instances of each type.

Snapshots & Manifests

Figure 1-2 reminds us that for each table, metadata exists as physical files collocated on the data lake alongside the actual data files that contain the contents of the table. This metadata layer allows tables to be versioned whenever content or structure is changed. A version is referred to as a *snapshot* which itself has a *snapshot identifier* that is unique to the table it is aligned to. Figure 1-2 also shows there are multiple types of files in the metadata layer.

Metadata file

An Iceberg catalog maintains the file name of the top-level table *metadata file* stored on the data lake. Table metadata files are stored in JSON format and the one referenced in the catalog is for the current version of the table. This is visualized by the arrow leaving the catalog in Figure 1-2. Example 1-2 shows one key bit of information stored in this table metadata file; a list of all known schema definitions that exist across all the snapshots.

Example 1-2. List of schemas.

```
"schemas": [ ❶
  {
    "type": "struct", "schema-id": 0, ❷
    "fields": [
```

```
      { "id": 1, "name": "cust_id", "type": "int" },
      { "id": 2, "name": "last_name", "type": "string" } ] },
  {
    "type": "struct", "schema-id": 1,
    "fields": [
      { "id": 1, "name": "cust_id", "type": "int" },
      { "id": 2, "name": "last_name", "type": "string" },
      { "id": 3, "name": "first_name", "type": "string" } ] }  ❸
]
```

❶ The collection of schema definitions.

❷ The first schema definition consists of two fields; `cust_id` and `last_name`.

❸ The second schema definition includes a third field; `first_name`.

Example 1-3 shows another key element in the table metadata file; the list of existing snapshots. Each snapshot indicates the schema it is aligned with.

Example 1-3. List of snapshots.

```
    "snapshots": [ {  ❶
        "sequence-number": 1,
        "snapshot-id": 7579480327941283080,
        ... additional details removed ...
        },
        "manifest-list": "s3://b/s/t/metadata/snap-123.avro",
        "schema-id": 0
    }, {
        "sequence-number": 2,
        "snapshot-id": 1959040240880497390,  ❷
        "parent-snapshot-id": 7579480327941283080,  ❸
        ... additional details removed ...
        },
        "manifest-list": "s3://b/s/t/metadata/snap-456.avro",
        "schema-id": 1  ❹
    }, {
        "sequence-number": 3,
        "snapshot-id": 894657535204501593,
        "parent-snapshot-id": 1959040240880497390,
        ... additional details removed ...
        },
        "manifest-list": "s3://b/s/t/metadata/snap-789.avro",  ❺
        "schema-id": 1
    } ]
```

❶ The collection of snapshots.

❷ The unique snapshot identifier.

❸ The parent snapshot identifier; if exists.

❹ The schema identifier for each snapshot.

❺ The file name of the *manifest list* associated with the snapshot. This file type is explained in the next subsection.

The table metadata file is stored in JSON format. It uses the naming convention of `*.metadata.json` (often referred to as simply `metadata.json`) and is stored in the `/metadata` folder shown in Example 1-1. As detailed in Example 1-4, it indicates the current snapshot identifier as well as any references to it across all of the table's metadata. These references are aligned with the *branching* and *tagging* features that are explained in Chapter 3.

Example 1-4. Identification of current snapshot.

```
"current-snapshot-id": 894657535204501593,  ❶
"refs": {  ❷
    "main": {
        "snapshot-id": 894657535204501593,
        "type": "branch"  ❸
    }
}
```

❶ The identifier of the current snapshot. Further details can be found in the snapshot information presented in Example 1-3.

❷ The list of branch and/or tag references.

❸ This single reference indicates the snapshot is present on the `branch` named `main`.

> Additional metadata exists; reference Table Metadata and Snapshots for the full schema definition of this JSON file.

Manifest list

As shown in Example 1-3, table metadata files also identify the file name of a *manifest list* for each snapshot listed. Each snapshot points to a single manifest list. This file, sometimes referred to as a *snapshot manifest*, is stored in the Apache Avro format. It is stored in the `/metadata` folder with the naming convention of `snap-*.avro`. A manifest list contains exactly that–a list of *manifest files* that make up the current snapshot's contents. The manifest list also tracks which manifest files belong together in each *partition*.

> Data lake table *partitioning* is the physical organization of large datasets into smaller-sized, more manageable directories based on a specific column value such as date, region, or category. The benefits of partitioning a table are presented in Chapter 3 and Chapter 4 identifies best practices aimed at optimizing performance.

The list of manifest files is not just a simple list of file pointers. Each entry shows summary information such as how many actual data files are being referenced, size of data, upper and lower bounds of key fields, and other metadata that can help a compute engine to process a query as efficiently as possible.

Manifest file

A *manifest file* is a pointer to one or more data files and rolled-up statistics for each. Much like the manifest list maintains summary information for all of its related manifest files, the manifest file maintains summary information for all of its related data files. These AVRO files are identifiable by their `*-m<N>.avro` naming convention. Walking the tree structure of Snapshot 0 in Figure 1-2, you can see that the content for the current version of this Iceberg table is made up of four data files.

All of the files in the metadata and data layers are immutable and cannot be modified once created. Figure 1-3 shows that additional metadata layer content is created when a new snapshot is created. Note the following:

1. The catalog pointer is changed to the new table metadata file.
2. The new table metadata file contains the list of snapshots and identifies the current as Snapshot 1.

3. Snapshot 1 is pointing to a new manifest list.
4. The new manifest list includes the prior 2 manifest files and the combined 4 data files they represent.
5. The new manifest list also includes a new manifest file which points to 2 new data files.

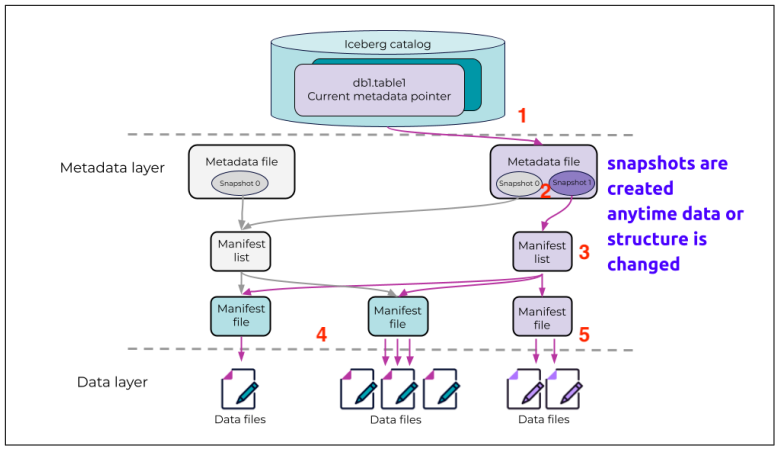

Figure 1-3. Additional metadata created with a new snapshot.

Metadata Pruning

Data lake tables are ultimately made up of many data files. Regardless of which table format, such as Apache Hive or Apache Iceberg, is being utilized, the worst-case scenario is when a compute engine needs to read all underlying data files. The general strategy is to look for opportunities to skip, or *prune*, files to be accessed.

As Figure 1-4 shows, file formats such as Parquet contain metadata that includes summary information of the records held within. It visualizes 5 data files and displays the upper and lower bounds for all customer identifier column values within each file.

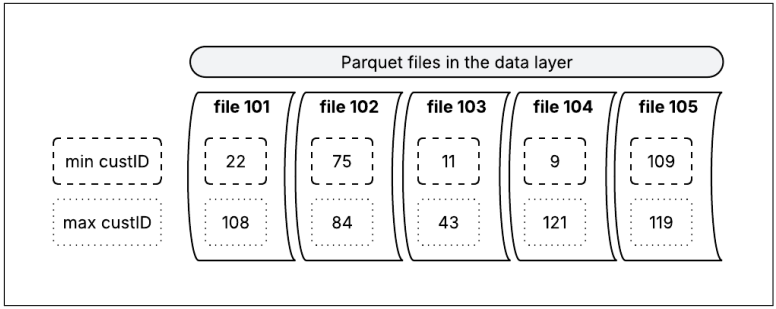

Figure 1-4. Min and max column values by data file.

Having this information stored in the footers of the data files allows a compute engine to quickly check each data file to determine whether the query conditions warrant a more thorough inspection. Table 1-1 identifies query predicates and what data files, from Figure 1-4, require more exhaustive reads to attempt to find relevant information for the query.

Table 1-1. Relevant data files based on predicate.

Query predicate	Relevant data files
`WHERE custID = 22`	Files 101, 103, 104
`WHERE custID BETWEEN 82 AND 84`	Files 101, 102, 104, 105
`WHERE custID = 12`	Files 103, 104
`WHERE custID IN (2, 122, 132)`	No files

As the third entry in Table 1-1 shows, if a query was searching for customer ID 12, then only two of the five data files warrant a more exhaustive look than just reading the summary information in the file footer. With only five data files present, you would not see much of a benefit with any clustered compute engine, but imagine if each of these five entries represented 10,000 files. In that case, having to exhaustively read 20,000 files instead of 50,000 will yield significant performance gains.

The last entry in Table 1-1 shows that with this query predicate, no files need to be considered for the results of the overarching query it is part of. That provides even more significant performance gains than any of the other 3 examples. With the original table format, Apache Hive, the compute engine has to access all 5 files for each query. The footer information has to be read to determine if a more exhaustive read is required. As before, not an issue with only 5 data

files, but a significant amount of work when there are 50,000 data files.

Apache Iceberg improves on this because summarized statistical metadata for data files is stored in the manifest file that references the data file as shown in Figure 1-5. This allows the compute engine to prune entirely any data file that is not relevant for the query.

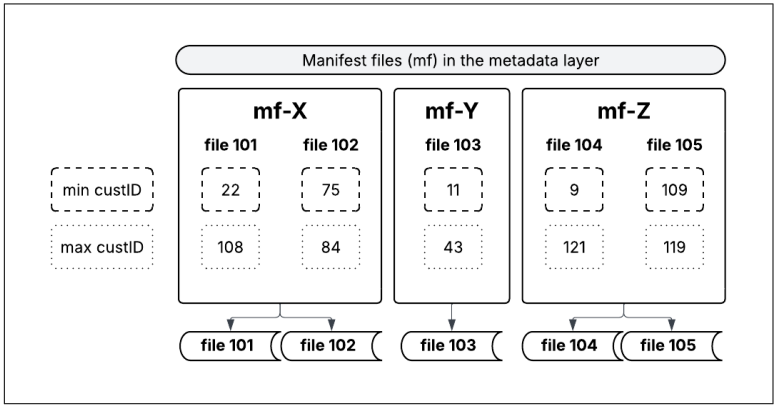

Figure 1-5. Min and max column values by manifest list.

With metadata-based *pruning*, the same data files identified in Table 1-1 are in-scope. The benefit is that the initial pass of accessing the footer information from all of the data files is eliminated. In the exaggerated example of each of these data files representing 10,000 files, that would eliminate the need to access 50,000 data files to read their footers in order to determine which files can be *pruned*.

> File pruning is a type of *predicate pushdown* which is defined as a query optimization technique that pushes filter conditions down to the data storage layer thereby reducing I/O and improving performance. It eliminates the need for a *table scan* – the need to read all records in a table.

Inspecting Metadata

As Example 1-1 visualized, if you have access to the repository location that the Iceberg table is stored in you can find these files in the `./metadata` subdirectory. Retrieving and reading these JSON and Avro files can get confusing as new snapshots are created for the

table. Fortunately, most compute engines that integrate with Iceberg feature easier ways to view a table's metadata.

The official Iceberg documentation details how to do this with Apache Spark in the Inspecting tables section. For the hands-on examples provided in this section, you will leverage the features presented in the Trino Iceberg connector documentation's section on Metadata tables which offer a SQL interface to the table's metadata.

> The SQL used in this section is available in the GitHub repository identified in Chapter 1 under the `./Chapter2` folder. All output presented in this chapter is discussed thereby allowing you the same learnings if you do not perform the hands-on exercises.

In your exploratory environment described in Chapter 1, create a schema and an empty Iceberg table.

```
CREATE SCHEMA optimize_ice.ch2;
USE optimize_ice.ch2;

CREATE TABLE my_iceberg_tbl (
  id integer,
  name varchar(55),
  description varchar(255)
) WITH (type='iceberg', format='parquet');
```

Use the query in Example 1-5 to review the contents of the `$snapshots` metadata table aligned to the newly created `my_iceberg_tbl` table.

Example 1-5. Querying the `$snapshots` metadata table.

```
SELECT
 snapshot_id, parent_id,
 substring(manifest_list,
           position('/metadata/' IN manifest_list) + 10)
  AS manifest_list
FROM
 "my_iceberg_tbl$snapshots";
```

The results from Example 1-5's query are shown in Table 1-2, and identify that a single snapshot is present, which aligns with the table creation operation.

Table 1-2. The first snapshot.

snapshot_id	parent_id	mainifest_list
9195...4485	NULL	snap-9195...4485-1...bc1b.avro

> The snapshot identifiers and names of data and metadata layer files that you will see in your exploratory environment will be different from those displayed in this section. They are independent across environments as well as for each iteration of these steps within the same environment.

Insert a few records into the table.

```
INSERT INTO my_iceberg_tbl (id, name, description)
VALUES
  (101, 'Leto', 'Ruler of House Atreides'),
  (102, 'Jessica', 'Consort of the Duke'),
  (103, 'Paul', 'Son of Leto (aka Dale Cooper)');
```

Verify that 3 rows are present as shown in Table 1-3 as a result of executing the following query.

```
SELECT * FROM my_iceberg_tbl;
```

Table 1-3. The initial 3 records.

id	name	descriptions
101	Leto	Ruler of House Atreides
102	Jessica	Consort of the Duke
103	Paul	Son of Leto (aka Dale Cooper)

Run the query again in Example 1-5 to see that there are now two snapshots present as seen in Table 1-4.

Table 1-4. The second snapshot.

snapshot_id	parent_id	mainifest_list
9195...4485	NULL	snap-9195...4485-1...bc1b.avro
1621...6722	9195...4485	snap-1621...6722-1...5e27.avro

The second row in Table 1-4 shows that a new `snapshot_id` was created for the initial three rows added. This row's `parent_id` value indicates that its parent is the original snapshot. The `manifest_list` column identifies the manifest list file name for each snapshot.

The `$manifests` metadata table provides a detailed overview of the manifests used in the current snapshot. This metadata table has multiple columns, but let's focus on those that help us understand characteristics of the data files that each manifest references.

Run the query in Example 1-6 to see details about the manifest files that are linked to the latest manifest list. The results are shown in Table 1-5.

Example 1-6. Querying the `$manifests` metadata table.

```
SELECT
 substring(path,
           position('/metadata/' IN path) + 10)
  AS path,
 added_snapshot_id AS add_snap,
 added_data_files_count AS add_file_cnt,
 added_rows_count AS add_row_cnt
FROM
 "my_iceberg_tbl$manifests";
```

Table 1-5. Initial manifest file listing.

path	add_snap	add_file_cnt	add_row_cnt
76849afc...5e27-m0.avro	1621...6722	1	3

The following details, by column, are represented in Table 1-5.

- `path` - The file name of the manifest file that was created.
- `add_snap` - The snapshot identifier that was created when this particular manifest file was created. Notice that it is the same identifier from the second row in Table 1-4.
- `add_file_cnt` - This manifest file only has a single data file associated with it.
- `add_row_cnt` - Aligns to the 3 rows that were inserted into this table for the given snapshot.

Run the query in Example 1-7 to see details of all referenced files for the current snapshot. The results are shown in Table 1-6.

Example 1-7. Querying the `$files` metadata table.

```
SELECT
  substring(file_path,
            position('/data/' IN file_path) + 6)
    AS file_path,
  record_count as recs,
  value_counts AS value_cnts,
  null_value_cnts AS null_value_cnts,
  lower_bounds, upper_bounds
FROM
  "my_iceberg_tbl$files";
```

Table 1-6. Initial data files listing.

file_path	recs	value_cnts	null...cnts	lower_bounds	upper_bounds
20...aa.parquet	3	{ 1 = 3, 2 = 3, 3 = 3 }	{ 1 = 0, 2 = 0, 3 = 0 }	{ 1 = 101, 2 = Jessica, 3 = Consor... }	{ 1 = 103, 2 = Paul, 3 = Son of... }

The following details, by column, are represented in Table 1-6.

- `file_path` - The name of the single data file linked to the manifest file in Table 1-5. The remainder of the columns relate to this particular file.
- `recs` - Identifies this data file has three records which is also shown as `add_row_cnt` in Table 1-5.
- `value_cnts` - The number of values in each column. Each of the columns has three values.
- `null_value_cnts` - The number of `null` values in each column. The count is zero for each column.
- `lower_bounds` - The lowest value in each column. For example, the first column, `id`, has `101` as a lower bound.
- `upper_bounds` - The highest value in each column. Staying with the same example, the `id` column has `103` as an upper bound.

Add additional records to the table.

```
INSERT INTO my_iceberg_tbl (id, name, description)
VALUES
```

```
(104, 'Thufir', 'Mentat'),
(201, 'Vladimir', 'Ruler of House Harkonnen'),
(202, 'Rabban', 'Ruthless nephew of Vladimir'),
(203, 'Feyd-Rautha', 'Savvy nephew of Vladimir'),
(301, 'Reverend Mother Gaius Helen Mohiam', null);
```

After verifying 5 more rows are present in the table for a total of 8, run the query from Example 1-5 to see the third snapshot listed as the last row in Table 1-7.

Table 1-7. The third snapshot.

snapshot_id	parent_id	mainifest_list
9195...4485	NULL	snap-9195...4485-1...bc1b.avro
1621...6722	9195...4485	snap-1621...6722-1...5e27.avro
4498...8325	1621...6722	snap-4498...8325-1...bbe8.avro

Run Example 1-6's query again, whose results list 2 manifests as shown in Table 1-8.

Table 1-8. Second manifest file listing.

path	add_snap	add_file_cnt	add_row_cnt
7887ee42...bbe8-m0.avro	4498...8325	1	5
76849afc...5e27-m0.avro	1621...6722	1	3

The row listing an `add_snap` value of `4498...8325` shows the linkage back to the third snapshot in Table 1-7. Adding up the values of the `add_file_cnt` and `add_row_cnt` columns shows that there are two data files present now with a total of eight rows.

Execute the query in Example 1-7 again to see statistical details of these 2 files as displayed in Table 1-9.

Table 1-9. Second data files listing.

file_path	recs	value_cnts	null...cnts	lower_bounds	upper_bounds

20...aa.parquet	3	{
		1 = 3,
		2 = 3,
		3 = 3
		}
			{
			1 = 0,
			2 = 0,
			3 = 0
			}
				{
				1 = 101,
				2 = Jessica,
				3 = Consor...
				}
					{
					1 = 103,
					2 = Paul,
					3 = Sonof...
					}

file_path	recs	value_cnts	null...cnts	lower_bounds	upper_bounds

20...1e.parquet	5	{
		1 = 5,
		2 = 5,
		3 = 5
		}
			{
			1 = 0,
			2 = 0,
			3 = 1
			}
				{
				1 = 104,
				2 = Feyd-...,
				3 = Mentat
				}
					{
					1 = 301,
					2 = Vladimir,
					3 = Savvy…
					}

The following details, by column, are represented for the last row in Table 1-9.

- `recs` - The new data file has 5 records, which is also shown in Table 1-8.
- `null_value_cnts` - One of the columns has a `null` value.
- `lower_bounds` - The id column shows `104` as the lower bound for this new file.
- `upper_bounds` - The id column shows `301` as the upper bound for this new file.

As mentioned at the beginning of this subsection, the logical metadata tables offer a SQL interface to the physical metadata layer's content. The upper/lower bounds details from Table 1-9 are representative of the types of metadata visualized in Figure 1-5. As stated earlier, values such as these allow the compute engine to perform metadata-based pruning without the expense of reading the actual footers of the data files.

Key Strengths and Weaknesses

No architecture exists without pros & cons. This rich metadata-driven architecture offers multiple compute engines to function alongside each other on the same tables. This same metadata can end up being a consequence once many snapshots are created.

Table Maintenance

The benefits of the enhanced metadata layer available in Apache Iceberg, coupled with the immutable nature of the table's data and metadata layers on the data lake, create suboptimal situations over

time. Metadata layer files benefit from consolidation and rebuilding periodically.

Differences in velocity & volume of data ingestion and content modifications often result in data files of widely varied numbers of rows contained within. Chapter 4 describes the "small files problem" that often plagues performance. Conceptually, consolidating and rewriting can resolve these issues. These issues can be kept well within control by appropriate *table maintenance*.

Table maintenance is an umbrella term for multiple optimization-oriented actions that can be triggered manually or scheduled for periodic execution. Each compute engine that integrates with Iceberg may have different mechanisms or syntax to initiate table maintenance activities, but the updated results are consumable by other compute engines.

Most table maintenance activities, or *jobs*, result in a new snapshot being created. This means that maintenance jobs could be running concurrently alongside other activities such as inserting, deleting, and/or updating records.

Iceberg supports concurrency, but it does not feature the feature richness that traditional RDBMS such as Oracle and PostgreSQL do in this area. It relies on an optimistic locking strategy and requires compute engines to implement a retry model when contention with that strategy occurs. More details can be found in the concurrent write operations Iceberg documentation.

The partitioning strategy defined for the table can also cause table maintenance to be difficult. Chapter 4 will identify partitioning guidelines that can lessen, or eliminate, this concern. While it is not always possible, one goal of the partition definition is to allow data within a given partition to eventually no longer need maintenance.

Lastly, data maintenance is not always easy on the wallet. You incur compute and storage costs as a result of these jobs. Finding a situation where the costs and the benefits of table maintenance are optimal is often highly dependent on multiple factors and will likely be unique for each of your largest tables.

Fortunately, as detailed in Chapter 7, most data platform providers have automated table maintenance features available to offload the responsibility of defining, tuning, and scheduling these jobs. Most data lakehouses have a multitude of tables that span a range of

size, modifications, and access. I recommend using automated table maintenance services for the majority of your tables.

For the very largest tables and tables of any size that have the highest query & modification concurrent activity, it is advisable to compare an automated solution with your own scenario-specific configuration and scheduling. Fortunately, these automated services will continue to mature. The details they log become the context that will likely allow them to understand patterns better and ultimately offer optimal table maintenance for all tables.

Chapter 6 will provide a more exhaustive focus on table maintenance explanations and recommendations. As an introduction, Table 1-10 lists the most widely known and utilized table maintenance activities.

Table 1-10. Table maintenance types.

Operation	Description	Trino function	Spark function
Compaction	Reads many small files and rewrites the contents into fewer, larger ones to allow compute engines to read from the table more efficiently.	`optimize`	`rewriteDataFiles`
Snapshot expiration	Deletes old table snapshots and their associated data & metadata layer files to free up storage space.	`expire_snapshots`	`expireSnapshots`
Orphan file deletion	Deletes any data files that are no longer referenced by any snapshot.	`remove_orphan_files`	`deleteOrphanFiles`
Rewriting manifests	Optimizes manifest files to speed up table planning and query performance.	`optimize_manifests`	`rewriteManifests`

The majority of these fundamental issues existed in some form or another prior to Iceberg and other modern table formats. Iceberg fortunately offers us structured ways to solve these problems. Solving them in a standard way leads to the one of the most important benefits of Iceberg - allowing multiple compute engines to work on the same tables as others.

Compute Engine Interoperability

The Apache Iceberg website's home page, *https://iceberg.apache.org/*, identifies this technology as "the open table format for data analytics." It further declares one of its major goals is "making it possible for engines like Spark, Trino, Flink, Presto, Hive and Impala to safely work with the same tables, at the same time." These are core tenants to the Iceberg architecture and key reasons for its wide adoption in the data lake technical domain.

No less than 30 engine integrations for Iceberg are listed on the project website. Some of the most popular integrations include the following.

- Programming-based solutions
  - Apache Spark
  - Apache Flink
  - PyIceberg
  - Iceberg Java API
  - Iceberg Rust
- SQL-focused engines
  - Trino
  - Starburst
  - Snowflake
  - Dremio
  - Hive

The interoperability goal has full integration in mind. Ideally, all Iceberg-enabled compute engines would be able to accomplish all of the following features based on allowed actions from applicable access policies.

- Create and drop tables
- Query existing tables
- Modify contents
- Perform maintenance activities
- Alter the structure of existing tables

Unfortunately, there are multiple challenges that could impede full interoperability. Catalog and policy management services could provide friction. Additionally, different features and/or versions of the Iceberg specification might provide additional issues.

As stated earlier in this chapter, all catalogs strive to allow multiple compute engines the ability to query Iceberg tables, but some catalogs may prevent modifications to, or creation of, tables. These limitations may be addressed in the development roadmap of a given catalog. Conversely, some catalogs are designed to limit anything other than reading capabilities to engines outside the data platform they are a part of.

The core Iceberg specification does not inherently include access policy management or security governance. Some catalog implementations, such as Apache Polaris, add policy management features. Alternatively, as is often done with Trino-based solutions like Starburst, security can be implemented at the engine level. This hurdle can often be resolved with integration configuration or by duplicating access control policies.

Some advanced features, such as all possible data file rewriting options, might not be supported in all compute engines. Chapter 5 will detail an example of this with file sorting options between Trino and Spark. Fortunately, this will not affect read/write compatibility after the maintenance task is complete. It could, however, mean that you may be forced to use more than one compute engine to implement a desired outcome. An example of this would be using Spark for data maintenance jobs while running your queries via Trino.

Lastly, the Iceberg specification is continuing to be enhanced. Special care needs to be applied when using a new version of the specification on any given table. To prevent issues, you should only use a specific version if all compute engines that will access the table support that version. For example, you can upgrade a version 2 table to version 3, but once that happens, an engine that does not support version 3 will be unable to access the table at all.

Many large enterprises already have multiple compute engines deployed in their environment. They also have a variety of data sources and ingestion frameworks. Figure 1-6 is an example of a variety of technologies that could be deployed in a modern enter-

prise. This includes transformation processing tools, data access frameworks, and data applications.

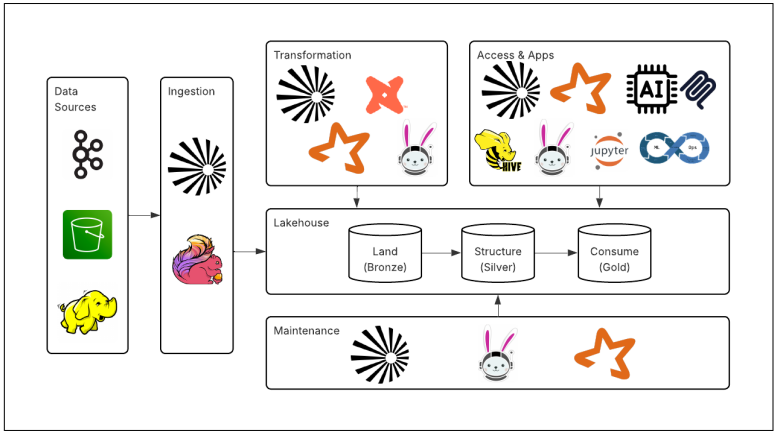

Figure 1-6. Multi-engine architectural diagram.

This technical optionality can support a multitude of use cases. One example, a connected car scenario, could be implemented with the defined technologies in these system specifications.

- Automobile sensor information is published to an Apache Kafka topic by a third party service provider.
- Apache Flink consumes the sensor data for two purposes.
  - — Perform streaming data analysis to detect crashes and theft.
  - — Write the sensor information to an Iceberg table in the Land zone.
- Perform transformation processing with Trino to create an additional table in the Structure zone that represents validated and standardized information.
- Trino is also used to create multiple aggregation tables in the Consume zone to aid the data access clients.
- Analysts use the Trino Python client in a Jupyter notebook for their analysis.
- Data maintenance activities are performed with Spark jobs.

The environment needed for a second example scenario centered on data science analysis of airline flight data with the specifications

below could also be implemented with a subset of the technologies shown in Figure 1-6.

- Airlines post their daily flight information into a common AWS S3 bucket.
- Starburst's file ingestion service reads the files and stores the raw data in a Land zone table and also performs the transformations needed to populate the Structure zone table. This ingestion service, presented in Chapter 7, also automatically performs table maintenance.
- Spark MLlib jobs run against the Structure zone table for a variety of machine learning operations.
- A data product is created in Starburst and stored in the Consume zone to improve performance for data access users. Rich metadata is captured, thereby ensuring the data product's datasets are AI-ready.
- The data product's metadata provides additional context to Starburst's own conversational AI agent and to third-party AI agents interfacing with Starburst's MCP server.

Apache Iceberg was designed from the beginning to allow this level of flexibility, and allows technologists to use the right compute engine at the right time. The open approach to how metadata itself is managed allows the optionality discussed in this chapter. The consequence of the metadata architecture is the need to perform table maintenance tasks periodically.

The next chapter will focus on the features and benefits of this open metadata architecture, including changes to the structure and content of tables as well as leveraging snapshots to query or roll back to prior versions of the table.

About the Author

Lester Martin is a seasoned developer advocate, trainer, blogger, developer, and architect with three decades of experience in software development and data engineering. He has spent the last decade focused on data pipelines and data lake analytics using Trino, Iceberg, Hive, Spark, Flink, Kafka, NiFi, NoSQL databases, and, of course, classical RDBMSs. He has used SQL in all stages of his career. Lester is a polyglot programmer, but feels most productive in Java and Python. He is well-versed in a variety of Dataframe APIs. Having consulted, taught, written, and presented on Apache Iceberg since 2022, Lester Martin is an expert on optimizing Iceberg lakehouses.