Hive Metadata Caching Proposal

Why Metastore Cache

During Hive 2 benchmark, we find Hive metastore operation take a lot of time and thus slow down Hive compilation. In some extreme case, it takes much longer than the actual query run time. Especially, we find the latency of cloud db is very high and 90% of total query runtime is waiting for metastore SQL database operations. Based on this observation, the metastore operation performance will be greatly enhanced if we have a memory structure which cache the database query result.

Server side vs client side cache

We are thinking about two possible locations of cache. One is on metastore client side, the other is on metastore server side. Both client side and server side cache needs to be a singleton and shared within the JVM. Let’s take Metastore server side cache as an example and illustrated below:

blocked URL

Here we show two HiveServer2 instances using a single remote metastore. The metastore server will have a cache and thus shared by both HiveServer2 instances. In practice, we usually use HiveServer2 with embedded metastore. In this picture, metastore server code lives inside HS2 JVM instance and metastore server cache is shared within the HiveServer2:

blocked URL

On the other hand, Metastore client side lives in client JVM and will go away once the client is gone.

blocked URL

If we are using HiveServer2 with embedded metastore, both client side and server side cache doesn’t make much difference as we only use one copy of the cache on HiveServer2. We need to make sure if we choose to implement both client side and server side cache, only one cache is used so that we don’t waste extra memory.

In general, metastore client cache has better performance in case of a cache hit since it further avoids a network roundtrip to the metastore server. Metastore server cache has a better cache hit rate as it is shared by more clients. Client and server side cache is independent and there is nothing to prevent us to implementing both. However, our experiment shows the major bottleneck is the database query not the network traffic. That favors us to focus on metastore server cache in our initial effort.

Cache Consistency

If we have cache in two or more JVM instance, we need to deal with cache consistency problem. For example, suppose both metastore server A and B caches table X, and at a moment, client changes table X via metastore A. Metastore A could invalidate the cached table X and maintain a consistent cache. However, metastore B does not realize the change and continue to use the stale cached table X. We can certainly adopt a cache eviction policy to invalidate old entries, but there is an inevitable lag.

To address this issue, we envision several approaches to invalidate stale cache on remote JVM:

1. Time based synchronization. Cache will be refreshed periodically from the database. However, we need to make sure cache is consistent during the refresh. This is part of the current implementation.
2. Metastore has an event log (currently used for implementing replication v2). The event log captures all the changes to the metadata object. So we shall be able to monitor the event log on every cache instance and invalidate changed entries (
   ~~HIVE-18056~~ - CachedStore: Have a whitelist/blacklist config to allow selective caching of tables/partitions and allow read while prewarming CLOSED
). This might have a minor lag due to the event propagation, but that should be much shorter than the cache eviction.
3. Maintain a unique id for every object in SQL database (eg, modified timestamp, version id, or md5 signature), which is different every time we change the object in SQL database. We will check the DB if the object is changed for every cache access. However, even check the timestamp in SQL database might take some time if the database latency is high
4. In addition, we might optionally add a “flush cache” statement in Hive in case user want to enforce a cache flush. However, this should be an admin privilege statement and will complicate our security model.

If the requirements present, we can also work on implementing a cache consistency protocol among multiple metastore instances. Such a protocol will need to replicate changes to all the active metastore before finally committing the change and responding to a client write/update request (perhaps using something similar to a two phase commit protocol).

Case Study: Presto

Presto has a global metastore client cache in its coordinator (HiveServer 2 equivalent). Note Presto currently only has 1 coordinator in a cluster so it does not suffer cache consistency problem if user only changes objects via Presto. However, if user also changes objects in metastore via Hive, it suffers the same issue.

Presto adopts a Guava based LRU cache which has the default expiration of 1h and default max entry of 10000 (tunable). The cached metastore is pluggable. The cached metastore client and non cached version are both an implementation of a common interface and either can be activated by a config.

Presto has the following cache:

- Point lookup cache
    - databaseCache
    - tableCache
    - partitionCache
    - userRolesCache
    - userTablePrivileges
- Range scan cache
    - databaseNamesCache: regex -> database names, facilitates database search
    - tableNamesCache
    - viewNamesCache
    - partitionNamesCache: table name -> partition names
- Other
    - partitionFilterCache: PS -> partition names, facilitates partition pruning

For every partition filter condition, Presto breaks it down into tupleDomain and remainder:

AddExchanges.planTableScan:

```
        DomainTranslator.ExtractionResult decomposedPredicate = DomainTranslator.fromPredicate(

              metadata,

              session,

              deterministicPredicate,

              types);

  public static class ExtractionResult

  {

     private final TupleDomain<Symbol> tupleDomain;

     private final Expression remainingExpression;

  }
```

tupleDomain is a mapping of column -> range or exact value. When converting to PS, any range will be converted into wildcard and only exact value will be considered:

HivePartitionManager.getFilteredPartitionNames:

```
     for (HiveColumnHandle partitionKey : partitionKeys) {

        if (domain != null && domain.isNullableSingleValue()) {

              filter.add(((Slice) value).toStringUtf8());

        else {

           filter.add(PARTITION_VALUE_WILDCARD);

        }

     }
```

For example, the expression “state = CA and date between ‘201612’ and ‘201701’ will be broken down to PS (state = CA) and remainder date between ‘201612’ and ‘201701’. Presto will retrieve the partitions with state = CA from the PS -> partition name cache and partition object cache, and evaluates “date between ‘201612’ and ‘201701’ for every partitions returned. This is a good balance compare to caching partition names for every expression.

Our Approach

Our design is a metastore server side cache and we will do metastore invalidation upon receiving metastore events. The reason for both choices are discussed in the above sections.

Further, in our design, metastore will read all metastore objects once at startup time (prewarm) and there is no eviction of the metastore objects ever since. The only time we change cache is when user requested a change through metastore client (eg, alter table, alter partition), and upon receiving metastore event of changes made by other metastore server. Note that during prewarm (which can take a long time if the metadata size is large), we will allow the metastore to server requests. If a table has already been cached, the requests for that table (and its partitions and statistics) can be served from the cache. If the table has not been prewarmed yet, the requests for that table will be served from the database (~~HIVE-18264~~ - CachedStore: Store cached partitions/col stats within the table cache and make prewarm non-blocking RESOLVED).

Currently, the size of the metastore cache can be restricted by a combination of cache whitelist and blacklist patterns (~~HIVE-18056~~ - CachedStore: Have a whitelist/blacklist config to allow selective caching of tables/partitions and allow read while prewarming CLOSED). Before a table is cached, it is checked against these filters to decide if it can be cached or not. Similarly, when a table is read, if it does not pass the above filters, it is read from the database and not the cache.

Quantitative study: memory footprint and prewarm time

The major concern in this approach is how much memory the metastore cache will consume and how much latency at startup time to read all metastore objects (prewarm). For that, we did some quantitative experiments.

In our experiments, we adopted some memory optimizations discussed below, which is separating table/partition and storage descriptor. We only cache database/table/partition objects and not count column statistics/permanent functions/constraints. In our setting, the memory footprint is dominated by partition objects which aggregates to 58M (shown in the table below). This does not discount the fact that many strings can be interned and shared by multiple objects, which could save an additional 20% according to HIVE-16079.

object	count	Avg size (byte)
table	895	1576
partition	97,863	591
storagedescriptor	412	680

If we have 6G memory reserved for metastore cache, we could afford 10,000,000 partition objects and which we think is enough for majority use cases. In case we are out of the memory boundary, we could switch to a non-cached metastore and don't crash.

We also tested the prewarm time in the same setting. It takes metastore 25 seconds to load all databases/tables/partitions from MySql database. Note in slow database such as Azure, the number could be much bigger but we have not tested yet. Metastore would only open listening port after prewarm. All metastore client request will be rejected before prewarm is done.

Memory Optimization

We plan to adopt two memory optimization techniques.

The first one is to separate storage descriptor from table/partition. This is the same technique we used in HBaseMetastore work and is based on the observation major components of storage descriptors are shared. The only storage descriptor components which might change from partition to partition is the location and parameters. We extract both components from shared storage descriptors and stored in partition level instead.

The second technique is intern shared string as suggested in HIVE-16079, which is based on the observation most strings in parameters are the same.

Cached Objects

The key objects to store in the cache are

- Db
- Table
- Partition
- Permanent functions
- Constraints
- ColumnStats

We will cache thrift objects in cache, so we can simply implement a wrapper on top of the RawStore API.

We don't plan to cache roles and privileges as we are in favor of external access control system such as Ranger, and put lesser focus on SQLStdAuth.

Cache Update

For local metastore request that changes an object, such as alter table/alter partition, the change request will write through to the wrapped RawStore object. In the mean time, we should be able to update the cache without further fetching data back from SQL database.

For remote metastore updates, we will either use a periodical synchronization (current approach), or monitor event log and fetch affected objects from SQL database (~~HIVE-18661~~ - CachedStore: Use metastore notification log events to update cache RESOLVED). Both options are discussed already in "Cache Consistency" section.

Aggregated Statistics

We already have aggregated stats module in ObjectStore (~~HIVE-10382~~ - Aggregate stats cache for RDBMS based metastore codepath CLOSED). However, the base column statistics is not cached and needs to fetch from SQL database everytime needed. We plan to port aggregated stats module to CachedStore to use cached column statistics to do the calculation. One design choice yet to make is whether we need to cache aggregated stats, or calculate them on the fly in the CachedStore assuming all column stats are in memory. But in either case, once we turn on aggregate stats in CacheStore, we shall turn off it in ObjectStore (already have a switch) so we don't do it twice.

getPartitionsByExpr

This is one of the most important operations in Hive metastore we want to optimize. The ObjectStore already have the ability to evaluate an expression against a list of partitions in memory. We plan to adopt the same approach. The assumption is even though we need to evaluate expression of every partitions of the table in memory, calculation time is still much lesser than database access time even though we can push some expression conditions to sql database. This is supported by some initial testing (shown below), but need to evaluate more in the future. In case it becomes a problem, a memory index to accelerate in memory expression evaluation is possible.

Architecture

CachedStore will implement a RawStore interface. CachedStore internally wraps a real RawStore implementation which could be anything (either ObjectStore, or HBaseStore). In HiveServer2 embedded metastore or standalone metastore setting, we will set hive.metastore.rawstore.impl to CachedStore, and hive.metastore.cached.rawstore.impl (the wrapped RawStore) to ObjectStore. If we are using HiveCli with embedded metastore, we might want to skip CachedStore since we might not want prewarm latency.

Potential Issues

There are maybe some potential issue or unimplemented featrue in the initial version due to time limitation:

1. Remote metastore invalidation by monitoring event queue. Discussed above and may not make it in version 1
2. getPartitionsByFilter may not be implemented. This API takes a string form of expression which we need to parse and evaluate in metastore. This API is only used in HCatalog and for backward compatibility. Hive itself will use getPartitionsByExpr which takes a binary form of expression and already addressed
3. It is better to have a cache memory estimation, so we can control the memory usage of cache. However, memory estimation could be tricky especially considering interned string. In the initial version, we might only put a limit on the number of partitions, as we saw the memory usage is dominant by partition.
4. In current design, once cache usage exceeds threshold (number of partitions exceeds threshold, or memory usage exceeds threshold in future version), Metastore will exit. If the metastore is still running with missing cache, some operations such as getPartitionsByExpr would produce wrong result. One potential optimization is we evict cache in table level, i.e., once memory is full, evict some tables alongs with all its partitions. When we want a missing table/partition, retrieve the table along with all partitions from SQL database. This is possible but adds a lot of complexity to the current design. We might only consider it if we observe memory footprint is excessive in the future.
5. In prewarm, we fetch all partitions of a table in one SQL operation. This might or might not be a problem. However, fetching partition one by one is either not an option as it would take excessive long time. We might need to find some way to address it (like paging) if this becomes a problem

Compare to Presto

In our design, we sacrifice prewarm time and memory footprint in change of simplicity and better runtime performance. By monitoring event queue, and can solve the remote metastore consistency issue which is missing in Presto. Architecture level, CachedStore is a lightweight cache layer wrapping the real RawStore, with this design, there's nothing prevent us to implement alternative cache strategy in addition to our current approach.