Pluggable File Filter interface for Apache Iceberg

Authors: Guy Khazma, Gal Lushi

Introduction:

This document describes a proposed enhancement to Iceberg to enable a pluggable file filter for the integration of data skipping indexes.

This document is written in response to Adobe Proposal and is inspired by some of the discussion in that document.

Goals

- Define a pluggable interface that will enable using custom data skipping index implementations
- The interface should enable data skipping across data formats that Iceberg works with (parquet, orc, avro)
- The interface should work with multiple engines e.g. spark, flink, presto (by using different implementations)

Non-Goals

- Index creation and management
- Non data skipping indexes (such as covering indexes e.g. as defined in Hyperspace)

While we believe the above are important, we think that they require a separate proposal and discussion in the community. Our view on these topics is described in the next steps section. In this document we focus on a pluggable interface for data skipping indexes to enable use of existing indexes in job planning.

Overview

Data skipping is a feature which can boost the performance of SQL queries by skipping over irrelevant files based on summary metadata associated with each object.
For every column in the object, the summary metadata might include minimum and maximum values, a list or bloom filter of the appearing values, or other metadata which succinctly represents the data in that column. We refer to this metadata as a *data skipping index* (or simply index), and it is used during job planning to skip over objects which have no relevant data.

In order to support data skipping indexes there are two basic APIs which are needed:

- API to retrieve all relevant files to be indexed
    - Such an interface already exists in Iceberg using the FindFiles helper API.
    - Using such an API each object can be indexed independently using any relevant framework (for an indexing example using Spark see the Usage example below)
- API to filter irrelevant objects during job planning according to given data filters
    - This API is called during the job planning in order to discard files which are deemed irrelevant according to the metadata

Iceberg already has support for filtering data files based on data file properties like location or row count as can be seen in this PR. This capability is currently used in the FindFiles helper API.
We propose to extend this capability with a pluggable interface that will enable custom logic that will consult external indexes in order to decide which files are relevant during job planning.

Implementation

A prototype of proposed changes to Iceberg can be seen here. See the next section for a usage example.

In order to enable pluggable logic for skipping files we propose to change the code in the DataTableScan so it can be configured to load the file filter implementation via reflection.
The configuration parameter value will be the fully qualified name of an Evaluator class which will be loaded by reflection.

The external file filter is an instance of an Evaluator which gets the following as an input for its constructor:

- An instance of the Iceberg table to filter
-
- The data filters applied to the file filter

It can then run any logic it needs in order to initialize its state (such as finding out all of the required and indexed files).
Then for each DataFile the `eval` function is called and it determines whether the DataFile can be skipped.

If the parameter is not set and a file filter expression is used (as for example in the TestFindFiles) then the default Evaluator is used which enabled to filter by data properties like location or row count will be used (as was added in this commit).
The DataTableScan passes to the ManifestGroup a FileFilterBuilder class which creates the relevant Evaluator according to the configuration.

Currently, The FileFilterBuilder has 2 implementations -

- BaseFileFilterBuilder - used instantiate the default Evaluator which was used so far (and enables to filterby data properties like location or row count)
- ExternalFileFilterBuilder - used to load the pluggable file filter using the class specified in the documentation.

Then, the ManifestGroup will use the evaluator as usual to filter the files.

Usage Example

The prototype code can be seen here.

To demonstrate the usage of such an API we implemented a POC of the proposed changes in order to integrate Iceberg with Xskipper.
Xskipper is a pluggable data skipping framework that currently works with Apache Spark and can be extended to work with additional engines in future.

In the prototype we implemented an external file filter which uses the index built by Xskipper in order to determine the indexed and required files for the given query predicate and then filter irrelevant files.

Using the prototype we are able to index an Iceberg table with the following indexes:

- Value List index - stores the list of unique values for the column
- Bloom Filter Index - uses a bloom filter to test for set membership

Note that Xskipper is extensible and defines APIs to allow easy addition of custom data skipping indexes.

An example runner which does the indexing and then runs a query which uses the indexes can be seen here.

The above example uses Spark to read the metadata.
Other implementations could use other engines to read the metadata.

Please note that the POC infers the metadata location from the iceberg table properties. A full implementation may read the metadata location from the table properties.

In addition, while the current bloom filter implementation in Xskipper uses Spark's bloom filter implementation it is easy to replace it with a bloom filter implementation which will be available cross engines.

Next Steps

Using the above interface, an external indexing system can be used to handle data skipping indexes creation and management enabling iceberg to leverage such indexes during job planning.

Our experience with data skipping indexes indicates that:

- Using data skipping as part of job planning enables better resource planning by the engine (for example, in Spark, it avoids allocation of partitions for files that can be skipped. Therefore, less tasks are provisioned).
- Storing the indexes in a consolidated fashion is beneficial opposed to using one index file per one data file, as it requires less requests to the file system. When using object storage (such as s3) where each request is a REST request, this can be crucial.
- Storing indexes in the same storage alongside the data alleviates the need to have an active metadata layer (same approach as with iceberg's table metadata).
- To broaden data skipping for data sources such as Iceberg it is beneficial to have execution engines push all predicates including UDFs as pushdown predicates (Spark for example doesn't push UDFs to the datasources)

Xskipper is one such system and it enables:

- Storing indexes in parquet (where each column is an index and each row is the metadata for a data file)
- Extensible and defines APIs to allow easy addition of custom data skipping indexes.
- Building indexes on multiple columns

With regards to non data skipping indexes, in particular covering indexes - the usage of such indexes is part of the logical optimization of a query engine and doesn't relate to optimization at the data source level. Therefore, such an optimization should be integrated into each engine separately..

References

- Supporting Secondary Indexing in Iceberg (Adobe proposal) - https://docs.google.com/document/d/1E1ofBQoKRnX04bWT3utgyHQGaHZoelgXosk_UNsTUuQ
- Xskipper - https://github.com/xskipper-io/xskipper
- Extensible Data Skipping - https://arxiv.org/abs/2009.08150