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

Copyright © 2027 O'Reilly Media, Inc. All rights reserved.

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
2026-06-23: Second Release

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Optimizing Your Apache Iceberg Lakehouse*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

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

979-8-341-67392-2

[LSI]

Table of Contents

Brief Table of Contents (*Not Yet Final*) ..... **vii**

1. Exploring the Apache Iceberg Architecture **1**
Metadata and Iceberg 2
Key Strengths and Weaknesses 18

2. Primary Features of Iceberg **25**
Mutability Options 26
Schema Evolution 42
Hidden Partitioning 47
Snapshot Features 56

Brief Table of Contents (*Not Yet Final*)

Chapter 1: Apache Iceberg Introduction (unavailable)

Chapter 2: Exploring the Apache Iceberg Architecture (available)

Chapter 3: Primary Features of Iceberg (available)

Chapter 4: Classic Lakehouse Pitfalls (unavailable)

Chapter 5: Advanced File Writing Strategies (unavailable)

Chapter 6: Data Platform Optimizations (unavailable)

Chapter 7: Starburst Icehouse Architecture (unavailable)

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. Please note that the GitHub repo is available at *https://github.com/starburst-data/optimizing-iceberg-book/tree/main*.

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.

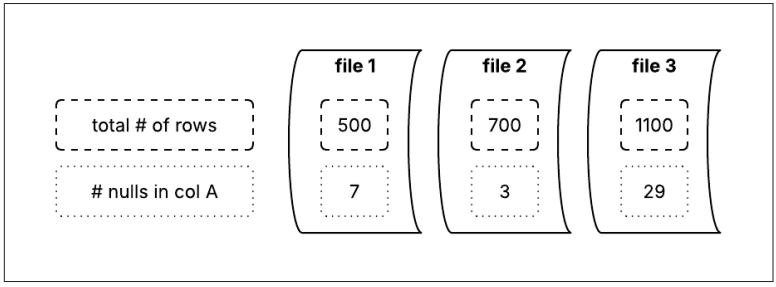


Figure 1-1. Statistical metadata stored in Parquet files.

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)

> **Note**
>
> 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.

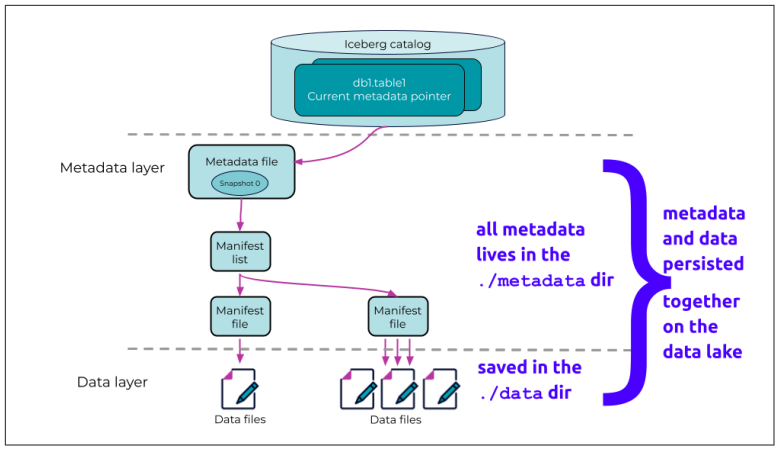


Figure 1-2. Iceberg architecture overview.

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.

> **Warning**
>
> 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 2.

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`.

> **Note**
>
> 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*.

> **Note**
>
> 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 2 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.

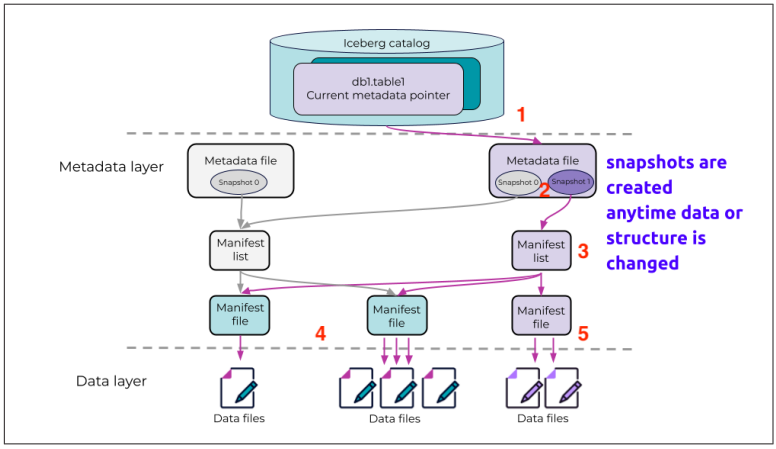


Figure 1-3. Additional metadata created with a new snapshot.

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.

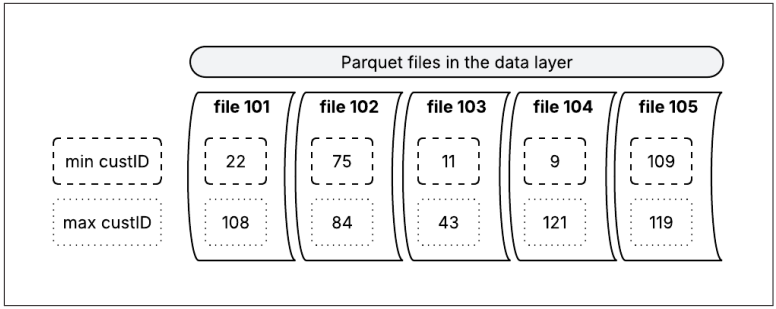


Figure 1-4. Min and max column values by data file.

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.

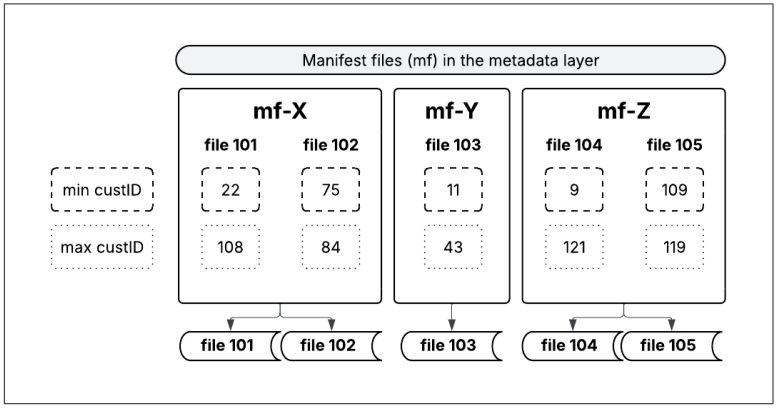


Figure 1-5. Min and max column values by manifest list.

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*.

> **Note**
>
> 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.

> **Tip**
>
> 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

> **Note**
>
> 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 = Sonof...
					}

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 = Son of...
					}

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 sit-

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

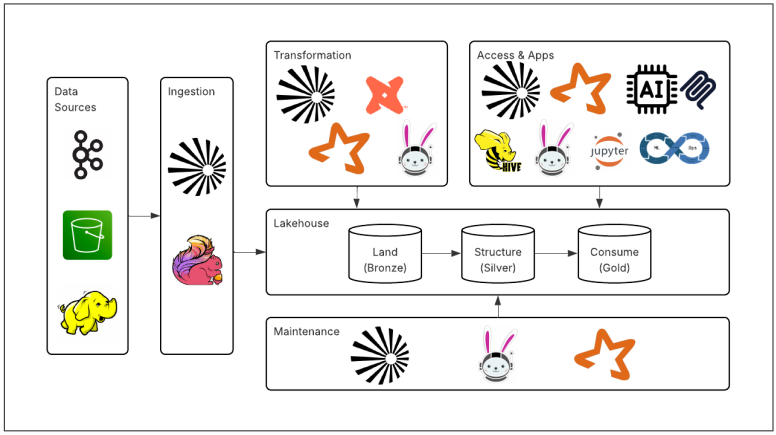


Figure 1-6. Multi-engine architectural diagram.

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.

CHAPTER 2

Primary Features of Iceberg

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 3rd chapter of the final book. Please note that the GitHub repo is available at *https://github.com/starburst-data/optimizing-iceberg-book/tree/main*.

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*.

This chapter details the primary features of Iceberg, starting with the opportunity to commit transactions that make content changes to tables. You will learn how this works despite the underlying data lake files being immutable. We'll also explore the two file writing strategies for capturing changes.

Table changes are not only limited to data as structural changes to the table definition are also allowed and tracked via metadata. Iceberg ensures that only changes that are compatible with prior snapshots are allowed to be made. These table alterations extend past column types and names to include modifying the partitioning strategy.

You will also understand the features available for working with multiple snapshots. This includes time travel querying and the ability to roll back the state of the table to a prior version. Advanced snapshot features include tagging and branching that can be helpful with testing, compliance, and data pipelines are also explained.

> **Tip**
>
> The SQL used in this chapter is available in the `./Chapter3` folder of this book's GitHub repository. All output presented is discussed thereby allowing you the same learnings if you choose not to perform the hands-on exercises.

Mutability Options

The majority of the original use cases addressed with data lake tables were enabled by storing copies of log files in the lake. Adding more records was easily accomplished by adding more data files. The concept of *mutating*, or making changes to, existing data involved removing and/or replacing the data files that the table's metadata was referencing. As the need to have more fine-grained changes at the record level, as opposed to the file level, surfaced, various solutions were implemented.

Often the best solution was to create file level merge operations to rebuild a better version of the data lake files. This was required due to the *immutability* of the files on the data lake. They could be deleted, but they could not be altered.

Apache Iceberg solves these problems by using standard SQL *DML* (Data Manipulation Language) statements; `INSERT`, `UPDATE`, `DELETE`, and `MERGE`. These operations can affect zero-to-many rows across multiple files. The Iceberg specification defines a standard approach to how all compute engines will persist changes made via these operations to enable the continued multi-engine support that is critical to adoption.

> **Note**
>
> The Iceberg Table Spec identifies 3 versions at the time of publication, with v4 currently under active development. This technical guide is aligned with v3 and key differences with earlier versions will be called out when appropriate.

Append-Only Tables

The initial version, v1, of the Iceberg specification only supported append operations. New records could be added, but they could not be removed or updated. This general pattern is still widely used, especially for immutable events that include a timestamp attribute for each record.

Despite not being the most recent version of the table spec, compute engines still allow new v1 tables to be created. Furthermore, compute engines will throw an error on v1 tables when a DML operation, such as an `UPDATE` SQL statement, is attempted.

> **Tip**
>
> While it may seem logical to create a v1 table to prevent unwanted mutations from being allowed, it is not recommended to use this configuration. Additional features are present in subsequent versions of the table spec that are advantageous for users and compute engines alike. Preventing unwanted statements such as `UPDATE` from being executed is best solved with a security access policy.

Another key element of Iceberg v1 is the identification of the supported file formats - they are Apache Parquet, Apache ORC, and Apache Avro which is a smaller set of file formats than Hive allows. This decision was made based on the analytic processing performance benefits of these file formats over simple formats such as CSV and JSON.

Transactional Tables (ACID Compliance)

Starting with v2, Iceberg tables allow for row-level updates and deletes in addition to the inserting operations allowed in v1. Table changes, including inserts, are persisted as part of a *transaction*. These adhere to the well-known ACID requirements for database transactions.

The Iceberg specification does not currently support multi-statement transactions enclosed within `BEGIN...END` blocks as implemented with RDBMSs. Some, often proprietary, catalog and compute engine combinations offer this support, but usually at

the expense of interoperability. As of v3, DML operations are auto-committing transactions.

DML changes can span many rows, but understand that Iceberg transactions are single-statement, single-table operations. The following sub-sections walk through the primary DML operations and explain conceptually what happens at write-time and read-time.

Run the SQL in Example 2-1 to generate a table for transactional operations experimentation. This *Create Table As Select* (CTAS) statement sources its data and structure from the Trino TPC-H connector's `region` table. If your exploration environment does not have this data generator connector you can build an appropriate `INSERT` statement based on the results in Table 2-1.

Example 2-1. Create a testing table.

```
CREATE TABLE dml_region
WITH (type='iceberg', format_version=2)
AS SELECT * FROM tpch.sf1.region;
```

The `dml_region` table was created as an Iceberg v2 table instead of v3 as it will be easier to explore the actual files being stored in the lakehouse. The differences related to ACID transactions between these versions will be explained in the removing records subsection.

The initial content of the `dml_region` table shown in Table 2-1 can be produced by executing the following query.

```
SELECT regionkey AS key, name, comment FROM dml_region;
```

Table 2-1. Initial content.

key	name	comment
0	AFRICA	lar deposits. blithely ... are according to
1	AMERICA	hs use ironic, even requests. s
2	ASIA	ges. thinly even pinto beans ca
3	EUROPE	ly final courts cajole furiously final excuse
4	MIDDLE EAST	uickly special accounts ... haggle furiousl

Execute the query in Example 2-2 to see details of all referenced data files for the current snapshot. The results shown in Table 2-2 identify a single file which contains five rows.

Example 2-2. Querying the `$files` metadata table.

```
SELECT file_path, record_count FROM "dml_region$files";
```

Table 2-2. Initial file details.

file_path	record_count
initial-5-rows-inserted.parquet	5

> **Note**
>
> The `file_path` values in this chapter have been manipulated to make them more closely aligned with the activity that created them.

If you retrieve the file shown in Table 2-2 from the lakehouse, you can choose a helper tool such as the VS Code parquet-viewer extension to see a JSON version of the Parquet file's contents. The following rows represent the contents of the `initial-5-rows-inserted.parquet` file listed in Table 2-2

```
{"regionkey":0,"name":"AFRICA","comment":"lar dep...ding to"}
{"regionkey":1,"name":"AMERICA","comment":"hs use...ests. s"}
{"regionkey":2,"name":"ASIA","comment":"ges. thi...beans ca"}
{"regionkey":3,"name":"EUROPE","comment":"ly final...excuse"}
{"regionkey":4,"name":"MIDDLE EAST","comment":"uick...iousl"}
```

Adding records

New records are primarily added via the `INSERT` statement. At write-time, one, or more, data files are added into the table's `/data` folder and appropriate metadata artifacts for this new snapshot are created in the `/metadata` folder as described in Chapter 1. The new data files contain the inserted rows.

At read-time, the compute engine reads the same data files that were part of the prior snapshot, plus the additional rows in the new data files.

Execute the `INSERT` statement in Example 2-3 to add four more rows so you can see this behavior firsthand.

Example 2-3. Add records SQL statement.

```
INSERT INTO dml_region
 (regionkey, name, comment)
```

```
VALUES
 (5, 'Middle-Earth', 'J.R.R. Tolkien'),
 (6, 'Narnia', 'C.S. Lewis'),
 (7, 'Erilea', 'Sarah J. Maas'),
 (8, 'Westeros', 'George R.R. Martin');
```

Run the following SQL statement to see the four newly added rows are now present in the `dml_region` table.

```
SELECT regionkey AS key, name, comment
  FROM dml_region WHERE regionkey > 4;
```

key	name	comment
5	Middle-Earth	J.R.R. Tolkien
6	Narnia	C.S. Lewis
7	Erilea	Sarah J. Maas
8	Westeros	George R.R. Martin

Running the `$files` metadata table query in Example 2-2 again identifies a second data file present which contains 4 more records.

file_path	record_count
initial-5-rows-inserted.parquet	5
added-4-rows.parquet	*4*

Example 2-4 shows the contents of the additional `added-4-rows.parquet` file and represents the four rows added via the `INSERT` statement in Example 2-3.

Example 2-4. Content of second data file.

```
{"regionkey":5,"name":"Middle-Earth","comment":"J.R....kien"}
{"regionkey":6,"name":"Narnia","comment":"C.S. Lewis"}
{"regionkey":7,"name":"Erilea","comment":"Sarah J. Maas"}
{"regionkey":8,"name":"Westeros","comment":"George R....tin"}
```

As more records are added to a table, new data files that contain the records being added are persisted in the lakehouse. This is fundamentally the same as the Hive table format sans the metadata artifacts that are also created at write-time and utilized at read-time.

Removing records

The `DELETE` statement allows existing records to be removed from a table. Since the data files in the lakehouse are immutable, compute engines cannot edit them in-place. The default solution with compute engines such as Trino is to create a change, or transaction, log indicating when records are removed. Individual references to rows affected by a `DELETE` statement are referred to as *deletion markers* and stored in a special type of data file called a *delete file*.

At write-time, one, or more, delete files are added to the table's `/data` folder and appropriate metadata for this new snapshot is also created. The newly added delete files identify the source file name and indexed position of the row being deleted. These delete files can contain more than one deletion marker, but a delete file cannot reference rows from multiple data files.

For Iceberg v2, this delete file is primarily subtyped into what is called a *positional delete* file. There is another subtype called an *equality delete* that identifies table columns along with specific matching values for values that need to be excluded from query results. This is conceptually similar to, and often sourced from, a `WHERE <col> IN (<values>)` clause that matches the records to be deleted. Query-oriented compute engines, including Trino, do not create equality delete files. They are part of the v2 specification, so all compute engines are required to be able to process equality deletes along with positional deletes at read-time.

Change Data Capture (CDC) frameworks, such as Debezium and Kafka Connect, are the primary writers of equality delete files. Unlike the query-oriented compute engines, these stream processing engines usually are not aware of the physical location for the rows that need to be deleted. Writing delete files this way is quick, but adds complexity, usually in the form of scanning, for readers. Improvements in positional delete files since v2 have resulted in most frameworks, including Flink, moving away from creating equality delete files.

At read-time, the compute engine reads the same data files that were part of the prior snapshot to get an initial representation of the current state of a table. Next, the compute engine processes the delete files so that any rows referenced by a deletion marker or an equality delete can be eliminated from the result to be returned, or processed further.

Execute the DELETE statement in Example 2-5 to remove a single row so you can see this behavior firsthand.

Example 2-5. SQL statement to remove a single record.

```
DELETE FROM dml_region WHERE regionkey = 7;
```

Run the following SQL statement to verify the third row identified in Example 2-4 was deleted from the results as displayed.

```
SELECT regionkey AS key, name, comment
  FROM dml_region WHERE regionkey > 4;
```

key	name	comment
5	Middle-Earth	J.R.R. Tolkien
6	Narnia	C.S. Lewis
8	Westeros	George R.R. Martin

Run the `$files` metadata table query in Example 2-2 again to verify an initial delete file was created that identifies 1 record.

file_path	record_count
initial-5-rows-inserted.parquet	5
added-4-rows.parquet	4
delete-op-delete-file.parquet	*1*

The JSON record below represents the contents of the additional `delete-op-delete-file.parquet` file which identifies the single row removed via the DELETE statement in Example 2-5. Leveraging zero-based numbering, you can verify that this deletion marker is referencing row 3 in `added-4-rows.parquet` shown in Example 2-4.

```
{"file_path":".../added-4-rows.parquet","pos":2}
```

As more records are deleted from a table, additional delete files are persisted in the lakehouse. Each of these contain references to the rows that should no longer be present. The final results returned are based on walking the logical change log which is implemented with these delete files that are linked back to normal data files.

As mentioned earlier, the testing table was created based on Iceberg v2. This was done as the delete files are relatively easy to peer into and understand what they represent. Iceberg v3 has modified how

delete files are persisted. The less frequently used equality delete files did not get enhanced in v3, but are still available. The concept of the positional delete file is still around, but the implementation has been enhanced for performance. The v3 delete files are referred to as *binary deletion vectors* and they are persisted in a new file format type named *Puffin* files.

In addition to equality delete files, compute engines still need to be able to read both kinds of position delete files. A table that gets upgraded from v3 will not automatically rewrite the positional delete files to use the Puffin format. Conversion happens lazily, during compaction and rewrite operations, not at table upgrade time.

To get some hands-on with the deletion vectors, run the SQL from Example 2-1 again changing the following items first and then perform the steps up until this point on the new table.

- Rename `dml_region` to `dml_region_v3`.
- Change the `format_version` value to 3.

> **Tip**
>
> Check out The Data Engineer's Guide to Iceberg v3 from Starburst in order to learn more about deletion vectors, the Puffin file format, and additional v3 enhancements and features.

Updating records

The standard SQL `UPDATE` command is the primary way to change the contents of one, or more, records in a table. The immutability of lakehouse files results in more complexity when updating records compared with deleting. Basically, each affected row needs to be deleted and then added back as a full, new record. This approach requires twice as many files to be created and the delete files need to be processed prior to the new data files' contents being incorporated.

At write-time, an update starts off looking a lot like the implementation of `DELETE` as one, or more, delete files are added to the table's `/data` folder identifying the affected rows. With the impacted rows now missing, the compute engine simply creates net-new data files that include the fully populated rows. These data files are also

persisted in the `/data` folder. The compute engine finally builds appropriate metadata artifacts for this newly created snapshot.

At read-time, the compute engine consumes the same data files that were part of the prior snapshot to get an initial representation of the current state of a table. Next, the compute engine processes the delete files so that any rows referenced by a deletion marker can be eliminated from the in-flight results. Finally, the contents from the newly created data files are included in the result to be returned, or processed further.

Execute the `UPDATE` statement in Example 2-6 to change the first three rows.

Example 2-6. Update three records SQL statement.

```
UPDATE dml_region
  SET comment = 'starts with A'
WHERE name LIKE 'A%';
```

Verify the rows impacted by the SQL from Example 2-6 were updated in the results displayed.

```
SELECT regionkey AS key, name, comment
  FROM dml_region WHERE name LIKE 'A%';
```

key	name	comment
0	AFRICA	starts with A
1	AMERICA	starts with A
2	ASIA	starts with A

Run the `$files` metadata table query in Example 2-2 one more time. Table 2-3 now shows there is a second delete file referencing 3 records. Additionally, there is a new data file with 3 rows. This accounts for the remove and rebuild process that Iceberg has to go through to handle mutable table records backed by immutable files.

Table 2-3. File details after updating three records.

file_path	record_count
initial-5-rows-inserted.parquet	5
added-4-rows.parquet	4
delete-op-delete-file.parquet	1
update-op-delete-file.parquet	*3*

file_path	record_count
newly-readded-3-rows.parquet	*3*

The JSON records below represent the contents of the new `update-op-delete-file.parquet` file listed in Table 2-3 which identifies the three rows being removed to start the implementation of the `UPDATE` statement in Example 2-6. Leveraging zero-based numbering again, you can verify that these deletion markers are referencing the first 3 rows of `initial-5-rows-inserted.parquet` shown in Table 2-2.

```
{"file_path":".../initial-5-rows-inserted.parquet","pos":0}
{"file_path":".../initial-5-rows-inserted.parquet","pos":1}
{"file_path":".../initial-5-rows-inserted.parquet","pos":2}
```

This took care of half of the work needed for the row updates. Now that these 3 rows are no longer present, the engine reads the contents of the `newly-readded-3-rows.parquet` file shown in Table 2-3 which represents the three rows added that are aligned with the `UPDATE` statement in Example 2-6 and shown in Example 2-7.

Example 2-7. Contents of second data file.

```
{"regionkey":0,"name":"AFRICA","comment":"starts with A"}
{"regionkey":1,"name":"AMERICA","comment":"starts with A"}
{"regionkey":2,"name":"ASIA","comment":"starts with A"}
```

Notice as explained earlier, the entire row is recreated even though only one column was being logically deleted. This write-time remove and rebuild model starts to create a long trail of files that need to be processed at read-time in the correct order before finally sharing results.

If you were to update the `AMERICA` record, you would get two more files with these details.

- A third delete file that identifies the second row of the data shown in Example 2-7.
- A fourth data file with the fully rewritten `AMERICA` record.

This long trail of `/data` files is joined by a sprawl of entries in the `/metadata` folder as the changes continue to be executed by the compute engine. Not to fret, these are well-known side effects created by how ACID transactioning is implemented and Chapter 6

will provide best practices around the necessary data maintenance activities to address these concerns.

Merging records

When a list of changes exists that could be processed by iterating through it to perform either an `INSERT`, `UPDATE`, or `DELETE` command on each record, utilizing the `MERGE` command is usually a better approach. It provides a concise way to capture the logic all within a single SQL statement and it commits all changes into a single snapshot upon completion.

The SQL statement performs a `MERGE` on each item in the list of changes (identified as a `source_table`) `INTO` a `target_table` by defining a match, or `search_condition`, in the `ON` clause. `WHEN` each item in the `source_table` gets `MATCHED` or `NOT MATCHED`, an appropriate `INSERT`, `DELETE`, or `UPDATE` operation is executed. Trino's `MERGE` documentation provides more details including syntax and examples of this powerful DML command.

> **Tip**
>
> To help prevent a table scan of the `target_table`, ensure the `search_condition` in the `ON` clause includes additional `WHERE` criteria to trigger a predicate pushdown by the compute engine. This additional criteria is usually dynamic based on the data in the `source_table`. Concrete examples are supplying min/max values on columns such as transaction dates or identifiers.

The concepts presented earlier in this section for write-time and read-time operations are still relevant when merging records in a single SQL statement. The difference here is that you will end up with fewer new delete files and data files than if you performed each row-level operation at a time as the compute engine can make the file additions in a more coarse-grained manner.

> **Note**
>
> A MERGE Example is available in this book's GitHub repository should you want to explore how this SQL statement handles data modifications and what changes on the lakehouse are made to persist the data modifications.

The examples in this section are aligned with a popular file writing strategy, but it is not the only one. The next section explores the two available options and presents tips on when each is most appropriate.

Copy-On-Write vs Merge-On-Read

As a reminder, the data files in the lakehouse are immutable. Compute engines cannot simply open a file, edit its contents, and resave it. Compute engine implementations of the SQL statements that change existing records have 2 strategies for persisting these changes for the next snapshot; *Copy-on-Write* (CoW) and *Merge-on-Read* (MoR). Table 2-4 provides a comparison of the options. The details within this table are explored deeper within this section.

Table 2-4. Summary of change writing options.

Feature	Merge on Read (MoR)	Copy on Write (CoW)
Write strategy	Creates delta files	Rewrites entire affected files
Read performance	Slower (requires merging delta files)	Fast (no need to merge files)
Write performance	Faster (only delta files are created)	Slow (entire files are rewritten)
Storage impact	Lower (full files are not rewritten)	Higher (unchanged data in multiple files)
Compaction requirement	Required periodically to merge delta files	Not required frequently
Best for	Update-heavy workloads, streaming ingestion, incremental updates	Read-heavy workloads, analytics, bulk updates

The Transactional tables section's description of read & write operations for removing, updating, and merging records was leveraging the MoR strategy.

> **Note**
>
> SQL clients do not need to make these changes directly. The compute engines do the work of persisting any needed files at write-time and tackle the burden of understanding them at read-time.

To help explain these options, Figure 2-1 shows an example of an Iceberg table whose current snapshot consists of content from 2 files, each with 3 records.

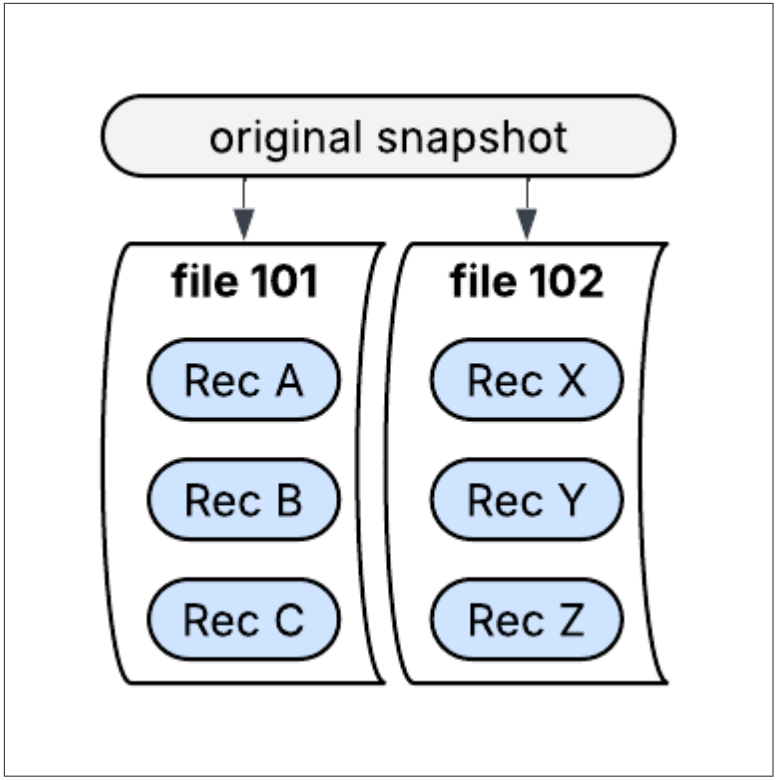


Figure 2-1. Original snapshot.

MoR overview

The Transactional tables section showcased the MoR strategy which puts the burden of reconciling the changes on the reader. Essentially, the compute engine writes a change log of what modifications are needed. That change log is persisted in what are commonly referred to as *delta files*.

Figure 2-2 shows the before and after snapshots when an `UPDATE` command is run to modify the rows representing records X and Z.

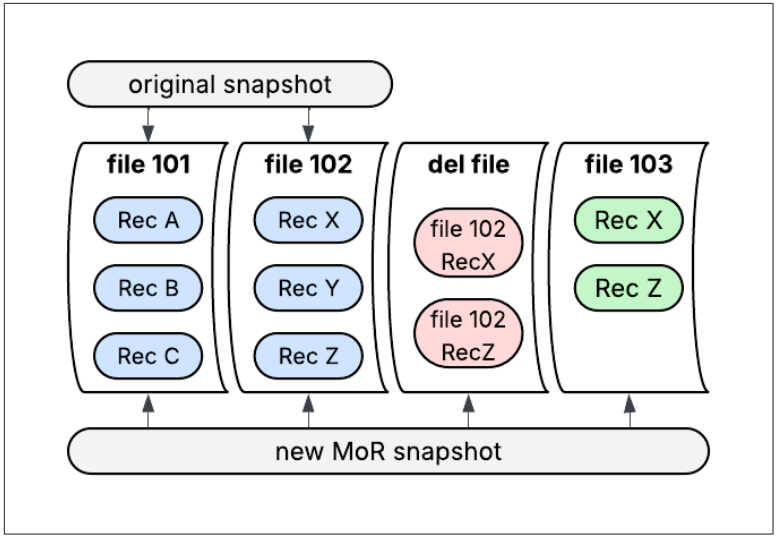


Figure 2-2. MoR snapshot.

Since the compute engine cannot update file 102, which contains records X and Z, a change log that identifies a full delete followed by a full replacement of the affected records must be created. Figure 2-2 identifies the write-time activities that must be performed in sequence.

1. A delete file is created referencing the rows that need to be modified.
2. File 103 is created that has the full row values that are required to meet the conditions of the `UPDATE` command.
3. Metadata changes are persisted that identify that all four files are part of the new snapshot.

At read-time, compute engines will perform these ordered steps.

1. The table metadata will inform the compute engine of all files that are part of the new snapshot.
2. Files 101 and 102 will be read to determine the base state of the table.
3. Reading the two deletion markers in the delete file will allow the compute engine to remove records X and Z from the logical contents of the table.

4. After reading the file 103, its contents will allow a full re-add of the X and Z records to finalize a complete and accurate representation of the table.

The MoR strategy is optimized for write performance and storage efficiency. Conversely, it can slow down reading performance due to the merging activities it must do. This strategy is best for update-heavy workloads such as streaming ingestion and data pipelines that require frequent incremental updates.

> **Warning**
>
> Periodic compaction maintenance jobs introduced in Chapter 1 will be required to rewrite the (generally) small-sized delete and data files that are generated by the MoR approach. Chapter 6 will cover best practices regarding table maintenance jobs.

CoW overview

The alternative to MoR is the CoW strategy. The implementation approach is to simply rebuild, and replace, any data files that have records that need to be updated or deleted. Figure 2-3 shows the before and after snapshots when the same `UPDATE` command as before is run to modify the rows holding records X and Z.

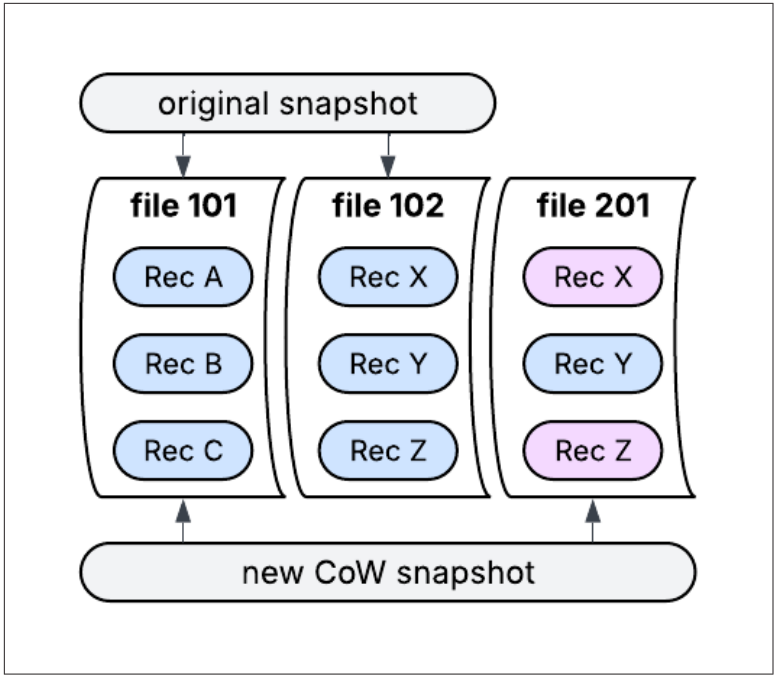


Figure 2-3. CoW snapshot.

Just as described before, compute engines are not able to simply update the data file that contains records X and Z. They can however create a new file that would look the same as a file that was edited. Ultimately, this new, updated version of the data file would be used in place of the one it was sourced from. Figure 2-3 identifies the write-time activities that must be performed in sequence.

1. File 201 is created that includes all unaffected rows plus the revised representation of the rows that needed to be updated.
2. Metadata changes are persisted that identify the unaffected files and the rewritten files make up the new snapshot.

At read-time, compute engines will perform these ordered steps.

1. The table metadata will inform the compute engine of all files that are part of the new snapshot.
2. File 101 and the new file 201 will be read.

This lack of complexity allows the CoW strategy to feature better query performance. It is best suited for read-heavy workloads such as analytics and that perform updates in bulk-sized batches. The consequence is that CoW requires more overall storage space and compute resources for writes.

> **Warning**
>
> Periodic snapshot expiration maintenance jobs introduced in Chapter 1 will be required to prevent the data size from growing out of control. Chapter 6 will cover best practices regarding table maintenance jobs.

Compute engines have different mechanisms for initially setting, and altering, the write strategies of MoR and CoW. These engines each have defaults for when a table creator does not identify which strategy to use. Additionally, some engines, such as Spark, allow fine-grained settings for `DELETE`, `UPDATE` and `MERGE` operations. Others, including Starburst, only have a single, coarse-grained setting for all operations which can be changed as needed. Table 2-4 can be used as a scorecard to help you decide which strategy is best for your particular use cases.

Your largest and/or most frequently updated tables are the most critical ones to identify the best write strategy for your use cases. Multiple strategies and/or multiple compute engines are worth considering to fine-tune your balance of read vs write performance.

> **Tip**
>
> As a heuristic, this decision matters most at data ingestion time since that is usually where the biggest changes to a table occur. For streaming solutions, a MoR strategy is often the best solution and for batch solutions that utilize the MERGE command, CoW produces more optimal results.

Schema Evolution

Over the lifecycle of a typical table, there are occasions when the definition of the table structure needs to be modified. Apache Iceberg provides a performant and robust solution by offering *schema*

evolution options that guarantee the structural changes can happen quickly and will be free of side-effects.

> **Note**
>
> The Iceberg specification identifies available primitive, nested, and semi-structured data types.

Supported Changes

In addition to the ability to change the name of the table, these column level changes are supported.

- New columns can be added to a table or within a nested `struct`.
- Existing columns can be removed from a table or within a nested `struct`.
- Existing columns can be renamed.
- Data types can be changed as long as they are widening options such as from `int` to `bigint`, or from `float` to `double`.
- The physical order of columns, and the fields within a nested `struct`, can be changed.

Schema evolution does not change the actual data files. If you have a need for data migration, such as breaking a single column containing a full name into separate columns for first and last name, you will need to create appropriate processing logic to actually change the data. This could be done with SQL using compute engines like Trino or programmatically with engines including Spark.

A data migration effort will require some, if not all, files to be rewritten and produce at least one more snapshot. The processing time & costs will be determined by factors such as the complexity of the change, the size of the data, and the amount of resources available to the compute engine.

Operationally Fast

By not requiring any data file to be updated when evolving a schema, this operation is exceptionally fast. It simply needs to make a new `metadata.json` file and update the catalog's reference for the table to point to the new metadata file.

It may seem logical that a schema change creates a new snapshot, but contrary to popular belief it does not. To help understand this, imagine you have a table with four columns; `nationkey`, `name`, `regionkey`, and `comment`. Example 2-8 shows the relevant schema and snapshot details in the table metadata file. A complete example file is available in the GitHub repository.

Example 2-8. Metadata file before schema modification.

```
"current-schema-id": 0, ❸
"schemas": [ { ..., "schema-id": 0,  ❶
      "fields": [   ❷
        { "id": 1, "name": "nationkey", ...},
        { "id": 2, "name": "name", ...},
        { "id": 3, "name": "regionkey", ...},
        { "id": 4, "name": "comment", ...} ]
  } ], ...,
"current-snapshot-id": 7105994075912274000,  ❽
"snapshots": [ {
       "sequence-number": 1,  ❹
       "snapshot-id": 7105994075912274000, ... },  ❺
       "manifest-list": ".../snap-...-ed78.avro", ❻
       "schema-id": 0  ❼
  } ]
```

❶ The collection of schemas has a single entry.

❷ This schema has four fields.

❸ The table metadata file references the single schema.

❹ The collection of snapshots has a single entry.

❺ The snapshot's identifier.

❻ The snapshot's manifest list file name.

❼ The single snapshot is using the single schema.

❽ The table metadata file references the single snapshot.

Now, imagine you evolve the schema by adding another column with a command such as this.

```
ALTER TABLE <table_name>
  ADD COLUMN new_col varchar;
```

Example 2-9 shows that while an additional schema exists, there is still only a single snapshot, and that the table metadata file itself is now referencing the new schema. A complete example file is available in the GitHub repository.

Example 2-9. Metadata file after schema modification.

```
"current-schema-id": 1,  ❽
"schemas": [ { ..., "schema-id": 0,
      "fields": [
        { ...1...nationkey... },{ ...2...name... },
        { ...3...regionkey... },{ ...4...comment... } ]
  }, { ..., "schema-id": 1,  ❶
      "fields": [
        { ...1...nationkey... },{ ...2...name... },
        { ...3...regionkey... },{ ...4...comment... },
        { "id": 5, "name": "new_col", ...} ]  ❷
  } ], ...,
"current-snapshot-id": 7105994075912274000,  ❼
"snapshots": [ {
       "sequence-number": 1, ❸
       "snapshot-id": 7105994075912274000, ... }, ❹
       "manifest-list": ".../snap-...-ed78.avro",  ❺
       "schema-id": 0  ❻
  } ]
```

❶ The collection of schemas now has a second entry.

❷ This new schema has an additional field.

❸ The collection of snapshots still only has a single entry.

❹ The snapshot's identifier has not changed.

❺ The snapshot's manifest list has not changed.

❻ When the snapshot was created it was using the original schema.

❼ The table metadata file is still referencing the original snapshot.

❽ The table metadata file now references the updated schema.

The only file that has to be created is a new `metadata.json` which accounts for the changes in the schema. No new manifest list or manifest files are created as no data files are changed to support

this feature. This is the key to the performant nature of schema evolution.

A change such as creating a new column boils down to adding a new schema to the list of existing ones. Looking back at Example 2-9 you can see that columns have unique identifiers. In that scenario when a query is run against the table just after the schema evolved, the previous data files do not have values for the new column. The compute engine simply includes the default value, null in this case, for the column value.

If this was followed up by a removal of a column then the compute engine will notice that the existing data files have a value for that currently absent from the schema column and will simply ignore it and not read or return it.

Enforces Correctness

This approach of using unique column identifiers is one of the key ways that anomalies don't occur as a result of schema evolution. The identifiers allow for independence from the column names and prevents scenarios such as *ghost columns*. This ghost phenomenon occurs when adding a new column with the same name as a column that was previously removed.

Without leveraging unique identifiers, a compute engine could inadvertently read an older file's value for the new column since it could find an existing value for that column name in the data file. Dropping or reordering columns does not affect the values in any other column either due to this independence quality of schema evolution.

Narrowing a data type for a column, such as from a `bigint` to an `int`, could cause anomalies. This is because the value of a given row's column could be outside the range of the more narrow data type. This is the same situation as moving from a `varchar` to one of the numeric data types. The existing value might not properly cast to the more restrictive data type. This is why only data type widening changes are allowed which ensures the data correctness that this modern table format promises.

Iceberg's approach of using column identifiers and not allowing narrow data type changes to occur provides a robust and performant solution that is free from side-effects. There are additional evolution

concepts such as partition evolution described in the next section and sort order evolution which is presented in Chapter 5.

Hidden Partitioning

Before we dive into the fascinating feature of partition evolution, let's refresh ourselves on the concept & benefits of partitioning. Then understanding how partitions are defined with transform functions gives you the basis of why Iceberg uses the phrase *hidden partitioning*. Armed with that knowledge you can then explore partition evolution.

This chapter focuses on the understanding of how hidden partitioning works. Chapters 4 and 5 present tips and best practices for defining, maintaining, and potentially evolving your tables' partitioning strategies with the primary focus on scale and performance.

Benefits of Partitions

As described in Chapter 1, 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. A partition is a way to divide data into subdirectories organized around logical groupings.

To help you understand, the file layout presented in Example 2-10 suggests a transaction history table with 1000 files in it. The table rows that are included in these 1000 files all have values identifying the year they were created. Assume there is no pattern regarding which year values are present within each file.

Example 2-10. Sample file layout of a non-partitioned table.

```
.../hist_tbl/data
              |-- file000
              |-- file001
              |-- file002
              |-- ... file003 - file996 omitted for brevity ...
              |-- file997
              |-- file998
              |-- file999
```

In the scenario described in Example 2-10, potentially all files need to be accessed when a query filter on the year column is present.

You did learn in Chapter 1 that Iceberg's metadata helps with this as the manifest files record columns' min/max values for each file they reference. Even with that, there are issues that still make this a suboptimal solution if the majority of your queries will use a filter on the year column.

First, the compute engine still has to scan all of the file-level min/max values from the manifest files. If the records across all the files in Example 2-10 were naturally persisted in a year sort order, the compute engine could determine a smaller number of files actually need to be read. Unfortunately, without a partitioning strategy defined there is no Iceberg-based gating mechanism to ensure that records for a given year are clumped together in a smaller number of files than the total number of files.

With this limited set of information, the year column appears to be the best solution for the partitioning strategy. Example 2-11 shows how the file layout might look with this partition defined if the distinct values for the year column across all rows range from 2023 - 2026

Example 2-11. Sample file layout of a partitioned table.

```
.../hist_tbl/data
              |-- /year=2023
                    |-- XXIII-aFile
                    |-- ... 200 more files present ...
                    |-- XXIII-anotherFile
              |-- /year=2024
                    |-- XXIV-aFile
                    |-- ... 250 more files present ...
                    |-- XXIV-anotherFile
              |-- /year=2025
                    |-- XXV-aFile
                    |-- ... 367 more files present ...
                    |-- XXV-anotherFile
              |-- /year=2026
                    |-- XXVI-aFile
                    |-- ... 175 more files present ...
                    |-- XXVI-anotherFile
```

Example 2-11 identifies the subdirectory names following a `year=<number>` naming convention. Iceberg guarantees that every row with a specific year column value will be persisted in one of the files present within the folder identified by that year. When adding a `YEAR = 2026` filter on a query, the compute engine will only have

to consider 177 files in the `year=2026` subdirectory instead of the entire 1000 files. This is called *partition pruning* and is another form of a predicate pushdown.

The partitioning solution is still faster than a non-partitioned table that persisted its records in a total order sort across all files backing it. It is true that the compute engine would end up targeting approximately the same number of files that contained the value for a given year. This would suggest that the performance would be very similar.

In reality, a partitioned table will be faster as the partition metadata is stored in the manifest list. This prevents the need to check all the manifest files individually and allows the compute engine to start reading the appropriate data files quicker.

A table can only employ a single partition strategy therefore your initial candidates for which column(s) should be utilized will come from identifying heavily-used query patterns with `WHERE` clauses that naturally allow a small percentage of the total number of rows to be returned.

> **Tip**
>
> As an example, consider immutable time-series datasets, including financial transactions and system events, that span a wide period of time. If the most important queries almost always filter on a much smaller period of time, then partitioning on some coarse-grained level of the timestamp column might be ideal. More partition strategy tips and best practices are presented in Chapter 4.

Now that you know the general concepts and benefits of partitioning, it is time for some hands-on exploration with partitioning. Using the CTAS approach and the Trino TPC-H connector presented in Example 2-1, execute the SQL in Example 2-12 to create and populate a partitioned table. The query at the end will provide some high-level information about the data in the partitions.

Example 2-12. Create a simple partitioned table.

```
CREATE TABLE customer  ❶
WITH (
  type = 'iceberg',  format_version = 3,
```

```
  partitioning = ARRAY['mktsegment']  ❸
)
AS SELECT custkey, name, mktsegment, phone
     FROM tpch.tiny.customer;  ❷

SELECT partition, record_count, file_count
  FROM "customer$partitions";  ❹
```

❶ Create a new table.

❷ Use the data and schema from this query.

❸ Define the partition on the `mktsegment` column.

❹ Query the `$partitions` metadata table.

Table 2-5 displays the results from the last SQL statement in Example 2-12.

Table 2-5. Contents from the `$partitions` metadata table linked to the simple partitioned table.

partition	record_count	file_count
{ mktsegment = MACHINERY }	288	1
{ mktsegment = AUTOMOBILE }	302	1
{ mktsegment = HOUSEHOLD }	294	1
{ mktsegment = BUILDING }	337	1
{ mktsegment = FURNITURE }	279	1

As identified in Table 2-5, the `customer` table has 1500 rows distributed relatively evenly across five partitions.

> **Tip**
>
> DDL syntax can vary among compute engines. The `partitioning` property's value shown in the Trino-based CTAS statement within Example 2-12 is included within an `ARRAY` construct. This is to allow more than one level of partitioning which creates additional levels of subdirectories. Fight the urge to over-partition and only use multiple columns when your testing proves out their worth.

All Iceberg partitioning definitions use transform functions. Not declaring a transform function on a column name simply uses the default function which is called an identity transform.

Transform Functions

The additional transform functions are one reason Iceberg uses the term hidden partitioning. Iceberg partitioning does not require SQL users to know the partition definition to take advantage of its benefits. This is especially true when rows have a timestamp value and you want to define the partition on some less-precise representation of this value such as the day the timestamp represents.

Create an empty table that leverages the `day(ts)` transform.

```
CREATE TABLE orders (
  order_id bigint, cust_id bigint, order_pri varchar,
  order_ts timestamp,
  order_amt double
)
WITH (
  type ='iceberg', format_version = 3,
  partitioning = ARRAY['day(order_ts)']
);
```

Execute the statement in Example 2-13 to load some data into the newly created `orders` table.

Example 2-13. Insert statement for day-level partitioned table.

```
INSERT INTO orders
  SELECT orderkey, custkey, orderpriority,
         current_timestamp(6) - interval '1' day,
         totalprice
    FROM tpch.sf1.orders;
```

Verify there are 1.5 million rows in the table.

```
SELECT format_number(COUNT(*)) FROM orders;
```

Run the SQL from Example 2-13 four more times, but only after incrementing the numeric value wrapped in single-quotes following the `interval` keyword each time. Increment the number by `1` each time so that you submit the SQL with a unique, single, value from the range of `2-5`. If this is confusing, the SQL helper file in the `./Chapter3` folder of the GitHub repository shows all five versions of the `INSERT` statement.

Obtain details on the table's partitioning status by running the query in Example 2-14.

Example 2-14. Query the `$partitions` metadata table linked to the transform partitioned table.

```
SELECT partition,
       format_number(record_count) AS row_count
  FROM "orders$partitions";
```

Table 2-6 displays the results from the execution of the query identified in Example 2-14. There are five partitions present with each containing 1.5 million rows.

Table 2-6. Contents from `$partitions` metadata table linked to the transform partitioned table.

partition	row_count
{ order_ts_day = 2026-03-31 }	1.5M
{ order_ts_day = 2026-03-30 }	1.5M
{ order_ts_day = 2026-03-29 }	1.5M
{ order_ts_day = 2026-03-28 }	1.5M
{ order_ts_day = 2026-03-27 }	1.5M

Notice that the `partition` column's values in Table 2-6 are not as simple as the identity transform partitioned table's `partition` values in Table 2-5. The `customer` table's partition property name is simply the name of the column used in the `partitioning` definition. In this case, the newer `orders` table partition property name is a concatenation of the field the transform function was passed in (`order_ts`) and the transform function name (`day`), with a _ between the two.

Fortunately, when you look in the `/data` folder on the data lake for the `orders` table, you see subdirectories in Example 2-15 that align with the values in Table 2-6.

Example 2-15. Data lake structure of partitioning subdirectories.

```
.../orders/data
           |-- /orders_ts_day=2026-03-27
           |-- /orders_ts_day=2026-03-28
           |-- /orders_ts_day=2026-03-29
           |-- /orders_ts_day=2026-03-30
           |-- /orders_ts_day=2026-03-31
```

Factoring in the number of columns present in a table, the number of available transform functions, and the fact that you can partition on multiple columns, there is a huge number of combinations that could be physically configured for your table's partitioning strategy. If that's not enough, you can actually change your partitioning settings at any point.

Partition Evolution

Historically, data lake tables have to be rebuilt if the partitioning strategy needs to be changed. Iceberg makes this a simple operation. Change the strategy of the `orders` table to see.

```
ALTER TABLE orders
SET PROPERTIES partitioning = ARRAY['order_pri'];
```

The operation is very fast because, as with schema evolution, partition evolution is a metadata change only, too. Verify the change was made by running `SHOW CREATE TABLE orders;` and you will see it called out in the `WITH` properties.

```
partitioning = ARRAY['order_pri'],
```

Running the query in Example 2-14 will create the results shown in Table 2-7. The results still list each of the older, day-based, partitions and continue to have 1.5 million rows in each. Again, the change of the partitioning strategy did not rewrite any data. You will notice that the `partition` column value looks slightly different than it did in Table 2-6. It kept the `order_ts_day` designator from before, but also includes the name of the new partition column, `order_pri`. This is just the `$partitions` metadata table's way of showing you there are multiple partition strategies across the life of the table; especially considering that changing partition configuration did not rewrite any data files. The actual subdirectories on the data lake still look as they did in Example 2-15.

Table 2-7. Data was not rewritten when partitioning changed.

partition	row_count
{ order_ts_day = 2026-03-31, order_pri=NULL }	1.5M
{ order_ts_day = 2026-03-30, order_pri=NULL }	1.5M
{ order_ts_day = 2026-03-29, order_pri=NULL }	1.5M

partition	row_count
{ order_ts_day = 2026-03-28, order_pri=NULL }	1.5M
{ order_ts_day = 2026-03-27, order_pri=NULL }	1.5M

When new data is written, it will be persisted using the current partitioning strategy. Figure 2-4 is sourced from the Iceberg project site's partition evolution page. It visualizes changing partitioning at the end of a month and then switching to a day partition transform of a date field. As the figure shows, Iceberg determines which partitions to include in a query even when the data is stored in multiple partitioning strategies.

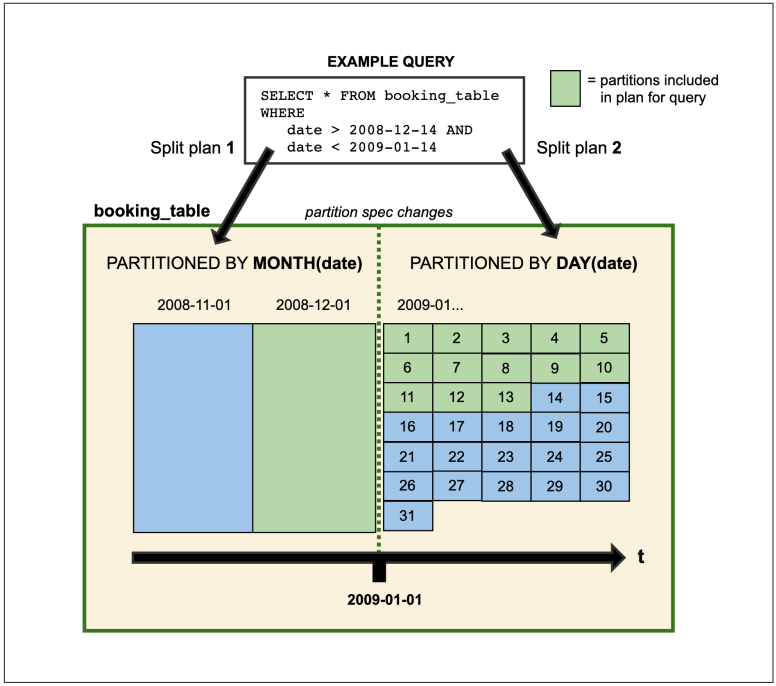


Figure 2-4. Example of data written to different partition strategies.

> **Note**
>
> While changing partitioning strategies does not automatically trigger a rewrite of existing data, primarily because this could be an overly expensive operation, Chapter 5 will present how you could rewrite the existing data if desired.

Add 1.5 million new rows into the table which will all be written using the current, `order_pri`, partitioning strategy.

```
INSERT INTO orders
  SELECT orderkey, custkey, orderpriority,
         current_timestamp(6) - interval '10' day,
         totalprice
    FROM tpch.sf1.orders;
```

Running the query from Example 2-14 shows five more rows are present in addition to the output in Table 2-7. These additional records from `$partitions` confirm that new writes use the current partitioning strategy. Just as before, the `partition` column is showing that there have been multiple partitioning configurations.

partition	row_count
{ order_ts_day = NULL, order_pri = 1-URGENT }	300K
{ order_ts_day = NULL, order_pri = 2-HIGH }	300K
{ order_ts_day = NULL, order_pri = 3-MEDIUM }	299K
{ order_ts_day = NULL, order_pri = 4-NOT SPECIFIED }	300K
{ order_ts_day = NULL, order_pri = 5-LOW }	301K

At this point, the subdirectories on the data lake still contain the ones identified in Example 2-15, but now there are five more folders.

```
.../orders/data
            |-- /order_pri=1-URGENT
            |-- /order_pri=2-HIGH
            |-- /order_pri=3-MEDIUM
            |-- /order_pri=4-NOT+SPECIFIED
            |-- /order_pri=5-LOW
```

> **Tip**
>
> Partition evolution is incredibly easy to do, but actually changing your partitioning *strategy* warrants the same level of strong analysis as when you set it initially. Chapter 4 will provide recommendations, patterns, and best practices for this critical analysis.

With the ability to change data and evolve the schema & partitioning strategies, you need to have a mechanism to isolate testing without changing everything immediately or maybe even a way to back out if you do realize you are worse off than before you started changing everything. Fortunately, those snapshots that get created actually offer ways to do just that.

Snapshot Features

Every time data is changed a new snapshot is created. Since files on the data lake, including metadata files, are immutable, Iceberg had to create replacement files for anything that required a change. At this point, conceptually it was extremely easy to just remember what the metadata looked like before a change. I often wonder if this was part of the initial design or more an epiphany when the creators realized this left them with version history for tables.

By having these snapshot versions of the table we get to actually peer back to a specific snapshot or a general time frame and query the table as it existed in the past. This is called *time travel*. It even has a cool name! Another valuable feature present is the ability to roll back a table to a prior snapshot. So cool!

Those snapshots also offer us features leveraged by software development teams for decades; branching & tagging. This is some pretty cool stuff!

Time Travel

You heard that right! We can run queries that execute on prior states of the table. Let's give it a try.

If you have run all the SQL in the partition evolution section, you have nine million rows in the `orders` table at this time. Using the query in Example 2-16, verify that you have some records for customer identifier `82661`.

Example 2-16. Check for a specific customer in the current snapshot.

```
SELECT * FROM orders WHERE cust_id = 82661;
```

If records are present then we can run our first time travel query. Example 2-17 shows how to reference a point in time for the state of the table. Specifically, it checks the state as of five minutes ago.

Example 2-17. Check for a specific customer as of a point in time.

```
SELECT * FROM orders FOR TIMESTAMP AS OF
    current_timestamp(6) - interval '5' minute
 WHERE cust_id = 82661;
```

If the query in Example 2-17 returned some matching rows, skip the next paragraph.

If the query did not return any rows, or you received a `No version history...at or before...` error message, then you are working very fast. You need to wait until it returns records before you can continue with these hands-on activities. I promise not to punish you again if you are working "too fast."

Go ahead and delete all of those records and then run the query in Example 2-16 again to validate that they are gone.

```
DELETE FROM orders WHERE cust_id = 82661;
```

Try the time travel query in Example 2-17 again to verify that the records have returned. If they have, skip the next paragraph.

If no rows are returned then you have waited more than five minutes since you deleted the records. The snapshot that was current five minutes ago is the one that was created when you deleted them. You are now working "too slow". I should have never chastised you for working too fast! Fortunately, this is an easy fix. Just raise the minute interval until it is higher than when you deleted the records and you will see some rows.

While it may seem confusing due to those potential problems that could occur with our hands-on experimentation, realize that this is still very valuable as often queries are being `AS OF` a very specific point in time and not a sliding window as you just did.

> **Tip**
>
> This is some pretty powerful stuff, but understand that the more snapshots we have, the more storage on the data lake we consume. Chapter 6 offers tips and best practices for table maintenance activities that can expire snapshots to reduce your storage needs.

A more precise way to perform time travel queries is `AS OF` a specific snapshot identifier. Run the query in Example 2-18 to get the list of snapshots that have been created until now.

Example 2-18. List of snapshots.

```
SELECT committed_at, snapshot_id, parent_id
  FROM "orders$snapshots" ORDER BY committed_at;
```

The last two rows in the output from Example 2-18 are shown in Table 2-8. The values have all been generalized so they can be leveraged in additional examples in this section. You will use the specific `snapshot_id` values that you receive in their place.

Table 2-8. Last two records from the `$snapshots` metadata table.

committed_at	snapshot_id	parent_id
...ts for last insert...	44444444444	33333333333
...ts for delete stmt...	55555555555	44444444444

The last two snapshots are shown in Table 2-8 and the final record is the current snapshot. Snapshot `55555555555` indicates its parent is the snapshot for the last `INSERT` statement. Using your specific `snapshot_id` for what is shown as `44444444444` above to time travel to a specific snapshot and see the deleted rows again.

```
SELECT * FROM orders FOR VERSION AS OF 44444444444
 WHERE cust_id = 82661;
```

Pretty cool stuff, but what else can you do with these snapshot identifiers?

Roll Backs

Iceberg provides the option to roll back to a specific table version. This can be very handy for non-production scenarios where you simply want to go back to a prior snapshot and extremely valuable in production if a human or a process committed a change that simply was wrong. Unlike time travel, you can not roll back to a specific point in time. This feature only works when presented with a valid snapshot identifier.

Roll back your table to `snapshot_id` logically referenced as `44444444444` in Table 2-8 and then execute the query in Example 2-16 to see the records for this customer.

```
ALTER TABLE orders EXECUTE
  rollback_to_snapshot(44444444444);
```

> **Warning**
>
> Be aware that if you roll back to a prior snapshot, it becomes your responsibility to replay any changes you want to recover since the roll backed snapshot. As the saying goes, with great power comes great responsibility.

Up until this point it might appear that the most recent `$snapshots` record represented the current snapshot. That's not something you can count on as you can see by running the SQL in Example 2-18 again. Rolling back the `orders` table did not create a new entry in the `$snapshots` table, nor did it change the `committed_at` value for the record which was already there.

The following query identifies the table's current snapshot by leveraging the `$refs` metadata table.

```
SELECT name, type, snapshot_id FROM "orders$refs"
 WHERE type='BRANCH' AND name='main';
```

Okay, but what is so special about a branch named `main` and what is a branch anyway?

Branching

Branching is conceptually what it sounds like. Think of a tree with a *trunk* and branches being created off it as it grows. In Iceberg, the "trunk" is actually a special branch named `main` which has been a standard naming convention in source code repositories for decades. Branching can be applied to several use cases, including:

Experimentation
: Run transformation on a branch before merging changes into the trunk.

Backfill jobs
: Isolate large data rewrites to avoid impacting production.

Safe development
: Allow multiple teams to work on the same dataset concurrently.

What-if analysis
: Test queries on branched data without affecting the production dataset.

Additionally, branching can also be combined with a Write-Audit-Publish (WAP) pattern, allowing you to safely stage changes in a branch, audit them, and then publish to the trunk, providing a robust workflow for managing complex data updates.

As is the theme of this book, the best way to learn is to go hands-on. Let's check it out!

> **Warning**
>
> As of publication time, all branching features are not fully functional with open-source Trino or in Starburst Galaxy. The samples in this section can be run on Starburst Enterprise (release 476-e, or later).

Create a simple table with five partitions.

```
CREATE TABLE branching (data INT, part DATE)
WITH (partitioning = ARRAY['part']);
INSERT INTO branching VALUES
  (10,  DATE '2026-01-01'),
  (20,  DATE '2026-01-02'),
  (-30, DATE '2026-01-03'),
  (40,  DATE '2026-01-04'),
  (50,  DATE '2026-01-05');
```

The data for `2026-01-03` appears to be incorrect. Create a branch to correct it and then list all branches that are present.

```
CREATE BRANCH dev IN TABLE branching;
SHOW BRANCHES FROM TABLE branching;
```

Remember that the trunk is technically a branch named `main`.

Branch
main
dev

As this example is a backfill job use case, empty out the partition being recreated and then perform DML operations to reload it with

the correct information. Add `@ <branch_name>` after the table name to target the changes to the branch.

```
DELETE FROM branching @ dev WHERE part = DATE '2026-01-03';
INSERT INTO branching @ dev VALUES (30, DATE '2026-01-03');
SELECT * FROM branching FOR VERSION AS OF 'dev';
```

Now the data looks correct.

data	part
10	2026-01-01
20	2026-01-02
30	*2026-01-03*
40	2026-01-04
50	2026-01-05

There are two ways to query the trunk.

```
SELECT * FROM branching;
SELECT * FROM branching FOR VERSION AS OF 'main';
```

When specifically looking at the `2026-01-03` data, the `main` branch clearly does not have the changes that were made on `dev`.

data	part
-30	2026-01-03

The `ALTER BRANCH...FAST FORWARD` statement can be used to update the `main` branch from `dev`. Fortunately, this statement will fail if the `main` branch has changed since the `dev` branch was created.

```
ALTER BRANCH main IN TABLE branching FAST FORWARD TO dev;
```

Now you can check to see if the data fix made it into the trunk.

```
SELECT * FROM branching;
```

Figure 2-5 provides a high-level visual representation of the relationship between snapshots and branches, aligned with the example presented in this section.

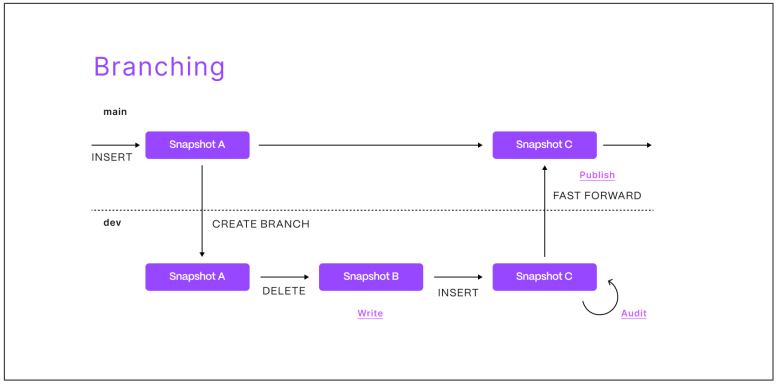


Figure 2-5. Visualization of snapshots and branching.

If this use case implementation were run regularly, it could be a good scenario to apply some kind of auditable label to the snapshot every time data changes are brought back into the trunk.

Tagging

Tagging is that auditable label just described. Tags are fully referenceable with SQL which essentially is a time travel query. This is the case as tags are applied to snapshot identifiers. By default, tags are permanently retained, but do allow a retention period to be set.

> **Warning**
>
> As of publication time, all tagging features are not fully functional with open-source Trino or in Starburst clusters. No formal hands-on examples are presented. Syntax from an alternative compute engine is provided to aid in your understanding of tagging functionality.

Spark SQL syntax for tagging operations is presented in Table 2-9.

Table 2-9. Spark SQL tag lifecycle commands.

Feature	Command
Create at the current snapshot	`ALTER TABLE table CREATE TAG \`historical-tag\``
Create for a specific snapshot	`ALTER TABLE table CREATE TAG \`v1-tag\` AS OF VERSION 123456789;`

Feature	Command
Create as of a point in time	`ALTER TABLE table CREATE TAG `june-data` AS OF SYSTEM_TIME '2024-06-01 00:00:00';`
Create with a retention period	`ALTER TABLE table CREATE TAG `monthly-audit` RETAIN 30 DAYS;`
Replace or update an existing tag	`ALTER TABLE table REPLACE TAG `historical-tag` AS OF VERSION 987654321;`
Delete	`ALTER TABLE table DROP TAG [IF EXISTS] `historical-tag`;`

Both tagging and branching are very powerful features and should be taken advantage of when you have a relevant scenario such as the use cases presented at the beginning of this section that can provide value to your lakehouse.

> **Tip**
>
> Use of branching and tagging will increase your data lake storage needs. It is a general practice to delete branches as soon as their need has been met. It is advisable to apply analysis rigor regarding tagging to determine if a retention period can be set.

In addition to branching and tagging, this chapter presented detailed explanations and hands-on walkthroughs of the power features of hidden partitioning and schema evolution. You were also presented with an understanding of how mutable operations in Iceberg work despite the underlying data lake files being immutable. All of these features create & utilize metadata and are key consumers of the snapshots that are created as data changes.

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.