DREMIO GUIDE

Quick Guide to the Apache Iceberg Lakehouse

Alex Merced - Senior Tech Evangelist, Dremio

dremio

Table of Contents

Apache Iceberg Resource Directory

What is Data Lakehouse?

The idea of a **data lakehouse** emerged as a response to the growing complexity of managing data across data lakes and data warehouses. Each has its strengths—data lakes offer flexibility and scalability for unstructured and semi-structured data, while data warehouses provide the performance and consistency needed for structured data and complex analytics. However, as data scales rapidly, the pain of maintaining these two separate systems becomes more acute. The separation means more effort in moving data, duplicating it across systems, and ensuring consistency between them, which leads to higher costs and operational inefficiencies.

Traditionally, a **data lake** is a centralized repository that stores vast amounts of raw data in its native format, making it an ideal choice for cost-effective storage and flexible data exploration. On the other hand, a **data warehouse** enforces structure, schemas, and query performance, giving you ACID guarantees and optimized analytics capabilities. But as data continues to grow exponentially, so do the challenges of reconciling these two systems. Moving data between a lake and a warehouse becomes more laborious, and keeping data up to date in both systems increases the risk of inconsistencies, delays, and operational bottlenecks.

This is where the **data lakehouse** comes in. By building a composable system where the data lake serves as the storage layer but the warehouse-like functionality is built on top of it, you eliminate the need to shuttle data between systems. You can maximize the flexibility of a data lake while maintaining the guarantees of a data warehouse—such as transactional consistency, performance, and structured query capabilities—all with a **single copy of data**. This allows you to tap into a wide ecosystem of tools without being confined to a specific vendor or architecture, making the lakehouse approach both agile and scalable.

At the heart of a data lakehouse are several composable components. These include the **storage system**, which serves as the foundation and is often based on cloud object storage or distributed file systems. Next is the **file format** that stores data efficiently, such as Apache Parquet or ORC, ensuring that data can be accessed quickly and in a structured way. Then comes the **table format**—such as Apache Iceberg or Delta Lake—which organizes files into tables while enabling features like time travel and version control. The **catalog** plays a crucial role by tracking metadata about these tables, ensuring consistency and ease of access. Finally, the **query engine** (like Dremio or Spark) allows you to run operations on these tables directly from the data lake without moving data, providing the best of both worlds: a lake's scalability with a warehouse's performance.

LEARN MORE: https://drmevn.fyi/icebergQG-lakehouse

What is Table Format?

A data lakehouse table format is a key component that enables the smooth functioning of a data lakehouse by treating a collection of data files as a singular, cohesive dataset or table. Traditionally, the idea of a table in a data lake was loosely based on directory structures. The Apache Hive table format, for instance, recognized all the files within a directory as part of the same table. While this was a flexible and straightforward approach, it introduced significant challenges in terms of performance and safety. When a dataset grew large, listing and managing files from directories involved a lot of I/O operations, slowing down queries and increasing the risk of failures. Furthermore, concurrent or multi-partition transactions were unsafe, leading to inconsistencies in the data.

The need for more robust and performant solutions gave rise to a new generation of table formats that rethought the way tables are managed in data lakes. Instead of relying on the directory listing model, these modern formats introduced the concept of metadata layers. In this approach, a table is no longer just a directory of files, but a set of well-structured metadata that tracks which files make up the table. This metadata-driven design enables important features that were previously missing, like ACID guarantees (ensuring consistency and durability of transactions), schema evolution (allowing the structure of the table to change over time without disrupting queries), and time travel (the ability to query historical versions of data).

These modern table formats, such as Apache Iceberg, Apache Hudi, Apache Paimon, and Delta Lake, have transformed how we manage data in a lakehouse architecture. By providing more efficient ways to handle large datasets, supporting transactional operations, and enabling optimizations like partition pruning and file compaction, they address the limitations of older table formats like Hive. This shift improves query performance and ensures data reliability, making the data lakehouse a more versatile and powerful architecture for managing large-scale data platforms.

Learn More: https://drmevn.fyi/icebergQG-tableformat

What is Apache Iceberg?

Apache Iceberg is a powerful data lakehouse table format originally developed by Netflix to address many of the limitations of the Hive table format. As data scales and becomes more complex, the shortcomings of the traditional Hive approach—such as slow transactions, lack of consistency, and difficulties with schema evolution—became more apparent. To overcome these challenges, Netflix created Apache Iceberg, which was later contributed to the Apache Software Foundation and has since become a top-level Apache project. Its innovative architecture redefines how tables are tracked and managed, making it a critical component in modern data lakehouse implementations.

At the core of Apache Iceberg's design is its ability to track tables through a history of snapshots, with each snapshot representing the state of the table at a particular point in time. These snapshots contain a list of files that make up the table during that time, allowing the system to maintain a reliable history of changes. This is achieved through a tree of metadata files, including the metadata.json, which holds critical information like schema, partitioning, and snapshot history. Manifest lists track which groups of files belong to a particular snapshot, while manifests maintain a detailed list of the files that make up each snapshot. This structured metadata approach dramatically reduces the need for costly file system operations, speeding up queries and operations.

This design enables ACID guarantees (ensuring reliable, atomic transactions), schema evolution (allowing for table structure changes over time without breaking queries), and time travel (the ability to query past versions of a dataset). These features make Iceberg a robust solution for managing data at scale. Additionally, Iceberg introduces unique capabilities such as partition evolution, allowing partition strategies to change over time without rewriting the table, and hidden partitioning, which automates partitioning based on table data without requiring the user to define partitions manually. These advanced partitioning features make Iceberg more flexible and ergonomic than earlier table formats, giving users the benefits of optimal data organization with minimal manual effort.

Learn More: https://drmevn.fyi/icebergQG-iceberg

Why Apache Iceberg?

1. Features

Apache Iceberg stands out for its powerful and unique features that go beyond what other table formats offer. **Partition Evolution** allows you to change partitioning schemes over time without rewriting the table, enabling flexibility as your data scales and requirements evolve. With **Hidden Partitioning**, Iceberg eliminates the need to define partitions manually, automating the process based on data characteristics, making it more user-friendly and efficient. Another standout feature is **Table Level Branching**, which enables version control for your datasets, allowing for safe experimentation and isolation of changes before merging them into production.

Iceberg's capabilities extend even further through its **catalog integration**, offering enterprise-grade features via tools like **Apache Polaris (incubating)**, which enables **RBAC (Role-Based Access Control)** and **catalog federation**. This allows multiple Iceberg catalogs to be treated as a unified system, streamlining management across environments. Additionally, through integrations with **Project Nessie**, Iceberg supports **catalog-level versioning**, providing a version-controlled catalog that makes managing data at scale easier and more reliable. These features combine to offer unprecedented flexibility, governance, and ease of use, making Iceberg a truly modern table format.

2. Ecosystem

Another major reason to choose Apache Iceberg is its rich and expanding **ecosystem**. Iceberg is supported by a wide variety of tools that go beyond basic read and write operations, offering solutions for specialized use cases like **table optimization**, **streaming ingestion**, and more. Whether you're working with batch or real-time data, Apache Iceberg gives you multiple options for every workload. For example, Iceberg supports streaming through engines like **Flink** and **Spark** while allowing batch processing with tools like **Dremio**, **Presto**, or **Trino**. In addition, Iceberg's integration with orchestration tools like **Apache Airflow** allows you to build and maintain complex data workflows with ease.

Iceberg's commitment to flexibility means you're never locked into a single solution. Whether you need to scale your data platform, optimize query performance, or manage transactional data, Iceberg provides an array of tools and integrations, ensuring that the format can meet any workload or architectural demand.

3. Community

As an **Apache Project**, Iceberg is built with a strong commitment to **transparent, community-driven development**. The project fosters an inclusive and open environment where anyone can contribute to its evolution. Development discussions happen in **public meetings** and on a **public mailing list**, ensuring that all voices are heard, and the direction of the project is clear to everyone involved. Iceberg's public repository is where development happens, with no hidden or private forks, making sure that the community can actively participate and contribute.

One of the key benefits of this community-driven approach is that any breaking changes or new features are communicated well in advance, preventing surprises and ensuring smooth transitions for users. Whether you're a casual user or a major contributor, the Iceberg community encourages participation, making it a transparent, collaborative, and user-friendly project that is always evolving in a way that benefits its entire ecosystem.

What is Apache Iceberg's Metadata?

Apache Iceberg's metadata system is one of its core strengths, offering a highly structured way to track the state of tables and files while maintaining performance and scalability. At the heart of this system is the **metadata.json** file, which acts as the central point of truth for an Iceberg table. This file tracks essential information such as the **table schema**, **partitioning strategy**, and a **history of snapshots**. Each snapshot represents a state of the table at a specific point in time, allowing you to track how the table has evolved. The **metadata.json** also records information about which snapshots are currently active and which files are associated with those snapshots, making it possible to efficiently query and operate on large datasets.

The **manifest list** plays a critical role by acting as an index that tracks which groups of files belong to a particular snapshot. Instead of Iceberg scanning an entire directory of files, the manifest list organizes the table into manageable groups, improving both query performance and file organization. Each manifest list references multiple **manifests**, which are essentially detailed lists of the files included in a snapshot. A manifest contains granular metadata about each file, including file paths, row counts, data ranges, and partitioning information. This level of detail allows Iceberg to perform optimizations like **partition pruning** and **data skipping**, reducing unnecessary I/O during queries.

Finally, **delete files** are used to track data that has been deleted without physically removing files from storage, a key feature that enables **fast row-level deletes and updates for tables** set as **"merge-on-read"**. Iceberg can apply delete files during queries to filter out deleted data, ensuring that you get the correct results without the need to rewrite large files every time a delete operation occurs. This allows Iceberg to handle deletions efficiently while still maintaining ACID guarantees and providing the ability to query historical data. By organizing all these components—metadata.json, manifest lists, manifests, and delete files—Iceberg achieves a high level of efficiency and reliability, making it a powerful tool for managing data in modern lakehouse architectures.

Learn More:

- **Metadata.json:** https://drm.dtlkhs.com/metadatajson
- **Manifest Lists:** https://drm.dtlkhs.com/manifestlist
- **Manifests:** https://drm.dtlkhs.com/manifests
- **Delete Files:** https://drm.dtlkhs.com/deletefiles

What are Apache Iceberg Catalogs?

In Apache Iceberg, catalogs play a crucial role in organizing and managing tables within a data lakehouse environment. However, it's important to distinguish between **lakehouse catalogs** in Iceberg and **metadata catalogs** like Colibra or Alation, as they serve different purposes. Iceberg's **lakehouse catalogs** are primarily designed to track **table references**, allowing data tools like Dremio, Spark, or Flink to easily discover, manage, and query tables. These catalogs enable tools to access the correct version of a table, handle transactions, and enforce consistency across multiple users and engines.

In contrast, **metadata catalogs** such as Colibra or Alation focus on **human discoverability**. They provide additional context, lineage, and business metadata, helping data teams understand the data's meaning, quality, and usage. These systems don't track the physical data files or snapshots but instead layer on top of systems like Iceberg to offer a more holistic view of data assets, often with governance and compliance features. So while Iceberg catalogs manage the technical side of data discovery for tools, metadata catalogs help people find, understand, and trust the data they are working with.

The **role of Iceberg catalogs** extends beyond simple table reference tracking—they also ensure that queries are accessing the **correct version of the data** and provide mechanisms to handle **concurrent transactions**. Iceberg's table format supports features like **snapshot isolation**, ensuring that multiple users can interact with a table simultaneously without causing conflicts or inconsistencies. The catalog acts as the gatekeeper, keeping track of which metadata.json of the table the engine should be working with. This capability is key to maintaining **ACID guarantees** and supporting operations like time travel, schema evolution, and safe, concurrent writes.

Iceberg catalogs typically fall into two categories: **file-system catalogs** and **service catalogs**.
File-system catalogs, such as the Hadoop catalog, store metadata directly in a file system like HDFS or cloud object storage. While simple to set up, they are not recommended for production environments because they don't scale well and lack the advanced transactional capabilities needed for large, multi-user workloads. File-system catalogs can suffer from bottlenecks in performance as they rely heavily on file system I/O, making them less reliable in distributed environments.

On the other hand, **service catalogs**, such as the **Hive Metastore**, **JDBC**, or **REST-based catalogs**, provide a more robust and scalable solution for managing Iceberg tables. These service-based catalogs are built to handle production workloads, offering features like **high availability**, **transactional consistency**, and better performance for concurrent operations. By using a service catalog, you ensure that your data lakehouse is able to handle the demands of modern data workloads, including large-scale analytics, streaming ingestion, and high-frequency updates. This is why service-based catalogs are strongly recommended for production environments, providing the scalability and reliability that file-system catalogs lack.

Learn More: https://drmevn.fyi/icebergQG-catalogs

How do Engines Read Iceberg Tables?

When a query is run on an Apache Iceberg table, the process involves multiple steps to ensure the engine retrieves the correct data in an efficient manner. Here's a breakdown of how this process unfolds:

1. **Request to the Catalog**
   The first step is for the engine (like Spark, Dremio, or Flink) to send a **request to the catalog** for the specific table it wants to query. The catalog, which manages references to the tables, acts as a directory of all the Iceberg tables and their current metadata.

2. **Catalog Returns Metadata Location**
   In response, the catalog returns the **file-system address** where the latest metadata.json file for that table is stored. This file contains all the critical information the engine needs to understand the table's structure and history.

3. **Engine Reads Metadata**
   The engine retrieves and reads the **metadata.json** file from the provided address. This file contains essential information about the table, including its **schema**, **partitioning strategy**, and **snapshot history**. The snapshot history is key, as it lets the engine know how the table has changed over time and which version of the table to query.

4. **Target Snapshot Identified**
   Within the metadata.json, the engine identifies the **target snapshot** for the query. This snapshot represents the state of the table at a particular point in time. The metadata file also provides a **file address** to the **manifest list** that corresponds to this snapshot, which contains references to the files in the snapshot.

5. **Manifest List Pruning**
   The engine retrieves the **manifest list** and begins analyzing it. This list contains multiple **manifests**, each representing a group of files in the table. The engine prunes the manifests by checking the **partition values** against the query's filter criteria. Any manifest that contains partitions outside the scope of the query is ignored, which helps narrow down the set of files that need to be processed.

6. **Manifest File Pruning**
   Once the relevant manifests are identified, the engine retrieves the individual **manifests**. Each manifest contains **column statistics** for the files in that snapshot, such as minimum and maximum values for each column. Using these stats, the engine can **prune individual files**, further refining the list of data files by discarding any that don't match the query's conditions.

7. **Scan Plan and Query Execution**
   After pruning, the engine generates a **scan plan**, which is the final list of files that need to be read to satisfy the query. This plan is passed to the engine, which executes the query, scans the relevant files, and returns the results back to the user.

This process ensures that Iceberg tables are read efficiently by leveraging metadata, partition pruning, and file-level statistics to minimize the amount of data that needs to be scanned. It's a key part of how Iceberg maintains high performance even when working with large datasets.

How does an Engine Write to an Iceberg Table?

Writing data to an Apache Iceberg table is a carefully orchestrated process designed to ensure transactional consistency, even in the presence of concurrent operations. Here's how the process typically works:

1. **Request Metadata Location from Catalog**
   The first step in writing data to an Iceberg table is for the engine to send a request to the **catalog** to retrieve the location of the latest metadata.json. This file contains the most recent state of the table, including schema, partitioning, and snapshot history, which will guide the upcoming write operation.
2. **Project Sequence Number for the Next Snapshot**
   After retrieving the metadata.json, the engine uses it to **project the next sequence number** for the new snapshot. This sequence number ensures that every snapshot of the table is uniquely identified and allows Iceberg to maintain the correct order of operations.
3. **Engine Writes New Data Files**
   With the sequence number in mind, the engine proceeds to write the **new data files**. These files represent the actual dataset being added to or updated in the table.
4. **Write Manifests Listing New Data Files**
   Once the data files are written, the engine creates new **manifests**, which are files that list the newly added data files. These manifests contain file-level metadata, including statistics like row counts, column ranges, and partition information, which will help the system optimize future reads.
5. **Write Manifest List for the New Snapshot**
   After the individual manifests are written, the engine compiles them into a **manifest list**. This list acts as an index for the new snapshot, grouping the manifests together and providing a complete view of all the files that make up the new snapshot.
6. **Write a New Metadata.json for the Updated Table State**
   The engine then writes a new **metadata.json** file to represent the new state of the table. This updated metadata file contains the new snapshot and tracks the new manifests, as well as any changes to the table's schema or partitioning strategy.
7. **Double Check Projected Sequence Number with Catalog**
   Before committing the changes, the engine requests the metadata.json from the catalog again to ensure that the **projected sequence number** is still valid. This step ensures that no other concurrent operations have modified the table state during the write process, which could potentially cause conflicts.
8. **Commit to the Catalog or Retry**
   If the projected sequence number is still valid, the engine commits the changes to the catalog, updating the reference to point to the new metadata.json and thus the new snapshot. However, if the sequence number is no longer valid—due to another operation modifying the table simultaneously—the engine will retry the transaction. This process will continue until the engine successfully commits the new state or the maximum number of retries is exhausted.

By carefully handling metadata updates and ensuring that sequence numbers are checked before committing, Apache Iceberg provides **ACID guarantees** during write operations, even in a distributed and concurrent environment. This ensures that the table remains consistent and reliable, no matter how many operations are occurring at the same time.

What is the Apache Iceberg REST Catalog Specification?

The **Apache Iceberg REST Catalog Specification** is an OpenAPI standard that defines a set of REST API endpoints for Iceberg table operations. This specification enables any catalog to implement a uniform interface for **reading, writing, and altering tables**, making it easier for Iceberg clients to interact with a variety of catalogs without needing custom integrations. By standardizing these operations, the REST Catalog Specification creates a seamless way for any client to work with multiple catalogs that support this interface, such as **Polaris**, **Nessie**, **Gravitino**, and others.

One of the key benefits of using a catalog that supports the **REST Catalog Specification** is the assurance of **maximum ecosystem interoperability**. Since the REST API endpoints are standardized, a single Iceberg client can interact with multiple catalogs regardless of the underlying implementation. This means that whether you're working in a cloud-native catalog like Polaris or a versioned catalog like Nessie, your Iceberg lakehouse can leverage a broad ecosystem of tools, ensuring flexibility and scalability as your platform grows.

Another innovative feature of the REST Catalog Specification is the concept of **vended credentials**. The specification includes endpoints that allow catalogs to provide **short-lived storage credentials** at runtime, reducing the need for complex client-side configuration. This is especially useful for environments where security and compliance are critical, as it minimizes the risk associated with long-lived credentials and makes client setup more intuitive. By simplifying access control and credential management, the REST Catalog Specification significantly enhances the operational efficiency of working with Iceberg tables across diverse storage environments.

In summary, the **Apache Iceberg REST Catalog Specification** offers a standardized, interoperable way to manage Iceberg tables across different catalogs. It ensures that your lakehouse architecture remains flexible, while also introducing innovations like vended credentials to make management more secure and user-friendly.

Learn More: https://drmevn.fyi/icebergQG-restcatalog

What is Partition Evolution?

One of the standout features of **Apache Iceberg** is its ability to support **partition evolution**, which allows you to modify the partitioning scheme of a table without the need to rewrite existing data. In Iceberg, many key details about the table, including partitioning information, are tracked in the **metadata**, rather than being embedded within the data files themselves. This decoupling of partitioning logic from the physical data files gives Iceberg a level of flexibility not available in other table formats.

When it comes to partitioning, the **partitioning scheme**, the **partition values**, and which partitioning scheme was in use when a particular data file was written are all recorded in the Iceberg table's metadata. This means you can alter the partitioning strategy over time—say, starting by partitioning your data by **year**, and then later refining it to **month** or even **day**—without having to rewrite or reorganize the entire table. The table's metadata keeps track of the different partitioning schemes used at various points, ensuring that queries can still access all the data correctly, regardless of when it was written or under which scheme.

This flexibility makes **partition evolution** a powerful feature, especially in dynamic environments where data needs and query patterns change over time. You can experiment with different partitioning strategies or adjust your partitioning as your use cases evolve, without incurring the expensive overhead of reprocessing the entire dataset. By enabling this kind of adaptability, Iceberg offers a more ergonomic and scalable approach to partitioning than other data formats, where changing partitioning often requires significant manual effort and data movement.

The ability to evolve partitioning is a unique feature in Iceberg, setting it apart from other table formats like Delta Lake or Apache Hudi. It allows organizations to optimize their data layout as they scale, ensuring that performance remains high and storage costs are minimized, all without disrupting existing workloads.

Learn More: https://drmevn.fyi/icebergQG-partitionevolution

What is Hidden Partitioning?

In **Apache Iceberg**, partitioning is more than just tracking the raw value of a particular column—it also supports **partition transforms**, which allow for more flexible and efficient partitioning without persisting the transformed values in the data files themselves. Instead, Iceberg tracks the **ranges of transformed values** in the **metadata**, enabling **partition pruning** based on these values without needing to store or query the transformed data directly. This concept of **hidden partitioning** simplifies the process for both data engineers and analysts by abstracting the complexity of partition transformations.

Iceberg provides several partition transform options, such as **YEAR**, **MONTH**, **DAY**, **HOUR**, **TRUNCATE**, and **BUCKET**. When a transform is applied to a column, Iceberg records the transformed values in the metadata rather than embedding them in the data files. For example, suppose you partition a table by the YEAR of a timestamp column. In that case, Iceberg doesn't need to store the year separately—it simply tracks the year partitions in the metadata. As a result, queries can take advantage of partition pruning based on the transformed value without having to modify the data files or include additional columns in the query filters.

For example:

- **YEAR(timestamp_column)**: Instead of storing the year separately, Iceberg will track the partition by the year in the metadata, allowing for efficient queries on specific years without altering the original data.
- **MONTH(timestamp_column)**: Similarly, you could partition by month, enabling faster queries for specific months without persisting the month value in each row.
- **BUCKET(id_column, 10)**: You can bucket a column into a specific number of partitions (e.g., 10), distributing the data across buckets for more even distribution and faster access.

With hidden partitioning, **data engineers** can easily implement and modify complex partitioning strategies without physically restructuring the data, while **data analysts** can query the original columns without needing to understand or apply the partition transformations themselves. This makes partitioning more intuitive, scalable, and efficient, ensuring that even as partitioning strategies evolve, performance remains high and queries stay simple.

Learn More: https://drmevn.fyi/icebergQG-hiddenpartitioning

Migrating to Apache Iceberg

Migrating your existing data to **Apache Iceberg** offers a range of benefits, including ACID guarantees, time travel, schema evolution, and optimized query performance. When it comes to performing the migration, there are generally two main approaches: **in-place migration** and **rewrite migration**. Each approach has its own use cases and trade-offs, depending on your requirements and existing infrastructure.

In-Place Migration

An **in-place migration** allows you to take your existing **Parquet** (or other file formats) data and convert it into an Iceberg table **without rewriting** the data. This approach is ideal for large datasets where rewriting would be time-consuming and resource-intensive. In an in-place migration, you simply add the existing Parquet files to an Iceberg table using either the add_files or migrate procedures in **Apache Spark**. Another tool, **Apache XTable**, can also be used to automate this process.

Rewrite Migration

The second approach involves rewriting the data into a new Apache Iceberg table. This method offers the advantage of fully reformatting the data according to Iceberg's optimized structure, potentially resulting in better performance and the ability to take full advantage of Iceberg features like **hidden partitioning** and **partition evolution**.

Engines like **Dremio** make this process easier by providing query federation capabilities, allowing you to move data from various sources into an Iceberg table seamlessly. For instance, Dremio's **COPY INTO** command makes it straightforward to ingest files in formats like **JSON**, **CSV**, or **Parquet** into an Iceberg table, all while handling schema detection and optimization automatically. This method is particularly useful when you want to consolidate data from multiple formats into a single Iceberg table or restructure your existing data for improved performance.

Blue/Green Migration Plan

Whichever migration strategy you choose, it's essential to implement a **blue/green migration plan**. In this approach, the **blue environment** represents the existing production system, while the **green environment** represents the new Iceberg table and ecosystem. Both environments operate side by side during the migration, allowing you to validate the new system without disrupting your current workflows.

The process typically involves:

1. Migrating a subset of data to the **green environment** (Iceberg table).
2. Running both systems in parallel for a period, validating that the new Iceberg table behaves as expected in terms of data correctness, performance, and query results.
3. Once the new system is validated, redirecting all traffic from the blue (old) environment to the green (new) environment.

This **blue/green** migration approach minimizes risk, ensures a smooth transition, and provides a fallback in case any issues arise during the migration process. By carefully planning and executing your migration with this strategy, you can confidently transition your data platform to Iceberg while maintaining the integrity and availability of your data.

Maintaining Apache Iceberg Tables

Maintaining an **Apache Iceberg** table is a key responsibility for data engineers in a lakehouse architecture. Since Iceberg-based lakehouses are **composable** systems, they don't automatically abstract optimizations like some tightly coupled data warehouse solutions. Instead, the optimization tasks fall on the data engineers, who need to ensure that tables are performing efficiently and maintaining a manageable storage footprint. There are three primary ways to optimize your Iceberg lakehouse tables:

1. Compaction

Compaction is the process of rewriting many smaller data files into fewer, larger files to reduce **I/O latency**. Over time, especially when ingesting data incrementally or through small streaming batches, tables can accumulate a large number of small files. This leads to inefficient query performance, as each file incurs overhead during reads. Compaction consolidates these smaller files, making queries faster by reducing the number of files that need to be scanned.

2. Clustering

While rewriting files during compaction, you can take optimization a step further with **clustering**. This involves sorting data within the files based on commonly queried columns, so that related data is stored together. This reduces the number of files included in the scan plan when running queries that filter on those columns. By clustering data effectively, you can significantly reduce query times by limiting the data that needs to be read for a specific query, thereby lowering overall I/O.

3. Expiring Snapshots

Iceberg supports **time travel**, which allows you to query historical snapshots of your data. However, files associated with older snapshots are retained indefinitely unless they are manually expired. To manage storage efficiently, it's important to periodically **expire snapshots**. This process removes unneeded files that are no longer part of any active snapshot, freeing up storage and reducing long-term data management costs. Expiring snapshots ensures that old data doesn't bloat your storage unnecessarily.

Approaches to Maintenance

There are two main approaches to maintaining Iceberg tables—**manual** and **automated**. Each has its advantages depending on your workload and infrastructure needs.

Manual Maintenance

You can manually orchestrate compaction, clustering, and snapshot expiration jobs using available commands in Iceberg-compatible engines. For instance, in **Apache Spark**, you can use procedures like rewrite_data_files for compaction and expire_snapshots to clean up old data. In **Dremio**, you can leverage commands like OPTIMIZE to compact files, and VACUUM to expire old snapshots. While manual maintenance offers granular control over when and how you optimize tables, it can be time-consuming, especially in larger environments with frequent data changes.

Automated Maintenance

For more efficient management, automated solutions are increasingly available to help maintain Iceberg tables. Platforms like **Dremio** and **Upsolver** offer services to automate the optimization of your Iceberg tables. For example, **Upsolver** can **stream data** into your Iceberg tables while simultaneously optimizing them as part of the streaming process. Similarly, **Dremio** allows you to set up **scheduled optimization** jobs, running compaction, clustering, and snapshot expiration on a daily, weekly, or custom interval. Automation reduces the operational burden of maintaining tables, allowing data engineers to focus on more strategic tasks while ensuring that tables remain performant and storage-efficient.

As the Iceberg ecosystem grows, more options are becoming available to streamline and automate these essential maintenance tasks, making it easier to manage Iceberg tables at scale.

Learn More: https://drmevn.fyi/icebergQG-maintenance

DataOps, CI/CD and Data-as-Code

Implementing **DataOps** practices with **Apache Iceberg** tables enables you to automate and orchestrate complex workflows across your data platform, ensuring that all operations are executed consistently, reliably, and with built-in validation. By integrating tools like **Apache Airflow** and **dbt (data build tool)** into your workflow, you can orchestrate operations on Iceberg tables across various engines such as **Dremio**, **Apache Spark**, **Snowflake**, and others. These tools allow you to set up automated pipelines for tasks like data ingestion, transformation, validation, and optimization, ensuring that every operation occurs in the correct sequence and with the necessary data checks.

For example, with **Apache Airflow**, you can schedule and manage complex workflows that involve running Iceberg table operations across different environments. Whether it's optimizing your tables, running validation queries, or orchestrating data migrations, Airflow ensures that tasks are run in the right order with dependencies clearly defined. Meanwhile, **dbt** can be used to transform and model data in Iceberg tables, allowing you to manage your SQL-based transformations with version control and built-in testing, ensuring data quality throughout the pipeline.

In addition to workflow orchestration, **Dremio's integrated open-source powered lakehouse catalog** offers powerful features to enhance your DataOps processes. With **catalog versioning** that uses **Git-like semantics**, you can **branch, tag, and merge** changes to your catalog. This means you can isolate workloads by creating separate branches for development or testing, without affecting the production environment. You can also create **zero-copy environments**, which allows you to work on the same data across different branches without duplicating it, saving storage and enabling agile development practices.

This version-controlled approach also provides robust disaster recovery options. In the event of an issue, you can easily recover from failures by performing a simple **rollback** to a previous version of the catalog. This ensures that your Iceberg tables remain stable, and you can quickly restore your environment to a known good state without losing data.

By integrating DataOps practices with Apache Iceberg and leveraging tools like Airflow, dbt, and Dremio's catalog versioning, you can automate, validate, and scale your data operations while maintaining flexibility and control over your data workflows. This combination ensures that your data processes are agile, repeatable, and resilient to change or failure.

Table Format Interoperability

As organizations adopt a variety of table formats in their data lakehouse architectures, ensuring **interoperability** between formats like **Apache Iceberg**, **Delta Lake**, and **Apache Hudi** becomes increasingly important. There are three main approaches to working with data across different table formats, each offering different levels of flexibility and integration, depending on the tools and technologies in use.

1. Engine Support for Multiple Formats

One of the most straightforward ways to achieve table format interoperability is by using **engines that support multiple formats**. For example, **Dremio** can natively read both **Apache Iceberg** and **Delta Lake** tables, making it easy to join tables and perform operations across these formats without having to worry about data conversion. This flexibility allows data teams to work with datasets stored in different formats seamlessly, providing access to the unique features of each format while running queries or analyses in a unified way.

2. Using Apache XTable

Apache XTable is another approach that facilitates working across table formats by allowing you to generate **metadata** for a table in one format (such as Iceberg, Hudi, or Delta) in another format. This means you can create metadata in **Apache Iceberg** for a table that was originally stored as **Delta Lake** or **Apache Hudi**, making the table readable in the new format. While this approach offers a degree of flexibility for reading tables across different formats, it's important to note that it doesn't involve rewriting the underlying data. As a result, you can read the data in the target format but may not have full access to the advanced features of that format unless the data is rewritten accordingly.

3. Delta Lake's Unity Catalog and Uniform Feature

For **Delta Lake** users, **Databricks' Unity Catalog** offers a feature called **Uniform**, which enables Delta tables to asynchronously generate **Apache Iceberg** metadata. This allows tables to be read as both Delta and Iceberg formats, although with some limitations. While you can read the table as Iceberg, all writes must still occur in the Delta format. This capability is particularly useful for organizations transitioning from Delta to Iceberg, as it allows for cross-format reads without fully committing to an immediate migration.

Trade-offs and Considerations

While tools like **XTable** or **Unity Catalog's Uniform** make it possible to read tables in multiple formats, there are trade-offs to consider. If you aren't writing your data in Apache Iceberg, you lose out on some of its most valuable features, such as **partition evolution** and **hidden partitioning**. These features are key to Iceberg's efficiency and flexibility and are made possible by how Iceberg writes and manages data. By only reading data in Iceberg and not writing to it, you won't fully benefit from Iceberg's advanced capabilities in terms of partition management, schema evolution, and query optimization.

In summary, there are multiple pathways to achieving interoperability between table formats like Apache Iceberg, Delta Lake, and Apache Hudi, each offering different levels of integration. Whether through engines like Dremio, metadata translation with Apache XTable, or the Uniform feature in Unity Catalog, organizations can work across formats, though they should carefully consider the trade-offs, especially when it comes to leveraging Iceberg's full feature set.

Apache Iceberg for the Business User

While **Apache Iceberg** offers a robust solution for managing analytics data throughout its lifecycle, delivering that data in an **accessible and easy-to-use way** for business users remains crucial. This is where platforms like **Dremio** come in, providing a seamless experience for Iceberg-based data lakehouses. Dremio positions **Apache Iceberg** as a first-class citizen, enhancing its usability and unlocking even more value for data lakehouse architectures.

Here's how **Dremio** amplifies the benefits of Apache Iceberg:

- **Read, Write, and Optimize**: Dremio natively supports the ability to **read, write, and optimize** Iceberg tables directly within its platform. This enables users to leverage all the advanced features of Iceberg, like ACID guarantees, partition evolution, and schema evolution, while optimizing tables for performance without leaving Dremio.

- **Integrated Lakehouse Catalog**: With Dremio's **integrated lakehouse catalog**, you can manage your Iceberg tables efficiently, complete with **catalog versioning**. This allows for branching, tagging, and merging changes in the catalog, enabling version control for your datasets and easy recovery in case of issues.

- **Fine-Grained Access Controls**: Dremio offers **fine-grained access controls**, enabling you to manage who has access to specific datasets and tables within your Iceberg lakehouse. This is essential for maintaining data governance and ensuring that only authorized users can interact with sensitive data.

- **Data Enrichment Across Sources**: Dremio allows you to query not just your Iceberg tables, but also data from **databases, data lakes, and data warehouses**, making it easy to **enrich and model your data** from various sources in one place. This unified view helps in blending disparate datasets and creating powerful analytics solutions.

- **Automatic Optimization of Iceberg Tables**: Dremio automatically optimizes Iceberg tables, ensuring they are continuously maintained for optimal performance, without requiring manual intervention. This includes tasks like compaction and clustering, making your data operations more efficient.

- **Multiple Delivery Interfaces**: Dremio offers multiple ways to **deliver data** to any use case via its **REST API**, as well as **JDBC**, **ODBC**, and **Arrow Flight** interfaces. This makes it easy to connect to your preferred analytics tools or pipelines, whether you're building BI dashboards or integrating with machine learning models.

- **Semantic Layer**: Dremio's **semantic layer** allows you to define your key business data assets once and reuse them across various teams and applications. This creates a unified view of your data, ensuring consistent definitions and easier access for business users.

- **Data Acceleration with Reflections**: One of Dremio's standout features is its **Reflections**, which provide the benefits of traditional cubes and materialized views, but with far less manual effort. Reflections built on top of Iceberg data **refresh automatically and incrementally** when data changes, keeping everything up-to-date without requiring extra configuration or maintenance. This enables faster query performance for both exploratory analysis and production workloads.

With Dremio, you can easily connect to your **Iceberg tables**, model your **data warehouse** alongside data from other sources, **accelerate key data assets** through Reflections, and **deliver insights** to business users via BI dashboards or to data scientists using Python for AI/ML models. This comprehensive platform not only enhances Iceberg's capabilities but also makes it easier to deliver the value of your data to the business.

Learn More: https://drmevn.fyi/icebergQG-dremiolakehouse

Tutorials & Resources

Apache Iceberg Fundamentals

(FREE BOOK) Apache Iceberg - The Definitive Guide:
https://drmevn.fyi/icebergQG-icebergbook

Free Apache Iceberg Crash Course:
https://drm.dtlkhs.com/icebergcourse

Apache Iceberg Tutorials

Hands-on with Apache Iceberg using Apache Spark & Dremio:
https://drmevn.fyi/icebergQG-handsoniceberg

Intro to Apache Iceberg with Dremio, Nessie and Minio:
https://drmevn.fyi/icebergQG-introtoicebergtut

- **From Postgres to BI Dashboard:**
  https://drmevn.fyi/icebergQG-postgres2iceberg
- **From SQLServer to BI Dashboard:**
  https://drmevn.fyi/icebergQG-sqlserver2iceberg
- **From MongoDB to BI Dashboard:**
  https://drmevn.fyi/icebergQG-mongodb2iceberg
- **From MySQL to BI Dashboard:**
  https://drmevn.fyi/icebergQG-mysql2iceberg
- **From JSON/CSV/PARQUET to BI Dashboard:**
  https://drmevn.fyi/icebergQG-jsoncsv2iceberg
- **From Elasticsearch to BI Dashboard:**
  https://drmevn.fyi/icebergQG-elastic2iceberg
- **Using Kafka Connect with Apache Iceberg:**
  https://drmevn.fyi/icebergQG-kafka2iceberg

* All the above tutorials can be done locally from your laptop, the only requirement being the installation of Docker Desktop.