Getting Started With Apache Iceberg

TED GOOCH
STAFF SOFTWARE ENGINEER, STRIPE
CERTIFIED APACHE ICEBERG COMMITTER

CONTENTS

The amount of data produced today is staggering. Companies, both large and small, collect an ever-growing number of data points. Statista's research department estimates that the global data creation will reach more than 180 zettabytes by 2025.

This is aided by the rise of distributed file stores such as Hadoop Distributed File System (HDFS), as well as cloud storage solutions like AWS Simple Storage Service (S3), Azure Blob, and Google Cloud Storage (GCS). These object stores provide a simple way of storing and retrieving vast amounts of information at a relatively low price point, and with little operational overhead.

However, collecting data is just the first step. Apache Hive was initially released in 2010 to provide a SQL interface on top of raw data files. Hive enabled any developer with SQL skills to write jobs for processing inputs in distributed storage. However, due to the sheer size of data, a large amount of time was spent just scanning through data to find the relevant information.

To provide better data access efficiency, Hive introduced a system for organizing files into directories, and the union of all files in that directory path represented a table.

Additionally, tables could be partitioned so that each sub-directory of a path was mapped to specific data values. For example, each day's worth of data for a given table would be in a single sub-directory of the base path.

SEE FIGURE 1 IN NEXT COLUMN

Figure 1: Hive directory-based partitioning

dremio

Make Cloud Storage Your Data Warehouse in Minutes with the Dremio Open Lakehouse Platform

Data is needed everywhere. But with current data architectures, it's still too difficult to go from data to decision making. Data warehouses are expensive, lock your data into proprietary formats, lack flexibility of different engines and make only a subset of data available. Data lakes are flexible but difficult to manage.

Data teams love Dremio because the open lakehouse delivers the best of both worlds.

McKinsey & Company · HITACHI · pwc · Henkel · Allianz · Hertz · Unilever · Rakuten · SAMSUNG · NUTANIX · Prudential · BRIDGESTONE · BOSE · NOKIA · Moody's · Microsoft · NCR · Adobe · DOUGLAS

Dremio is the *only* open lakehouse platform with a forever-free edition.

Get started now at Dremio.com

When queries provide a filtering expression that includes the partitioning column, providing a method to prune files at the planning phase dramatically reduces the amount of data that needs to be scanned. However, this created a number of trade-offs and introduced the Hive Metastore as a key bottleneck within the data ecosystem.

Apache Iceberg was originally developed as a full redesign of the table abstraction for distributed processing systems. Its goal was to address the existing issues in the Hive ecosystem. Additionally, taking a page out of traditional relational database design, it focused on decoupling the underlying storage details from the user-facing presentation of the table.

Lastly, a key issue identified was the lack of a formal specification for the Hive table format. Since it co-evolved with a number of different processing engines, there were subtle differences in the behavior and semantics depending on which engine was used.

From its inception, Iceberg sought to create a well-defined, community-driven specification that would outline the key expected semantics. A clear specification allows all implementing code bases to determine the best possible approach within their own paradigms while maintaining consistent semantics. Regardless of the execution environment, users can expect the same interpretation of a given set of operations. Behaviors conform with the expectations for a relational database with no unpleasant surprises.

ABOUT APACHE ICEBERG

The Iceberg table format takes an opinionated stance regarding the separation between the logical table and the physical storage that underpins that representation. Users should not need to know about the underlying details of how the table is stored, and the Iceberg libraries should manage that complexity transparently.

Iceberg accomplishes this by creating a layer of metadata on top of the actual data. There are several tiers of metadata. The top-level tier is used to track:

- Schema
- Table organization (partitioning and sort order)
- Table state (i.e., snapshots)

The next two layers are concerned with table state. They contain detailed statistics about the set of files that compose a table at a given point of time. The library provides a rich expression-based interface for interrogating information from these metadata structures. Thereby, implementing clients are allowed to make intelligent decisions with respect to table operations.

Structuring the table and associated library code in this way provides several benefits. First, the fully open specification allows for broad support across many different query engines. It is designed from the ground up to provide well-defined, pluggable interfaces for key functionality such as File Format, I/O, and catalog implementation.

This extensibility allows Iceberg to be easily ported into new environments, while still maintaining consistent semantics. For example, Iceberg is supported in a variety of execution engines, including:

- Spark
- Trino
- Presto
- Flink
- Dremio

Schema evolution is a common occurrence in many data warehouse implementations. The Iceberg specification outlines specific rules around these changes so that they can be safely rolled out without having to re-write data. Furthermore, partitioning can be evolved as there is a change in table size or query demand.

All table changes occur by creating atomic swaps of the current table state. A linear history of these changes ensures there is transactional consistency across all environments accessing an Iceberg table. Table readers get the same view of data across the lifetime of an operation. Concurrent writers are able to write optimistically, and conflicts can be resolved systematically.

Furthermore, maintaining a history also means that rollbacks can be accomplished by simply swapping the table state back to a prior state. Similarly, previous states, so called “time-travel,” can be queried without changing the current state of the table.

Partitioning in Iceberg is accomplished by optionally performing transformations on column data. This produces what is known as *hidden partitioning* — the partition values are hidden from the end user and automatically applied when reading and writing to a table. Data writers write data to the table as normal, and the Iceberg client code handles mapping records to partitions. Similarly, table readers write queries against the column data, and predicates are automatically applied to provide partition filtering.

Table maintenance on Iceberg tables follows the same high-level design goals: simple to reason about and no unexpected surprises. To achieve these goals, the project ships with a number of built-in procedures for simplifying the managing of table data and metadata. Some examples include expression-based deletion, changing the current pointer to table state, optimizing file layouts, and clearing out data no longer reachable by an active snapshot.

KEY METHODS AND TECHNIQUES

WELL-DEFINED SPECIFICATION

Iceberg’s community-driven specification is fundamental to the project’s success. The specification defines the expected semantics for each feature and gives clients a framework for developing specific implementation. It relies on definition rather than convention to ensure that all implementing clients have consistent behavior. Versions 1 and 2 are ratified and currently in use in the codebase, while version 3 is still in development.

Version boundaries may break forward compatibility in the following way: Older readers cannot read tables written by a newer writer. However, newer readers are able to read tables written by older writers. The expected metadata changes for each version are outlined here.

ICEBERG SPECIFICATION V1

The initial Iceberg specification is oriented around managing large, analytic tables using open file formats such as Parquet, ORC, and Avro. The main goals are to provide a performant, predictable, and flexible abstraction for interacting with large scale data.

A key feature enabling these goals is a distributed hierarchical metadata structure. A self-contained, distributed metadata allows job planning to be pushed to clients and removes the bottlenecks of a central metastore.

The main objects in this hierarchy are as follows:

- **TableMetadata** → Information about schema, partitioning, properties, and current and historical snapshots
- **Snapshot** → Contains information about the manifests that compose a table at a given point in time
- **Manifest** → A list of files and associated metadata such as column statistics
- **Data Files** → Immutable files stored in one of the support file formats

ICEBERG SPECIFICATION V2

V2 focuses on providing capabilities around row-level deletion. Deletions on immutable files are accomplished by writing delete files that specify the deleted rows in one of two formats, either position-based or equality-based.

POSITION-BASED

Position-based deletes provide a file path and row ordinal to determine which rows are considered for deletion. To improve file scan performance, entries in the delete file must be sorted by file path and ordinal. This allows for effective predicate pushdown in columnar formats. Optionally, the delete file may contain the deleted data row values.

EQUALITY-BASED

Delete files using the equality-based approach contain a row for each expression that are used to match delete rows. The expression is constructed by using a list of `equality_ids`, where `id` is the column `field_id`, and a matching row that contains all of the `equality_ids` and optionally additional fields.

For example, a deletion where `country_code='US'` could be encoded as either of the following:

```
equality_ids=[3]
country_code: 3     |
--------------------
US                  |
```

Or:

```
equality_ids=[3]
customer_id: 1    | event_ts: 2          | country_
code: 3
-------------------------------------------------
12345             | 2022-01-01 00:00:00  | US
```

SELF-CONTAINED METADATA

Iceberg's metadata is defined using the native file abstractions for the file store used in a specific implementation. The only external service needed is the catalog. The catalog implementation requires a mechanism for atomically swapping the pointer to the current table metadata location. The entire state of a table is contained in a hierarchical tree of immutable files.

Listing is expensive in both S3 and HDFS. This cost is removed in Iceberg since the set of files is enumerated as a union of manifest entries. Planning is done by making parallel O(1) RPC calls instead of an O(N) listing of directories.

Snapshots contain an immutable set of files. This means that readers will always have a consistent view of a table without holding any lock. Readers will not see any different data until a new snapshot is committed and a new read operation is initiated.

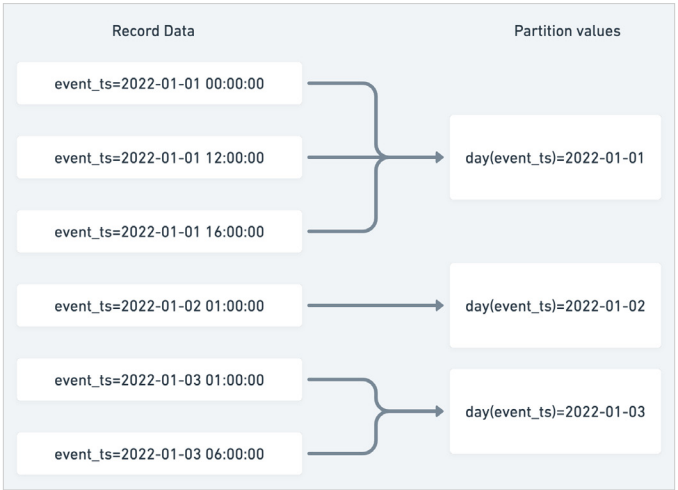
REMOVES ISSUES OF PREVIOUS DESIGNS

- Expensive listing operations
- Listing consistency
- Provide the ability for table data to not conform to a single path
- Commit retries
- Multiple levels of statistics available

HIDDEN PARTITIONING

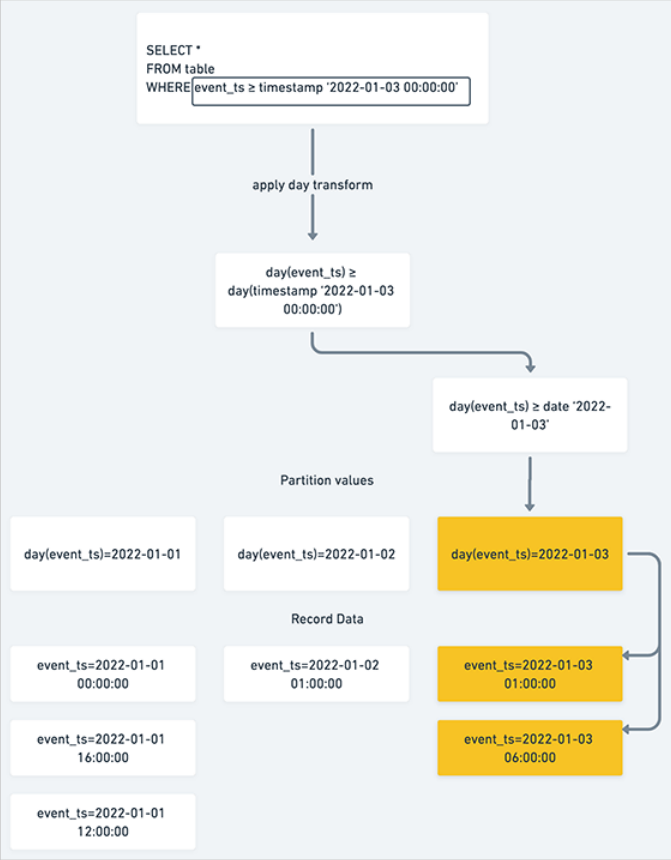
Hidden partitioning is a significant simplification improvement over the partitioning capabilities offered by Hive. As a concrete example, for a table with a timestamp column, the day transform can be applied to give partitions at the day grain.

Figure 2: Applying the day partition transform function



There are several benefits of this design. First, the mapping is applied directly to existing column data — no additional columns are needed to create the desired partition granularity. Producers and consumers do not need to take any additional action to ensure that partitioning is applied. Additionally, all partition resolution is an O(1) RPC call and can be planned in parallel.

Figure 3: Partition predicate evaluation



LINEAR SNAPSHOT HISTORY

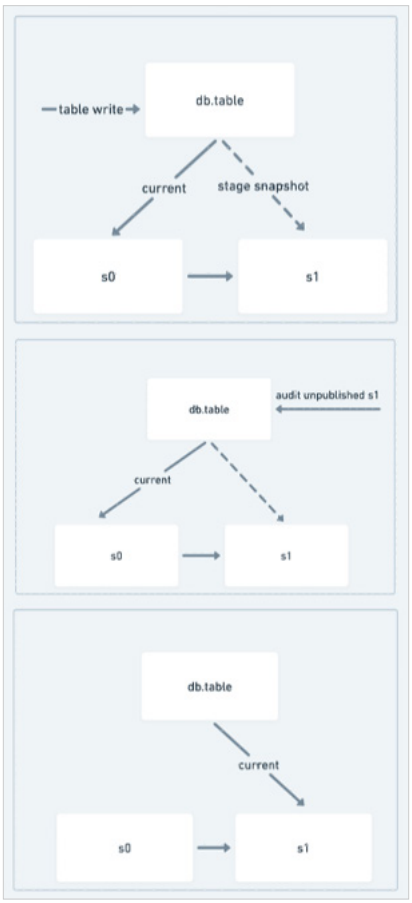
A fundamental problem with the Hive table format is that there is no native concept of table history. The table is simply whatever files happen to be in the defined paths at a given point in time. An early goal of Iceberg was to provide metadata primitives for tracking table state over time and decoupling the active state from the table paths. This is accomplished through the snapshot and snapshot log.

There can only be one current snapshot; however, there may be many snapshots in the log, both historical snapshots and yet-to-be-committed snapshots. This provides a linear history of the table state. The lineage of changes in a table is available by traversing backwards from child snapshot to parent snapshot until the point in time of interest.

It is now possible to service write-audit-publish workflows. Write-audit-publish (WAP) is a pattern where data is written to a table but is not initially committed. Then validation logic occurs — if validation succeeds, the uncommitted snapshot is promoted to the current table version. Downstream readers will never see data that has not passed validation.

WAP

Figure 4: WAP pipeline



TIME TRAVEL

Time travel queries are supported by reading previous snapshots, which give a point-in-time view of table state. End users are able to reconcile data changes at various points in a table's history through a simple interface provided by the Iceberg client as well as within Spark.

Below is an example Spark code snippet that retrieves a previous table version using epoch seconds (`2022-01-01 00:00:00`):

```
spark.read
    .format("iceberg")
    .option("as-of-timestamp", "1640995200000")
    .load("db.table")
```

ROLLBACK

Tables are easily reverted to previous states by simply making a previous snapshot the current snapshot. This is accomplished either directly through the Iceberg client or through engine specific commands. For example, Trino makes the `rollback_to_snapshot` procedure available and Spark has `rollback_to_snapshot` as well as `rollback_to_timestamp`.

Figure 5: Rollback mechanism

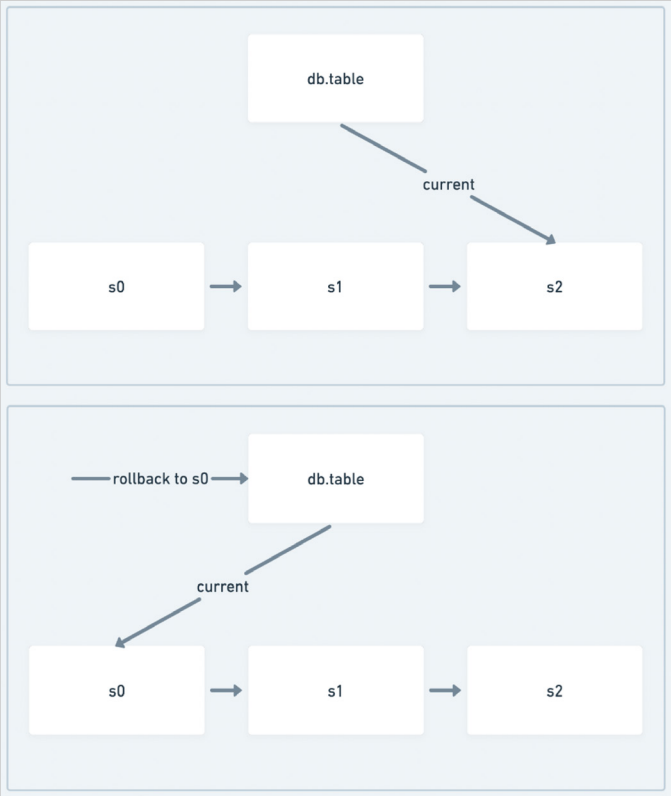


TABLE MAINTENANCE

The self-contained nature of Iceberg's metadata allows for simple interfaces for common table maintenance operations. Importantly, table owners do not need to have a deep understanding of the details of physical layout to effectively manage table data.

EXPRESSION-BASED DELETION

Starting with Spark 3, `DELETE FROM` syntax can leverage the same expression transformation used in `SELECT` queries for applying data removal. As an example, given the following DDL:

```
CREATE TABLE db.table(
event_ts timestamp,
event_id bigint,
customer_id bigint
event_category string,
country_iso_code string
) USING iceberg
PARTITIONED BY (day(event_ts))
```

Issuing the below statement will be a metadata-only operation and will remove all partitions that contain records where `event_ts` are between `'2021-01-01 00:00:00'` and `'2022-01-01 00:00:00'`:

```
DELETE FROM db.table where event_ts >= timestamp
'2021-01-01 00:00:00' and <= date '2022-01-01
00:00:00'
```

For V2 tables, expressions that do not match entire partitions can be used to perform row-level deletion operations. This will drop all partitions that match fully, and otherwise produce a delete file for the partially deleted partitions.

SPARK ACTIONS

Commonly performed operations are provided by Iceberg as Spark procedure calls. These procedures demonstrate canonical implementations for removing files that no longer are in scope for a table's current or past state. Additionally, the write patterns for a table may not be congruent with the read access patterns. For these cases, procedures are available for optimizing both metadata and data files to reconcile the incongruence between producers and consumers.

EXPIRE SNAPSHOTS

There are trade-offs involved in maintaining older snapshots. As more and more snapshots are added, the amount of storage used by a table grows both from a table data perspective and from a metadata perspective. It is recommended to regularly expire snapshots that will no longer be needed.

This is enacted by issuing the `ExpireSnapshots` action, which will remove the specified snapshot from the `SnapshotLog`.

REMOVE ORPHANED FILES

There is no longer a direct mapping between the files under a table's path and the current table state. Due to various reasons, job failures — and in some cases, normal snapshot expiration — may result in files that are no longer referenced by any snapshots in the table's `SnapshotLog`. These files are referred to as *orphaned files*, and Iceberg provides a `DeleteOrphanFiles` Spark action for performing this clean-up operation.

REWRITE DATA FILES

Tables that have many small files will have unnecessarily large table metadata. The `RewriteDataFiles` action provides a method for combining small files into larger files in parallel, thus improving scan planning and open file cost.

REWRITE MANIFESTS

The layout of manifest metadata affects performance. To get the best performance, the manifests for a table must align with the query patterns. The `RewriteManifests` action allows the metadata to be reoriented so that there is alignment with processes reading data from the table. Additionally, small manifests can be combined into larger manifests.

TRINO ALTER TABLE OPERATIONS

Trino also offers several of the above functionality through the `Alter` table statement syntax. The following capabilities and their Spark analogue are outlined below:

SPARK AND TRINO TABLE MAINTENANCE

OPERATION DESCRIPTION	SPARK	TRINO
Snapshot Removal	`ExpireSnapshots`	`Alter Table … execute expire_snapshots`
Data File Compaction	`RewriteDataFiles`	`Alter Table … execute optimize`
File Garbage Collection	`DeleteOrphanedFiles`	`Alter Table … execute remove_orphan_files`

More detail on these operations can be found in the Trino Iceberg connector documentation.

REAL-WORLD APPLICATIONS

Below are several use cases where the benefits of Iceberg are leveraged to enable use cases that were either impossible or difficult to manage in legacy table formats.

STREAMING → DATA MESH

Iceberg enables the data lake, which can be a part of a data mesh architecture in a simple, cost-effective way. Developers at Apple leverage streaming reads to decrease the total latency of data pipelines. In doing this, they have decreased their total pipeline processing delays from hours to minutes.

A combination of features allows Iceberg to effectively support streaming use cases. One common one is the ability to stream Change Data Capture (CDC) events from an online transactional system to a data warehouse. In legacy table formats, implementing a merge operation was either complex, computationally costly, or both. With the release of V2 tables, Iceberg reduces the complexity of creating streaming pipelines in several ways.

First, Iceberg supports streaming clients for both Spark Structured Streaming and Flink. Both streaming and writes and reads are supported. These clients are the simplest path to enable Iceberg as a participant in an event-based architecture. Iceberg can be used similarly to other sinks using either streaming client. Storing large amounts of history in Kafka can be prohibitively expensive. In such cases, an Iceberg streaming source table can be a lower-cost, lighter infrastructure alternative to storing the historical data directly in Kafka.

Additionally, Iceberg implements SQL constructs that can operate on row-level modifications such as merge, update, and delete. The capability of operating on a single row reduces the complexity for job writers when true streaming is not needed but the semantics of frequent updates are needed. The metadata-based approach for writing changes allows the minimum amount of data to be processed and lets pipeline owners determine the frequency of compaction. This enables developers to balance the cost and write latency with the read-time scan performance. In cases where read performance is critical, a copy-on-write approach can be taken. Conversely, when write latency is a larger issue, merge-on-read can be used with background compaction jobs.

HIGHLY SELECTIVE FILTERING

Netflix migrated microservice telemetry data from Hive to Iceberg. The telemetry tables contain millions of files per month. Switching from Hive to Iceberg reduced query planning time from 9.6 minutes to 10 seconds.

The statistics available to Iceberg reduce the amount of data that needs to be read at execution time. And, to better support the upfront elimination of files scans, Iceberg's *hidden partitioning* enables more granular partitioning than is practical in a strict directory hierarchy scheme. When combined, these two strategies allow queries on Iceberg tables to provide lower latencies and the capability to eliminate the reading of files that do not contain relevant data.

The following subsections describe the methods used to create more performant scanning and filtering.

FILE SCAN PRUNING

Iceberg uses internal statistics to perform file pruning during the scan planning phase. Furthermore, the hierarchical nature of statistics provides the opportunity to not only prune the amount of data files accessed but also the metadata files used for planning. First, the manifest list is used to eliminate all manifests that do not contain partitions involved in the current scan.

Next, the scan predicates are applied to each entry in the remaining manifests from the previous step by using the upper and lower bounds of column values for each file referenced. In the case where a file has a sorting by a scan predicate, it will eliminate all files that do not satisfy the predicate.

ADVANCED PARTITIONING

Partitions on high-cardinality fields are not practical in Hive. This is due to the one-to-one relationship between partition values and the corresponding file system structures, as well as the representation in the database backing the Hive metastore. Iceberg provides transform functions, which have a many-to-one mapping from column value to partition.

As an example, the Bucket transform gives developers a tool for converting a column value with a large number of unique values, such as a user id into a configurable, predictable number of partitions. This unlocks a few novel patterns; first, highly selective predicates are able to leverage the plan-time manifest pruning mentioned in the previous section. Secondly, it arranges data in such a way that more advanced join strategies are available to engines.

Specifically, storage partitioned joins in Spark breaks up a large join into a series of smaller joins that occur between like partition values, thus dramatically lowering the memory requirements.

ROADMAP

File scan pruning and advanced partitioning techniques go a long way, but there are many exciting new capabilities being added:

- Bloom filters → Predicate pushdown on high-cardinality columns
- Z-order partitioning → Equal weight partitions
- Advanced statistics → Number of distinct values for cost-based optimizers

AUTOMATED SCHEMA EVOLUTION

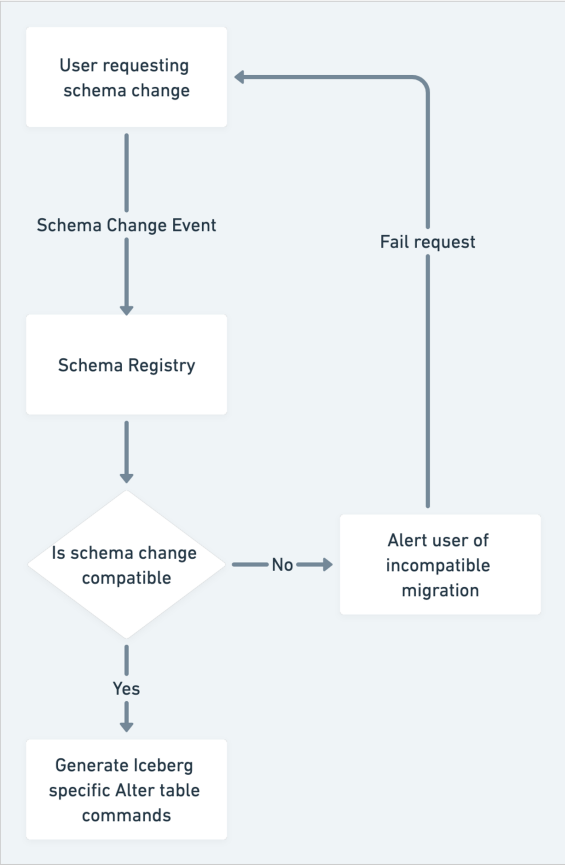
Iceberg handles schema evolution in a precise manner with the expected behaviors defined in the specification. Given reliable evolution rules, it is possible to provide build-time checks against table structure migrations. Specifically, changes to an upstream source can automatically create schema migrations in downstream dependencies. In cases where incompatible migrations are produced, it is flagged in integration testing.

CANONICAL SCHEMA

A canonical schema stored in a schema registry can be used to represent a model across the lifecycle of a data product. Each participating data component reads the current schema from the registry in a single, normalized format such as *protobuf*. All changes to that schema must be made through the registry and are validated at the time the change request.

SCHEMA MIGRATION PROCESS

Figure 6: Schema change process flow



This reduces the amount of manual effort for data pipelines maintainers. The tables in their pipelines can automatically evolve with their upstream sources. Additionally, pipeline breakages are reduced by validating schema changes proactively at build time instead of job execution time.

GETTING STARTED

As a component in a larger data lake ecosystem, Iceberg needs some supporting infrastructure to be in place to demo properly. If these components are not already available, there are several different approaches to bootstrapping a complete environment.

AWS-BASED APPROACH

One of the simplest ways to stand up an Iceberg demo that can also be transitioned into a production-grade data lake is to leverage the AWS ecosystem. AWS has published an excellent set of instructions on launching an entire data lake solution using Iceberg. This uses EMR, Glue, and Athena to provide a comprehensive set of integrated tools for demoing Iceberg's features.

DOCKER-BASED APPROACH

It is also possible to stand up a simple Docker-based stack that uses a relational database as a backing store for the catalog. The fastest way to get started in this approach is to use one of the community-maintained Docker images with all the necessary software already properly bootstrapped.

Docker and Docker Compose must be installed if they are not already available. Next, a composed YAML file is needed that contains the following:

```
version: "3"

services:
  spark-iceberg:
    image: tabulario/spark-iceberg
    depends_on:
      - postgres
    container_name: spark-iceberg
    environment:
      - SPARK_HOME=/opt/spark
      - PYSPARK_PYTON=/usr/bin/python3.9
      - PATH=/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/sbin:/bin:/opt/spark/bin
    volumes:
      - ./warehouse:/home/iceberg/warehouse
      - ./notebooks:/home/iceberg/notebooks/
notebooks
    ports:
      - 8888:8888
      - 8080:8080
      - 18080:18080
  postgres:
    image: postgres:13.4-bullseye
```

CODE CONTINUES ON NEXT PAGE

```
    container_name: postgres
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=demo_catalog
    volumes:
      - ./postgres/data:/var/lib/postgresql/data
```

These commands will create the Docker image and launch a minimal execution environment:

```
docker-compose up -d
docker exec -it spark-iceberg pyspark-notebook
```

This will bring up a Spark Notebook server running at `http://localhost:8888`. At this point, data can be read and written to Iceberg tables using either SparkSQL, Spark shell, or PySpark. A detailed overview of this environment is outlined in the Iceberg Quickstart and in Tabular's blog post on Docker, Spark, and Iceberg.

HYBRID APPROACH

Trino provides an easy-to-use SQL interface on top of Iceberg tables. The basics of configuring Trino are covered in the Iceberg connector documentation. Trino's Iceberg connector requires access to either a Hive Metastore or an AWS Glue instance. Due to this limitation, it will not work out of the box with the setup described above. However, with some small tweaks to the Docker image, AWS Glue can be used as the backing catalog — and it is even possible to demo Trino as well.

To do this, the Spark config files in the Docker image will need to be updated as follows:

```
spark.sql.catalog.demo.catalog-impl    org.apache.
iceberg.aws.glue.GlueCatalog
```

And the line referencing the Postgres connection info should be removed. These instructions can be used to ensure that Glue is properly configured for Iceberg.

In Trino, the catalog will need to also be set to use Glue with the following parameters:

```
connector.name=iceberg
iceberg.catalog.type=glue
```

The full set of Glue configuration options can be found here.

CREATE A TABLE

Now, with Trino configured, it's possible to issue SQL queries as normal and leverage Iceberg's capabilities.

```
CREATE TABLE iceberg.demo.test_table(customer_id
bigint, country_code varchar, event_ts timestamp(6))
WITH (partitioning = ARRAY['day(event_ts)',
'bucket(customer_id, 10)'])
```

LOAD DATA

This SQL statement generates a sequence of data from 1-10000, which is used as the basis for data within the other columns:

```
INSERT INTO iceberg.demo.test_table
SELECT b.id, CASE WHEN id % 100 < 25 then 'US'
WHEN id % 100 < 50 then 'IN'
                WHEN id % 100 < 75 then 'CA'
                ELSE 'AA' END,
            dateadd('minute', id, TIMESTAMP '2022-
01-01 00:00:00')
FROM (SELECT sequence(1,10000) as seq) s
CROSS JOIN UNNEST(s.seq) as b(id)
```

This will create data across partitions for the days between `2022-01-01` and `2022-01-07` inclusive.

QUERY

Now that there is data, we can query it as normal. This will push the partition evaluation into the Iceberg client code:

```
SELECT COUNT(*)
FROM iceberg.demo.test_table
where event_ts >= timestamp '2022-01-01 00:00:00'
and event_ts < timestamp '2022-01-02 00:00:00'
```

Now, there is also access to tables that display the metadata information, which are queried in the same way as the base table. But with a special modifier, the following shows an example query to get the current partitions in the demo table:

```
SELECT *
FROM iceberg.demo."test_table$partitions"
```

It is also possible to get the snapshot information using a similar construct:

```
SELECT *
FROM iceberg.demo."test_table$snapshots"
```

CONCLUSION

Iceberg is a significant advancement over previous table formats. It is designed from the ground up to address key reliability and efficiency issues within the Hive table format. Slow operations such as directory listing and partition resolution are removed in favor of fast and easily parallelizable RPC calls.

Well-defined evolution of both schema and partitioning ensure that the table layout can update as the requirements change. Serializable isolation and optimistic writes establish both a linear history of table state, as well as a predictable scheme for resolving concurrent write conflicts automatically. The hierarchical metadata structures supply clients with the capability to quickly interrogate table state. Furthermore, the flexible nature of the metadata model forms a malleable substrate for further feature development.

As a result of Iceberg's collaborative, open specification, there is consistent behavior across many processing systems such as Spark, Trino, and Flink. The technology best suited for a given use case can be used without having to transform or otherwise move data. Companies have the flexibility to design a data lake that matches the needs of their organization without having path dependence on a single processing component.

Lastly, Iceberg has a vibrant and active community. Development is happening at a rapid pace with a 1.0 release slated for some time in late 2022. The project has consistently seen code contributions grow year over year. And there are contributors across many companies, including Netflix, Apple, AWS, and more. There are regular monthly community syncs where there is an open discussion on the current items of development and highlights of recently merged features. The Iceberg community has fostered a very friendly and engaging culture, which has been a key part of the project's success.

ADDITIONAL RESOURCES

- Apache Iceberg website
- GitHub
- Google community groups:
  - Iceberg sync
  - Iceberg-Python sync

WRITTEN BY TED GOOCH,
STAFF SOFTWARE ENGINEER, STRIPE

Ted is a seasoned data professional with over 15 years working in the data space. In that time, he has worked with many organizations to improve infrastructure and best practices around data. While at Netflix, he worked on the initial implementation of Iceberg and is currently a committer on the Iceberg open-source project. In his current role at Stripe, he is helping Stripe to build the next generation of data tooling with an emphasis on security, compliance, and user experience.

DZone
600 Park Offices Drive, Suite 300
Research Triangle Park, NC 27709
888.678.0399 | 919.678.0300