High Availability Design for reads

12/02/2013

Hortonworks, Inc.

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

Issue : HBASE-10070
Doc version : 1.0

Background and Requirements

In the present HBase architecture, it is hard, probably impossible, to satisfy constraints like 99th percentile of the reads will be served under 10 ms. One of the major factors that affects this is the MTTR for regions. There are three phases in the MTTR process - detection, assignment, and recovery. Of these, the detection is usually the longest and is presently in the order of 20-30 seconds. During this time, the clients would not be able to read the region data.

However, some clients will be better served if regions will be available for reads during recovery for doing eventually consistent reads. This will help with satisfying low latency guarantees for some class of applications which can work with stale reads.

With distributed log split, the regions will not available for writes, however this high availability design for reads can be combined with distributed log replay, and write availability feature to provide a much better availability story.

Proposed design - Overview

For improving read availability, we propose a replicated read-only region serving design, also referred as secondary regions, or region shadows. Extending current model of a region being opened for reads and writes in a single region server, the region will be also opened for reading in region servers. The region server which hosts the region for reads and writes (as in current case) will be declared as PRIMARY, while 0 or more region servers might be hosting the region as SECONDARY. There may be more than one secondary (replica count > 2).

There won't be a distinction between region server processes. The region server might be hosting some regions as primary, while hosting other regions as secondary. A region server is said to be secondary for a region, or primary for another region.

The secondary region servers for the region will open the region in secondary mode. This means that the write requests will be denied, and read requests will be answered with whatever the data

is available to this regionserver, and with a flag indicating that the results are coming from possibly stale data.

The consistency guarantees for reading and writing from the primary will not change. The writes will NOT be propagated synchronously to the secondary region servers, but secondary region servers do a best effort to keep up to date.
Secondaries might be out of date with the primary node, but they will always be seeing a consistent (but older) snapshot of the region data.

The client can decide which replica it can go to execute the query. For writes, only the primary will be used. For each read request, a flag (get or scan flag) will be introduced to indicate whether reading stale data is desired. If flag is not set, the request can only go to the primary replica. Otherwise, client can execute any of the following or similar strategies for executing the query.

- Primary timeout : client will execute a read from the primary replica. After a short timeout, if rpc response is not back, parallel requests are sent to secondary replicas. Whichever rpc result (including from primary) response is returned to the client
- Parallel : client will execute read from all replicas (primary and secondaries) in parallel. Response from whoever responds first, will be returned to the application
- Parallel with delay: client will execute read from all replicas in parallel. Client will wait for some short time for the response from primary. If not, it will return whatever is available.

Tradeoffs

Having secondary regions hosted for read availability comes with some tradeoffs which should be carefully evaluated per use case. The main advantages of this design are

- High availability for read-only tables
- High availability for stale reads
- Ability to do low latency reads with <20ms 99.9% latencies for stale reads

The downsides for this feature are

- Double / Triple memstore usage (see region changes section)
- Increased block cache usage (see region changes section)

Proposed design - Details

In the high level flow, we identified the list of areas that needs to change as follows.

1. Configuration changes
2. LoadBalancer Changes
3. Region Assignment Changes
4. Region Changes
5. Client Changes

Configuration Changes

The number of secondary replicas will be configurable per table and kept in the table descriptor. All of the regions of the same table will have the same replication count. It will default to 1, meaning only 1 replica (primary) will be assigned. This will allow multiple tables having different replication levels to co-exist. A table's replication count might be changed with disable table + alter table + enable table.

LoadBalancer changes

LoadBalancer right now, handles most of the placement policy logic for regions. The proposal is to make the LoadBalancer aware of the secondary regions, and make the changes necessary so that all assignment decisions go through the LB (not for example from AM). As per today, LB will only create region assignment plans, but not carry out the actual assigning.

Assignment of replica copies for a region will be independent of each other. Each assignment of a region to a server (RegionPlan) will contain the *replica_id* (0 for primary, 1 for secondary, 2 for tertiary, etc). LB will create individual plans for all the replicas of the region, and AM will carry out the assignment. LB will also get a replica count per region (obtained from the table descriptor).

In assigning regions to regionservers, there will be some basic constraints that has to be honored (hard|soft). Region placement will take these hard or soft constraints:

- Same region server : Two region replicas cannot be hosted at the same RS (hard)
- Same host : Two region replicas cannot be hosted at the same host (hard)
- Rack : Similar to block assignment for hdfs, we may want to prevent assigning secondaries in the same rack as the primary (soft)

In some cases, above hard constraints might not be satisfied immediately. For this, we will add an in-memory map of under-replicated regions for which there is not enough region servers to choose from. If LB cannot assign secondary replicas due to above hard constraints, the replica will be added to this map. In case of new region servers joining the cluster, LB will be invoked over this map.

Region assignment changes

Assignment Manager changes

Region assignment mechanics will not change fundamentally, however, for each region, there will be multiple assignment states per replica. Each replica will be assigned individually and asynchronously from the other replicas. Conceptually, it will be similar to one more region for each replica. AM internal structures (like RegionState) will be amended to have replica_id to differentiate between different replicas. Similarly RegionStates maps will be keyed by <region_id, replica_id> instead of region_id.

RPC changes

OpenRegionRequest will have an extra field for replica_id. The regionserver will open the region in primary mode or secondary mode depending on whether replica_id == 0.

ZK Assign changes

The RIT zookeeper znodes need to be updated to handle the case of assignment of secondaries. The znode name today is the name of the region in regions-in-transition znode. For replica_id's > 0, we will append the replica_id as a suffix to the region name with a delimiter. The rest of the assignment mechanics (region states, etc), won't change other than handling the assignment per replica per region.

Meta Changes

When an RS opens the region (as primary or secondary) it will update hbase:meta for writing the location to be discovered from clients. Saving of the primary location will be unchanged.
When another replica opens the region, it will update the meta for the server_replica_id column in the same row. The client will discover the primary and secondary hosts the same way. Meta table's regions itself won't be replicated more than once in the first phase (need to change the ZK handling of META znode for that).

Server shutdown handler changes

Similar to current design, ServerShutdown handler will consult to meta and in-memory state to map out the regions, and replica_ids for those regions. For regions hosted as primaries, it will carry out the current flow. For secondary replicas, no log replay will be needed. Only assignment will be carried out.

Table operations

Table operations, like enable / disable table, alter table etc will not be much affected. However, disable table should ensure that all replicas are closed, and enable table has to ensure that all replicas are deployed, or they are in the under-replicated regions list.

Web interface changes

Region assignments for replicas will be displayed in the master and region server web interfaces.

Region changes

Region will be opened for serving in primary or secondary mode. In primary mode, no changes are necessary. Flush and compaction decisions will still happen locally without coordination with the secondaries.

For secondary mode, the region will have a flag indicating this case. Write requests will be rejected immediately. Read requests will be served from the data available.

The region will still use the block cache for serving the region in secondary mode. At least the index blocks will be kept in the block cache for better performance. For minimum disruption, we

will add additional configuration for whether or not caching data blocks served as secondaries. For some use cases, having a hot block cache is more desirable, versus, for others, not duplicating block cache usage might be desirable. For having a hot block cache, the client should send all or most of the queries to all replicas, so that the replicas can keep the blocks of the region cached.

For propagating edits to the secondaries, there are three designs considered: region snapshots, wal tailing, and async replication.

Region snapshots

In this design secondary regions will serve read requests from a snapshot of the region store files. In this approach, the secondary region will not get wal edits from any source, but can only see the data in the store files. This will be ideal for an initial implementation, but also for read only tables (for example from bulk loaded data).

As discussed above, primary will not sync the changes to the set of region files with the secondaries. Flushes, compactions and bulk loads can change the list of files. This means that the secondaries will need to discover the file changes, but until all secondaries are in sync, the old store files should not be deleted.

For discovering the list of store files, secondary regions will do periodic list files operations to the store, and update their view of the list of store files accordingly. The period will be configurable with a 1 minute default. The extra load on the namenode will be negligible even with thousands of regions.

Thanks to snapshots, store files can already be moved to the archive folder even when there are open readers. This allows the primary to move the files after compaction, even though secondaries might be reading from these. For ensuring that store files are not deleted while secondaries are serving, we will simply rely on TimeToLiveHFileCleaner, which ensures to keep the data files for some time. The ttl time will be configured to be a large multiple of the refresh files period (5-10). If the region server cannot refresh the files for more than ttl time, it will close the secondary region.

In this design, secondaries will not see the edits unless they are flushed to HFiles. Therefore, there is no need for the secondary regions to have memstores. The flush interval can be configured on a per table basis - this will limit how old the oldest edit can be in a memstore.

WAL tailing

A wal tailing design makes sense only in a wal-per-region world. We will not want to do group-based assignment for co-locating primary regions from one regionserver to another region server as secondaries, because the failure handling case would be much more complex. Without group-based assignment, for tailing the logs of the primary regions, every region server will have to tail logs of every other region server which is also not desired.

If wal-per-region is implemented (not covered in this document), tailing the logs of the region becomes trivial. If the logs per region is kept inside the region directory with the sequence numbers, each secondary region can open the log files and tail. In case of primary failover, new region server will close and roll the previous log, and continue with a new file. Secondaries will just do file listing to discover new logs (sorted by seqId) and continue tailing.

Whenever a region is opened as a secondary, the max seqId will be found from the store files. The RS will tail the logs after this seqId, and filter out edits before this. In this design, the region will have memstores which holds the recent edits. The memstore is never flushed. If the memstore is full, and cannot accept any more writes, the tailing thread pauses, and waits for memory to clear from other flushes.

In this design, flushes, compactions and bulk loads are also written to wal [2, 3] so that secondaries will apply those changes whenever they encounter them. Compactions and bulk loads are relatively easy to replay. For flushes, the secondary region has to efficiently drop all the entries from its memstore for the edits in the new store file. For doing this, we will replicate the memstore snapshots of the primary region. The flush will drop entries to the wal for flush init (with seqId), flush abort and success. Whenever the secondary sees init record, it will also create a memstore snapshot, and merge or drop the snapshot when it sees the flush abort or success edit.

Unlike previous case, flushes does not have to block log rolling. In case of a failure of the primary, the flush abort or success might not be written at all. Secondaries tailing the wal, can keep the memstore snapshot from the first flush init record. A subsequent flush init record will indicate that previous one failed, so they will merge the snapshot with the current, and create another snapshot.

In case the memory is full, and the edit cannot be applied, tailing cannot drop memstore entries, so it will simply block until more memory is available. Note that memstores for secondaries also cannot be flushed normally.

In a wal tailing design, special case be taken for log deletion. We can implement a HLogCleaner plugin, which will guarantee that the log will not be deleted until all active secondaries are finished with processing the log file.

Async WAL replication

This design will build on the region snapshots and current replication / log replay features. This does not necessitate a wal-per-region approach as with wal tailing. In this design, the primary will have a replication source for tailing it’s own log and a replication sink for each secondary replica that is alive. The edits will be replicated to the secondaries from the async replication thread. In that regard, this design resembles in-cluster replication, but also sharing the same data files for the region, instead of duplicating the data.

Similar to wal tailing, secondaries will have associated memstores, and will be replaying flushes and compactions and bulk loads.

Client side

Client side will be amended to be aware of the secondary replicas for the region. Write requests will go through the primary including checkAndPut and coprocessor endpoints. Read requests will go through the primary if the high-availability feature is not enabled, or table does have replication count == 1, or stale mode reads are not set for the read request. Otherwise, one of the strategies detailed in the high level design is employed to execute the request. We may want to add ways of cancelling requests.

We need to do the following changes for the client.

API Changes

We will need an API for each read request to indicate whether reading from stale data is okay. We will add a `Consistency` enumeration (similar to `Durability`) for the client to indicate the consistency requirements for the query. On Result objects, the client can inspect whether it is a stale result by calling `Result.isStaleData()`. There will not be any API support for getting how stale the data is (because we cannot know this) or which replica the results are coming from (except from primary replica).

We also want to make the execution strategy to be defined per table or per query.

Client-Meta interaction changes

Client discovering the locations from the meta will be similar, but have to be amended for locations for other replicas of the regions. As discussed previously, assignment for each region replica will be handled independently and asynchronously. Thus, the clients might discover locations for each region replica independently as well.

However, the client sometimes has to know about how many replicas there are for a region. For this, we will employ basic caching of the replica count per table. This is to make sure that the clients will discover locations for new replicas, when for example replication count is increased. When replication count is decreased with an alter table, some clients might still look for locations for non existing replicas until they invalidate their cache. A time-based expiry might be employed for invalidation.

For batch operations that affect only one region, the queries does not have to be changed, but sent to possibly multiple region servers with different locations. For batch operations for multiple regions like multi-get (even all primaries on the same region server), the secondary replicas might not be all co-located together. For this case, the queries will be broken up per replica per server and executed.

RPC changes

Client would convert a single RPC to multiple RPCs (per replica) which are independently retried. Each sub-RPC would have the replicaID associated with it so that it can discover the specific server associated with it.

Shell changes

Shell interface for getting the locations of the secondaries should be provided. The region level commands could be augmented to have an additional argument for the “replica” (e.g. move <region> <replica> …).

HBCK changes

In phase-1 we should make sure HBCK doesn’t fail when there is more than one replica for regions.

Development Phases

We propose to develop this feature in two phases for faster delivery and better managing the change requests.

After the first phase, most of the functionality for getting the benefits of high available reads will be demonstrated. This phase will include most of the changes to region assignment, region load balancing, and client. The region changes related to secondary mode will be implemented together with region snapshot approach for secondary region data. This means that, after this phase, secondary regions will be seeing flushed data, which can be combined with limiting the max unflushed time for edits for predictable staleness for reads.

Second phase will include the remaining work left over which is not crucial from the first phase, plus either wal tailing or wal replication approach for keeping the staleness to very small intervals.

Apache code development process

We propose to do this work in a working branch at the main HBase repository, or a github clone and publish the working set of patches constantly. For easier reviews and development, we will break up the patches into sub tasks (as detailed in this doc) which should be individually applicable.

At the end of phase 1, we expect to have an end-to-end working system for replicated tables in a stable and immediately usable form. We expect that we can demonstrate the benefits of the approach after phase 1. Only after this, we will propose a merge vote for merging the branch in the main hbase trunk.

Future Development

HDFS Changes

HBase does not guarantee any SLA's currently, so achieving 99% 20ms latency is currently best effort. No parts of the stack (hdfs, jvm, os) has SLA's.
However, for more predictable latencies, we can still improve on the observed read latencies, by doing parallel reads for HDFS blocks[4]. This will allow to circumvent the timeout when the datanode for the primary block replica is dead or unavailable and also smooth out the read latencies.

Promotion for MTTR

It is obvious that with a wal tailing or wal replication approach with secondary regions, having MTTR recovery times in the couple of seconds range becomes possible after detection. The idea is to promote one of the secondary regions to become the primary in case primary is declared dead. The recovery in this case will only involve, secondary doing lease and block recovery on the remaining wal files, and apply the remaining edits to it's memstore. Faster recovery implies faster availability for write workloads and strongly-consistent read workloads. Eventually, the usage of favored nodes for choosing write replicas will be considered.

Client changes for multi-DC

In certain cases, tables are replicated to other data-centers for the purposes of availability. In those cases, the capability of querying multiple data-centers would be useful. Here, it would be querying the same tables in two different data-centers for the rows (just like within a DC/cluster, the client would be querying multiple regionservers for the rows).

Deep dive into how this affects coprocessors

In the first phase we are not looking deeply at how the co-processors would be affected by this architecture (if at all).

Region Splits and Merges

In phase 1, DisabledRegionSplitPolicy should be used for the tables that needs to have region replicas.

References

[1] HBASE-2357 Coprocessors: Add read-only region replicas (slaves) for availability and fast region recovery

[2] HBASE-7329 Remove flush-related records from WAL and make locking more granular

[3] HBASE-2231 Compaction events should be written to HLog

[4] HBASE-7509 Enable RS to query a secondary datanode in parallel, if the primary takes too long