Adding a File System based Consistent Method to Identify Iceberg Tables' Current Snapshot

Original Authors: Ashvin Agrawal, Russell Spitzer

Last Updated: Oct-28-2024

Goal

The goal of this proposal is to enhance the interoperability and self-sufficiency of Iceberg tables representation on the file system by replicating the snapshot's metadata file name, a.k.a. version-hint, from the catalog to the file system. This makes the table representation on the FS complete and eliminates the need for catalog dependency in certain read only scenarios. This proposal builds on the decision to move towards a catalog-centric approach to broaden the adoption and utility of Iceberg tables.

Use Case

Microsoft Fabric now supports Iceberg tables in OneLake[4], enabling users to leverage Iceberg tables in addition to Delta Lake tables with Microsoft Fabric's compute engines. OneLake RBAC assigns roles to manage permissions for its users. Users can assign these roles to either individual members or groups. To load a table, users can either integrate with a catalog or directly provide the Iceberg table's directory path, particularly when OneLake's directory-level ACLs are sufficient. Using the file system (FS) based integration reduces the number of components required in the read query execution path. This is especially beneficial when the catalog is inaccessible from Fabric (e.g. on-prem catalog) and during pre-production scenarios. Our internal engines that generate table metadata are designed to fully represent the FS and ensure the liveness and accuracy of metadata on the FS with minimal cost. This makes FS-based integration an effective solution for read workloads. Users are also able to bring in tables produced by other engines external to Fabric. However, in the absence of a standard, externally generated Iceberg tables may lack the *version-hint-like file* or contain non-standard custom metadata. This complicates integration and makes maintenance challenging. Saving the snapshot's metadata file name on the filesystem offers a practical alternative to the catalog for read-only situations. This strategy addresses common issues encountered in real-world applications, as evidenced in this use case.

Introduction

Iceberg is a widely used table format that enables seamless engine interoperability. Its primary strength lies in its ability to allow any query engine to process data written by another engine, provided the reading engine has access to the table's data and metadata files stored in an Object Store or File System (FS) and knows the desired snapshot's (typically the current snapshot) metadata file name. Since a snapshot is immutable, multiple engines can read it simultaneously without any conflicts enabling high concurrency. When data access needs to be governed, a Catalog component is used to manage and serve access control policies. While a catalog can enforce table-level ACLs by distributing file access tokens, column and row-level ACLs are generally enforced by the query engine [3]. In some scenarios, central catalog service is used by multiple engines, whereas in others where fine-grained ACLs are absent, a separate catalog may be avoided.

The use of a catalog is also recommended for achieving high efficiency and reliability in concurrent table write operations. The current Iceberg specification suggests that a catalog should manage a snapshot's metadata file name, referred to in this document as ***SFN***. This is because object stores lack certain features necessary for efficiently and atomically maintaining the SFN, making this design choice essential. However, the current specification requires that the SFN be stored exclusively in the catalog. This restriction introduces a significant limitation: ***it renders the table representation on the FS incomplete. Not providing a way to fully represent the table on the FS contradicts the objective of a table format.***

Iceberg metadata is composed of several key components, including manifest files, manifest lists, and snapshots (metadata json files). These are all stored in the FS, usually alongside the data files. If the query engine knows a snapshot's metadata file name (SFN), the FS metadata is enough for it to perform a read query. Unlike the previously discussed metadata components, the SFN, indicating the current state of the table, is not stored in the FS; rather, it is kept in the catalog only.

Although retrieving the most recent snapshot from the catalog is ideal, it need not be a strict requirement in all scenarios. In certain production environments, FS reliability and governance issues are absent, and requiring catalog integration impedes interoperability. The Hadoop catalog implementation **partially** addressed this by generating a version-hint file [2], however, it is not part of the specification. To address this issue, users often rely on ad hoc mechanisms, inspired by the Hadoop Catalog's **version-hint-like** file, to store SFN metadata in the FS. It is essential to clarify that we mention the Hadoop catalog, which has known limitations, merely to explain the concept.

This proposal aims to introduce an optional step that entails replicating SFN metadata present in the catalog onto the FS. This would make the table representation on the FS complete and self-sufficient, thereby enhancing its capability to support wider interoperability use cases.

The proposal decouples the SFN metadata file specification from the method of creating the SFN metadata file and provides various options for generating this information. To ensure simplicity and backward compatibility, generating the SFN metadata file is an optional step and is implemented with a less stringent consistency guarantee.

SFN Generation Options

It is important to note that the proposal does not add a new step in the transaction execution. Instead, the creation of SFN files will occurs after a commit, i.e. SFN generation will be asynchronous or not guaranteed. Possible methods for managing SFN file in this context include:

Management through Catalog using notification (proposal [5])

This approach involves configuring a notification listener that responds to catalog events for new table snapshot creation and initiates the SFN file generation on the FS. The notification listener can be hosted by the catalog, since the listener needs to have necessary authorization to add files in the table's directory. The listener approach ensures that the FS is updated in near real-time. This approach would leverage a feature that would be available in the catalog and would be customizable, without imposing any extra burden on the users.

Management through the Catalog

In this approach, the SFN file is generated after the commit by the catalog. Creating the SFN inline offers the same benefits as the notification listener method; however, it is more tightly coupled and less flexible.

Management using an external maintenance task

This method involves deploying a 3rd component to monitor the catalog for any changes and update the SFN information accordingly. This method allows for asynchronous processing and operates independently of the catalog's real-time functions. However, the maintenance task relies on polling and introduces a lag between when changes occur in the catalog and when they are reflected in the FS. Additionally, it also requires managing a new component and may increase the load on the catalog if a large number of tables are present.

Proposal for SFN Meta Naming Options

The second part of this proposal addresses SFN naming and contents. There are a few constraints to consider: 1) scenarios where multiple tables are stored in the same FS directory for performance reasons, and 2) the high cost of listing files in a directory. Various commercial systems, such as Fabric, BigLake, and SNOW, employ a metadata publish pattern. In these cases, a single writer publishes the table metadata to mitigate conflicts, with some latency expected. Additionally, the most common use case involves having the current snapshot, which

is generally sufficient for these purposes.The proposal aims to avoid ambiguity while maintaining simplicity and low cost. Several naming conventions for SFN meta files are suggested.

Table Identifier (namespace + name) based naming (Option 1)

One straightforward approach is to use a fixed name for the SFN meta file, i.e. base the filename on the table identifier. Deriving the SFN meta file name from the table identifier eliminates the need for file listing and ensures that the file can be easily located and updated. The table reader only needs the directory path as input. This approach borrows from the current HadoopCatalog implementation (which as mentioned earlier has known limitations), in which the SFN-file name is fixed to version-hint and is table name agnostic.

One limitation of this approach is that a table rename operation would result in creation of a SFN file with a new name. The consumer might continue checking the outdated file and serve old data. To mitigate this issue, we propose to link the new SFN file in the old file. The old file should expire after a certain period. This ensures that any processes or users referencing the old SFN file can still locate the updated metadata without immediate disruption. Optionally, in scenarios where different branches of the table are managed, a branch-specific naming convention could be employed. However, identifying SFN meta files for other branches would require file listing.

For e.g.
`<namespace>_<table_name>_<branch>.ver`
`<namespace>_<table_name>_main.ver` (for the main branch)

Ordinal-Based Naming (Option 2)

Another option is to incorporate an ordinal (like timestamp) into the file name. This method allows for easy tracking of the snapshot history over time and avoids the need to “replace” SFN files. This mimics a write-ahead-log behavior. However, it does introduce the need for file listing to identify the latest snapshot, which might be an acceptable trade-off in use cases where rename and table recreation are popular. Additionally, this approach could lead to a proliferation of files over time, potentially complicating file management and increasing storage costs. Regular cleanup and maintenance routines would be necessary to mitigate these issues, adding operational complexity.

A solution to address the many files issue is to store SFN files in a subfolder inside or alongside the metadata folder. This method limits the number of files at the sub folder level and reduces the payload of the list API, thus lowering the cost of correct SFN file identification. By segregating SFN files into a dedicated subdirectory, the system can maintain a cleaner and more organized file structure.

For e.g.: `<namespace>_<table_name>_20240128T153045.ver` (timestamp 20240128T153045 represents ordinal)

Table UUID-Based Naming (Option 3)

The table UUID naming method utilizes the table's UUID for SFN meta files, ensuring that each snapshot can be uniquely identified. The table GUID remains unchanged throughout the table's lifetime and does not conflict even if two tables have the same name. This method is useful for situations like renaming, table recreation, and directory sharing. However, it necessitates file listing when the table's UUID is not known to the user. This could be addressed by maintaining a sub-folder as in Option-2.

For e.g.: `<table_id>_123e4567-e89b-12d3-a456-426614174000.ver`

Considerations

Unique Directory per Table: In this scenario, only one table’s data and metadata files are present in a directory, ensuring that at most one CSID-file is present.
Directory is reused for Table Recreation: This occurs when a table is dropped and then recreated in the same directory path. During this process, orphaned files from the dropped table may remain in the directory.
Table Renaming: When a table undergoes a renaming operation without a change in its location, the table's CSID-file must still be deterministically discoverable.
Shared Directory: In this scenario, a single directory stores files for multiple Iceberg tables. Each table must have its own unique CSID-file to avoid unnecessary coordination between table managers.
Multi-environment: Table writers across dev, staging, and test environments may use the same table identifier and can lead to a risk of conflicts.

Structure of the SFN meta file

The SFN meta file is structured as a JSON file that encapsulates essential information to identify and manage the table’s current state. The JSON file includes the following key elements:
Version: version of the SFN meta file specification
Table Identifier: composed of the table namespace and table name
GUID: This allows readers a way to validate that the correct table is read through table updates.
Metadata File path: Absolute path of the snapshot metadata.

Here is an example of the JSON structure for the CSID file:

```json
{
"version": 1
"table_identifier": "sales.customer",
```

```
"guid": "123e4567-e89b-12d3-a456-426614174000",
"metadata_file_path": "/foo/bar/abc.json",
"Ordinal": "20240128T153045”
}
```

Limitations

Staleness: The table manager produces the CSID file in a post-commit step. As such, there may be a delay in the visibility of new table updates. This staleness can affect the real-time accuracy of data available to the readers.

Loss of Commits: In cases where the manager fails before updating the CSID file, the new commit may not be visible to the reader. This could result in data inconsistencies. Although a periodic maintenance job could bring the CSID file to a consistent state, this is not enforced and relies on the robustness of the table manager.

Incomplete Timeline: The CSID file cannot be used to track all commits on a table comprehensively. For example, in a high transaction scenario, the table manager may choose to update the CSID file only once, missing intermediate commits. This could lead to gaps in the commit history available to readers.

Cost: Unless an easy file naming mechanism is adopted, readers may need to execute a list operation on the storage engine. This can be a costly and high-latency operation, potentially impacting the performance and efficiency of the system.

References

[1] Snowflake: CREATE ICEBERG TABLE
[2] Version-hint file discussion
[3] Spark ACL
[4] OneLake Iceberg Support
[5] IRC notification proposal