

IMPALA-7954 : Impala Catalogd Auto Metadata Update

Status: in-progress

Authors: Bharath K, Vihang, Parna

[Introduction](#)

[Goals](#)

[Non-Goals](#)

[Proposed Auto-Update design](#)

[Design Considerations](#)

[HMS Change Log](#)

[Update speed](#)

[Concurrent Impala / Non-impala updates](#)

[Changes from Sentry](#)

[Self-events](#)

[False positive](#)

[False negative](#)

[Alternative proposal to detect self-events](#)

[Support for DML](#)

[Fine-Grained / Direct Update](#)

[Batching multiple partition events](#)

[Testing Consideration](#)

[Scale and Load](#)

Introduction

Impala maintains a cache of table metadata that are being queried by users. This cache has been pretty static where it is not really updated unless users manually issue refresh or invalidate commands. Issues related to manual metadata updates have been captured in length earlier.

In 3.1.0 we introduce mechanisms to invalidate table metadata cache from Impala's Catalogd when certain conditions are met. These conditions are non usage of table over period of time and in case of memory pressure in CatalogD. While the mechanisms are designed to reduce memory pressure and avoid crashes due to OOM in catalogd, they do not help in exchange of metadata when change happens.

Most of the users bring in data using ingest tools such as Sqoop, Flume, Streamsets etc. The batch transformations happen via Hive or Spark and the application and BI tools use Impala for accessing the data. Hence the exchange of metadata is critical for smooth operation of the data warehouse. It is easy to see why Impala's manual metadata refresh could be challenging.

This process is already cumbersome in an environment where Hive, Impala, Spark and other tools are running in the same cluster. The challenge of exchanging metadata from ETL tools such as Hive and Spark to Impala is magnified in multiple cluster environment. How to refresh all the catalogs? Without the automated mechanism for update the operation of multi-cluster deployment would quickly become complex.

Goals

- Automatic update of Impala Catalogd when new data arrives in the table and such changes are recorded in the HMS changelog.
- The update is applied to the table metadata that are in complete state within catalogd.
- The updates are applied within 5 seconds. Note this is the ultimate goal and might not be achieved in first version.
- Reduce the need for users to do refresh and invalidate.

Non-Goals

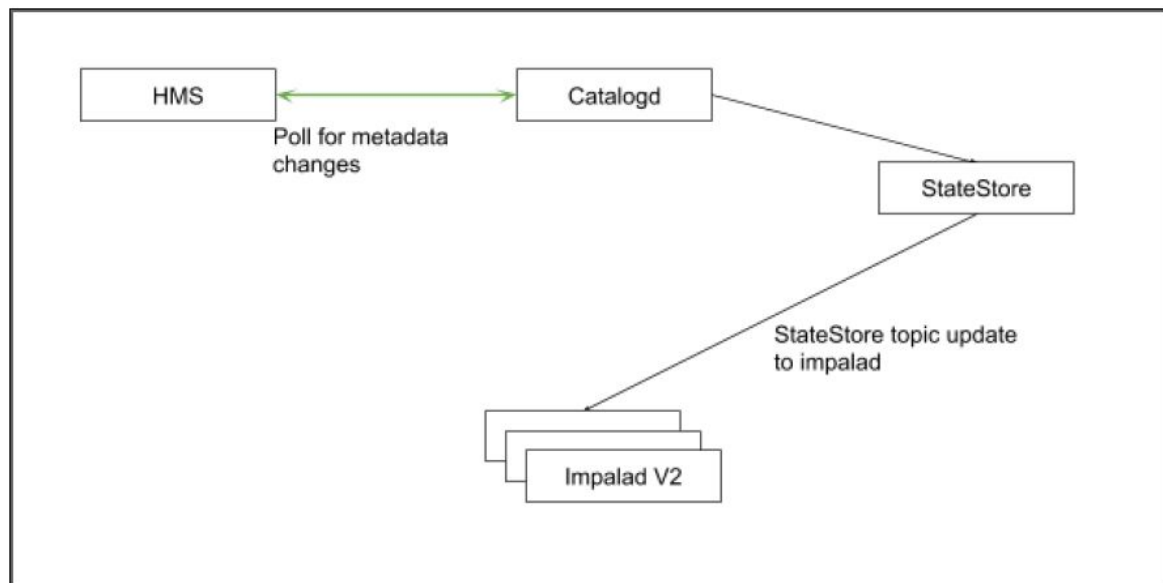
- If the data is being added to the table via non-sql mechanism. I.e. adding files to partition directories out of band, then these will not generate updates in Catalogd.
- Completely remove the Invalidate and Refresh commands.

Proposed Auto-Update design

To address these challenges we propose the following enhancement

- Impala will poll HMS for changes to metadata. HMS provides a notification fetch API to capture all metadata changes from a given EVENT_ID.
- Invalidate or refresh the relevant catalogd cache based elements based on the events from HMS.
- The metadata change updates should be available to Impalad's within 5 seconds.
- Upon detecting the change in table metadata, the initial implementation may just invalidate the table in catalogd.
- In future, changes from certain partition level events will be applied directly to the table metadata instead of invalidating the whole table.
- The primary goal is to make poll based invalidation work with V2 catalogd.

- Making poll based invalidation work with V1 would be a stretch goal.



Design Considerations

HMS Change Log

Hive Metastore has a mechanism to configure multiple listeners (transactional or non-transactional), as plugins, which are called upon changes to metadata. A transactional listener is called within a transaction and a non-transactional listener is called after the transaction is successfully completed. At a high-level, this plugin logs CREATE, ALTER, DROP events for various metastore entities (Database, table, partition, function, index etc) in a table when such API calls are made to change the metadata by any client. Note that this is a transactional listener which means that it is guaranteed that the event is published only if the metadata API call is successful. All the events have an EVENT_ID which is monotonically increasing long value and guaranteed to be unique across HMS instances.

Currently, the notification mechanism is pull based. Any client which needs to keep in sync with the changes in HMS can poll the notification log using HMS API *get_next_notification*. The notification fetch API is fairly primitive in the sense it does not provide the ability to filter events. It returns all the NotificationEvents from a given an event ID upto a maximum number of size specified in the request. In order to find the current notification id, HMS provides another API called *get_current_notificationEventId*

Update speed

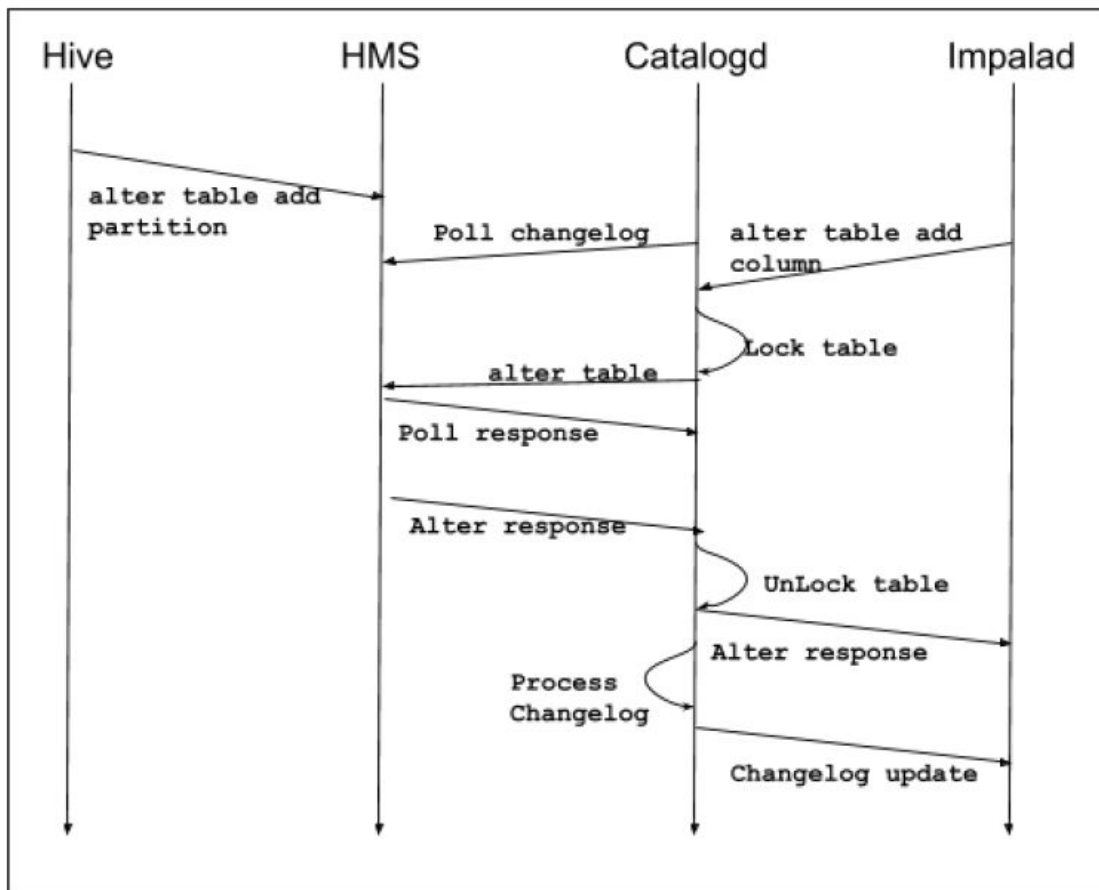
Catalogd sends metadata changes to impalads via statestore. By default Statestore gets the catalog delta and pushes to impalad's every 2 seconds. In case of V2, the change message is very small and is used to invalidate the table metadata cache in each impalad. We do not plan to change this mechanism as of now. When Catalogd polls for change log and applies changes to its copy of metadata, these changes will still take some time to get propagated to Impalads. This is not necessarily a limitation in the first phase. By default, the updates go out every 2 seconds which is sufficient to meet our design goals of update within 5 seconds. We can potentially look at doing a direct update to Impalad's in case StateStore propagation becomes a bottleneck, however we do not envision this in the first release. Alternatively, we can increase the frequency of statestore update to lower than 2 sec.

Impalad's CatalogTableInvalidator will poll HMS change log APIs at fixed intervals. The default poll will happen every second. This should be sufficient to meet with 5 seconds design goal for updates. The polling **every few seconds** does add additional load on HMS. However, we are expecting only a handful of high-frequency pollers hence the current HMS should have sufficient capability to handle the load. Note that there is a trade-off on the load of HMS since this can potentially get rid of multiple fetch requests from CatalogD when users issue invalidates. It is likely, that overall load on HMS will decrease if we can get rid of user-initiated invalidates and issue system generated invalidate commands only they are really needed. In case the testing reveals that HMS may possibly get overloaded, we plan to optimize the HMS APIs with a more filtered poll from Impala. The polling interval should be configurable by the users.

Concurrent Impala / Non-impala updates

It's possible to alter the same table from both Impala and Hive simultaneously. This may not be a common case on user deployments. If it does, we need to consider what happens in Catalogd when Impalad is trying to modify an object and the polling in Catalogd is modifying the object based on changes from Hive. Note handling the concurrency may not be required for the set of use cases we are targeting in the initial implementation. However, for the completeness of the considerations, we would describe it in detail here.

Concurrent updates to a table from within Impala are serialized in CatalogD by taking table-level locks in catalogD.



In the above diagram if the alter table is initiated by Impalad then it hold the lock on the table in catalog. This lock is not released until the changes are propagated to HMS and applied to the internal state of the table representation in Catalog. Hence if there is a change from outside (as indicated by a event) on the same table is seen, Catalogd will block such updates from the events till the self-initiated alter command completes. It's possible that the alter command to fails in HMS, however that does not change the semantics. Hence the current table lock mechanism would be sufficient to handle concurrent update case.

Changes from Sentry

Sentry is already being polled by Catalogd. Though the default interval for sentry polling is currently 5 seconds. This is still within the design goal of update within 5 seconds. In practical situation, for new tables if the sentry information is not available in catalogd, then an RPC is generated to fetch the information. The issue could be with having stale sentry permissions, however without any additional work it would be as it is today where the information is updated every 5 seconds.

Self-events

It is possible that an event received from the HMS notification log is generated by the same impala cluster which is trying to apply such an event. Such “self-events” need to be distinguished from the events which are generated from other systems (Hive or other impala clusters) for avoiding unnecessary invalidates.

Having a full-blown optimistic locking using version number described below is a fairly complex feature which will need database schema changes along with new HMS APIs.

A much simpler alternative to achieve the same results without code changes to HMS APIs and schema is to add a changelD (combination of catalog serviceld and catalog version) to the object parameters when Impala creates/alters the metastore objects. These UUIDs could be seen as transaction ids for the metadata update operations. When a event is received later, we should compare the changelD in the parameters with the list of catalog version numbers which catalogD has for that object. If there is a match, it can be assumed that the event is a self-event and we can ignore the event. Once the changelD is matched, it can be removed from the cached object. If the changelD does not match, event is determined as not a self event and need to be acted upon. The advantage of this approach is that it is completely self-contained with no changes needed on HMS side. Note, that this approach provides a best-effort mechanism to avoid unnecessary invalidates. There are following cases to consider:

False positive

A false positive is a wrong determination of self-event. This could happen in case of collisions of changelD which are by definition not possible. Another case when a same changelD is present in a event when a non-Impala client makes change to a object. Consider the following example: Table A is altered by Impala cluster A which has a serviceld a1 and version number 10. These two values are added to table parameters before the alter RPC is made to the HMS. The event received contains a1 and version number 10 in its parameters. Now lets consider a client like Apache Hive. It also makes a alter call on the same table A. Since Hive does not recognize these serviceld and version number, it keeps them as it is. The event generated by this second alter operation also presents the same serviceld a1 and version number 10. In order to handle this case, it is critical to clear the pending version numbers from the catalog table when the first event is received.

False negative

A false negative is the case when we determine self-event as not a self-event. This could happen if the order to commits on the HMS server is different than order of API requests to update the same object from multiple systems. For example, Hive and Impala issue

alter_partition on same partition in the order t1 and t2 but on the HMS side they are committed in the order t2 and t1. In such a case we will receive the events which may trigger unnecessary refreshes on a partition. This is a corner case which is less likely to be seen commonly and it is not handled in the current design. Refresh operations being idempotent will cause unnecessary work done in such corner cases without any loss of correctness of functionality. Another case when there could be a false negative is we see an event with a changeld that we could not keep track of at the table level. In order to bound the memory consumption all the in-flight changelds, the current design only keeps track of first 10 version numbers which are in-flight at each table. This means that if there are more than 10 events generated on a table between two successive polling operations, it is likely that the 11th event seen on the table would be wrongly determined as a self-event. This is okay, since the actions taken on self-event is to invalidate the table. In the unlikely case of this happening this is a performance penalty and not a correctness problem.

Support for DML

Not all metadata objects are created using DDLs. “insert into” or “insert-select” (with overwrite option) queries add new files to existing partitions or tables. This new file information is of interest to Impala as well. In order to detect such events we propose to make use of INSERT_EVENT. The InsertEvent provides the partition object and as well the file information which is added by the insert query. This is an on-demand event which clients can generate using the HMS API fireListenerEvent() to fire such an event. Hive already has the capability to generate such events when data is inserted in existing tables and partitions provided hive.metastore.dml.events is set to true. However, in order to support automatic updates from one Impala cluster to another, we will need to modify Impala’s code to generate such events when insert queries change existing tables or partitions.

Fine-Grained / Direct Update

Invalidating a large table which has thousands of partitions is not very efficient, particularly when the events suggest that only some of the partitions of the table have been changed. In case we see ADD_PARTITION, ALTER_PARTITION and DROP_PARTITION we should only add/refresh/remove the partition involved instead of invalidating to avoid unnecessary refetch of large amount of data. In addition to adding just the Partition objects, these events should also be used to update the file metadata information in the catalog.

If the table corresponding to the add/alter/drop partition events is incomplete, we can ignore the events since the table is not being used at the time when the event is received since the table will be loaded using the latest information from HMS lazily on an on-demand basis. However, if the table is not Incomplete following actions can be taken:

ADD_PARTITION

In case of add partition the event has the tablename and dbname information. This can be used to get the HDFSTable object. The partition object itself can be deserialized from the event message. HDFSTable provides a method to createAndLoadPartitions which loads the partition to the HDFSTable as well as loads the file metadata information for each of the partition. This method can be reused to add the partition object in HDFSTables's cache.

Note: If the added partition has HDFS caching enabled/disabled catalogD invokes alter_partition call on the added partition to update the cache directive in the partition parameters. The events generated by such alter_partition calls can be determined as self-events as described in the section above.

ALTER_PARTITION

Alter partition events are generated when the partition schema is changed (from Hive or other ETL tool) or more commonly when new data is inserted or overwritten in a existing partition. In case of Hive it triggers alter partition events in order to update the partition statistics in the metastore. <TODO> Does the other tools like spark also generate alter partition events when data is inserted into existing partitions? </TODO> In case of alter partition events we can use reloadPartition method in HDFS Table which drops the partitions and adds it again while reloading filemetadata.

The altered partition object could have incremental statistics in its parameters. In such case if the event does not have statistics in the parameters, care should be taken to not overwrite the existing statistics which could result in performance degradation. Instead we can create a new HDFSPartition object using the existing statistics while still reloading the filemetadata for the partition so that it has the latest information.

DROP_PARTITION

In case of drop partition events we can rely on HDFS table dropPartition method which removes the filemetadata as well as fileMetadataStats along with removing the HDFSPartition from the partitionMap.

Batching multiple partition events

HMS does not have ADD_PARTITIONS or ALTER_PARTITIONS event type. Instead it creates many instances of such events generated in one transaction. It makes sense to batch such events together so that we minimize multiple API calls (each of which will hold a table lock). All the above mentioned events can be batched together for each table and catalogD can be updated to add/refresh/drop the list of Partitions coming from the event stream. It should be noted, that batches of events are done for each polling duration. If a certain batch of events spans across the default fetch size, it may still be updated in multiple polling intervals. For

example, if a Hive query adds 1000 partitions dynamically during query execution and the fetch size from Impala is 100 all the events pertaining to this hive query will be fetched over 10 consecutive polling intervals. An alternative to this approach is have the default fetch size reasonably high (~10000) so that such cases are uncommon. It is possible to get all the events from a given eventId if the fetchSize is set to -1 but it is not recommended to do this since this put unnecessary pressure on HMS and we have seen escalations on HMS related to this.

Testing Consideration

Scale and Load

The changes made are relevant to both single and multi-cluster. In a single cluster, we expect performance improvements because we are not relying on updates from StateStore. Hence, it is good to measure performance improvement in a single cluster setup which uses notifications to create a baseline.

The primary step would be to get a good understanding of

1. how many API calls should be generated
2. how many concurrent queries should be run
3. what range of batch size/poll_interval do we intend to support
4. how many clusters should we test for in the multi-cluster case.

Looking at some customer workloads will give an idea about the above numbers.

With these numbers in mind, we can create tests for the following areas:

1. **TESTING NOTIFICATION PROCESSOR** - We are going to process the notifications in batches from Impala side. So we need to test with large batches of notifications and test how the notification processing scales with batch size.

TEST STEPS

1. Create notifications of different batch sizes. This may be done by changing the POLL_INTERVAL so that different batch sizes are generated. Or create dummy notifications and use it in batches for the test.
2. Process these notifications using the Notification Processor and benchmark the time taken to process all the notifications.
3. If we implement logic to optimize "Multiple events on the same object", we can use this test to measure the improvement achieved by the implementation.

2. **TESTING HMS** - When Impala starts consuming notifications, HMS gets more API calls, as to get notifications and possible further requests for the Table/Partition objects. As of now, the plan is to poll the HMS notifications from Impala in specific intervals (POLL_INTERVAL). The POLL_INTERVAL which is configurable should be modified to different values. HMS can get

loaded when the POLL_INTERVAL is small and there are multiple instances of Impala polling HMS at the same time.

TEST STEPS

1. Have two variables number_of_impala_instances and POLL_INTERVAL.
2. Change the number of instances and POLL_INTERVAL to see how the HMS scales.
3. Benchmark the load on HMS based on the frequency of polling and number of Impala instances.
4. We can have Hive clients as well running Hive queries and compare how it affects the overall performance when Impala and Hive both exist in the environment.

It would be useful to benchmark the HMS notification APIs to find out how well they scale under load (number of events and/or concurrency)

3. **TESTING CONCURRENCY** - Performing Impala queries while invalidating/updating catalogd to see bottlenecks caused by a possible lock here.

TEST STEPS

1. Write a set of Impala queries which creates HMS notifications, like Create, Alter, Drop etc. on multiple sets of tables and partitions.
2. Execute these queries in parallel.
3. Benchmark the time taken to complete the queries, we can compare this with the Baseline(model without notifications) and see the performance improvements.

4. **TESTING MULTI-CLUSTER**- Adding multiple clusters and run queries to check if the solution scales.

TEST STEPS

1. Create a multi-cluster environment where there exist multiple Impala instances in each cluster.
2. Perform a set of queries from these Impalas, in parallel and benchmark in general how the whole system scales with the number of clusters.