Phase II Design for HBASE-10070

04/14/2014

Enis Soztutar (enis@apache.org)
Devaraj Das (ddas@hortonworks.com)

Issue : HBASE-10070 (parent issue)
Doc version : 1.0

Related docs

Parent doc

https://docs.google.com/a/hortonworks.com/document/d/1CgEn_b1we5NksR09xuf0PLL3Ea_IC4nBsz-VjunXK8k/edit#heading=h.8xotlio2ebrc

User level guide

https://docs.google.com/a/hortonworks.com/document/d/1N3VWLeIOxz7JqNybusnwOGkrQ1IAhiQlfTKotnZSJlg/edit#heading=h.to8tdc3x6htg

Introduction

This doc will cover changes scheduled for Phase II of HBase-10070, that builds upon Phase I features. We assume Phase 1 features as they are committed in HBASE-10070 branch as of 05/14/2014.

Async WAL Replication

“Async WAL replication feature” as defined in Region Changes section in the parent doc is one of the three proposals for the secondary region replicas to update their statuses with the new data from primary region replica. The three approaches are

- Region Snapshots
- WAL tailing
- Async WAL replication

WAL tailing, and async WAL replication are similar with a major distinction. WAL tailing can be thought as a pull-based data replication scheme, versus Async wal replication is push-based.

Region snapshots

Region snapshots have already been implemented in the HBASE-10070 branch in issues HBASE-10352 and HBASE-10859. With this feature when we open the secondary region replicas, we do list the store files of the primary, and open the files for reading. We do ensure that we can only do reads in directory, and not disrupt the primary region. This feature (as it is committed in branch hbase-10070) is useful for read-only use cases, where after writing the data to the table, we can do a flush, and reopen the table (increase region replication). Then this read-only data will be served by multiple region replicas for higher availability. If the data is read-only, all reads can be done with TIMELINE consistency to have high availability for the data.

The second use case for region snapshots is to enable an automatic file refresher for secondary regions. If enabled, this refresher chore will be run in every region server periodically. It does a file list for primary region's files for every region hosted in secondary mode. When the files of the primary region changes (due to flushes and compaction), the file will be picked and opened in the secondary region as well. This means that the new data will only become available after the primary does a flush, then after some time the secondary does the periodic file list and sees the new file. However, we think that this is still useful for bulk-load only tables, and for read-write tables that the user is fine with seeing more stale data.

WAL Tailing

As discussed in the parent doc, we think that WAL tailing only makes sense in a WAL-per-region implementation. Since every secondary region will want to tail the logs of the primary region, without having a separate file for each region, the cluster will end up with all region servers tailing the logs of every other region server, which is not desired. That is why for this work, we will propose the push-based async wal replication feature instead of pull-based wal tailing.

Async WAL Replication (design)

This design will build on the region snapshots and current replication / log replay features.
We envision that the tailing of WAL from region servers will be built upon the replication code path, while actual replication to secondaries will built upon the log replay features.

Async WAL replication will define its own ReplicationConsumer (see HBASE-10504) and inject it in primary regions RS. The replication infrastructure on the region server side for tailing the WAL, failing over from dead RSs, and zookeeper interactions etc (mostly ReplicationSource, etc) will

be used without much changes. However, instead of sending the edits to a peer remote cluster, each primary will replicate to every secondary. This means that there will be a separate replication queue per Table per region replication count.
We think that this will fit nicely with the work at HBASE-10504 (Define Replication Interface), and will hopefully help define interfaces and injection points better so that the actual multi-DC replication, and wal replication for secondaries will share the wal-tailing code cleanly.

Semantics

Edits from the primary will be unaffected since the secondaries will receive the edits from the asynchronous wal replication. This also implies that the secondaries will still have (possibly) stale data, although the staleness will be reduced to a couple of seconds, and will be similar in characteristic to multi-DC semantics and performance).

Memstore and file management

It is critical that the secondaries share the data files with the primary so that the data is not replicated. Only the memstore data should be replicated. We also do not want to secondaries to persist the edit to their own WAL files because this will multiply the WAL writes, which will result in significant performance loss on write side.

Memstore snapshots on the secondary (no flushes for the secondary regions)

An edit will be written to the primary's memstore, and the WAL of the primary region server without any changes. Once this edit is sent over to the secondary, it has to be put to the memstore of the secondary. Now the issue that has to be dealt with is what happens when there is memory pressure on the RegionServer hosting the secondary.
If we go with the approach where we strictly disallow flushes for the secondary regions, the secondary cannot perform a flush to free up its memory. Instead, the secondary can only free up some memory corresponding to an already flushed file. The secondary cannot do flushes of the secondary memstores, however the region server should be hosting some primary regions as well. In case the edit cannot be put to memstore, because of mem pressure, we will trigger a flush of a primary region, and throw RegionTooBusyException which should throttle the client. When memory is an issue, the replication will slow down this way.
Memstore snapshots already keeps track of the seqId, and saves this in the flushed hfiles. However, the individual cells inside memstore do not keep track of seqIds. This means that the only way to efficiently discard some of the memstore would still be to mimic the memstore snapshots of the primary region in the secondaries. For this, we will reintroduce WALEdits for persisting flush actions in WAL. There will be three new types of WALEdits, (a)start flush, (b) commit flush, and (c) abort flush (we already had these, but got removed some time ago). These will be sent over to the secondaries. Whenever a secondary sees a start_flush entry, it will create a snapshot of its own memstores by that seqId. Since the edits are already replayed

in order, the memstore snapshot contents should be identical to the primary. If the secondary sees the commit flush entry, it will open the the hfile readers and discard the memstore snapshot. In case of abort, it will mimic what the current behavior (keeping the snapshot around for a retry).
The primary region might actually be killed or aborted before the ongoing flushes are committed or aborted. This means that the secondaries will have a memstore snapshot that is already abandoned. The primary will be opened in some other region server, and it will execute another flush which may contain more data than the aborted snapshot. For dealing with this, we can do:

1. Allow multiple memstore snapshots: This will allow us to keep the memstore snapshot from aborted flush as it is. Whenever a new flush is seen without the previous one being complete, we can create another snapshot of the secondary memstore. The queries will be answered from all memstores which are still ordered by seqId. When a commit flush or abort flush is seen, then we can discard all snapshots.
2. Merging snapshot with current memstore: If we see another flush start before the previous one is finished, we can block all writes and merge the current data to the snapshot and override the snapshot.

We think that approach (2) will be more easier to implement, but it is more costly because of the skip list merge.

Compaction events will also be picked from replayed WAL entries. Whenever a secondary sees a compaction entry, it will go ahead and apply the compaction. Compaction and flushes being written to WAL implies that there is no more need for a store file refresher as defined and implemented in region snapshots section.

Allow Flushes for the secondary regions

The approach here is to instead allow the secondaries to "spill-to-disk" (in this case it is a simple flush). That is, the regionserver considers secondaries like any other region for flush candidates. However, the spill is done to a configured place on the local disk since these files are really temporary and even if the machine dies or something, the data is not lost since the data is still there on the primary side (in storefiles and in WALs). In either of the approaches, new secondaries can be brought up to speed based on that data. The secondary serves reads from a merged view of primary's store files, secondary's memstore and secondary's spill file (which would be yet another storefile).
Periodically, or on certain events, the spills should be cleaned up. There are multiple approaches here, for example, periodic scan of the files and discard them based on the seqId persisted (just like regionserver handles cleanup of WAL files), or, when we see a flush-commit message from the primary (if we reintroduce the flush-commit message as discussed in the previous approach), we archive/discard some of those spilled-files (that’s a logical point since the secondary updates its view of the store files then). We feel that there won’t be that many spills accumulated if we follow a simple cleaning up process. This approach has associated writes to local disk, which is something that’s not needed in the previous approach. But it doesn’t have

any (or little, if any) state management and co-ordination to do with flushes (client throttling etc) on the primary side as discussed in the previous approach, and flows pretty well with the normal regionserver operations (very less special casing of memstore handling etc).

WAL changes

Actions regarding flush and compaction will be written to WAL. We may also want to write region opening and region closing as well.

As per above the WAL replication will plug in to the multi-DC replication interfaces and share the same ReplicationSource/ReplicationConsumer.

Failure handling (Primary and Secondaries)

In case of the RS hosting the primary region going down, the replication queue already handles failover, and ensures that recoveredLease() is called to see all the data. Though as already reported elsewhere, region moves and failures can cause concurrent or out-of-order delivery of wal edits to the receiving side. We will work on fixing this as well for general replication case and secondary regions case. Details are TBD.

In case of RS hosting a secondary region going down, the replication queue for this secondary cannot continue from where it is left because the edits are not persisted at the secondary through WAL (we skip WAL at replay). However, the secondary region will be opened elsewhere, and the store files opened will contain the latest seqId that is persisted. From this point on, replicated edits will start arriving, however the secondary will not start serving data until it sees a next flush commit from the primary. The reason is that, we do not want the secondary to go back in time due to lost WAL entries that is not persisted in its own WAL. Whenever a secondary starts serving, it will trigger a flush from the primary region which should write a new file, or if there is nothing to flush, mark an entry in the WAL. In this design, there will be a replication queue per table per region replication count.

In terms of the WAL file lifecycle, this schema will work with the offset based tracking in replication queue as well. If secondaries lose the edits because of a failover, they will continue to wait until the next sync point (flush) to start serving data.

Client side changes

For replicating wal entries to secondaries, the client has to send the WALEdit/WALEntry objects. Currently the client cannot perform a write to a secondary region. We will implement writes to secondaries but the difference would be that the write (log replay) will be scheduled for a specific replica id (rather than primary replica). There will not be multi-replica RPC requirements for this, only one RPC to a specific replica at a time.

Region Splits and Merges

Phase 1 requires to use the `DisabledRegionSplitPolicy`.

For Phase 2, we can implement region splits using either sync or async approach detailed below. The designs differ in whether the region replicas will be split/merged in sync with the split/merge of the primary region(s). Below only split path is detailed where merge is similar.

Sync design

While the split is executed, the region is closed for a short while, and the daughters reopened at the same region server. A meta multi-mutation decides on the fate of the split. If that is successful, it is only rolled forward. In the sync design, although the secondaries are not in the same region server, the master can ensure that the secondaries are closed before it ack's the split. The primary region server will also do the META changes required for secondary regions as well as a part of its own META edit. After split is committed, the master then assigns the secondaries for daughters.

Async design

In the async design, the primary region who is doing the split is unaware of the secondaries with no change in the execution except that the split still has to create META edits for the secondary replicas of daughter regions. After the split is done, the master will create a task to close the secondary parent regions and open the daughter regions. If the master dies during execution of this task, the meta will still contain info about the split (as in the case for RS failure after PONR). The master initialization should recreate this task.

We feel that sync design is more involved in master operations, and has a major drawback that during split, the client cannot do reads against secondary regions since they will also be offline as well creating unavailability.

The async design does not have this availability problem since the secondaries will be offlined after primary becomes online. The downside would be on the client-side meta caching layer. This implies that the client should be able to do requests against primary of the parent region, and secondaries of the daughter regions (or vice versa).

Performance

We will continue on the performance analysis, and ensure that there is no regression and the overhead of extra RPCs are worth the tradeoff for the use cases.

Warming Block caches of secondaries

This will be implemented as an optional feature iff testing proves that this is a requirement. In case of the primary region server handling a large percentile of the request without latency issues, the block caches of the secondaries will not be hot with the data. In case of a failover, then the secondaries will be getting a lot of request. With a cold cache, the latency spikes might be intolerable for the application.

For this we can optionally send a small percentage of the requests to the secondaries immediately to pre-warm the block caches and keep the blocks mostly cached. For these requests, the primary response will still be waited until the primaryTimeout has passed so that it will be transparent.

Other Changes

For LoadBalancer, we do not expect any more changes on top of Phase1.

For region assignment / META, the changes will only be related to region split / merges.

API-wise, the changes already there in the branch should be enough to cover the use cases. There might be small usability improvements.

Testing, stabilization etc work will continue as expected.

Promotion of secondaries will be detailed later. Since this is a MTTR improvement, and requires WAL replication, we can do this after Phase 2 work is complete.