Proposal for new approach to Impala catalog

Current design overview

The Impala frontend requires access to various pieces of metadata to plan queries:

- Table and partition schemas, definitions, locations, etc (stored in the HMS)
- Lists of files for each partition referenced by the query, along with the physical block locations of those files (stored in the HDFS NameNode or other filesystems such as S3)
- Security metadata such as roles and grants (stored in Sentry)

Since the inception of the Impala project, the design has been for Impala to aggressively cache these pieces of metadata such that a typical query does not need to obtain any information from any external systems prior to or during the planning process. Some of the original goals of this design were:

- Reduce query latency and improve throughput by avoiding RPCs to external systems
- Reduce or eliminate the impact of failures of remote systems (HMS and HDFS previously had single points of failure)

Since Impala 1.2 (2013), the integration between Impala and the external metadata services has been centered around the Catalog Server (catalogd), as follows:

- The catalogd fetches and consolidates all information from remote services such as HMS and HDFS
- Upon a user request to refresh the information for a table, the catalogd attempts to incrementally detect changes made in external systems and incorporate them into Impala's cache.
- Operations that modify metadata (eg DDLs) are routed from the coordinator to the catalog service, which performs them and modifies the metadata cache in-place to reflect the changes.
- When new information is available about a table, the information is collected in a thrift "catalog object" structure and broadcast to all impalad daemons using the statestore.

Issues with current design

Over the 4-5 years since the current design was first introduced, production experience with Impala has identified a number of shortcomings:

Catalog memory usage

The current catalog does not have any notion of policy-based cache eviction (either by size or LRU-like policy). As Impala scales to larger user deployments and datasets, the amount of metadata has continued to scale. There are reports in the wild of catalogds requiring many tens of GBs of RAM to cache the whole catalog. Additionally, users have complained that their catalog requires periodic restarts in order to flush the cache and “start fresh” in an attempt to avoid OOM.

In addition to sizing the catalogd heap to be large enough to accomodate the entirety of the catalog, the catalog ends up being replicated to all impala daemons that act as coordinators. This means that a large amount of cluster memory is devoted to caching catalog information, despite the fact that many of the tables in the cache are infrequently accessed.

Catalog object granularity

Currently, the catalog implementation in Impala fetches, distributes, and caches information at the granularity of a table:

- When any change is made to the information for a table on the catalogd, *all* metadata for the table is re-serialized and distributed via the state store to impalads.
- If an impalad requires any information about a table (eg to satisfy a ‘DESCRIBE’ query), it requests a full metadata load of that table.
- The catalog service modifies the metadata for tables in-place during a refresh or DDL, and acquires table-level locks. These locks can prevent the propagation of catalog information for unrelated tables, causing a system-wide apparent “lock-up” of impalads.

The above results in various issues:

- The size of metadata for a single table can become quite large in the case that the table has a high number of partitions or files. For example, the serialized Thrift object has been seen to exceed the 2GB size limit for a Java array which prevents the metadata from being transferred.
- During deserialization, we have seen the peak memory usage and allocation rates cause long garbage collection pauses or OOMs in the impalad process.
- When the cache is cold (eg after an invalidation) the performance of seemingly-simple operations is unexpectedly slow. For example, a ‘DESCRIBE’ query which the user expects to fetch only a small amount of information may take several minutes.

Anecdotally, we have heard of a case where a user has a single very large table with so many partitions that any attempt to load it will cause all impala daemons to OOM. They have resorted to educating the cluster users that this table must not be referenced in any Impala query (even a seemingly innocuous ‘DESCRIBE’)

Usability issues

Apache Impala is often deployed in coordination with other systems such as Apache Hive and Apache Spark. Those systems can independently modify the metadata in the Hive Metastore, or modify the set of files in partition directories. Because of Impala's aggressive caching of metadata, users often experience surprising semantics, such as:

- A user creates a table in Hive but the table does not appear in Impala. They must issue "invalidate metadata <tablename>" (referencing a table that does not exist) for it to appear. Alternatively they may invalidate the entire cache using 'INVALIDATE METADATA'
- A user swaps out the contents of a partition directory using 'INSERT OVERWRITE' in Hive or Spark. Impala queries fail with FileNotFoundExceptions attempting to access now-removed files until the user issues an appropriate invalidate or refresh statement.

Because the end users of Apache Impala are often not Hadoop ecosystem experts, we have empirically seen them resort to policies such as "every time you query a table, issue a REFRESH first" or "whenever you have some issue, try invalidate metadata and rerun the query". As such, despite Impala's efforts to obtain high cache hit rates due to aggressive caching, a large number of queries end up hitting a completely cold cache and re-fetching metadata inline with planning.

Restart time

In the current design, when the impalad restarts, it downloads a full copy of the cached catalog from the catalogd before it can start receiving queries. This can take tens of minutes on clusters with a lot of data. We have also seen this cause the receiving impalad to OOM while attempting to parse the large thrift objects.

Workload isolation

Because the current architecture centralizes the updates and distribution of metadata, different workloads can have a high impact on each other. For example, if one user runs 'refresh' on a table containing 100k partitions, another user's attempt to load a table containing just one partition may be blocked on central resources such as threadpools or the statestore update locking.

This increases risk when cluster operators consider adding new workloads to existing multi-tenant clusters. A new "misbehaving" tenant (eg who creates tables with 100k+ partitions) can have an extremely negative effect on the performance of another tenant's queries.

Previous improvements

A large amount of work has been invested in attempts to improve the above issues in areas such as:

- Memory usage (prefix-compression of file paths, manual dictionary coding of hostnames, disk IDs, etc)
- Attempts to improve GC pressure by partially using flatbuffers to reduce object counts.
- Attempts to compress catalog updates on the wire
- Attempts to make the locking more fine-grained in the catalogd
- Attempts to avoid large metadata refreshes from blocking smaller “urgent” refreshes

However, the increasing complexity of these workarounds has also caused a number of bugs which have been difficult to debug, reproduce, and fix. The general feeling among Impala developers who have been working on these improvements is that the current design has reached a point of complexity beyond which any further attempts to incrementally improve it have an amount of risk that outweighs their potential benefits.

Proposal for a new design

Functional Goals

- Maintain existing feature-set of Impala: all queries and DDL operations that worked previously should continue to work with the same results
- Maintain support for all currently-supported file formats, partitioning strategies, and scale dimensions.

Performance Goals

The following goals are assuming that the memory budget allocated to impalad daemons is unchanged. That is, these goals must be achieved without an increase in memory consumption.

- No more than a 5% average throughput regression for a *best-case* workload (eg read-only workloads with no invalidation or modification of metadata)
- Substantial improvement in 75th, 99th, max planning latencies as measured by the coordinator for *typical* workloads involving some mix of INVALIDATE/REFRESH, mutative DDLs, etc.
- No more than a 10% increase in load (CPU cycles, etc) placed on HDFS NameNode, HMS, or cloud service API, for a typical workload as described above.
- Substantial improvement in cluster startup/restart time.

- No throughput regression for *typical* workloads (as described above)

Non-Goals

- No changes to execution path. Generated plans should remain identical.
- No improvements to “snapshot” semantics or automatic metadata discovery. I.e. maintain existing feature set but not extend.

High-level design overview

This document proposes that we introduce a new architecture for Impala’s treatment of metadata. The high level design is as follows:

- Catalogd is removed. Each coordinator manages its own caching[1] and fetches metadata directly from source systems as necessary. The cache is populated as needed when queries are submitted to that catalog, and implements an eviction policy based on size and/or TTL.
- The existing read-your-writes semantics are maintained by distributing cache invalidation messages around the cluster after mutative changes (eg DDL). Upon receiving such an invalidation message, the coordinator will remove any affected items from its local cache, such that the next query to reference that table/partition will fetch new information that reflects the change.
- All caching is as fine-grained as possible and lazy-loaded as late as possible during planning.

Impalad changes

- Frontend incorporates a cache which holds the raw results from external API calls (eg metastore Partition thrift objects or HDFS FileStatus objects)
- Each query starts with a fresh ‘Catalog’ instance. As tables are referenced, information is fetched from the cache.
- Partition details and file listings are deferred until after partition pruning.
- A “LocalCatalogExecutor” is introduced to run DDL statements. Cache invalidations are broadcast via the statestore in a new topic. Upon receiving an invalidation, the impalad removes any affected items from its cache.

[1] The Future directions/optimizations section mentions the potential to later use a distributed cache such as memcached for certain pieces of data. The initial plan is to use only local caches despite the redundancy of information, and encourage users to use a relatively small number of coordinators (perhaps two or three for each tenant)

Anticipated benefits

Memory usage

Because we will implement a cache eviction policy[2], the memory usage can be bounded. Based on decades of database research as well as some user traces, we believe that most workloads exhibit strong temporal locality of reference -- i.e a vast majority of queries reference a vast minority of tables, and that the design will yield a net reduction of cache size on each impalad with a similar hit rate.

Start time

Because the impalad can run with an empty cache, it starts up nearly immediately. There is no need to wait for any data to transfer from metadata sources in order to begin servicing queries.

Cache granularity

One of the largest culprits for metadata issues today are tables with tens or hundreds of thousands of partitions. We believe that users typically use such fine-grained partitions to achieve effective partition pruning for their query workload. In other words, despite having a large number of partitions, the majority of queries access only a very small number of partitions after pruning. Therefore, we believe that fine-grained caching of partition and file metadata will result in much-improved performance with cold caches following service restarts or cache invalidations.

We also believe that lazy on-demand fetching of metadata will vastly improve the perceived performance and usability of Impala. For example, we can guarantee that a small query such as ‘DESCRIBE t’ does a constant number of operations against metadata sources and that such queries will therefore respond much more quickly.

Data freshness/usability

By incorporating a TTL into our cache, we can give users a soft guarantee that any changes made in external systems such as Hive or Spark will become visible in Impala within a bounded

[2] The current prototype detailed below uses the Guava LoadingCache, which implements LRU and time-based eviction. In the future we expect to need an improved cache policy that prevents a single query which references a large amount of metadata from blowing out the cache of hot metadata. Candidates such as W-TinyLFU or LHD seem like good options.

amount of time. This will reduce the need for users to manually issue REFRESH/INVALIDATE commands in a large number of scenarios.

The TTLs used for different pieces of information may differ. For example, since block locations are important for performance but not correctness it might be desirable to use a longer TTL. Items that are relatively cheap to fetch might use a shorter TTL since misses are less expensive.

Note: the proposed solution still provides no *hard* guarantees on data freshness unless the user explicitly issues a REFRESH/INVALIDATE. This is not a regression from today, and improving these semantics, while desirable, is not a goal of this project. See "future work" section below.

Tenant isolation

Because each coordinators manages its own cache, we will be able to segregate different tenants to different coordinators and thus segregate their caches. This will provide a few benefits:

- Avoids issues such as lock contention in the catalog daemon update protocol (user A refreshing table T won't affect user B refreshing table U on a different coordinator)
- Avoids cross-workload contention of shared thread pools such as impalad-wide HMS clients.
- Static partitioning of cache capacity: if tenant 1 loads a very large table, it cannot evict items from the cache for tenant 2 on a different impalad.

Architectural simplicity

Because the new cache is loaded on-demand and never updated in place, we anticipate a net reduction in code complexity. For example:

- Each query performs its own fetches from the cache and creates its own query-local catalog objects. Thus, the catalog objects do not need to be thread-safe. Concurrent fetches of the same underlying object are managed by synchronization within the caching layer.
- Because catalog objects are never updated in place, there is no need to compute 'incremental' changes or worry about coordinating locking. Instead, only simple idempotent "invalidation" messages need to be broadcast. The invalidation messages can be granular based on the changes that were made (eg adding or dropping a partition need only invalidate the cached partition list, but not the cached block locations)
- Because there is no distributed update protocol, there is no need to aggressively optimize the transfer of large catalog objects over the network.

Note: it's possible that certain optimizations made in the current design could still apply to the new design. For example, use of flatbuffers for certain pieces of data may still be beneficial for

reducing GC overhead. However, we intend to only re-introduce these optimizations as deemed necessary, driven by empirical data, rather than re-introduce them “by default”.

Anticipated risks

The following potential risks have been identified with the new design:

Load on source systems

With no central coordinator, we now have potentially many more daemons directly accessing source systems such as the HMS or NameNode.

We believe that with the more fine-grained fetching of information, this will be mitigated. Previously, an invalidation of metadata on a large table would require re-fetching thousands of partitions worth of information. Query traces from users indicate that much of this information was never used before being fetched again. In the new system, we would avoid fetching this information at all.

We also plan to investigate optimization opportunities on the source systems. For example:

- Batched listing APIs to reduce the number of RPCs to the namenode (eliminating fixed per-RPC overheads)
- Selective partition-information fetching APIs in HMS (e.g. to eliminate fetching incremental column statistics and HLL blobs for queries which do not require them)
- Investment in previously-identified points of overhead in source systems. For example:
    - Continued profiling and improvement of the HMS “Direct SQL” code paths
    - Implementation of AESNI-assisted wire encryption for HDFS NameNode RPCs
    - Micro-optimization of important HDFS RPCs such as listFiles

We believe based on early prototypes that the above API-level improvements and optimizations can reduce load on source systems by 5-10x.

Dependencies on current semantics

Currently, most users of Impala consider the aggressive caching to be detrimental: explicit REFRESH/INVALIDATE commands are necessary in order to see changes made in external systems such as Hive or Spark. However, there is a risk that some users may be taking advantage of the semantics of this caching using a workflow such as:

- REFRESH t1 ; REFRESH t2
- Start running a BI workload which references t1 and t2.
- Start running an ETL job in Hive which adds partitions or files to these tables.

    - (BI workload continues running concurrently with the Hive job, and sees a "snapshot" of the tables with no concurrent changes visible)
- ETL job finishes and runs REFRESH on both tables.
- The BI workload now sees a new "snapshot" of the tables.

We have heard mixed reports whether customers currently rely on the semantics above. In a tightly controlled environment, the above workflow could work. However, the workflow is already highly risky due to several possibilities:

- A catalogd restart during the ETL workload would cause new versions of the metadata to be fetched.
- Any other tenant issuing a 'INVALIDATE METADATA' command would cause new versions of the metadata to be fetched.

We have not yet identified a workaround for workflows like the above. We are considering whether it will be necessary to add an explicit 'snapshot' feature. However, the effort of designing and implementing a snapshot feature is likely to be large, so we are considering it out of scope for this document.

Comparison with other SQL systems

To some extent, there is an "existence proof" that a design like the one proposed can work well when considering other SQL-on-Hadoop engines:

Hive

Hive itself does not currently perform any metadata caching. Each query directly accesses the HMS. This has recently been identified as an issue, particularly when running with the HMS connected to a remote RDBMS in the Cloud environment. There is an active proposal to add a metadata cache in which HS2 maintains a full cache of the metadata from the HMS. Their proposal differs from this one in that it does not include eviction, but also does not cache non-HMS metadata such as block locations.

Presto

Presto has a time-based cache of Hive metadata. However, as of April 2017 the cache TTL has defaulted to 0, meaning that the cache only prevents repeated HMS accesses within the scope of a single transaction. Presto's transaction support appears rudimentary.

Spark SQL

Spark currently does not perform any metadata caching. Each query separately accesses the sources of metadata.

Summary

Although Impala has traditionally focused more on interactive SQL applications with sub-second response times compared to the above engines, we believe that in most of those applications there is strong locality of reference and we can achieve a good cache hit ratio even with a relatively short TTL. For example, as an interactive BI user “drills down” into a particular analysis, the same set of tables and partitions will be repeatedly used and performance will not suffer compared to today’s aggressive caching.

Risk mitigation

Impala is already a critical piece of infrastructure for many users. Therefore it is paramount that this project be developed in a way that minimizes any risk to users.

Treatment of existing code

- The new catalog code should coexist with the existing code, enabled only by a flag. Impala should continue to operate in the old mode by default.
- Minimal refactoring of the old code to introduce interfaces where necessary instead of direct dependencies between the planner/analyzer and the catalog. In cases where more complex refactoring would be necessary to share code, we will choose copy-paste instead of refactoring to avoid introducing risk to the existing code.

Incremental deployment option

It should be possible to deploy one or more coordinators within a cluster in the “new” mode, co-existing with coordinators running in the legacy mode. The plans generated from the new mode should be interoperable and run with no backend execution changes. This will allow users to test out the new mode without deploying an entirely new cluster by spawning a coordinator-only impalad that is not part of the production load-balancer pool.

Prototype results

Over the past two weeks we have developed a prototype of this design as follows:

- Refactored Catalog interactions from the Frontend into interface classes
- Implemented “read-only” functionality sufficient to plan queries against HDFS tables using a local cache with 60sec TTL
- Implemented REFRESH and INVALIDATE METADATA as cache invalidations within a single impalad

- Augmented the impalad with a flag which converts all queries into their equivalent “EXPLAIN” in order to benchmark the planning/metadata phases in isolation
- Applied several optimizations to the HDFS NameNode including a new batched listLocatedStatus API (patches to be contributed to HDFS separately)

Against this prototype we have run a workload in which multiple threads sample random subsequences of a query trace from a Cloudera customer and replay those subsequences against a coordinator which performs appropriate planning. The workload involves a mix of queries including REFRESH/INVALIDATE statements, queries from ‘COMPUTE STATS’, and various end-user workloads including ‘DESCRIBE’, ‘SHOW TABLES’, etc, as generated by BI tools. The corresponding NameNode and HMS metadata have been loaded into a small cluster approximately 2ms away (typical worst-case within-datacenter round trip). The NameNode has been locally modified to return randomized fake block locations for all files.

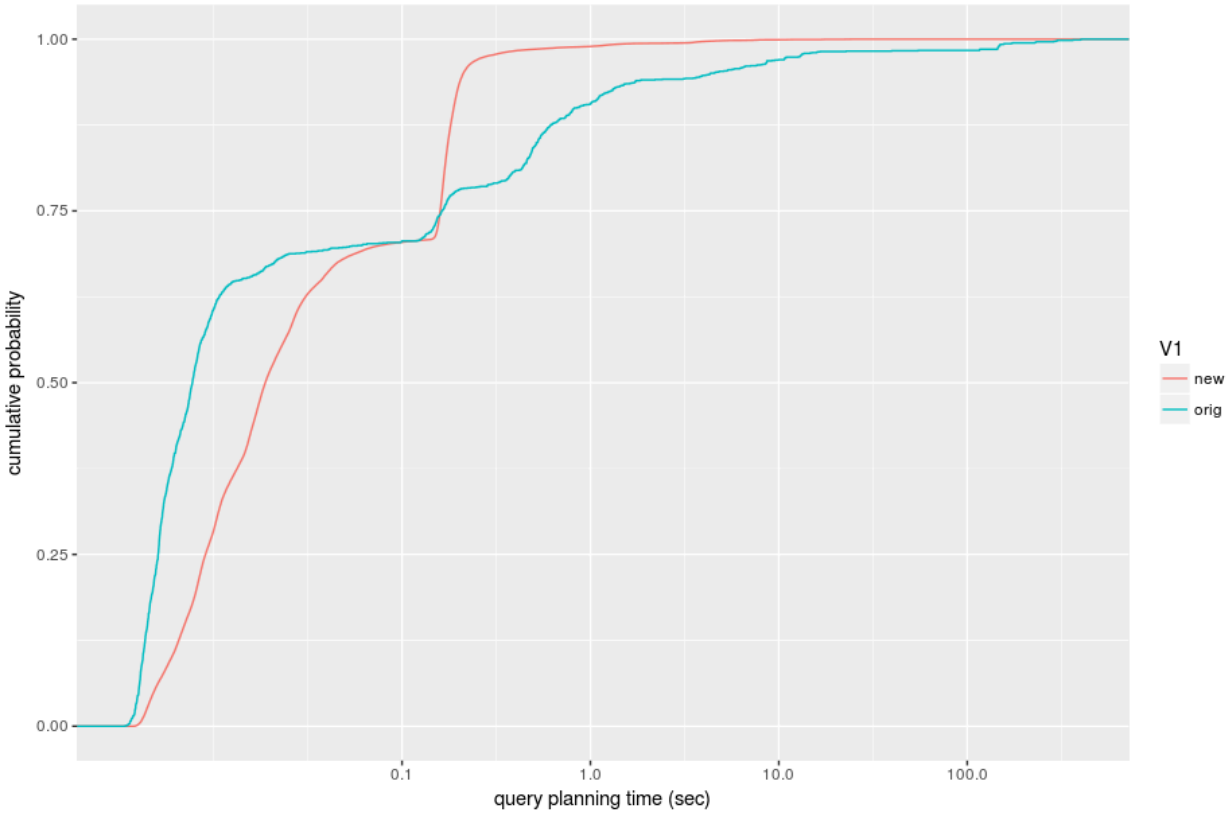
By running this workload for 100,000 queries we are able to compare the relative latencies of the existing catalogd-based implementation and the new design. Initial results are as follows:

Latency percentiles across all queries

```
percentiles for scenario:  new
        50%         90%         95%         99%       99.9%        100%
 0.01880598  0.18806386  0.21126556  1.07064593  7.82321382 25.16159415


percentiles for scenario:  orig
         50%          90%          95%          99%        99.9%         100%
8.158922e-03 9.416005e-01 5.192340e+00 1.729687e+02 4.018958e+02 5.993552e+02
```

Cumulative Distribution Function of Latency



(if you are unfamiliar with this type of plot, refer to Reading ECDF graphs)

Summary

The proposed design shows an approximate doubling of the median query planning time from 7ms to 18ms. However, given other constant overheads in Impala query execution such as codegen, fragment start time, etc, the end-to-end degradation for very short queries will be significantly smaller. We anticipate with further optimization of the “hot cache” case we can narrow this gap if necessary.

The proposed design shows significant improvements in 95th, 99th, 99.9th, and max planning times (24x, 161x, 51x, and 24x respectively). Qualitatively (based on inspecting logs, jstacks, etc) these improvements are due to:

- No need to load large amounts of metadata for small queries such as DESCRIBE or SELECT with a small number of referenced partitions
- No cross-query blocking (DESCRIBE on a table is not blocked by REFRESH on some other table)
- No waiting on statestore heartbeats

Though these results are based on an early prototype and only measure a single workload, the numbers indicate that this design is worth pursuing further.

High-level development plan

Per above, the goal is to be able to develop this code in parallel on Impala trunk without creating a new feature branch. Minor refactoring will be required to introduce interfaces as necessary, along with a new configuration flag to enable an impalad running in the new mode.

During early development, we will treat this feature as fully experimental/unsupported:

- We anticipate that early versions of the code will fail a large percentage of the existing Impala test suite. As such, the existing Impala precommit jobs will *not* enable the new catalog mode, but will continue to test the existing architecture. This will insulate the majority of Impala development from disruption caused by this project.
- If an Impala release is made while this feature is still under development, it will either be explicitly documented to be unsupported, or it can be hard-coded to “disabled” in the release branch.

After reaching an appropriate feature set, we will set up a separate job which runs the test suite with the new catalog mode enabled. We anticipate that the pass rate will initially be low, and we can track this rate as we approach 100% passing. The goal is to maintain existing semantics as much as possible.

Evaluation / success criteria

As this project proposes a fundamental change in the architecture of Impala, it is imperative that we have a clear set of criteria to evaluate it vs the existing architecture. We will evaluate the following high-level areas:

- **Average Planning Latency**: given some workload, we do not want to introduce a significant regression in the average planning performance of user queries. A small regression (eg from 20ms to 30ms) is acceptable so long as the query remains “human real-time”. Contrarily, a regression from 20ms to 500ms would not be acceptable.
- **High-percentile Planning Latency**: given some workload, we would like to improve the high-percentile latency of plan generation by several orders of magnitude. Initial experiments indicate that the 99th percentile planning can currently reach into the hundreds of seconds in highly concurrent scenarios which include REFRESH and INVALIDATE METADATA statements.
- **Bounded memory usage**: the new solution must be able to bound its memory usage, and achieve the above latency metrics using only a similar or smaller amount of memory compared to today’s implementation.
- **Semantics**: we must maintain critical semantics such as within-session read-your-writes, cross-session read-your-writes with SYNC_DDL enabled, etc.

- **Load on external systems:** given a test workload, we should measure the resource consumption of external systems (HDFS NameNode, cloud service, HMS, and underlying HMS RDBMS) and show no significant increase in key metrics like CPU, network traffic, monetary cost (due to cloud API calls), etc.

Long-term plan

Replace, not augment

The long-term goal of this project is to replace the existing architecture, not augment it. Given that, we expect that some number of releases of Impala will include both architectures. Depending on various factors, we may choose to have some number of cross-over releases during which either architecture can be configured using a flag, but in the long term we expect that the new architecture's feature set and performance will be a strict improvement over the old architecture and the old architecture will be removed.

Future directions/optimizations

In discussing this proposal, various other potential optimizations and improvements have been identified that could slot in at a later date. We'll explicitly **not work on these as part of the initial scope**, unless we find that doing so is necessary to achieve the goals stated above:

- **Distributed cache** - distributed caches such as memcached, infinispan, redis, etc, have been discussed as a potential way of (1) reducing redundant storage across impalads, and (2) keeping data synchronized across nodes. After the re-design proposed in this document, it seems like moving from a local cache to a distributed cache would be relatively straight-forward architecturally if deemed necessary.
- **Replacing HMS** - there is a movement in the Hive community to refactor the HMS out of Hive into its own project, and potentially replace its backend storage with something more efficient than the current ORM/RDBMS code. This is largely orthogonal to this proposal and would have compound benefits.
- **Standalone Impala** - there have been occasional past discussions around a "standalone Impala" that could run on a single node against local or remote filesystems. This proposal does not include that goal, but the refactoring may provide a useful stepping stone to alternate catalog implementations.
- **Removal of need for REFRESH/INVALIDATE** - it would be desirable to completely remove the need for users to ever issue REFRESH or INVALIDATE statements, perhaps by coordinating more closely with the notification mechanisms provided by HDFS and

HMS. The current project will *reduce* the need for these statements but still only provide a soft time-based cache expiration rather than any hard guarantees.

- **Dynamic scheduling of scan ranges** - currently, Impala needs all block location information during planning because it assigns all work to backend nodes up front before query execution begins. Instead, it might be possible to dynamically schedule ranges to available executors and pipeline the fetching of block information after starting the query. This would require major changes outside of the catalog component and frontend, however, so is out of scope for now.