Metastore split cache

The goal of this feature is to cache file metadata (e.g. ORC file footers) to avoid reading lots of files from HDFS at split generation time, as well as potentially cache some information about splits (e.g. grouping based on location that would be good for some short time) to further speed up the generation and achieve better cache locality with consistent splits.

Table of Contents

Caching file footers

General approach (minimum viable) and usage

Metastore database changes

File metadata will be stored in a separate HBase metastore table with one more CFs in the form of fileId => { metadata(footer)_pb, ppd_stats }. The details are as follows.

The key is going to be fileId. On HDFS, fileId is an unique id that changes when the file is overwritten; on other FSes, it can be generated but it is not bulletproof, so footer cache can be disabled by default on FSes that don't support fileId.

File metadata for ORC format is already protobuf (which is consistent with HBase metastore approach to values), so it will be stored as is. Additionally, to facilitate PPD (see below), column statistics per stripe may be deserialized once at storage time in separate column family.

Metastore API changes

Logically, metastore APIs for retrieving data will be as follows:

- `List<blob> getFileFooters(int[] fileId)`
  returns all the footers for a list of fileId-s that are present in the cache.
- `Struct<Map<fileId, Pair<blob, int[]>>, List<fileId>> filterFilesByExpr(int[] fileId, ExprGenericFuncNodeDesc predicate)`
  filters the incoming list of files by applying the predicate to statistics in file footer (see PPD), and returns file footer data and stripe indices for the files that match, as well as the list of files that were not found and could not be evaluated
  TBD: footers are used by filtering and by VectorizedOrcInputFormat. We need to evaluate if we can avoid returning them altogether from this call or minimize data transferred.
- `void putFileFooters(int[] fileId, List<blob> footers)`
- `void clearFileFooterCache(int[] fileId)`

Implementation-specific considerations, such as Thrift types, batching and flat objects to avoid high memory usage, changeability (request/response objects), dependency issues (expr serialization for metastore) are out of the scope of this document. This describes just the logical signature.

Cache semantics and cleanup

Due to fileId usage as key, we don't have to worry about any file overwrites – the file record after overwrite will not match the lookup, and will merely be stale. Such records can be cleaned up by a background thread slowly iterating thru the cache (see also cache design section), or explicitly via an API call.

ORC files are not modified after being written; therefore, any data by fileId will remain valid indefinitely until the file is removed (as long as HDFS fileId semantics are maintained).

Stripe-level PPD

Footer cache can deserialize the footer at time of store, and preserve some statistics in a separate column family. Then, if the caller supplies Hive expression that can filter on the columns, this expression can be applied to the statistics. The pipeline and general approach will be the same as with the calls that filter partitions by a predicate.
Metastore expression support is relatively limited (and will always stay so, e.g. user functions can never be run in metastore), so filtering will still happen on the client in such cases.

Passive cache, active cache, automatic generation

Initially, the cache will be passive – put… metastore API will be used by users to store the footers for the files that were previously missing.
Active (read-thru) cache is considered undesirable, as it can create a single point of failure under high load, when metastore tries to read footers for large number of files.
As the next step, we can add a parameter to tables that, when enabled on supported tables, would cause metastore to:

- Read footers in background when new files/partitions/paths are written, and populate the cache.
- Potentially, scan partition directories looking for new files, to read footers from them.

Delivery outline

1. Dumb passive cache; get/put/clear APIs and storage. Usage from split generation in ETL strategy. Off by default.
2. PPD into metastore, filter API. Usage from split generation.
3. Testing to make on by default on HDFS, expand to all cache strategies (rather, ETL cache strategy becomes the default IF cache is enabled).
4. Automatic generation in metastore.

Potential future work

- Track information about the nodes where particular split was read. It can help generating consistent splits for local (HDFS, LLAP) cache locality.
- Caching real data sizes/etc for encoding after reading the files, to account for raw data size differences not visible from file and stripe statistics.
- Moving split generation to metastore via plugin system? This will allow for more fine-grained in-memory caching of split groupings, etc., and controlled access to NameNode for the file data so that it doesn't fall over.

Backward compatibility and limitations

The limited support for this feature is ok, since it is a perf improvement and not mandatory for any other part of Hive. It is important, however, that we retain backward compatibility in all cases.

1. **The functionality will (initially?) be limited to HBase metastore.** We don't expect RDBMS based metastore to be able to scale to the amount of data. Response object for the API(s) will contain boolean field for backward compat that would indicate that the operation is not supported. However, since RawStore will need the methods that will be implemented both in HBaseStore and ObjectStore, the implementation for RDBMS metastore is possible and well isolated, should anyone want to contribute a patch.
2. **The automatic update functionality will initially be limited to ORC.** We need to choose one format for minimum initial implementation. As per above, ORC is a good format for our purposes.
3. **The code for split generation will need to use the cache explicitly.** Split generation is done in many places in Hive for many use cases. They will not all automatically benefit from this feature due to refactoring/architectural challenges. We will add the cache usage to different places in code as needed and desired.
4. **Not compatible with ACID.** Stripe filtering, etc. don't work with ACID.