**Snapshots in HBase 0.96 – v0.1 (5/20/12)**

The initial design document and implementation for snapshots was proposed in HBASE-50. The original, overall design is still valid on the current HBase trunk, but the underlying implementation is no longer reasonable to apply to the current codebase. In the last two years, too much has changed in the HBase implementation in terms of lower level things, e.g. locking mechanisms, and higher-level design paradigms, e.g. managers and monitors.

The goal of HBase-6055 is to port up the concepts from HBase-50, but applying them to the current codebase to enable **table-level, point-in-time** snapshots.

It is currently possible to achieve timestamp-based backup of a table by increasing the retention time on a table to outlast the time it takes to run the copy (essentially creating a ‘snapshot’). However, this is not truly ‘point-in-time’ as different servers are not perfectly clock synchronized, leading to potentially drastically different table views even over the same clock-tick, given the high-throughput and low-latency nature of HBase (at over a million writes /second, things can change a lot over the common clock skew seen in NTP). Further, this leads ‘snapshot’ method has lots of latency because of its use of M/R and leads to a large amount of data replication.

Point-in-time snapshots can be thought out the creation of file references, in the copy-on-write paradigm for each region in a table run a cross-server transaction limited by a timeout. Whew, that’s a mouthful!

What follows is a more in depth discussion of each of how taking a snapshot actually works. We’ll start with the general theory and then move to how the actual implementation works.

**Point-in-time Snapshots – the hand-waving theory**

If you have read the documentation in HBASE-50, you can probably skim, or outright skip, this section and instead jump right into the nitty-gritty implementation details.

For those still around, lets jump right into it!

It traditional SQL databases, snapshots are enabled via a mechanism known as ‘copy-on-write’. Copy on write semantics can be simplified to the following: when multiple callers ask for a resource (e.g. a cell in table), everyone gets pointers to the same resource. However, if any of those callers makes a change to that resource, they get their own local copy to change, while the other viewers still see the original.

This ends up working really well for snapshots because the snapshot always keeps a reference to the old version, while the database just keeps moving on with the newer copies. Taking a snapshot on a single-machine database then amount to writing a single reference to keep your place. It starts to get a bit more complicated when the database spans multiple machines and is one of the major concerns with the HBase implementation.

Don’t worry, we’ll get to that, but for the moment lets look at why copy-on-write is actually a huge boon to HBase as well.

The astute reader will have realized by this point that HBase is actually built around the copy-on-write idea combined with a merge-sort to get the most recent version. So it turns out, most of the work for snapshots is built into HBase already!

HBase (more specificially HDFS) is built around the idea of immutable files, so a point-in-time view is always maintained as long as can we find the right HFiles (possible if we don’t throw them away – the point of HBASE-5547) and keep track of the state in the memstore.

Then for a snapshot, all we need to do is store a reference to all those the HFiles and WALs for a table, so we can go back to the right set of files to regenerate the table state. To minimize latency and maximize throughput, a natural parallelization of the work falls out of implementation of HBase - the servers hosting the files we need to reference are delegated the work of creating those references.

What about compactions you say? Won’t that blow away the files we just referenced you say? In current implementation of HBase, yes, this would be a huge problem for snapshots. However, in HBASE-5547 the implementation was added to enable ‘backup mode’, wherein HFiles are not deleted, but instead moved to an hfile archive directory. The beauty here is that the reference file doesn’t even need to be updated since if you can’t find the hfile in the original reference, you can do the archive directory and read the hfile there.

So, all we need to do for adding HFiles to a snapshot is just enumerate the files currently being read for a region and write references (inherently small files, which are fast to write) to those files. Simple, right?

“Woah, hold on,” you might be saying now, “what about the in-memory state in the memstore? That isn’t going to be in the HFiles!” You would be correct - if we just referenced the HFiles, we would be missing data.

The simple idea would be to first flush the memstore to an HFile and just add that HFile to the set we are archiving. However, that is going to be really painful and dramatically increase the latency to take a snapshot. Instead, we can capture in-memory state by just keeing track of the WAL files as well. As long as we mark where the snapshot starts into the WAL file for each regionserver and then add references to those WAL files to the snapshot, we can rebuild the in-memory state by replaying the WALs.

The tricky part comes in the *point-in-time* semantics. Take the case of two servers A and B. Suppose A is getting hammered with writes but B is lightly loaded. At some point, we decide to take a snapshot. When they receive the snapshot request, A is likely to have a bunch of outstanding writes while B is going to be able to process the snapshot almost immediately. To get a point in time, we need to wait for A to process all its outstanding writes and take its snapshot (write references to all the associated files and put a marker in its WAL) before we let *either A or B receive more writes*. We need to get A and B to agree to a single point in time where they can say, “Here is all the data involved in snapshot X.” This creates a coordination bottleneck, where we need to block writes to the table for the amount of time it takes to agree on a stable state across the cluster.

Essentially, the process is the following:

1. Tell all the servers hosting a table to get ready to take a snapshot
2. All the servers prepare for the snapshot
   1. Stop taking any new writes
   2. Create references to all their HFiles
   3. Put a write in the WAL
   4. Create references to all WAL files for the server
   5. Notify the master that they are ready to snapshot
3. Wait for all servers to agree that they are read
4. Tell all servers to take a snapshot (they all agree on the current state)
   1. Release the write lock, allowing the table to start taking writes again.

In the general case, we are not likely to see long latency in snapshots because of the rate at which we can process requests on a server and because (as discussed above) taking a snapshot is really a lightweight operation. Note that while this is going on, we can continue reading from the table normally, we just have to buffer the writes until the table becomes available again.

*Note that a “single point in time” cannot actually be achieved just using clocks synchronized by NTP. We would require* perfectly synchronized clocks *to ensure that at a single point, we can place a snapshot marker across all servers. NTP is going to be off by a limited amount (generally less that 1 second), but it not close enough for perfect synchronization. As such, we need to use the above locking mechanism (limiting availability) to get a consistent snapshot across multiple servers. See* Future Directions *for higher availability alternatives for taking snapshots.*

**Failure situations**

In the situation where all the servers are up and evenly loaded, it’s likely that snapshots will proceed fairly quickly, with minimal locking. However, server failures or GC pauses can potentially cause noticeable cluster performance. Let’s then talk about how a failure can cause performance issues and how there we can safeguard against breaking SLAs.

Suppose that, while taking a snapshot, one of the servers crashes while preparing a snapshot. All the other servers involved in the snapshot are going to wait for that server to join the snapshot. If wait is unbounded, the table will be unavailable for writes until the regions are reassigned to other regionservers and those servers can handle snapshotting those regions. For anyone with any sizeable write-rate, this is completely unacceptable wait time.

To avoid crippling a table in an error situation, we enforce a time limit on each server that it is willing to wait for a snapshot to complete. If it cannot complete the snapshot in the allotted time, the snapshost manager on the regionserver will:

1. Fail the snapshot locally
2. Allow writes to proceed on all its local region hosting the snapshotted table
3. Propagate that failure to all the other servers

If a server receives a snapshot failure, it will propagate that failure down to its own snapshot manager and quickly fail the snapshot and start accepting writes again (if they haven’t failed already).

This gives us a hard limit on the amount of time a table can possibly block writes. Further testing on the per-cluster is required to determine the latency effects of taking a snaphot and if it is within acceptable SLAs (see future directions a zero-latency snapshot option, if results are not within requirements). Depending on the write volumes, it could still cause a noticeable impact, though there should be no appreciable impact to reads.

There are two other concerns that only occur for exceptionally large clusters (+1000 node). In smaller clusters, we can count on more hardware stability and generally more pronounced off-peak times.

First, there can be significant network/propagation latency for servers to receive updates at to the snapshot progress. This can lead to timeout failures just due to the fact there are a lot of things going on with the network. This concern is mitigated by two things: we use ZooKeeper which scales updates well beyond the current scale of the largest HBase cluster (but we can easily switch to an RPC update/synchronization mechanism if necessary) and running snapshots at low-write periods (i.e. nightly), to minimize the latency when attempting to find a ‘stable’ point.

With increasing cluster sizes running on commodity hardware, there is increasing probability for hardware failures - on larger clusters, it is a much smaller issue to just loose one box than on a smaller cluster. In a well-designed and balanced table, there is a high probability that the table will have a region on a server that fails.

Consider attempting to take a snapshot and a regionserver fails. Then you need to wait for the snapshot timeout before it fails. Then you attempt to take a snapshot again after the table rebalances and again have a probability that another region server will fail. In a larger cluster, this probability will be larger, leading to decreasing likelihood that a snapshot will succeed.

In this case, we rely on the snapshot timeout to minimize effect of the failure on the rest of the cluster as well running snapshots at low traffic times to minimize the potential window/probability of failure. However, cascading failures are not very likely and are likely indicative of larger problems in the cluster that need to be mitigated before worrying about the nightly snapshot.

**Implementation Details**

Now that you believe that snapshots actually can be done with minimal latency and effect on the cluster, lets get into the details of how they are implemented.

The client initiates snapshots via the HBaseAdmin on a given table. Each snapshot is given a name that is then used to label the output directory – this name must be unique among all other names snapshots taken or in progress across the cluster.

*Currently we only support* taking *one snapshot at a time on the cluster. This is a simplification on the master-side implementation, but is actually not limited on the regionserver. Only slightly more work needs to be done to enable multiple snapshots at the same time. The other consideration here is* if *multiple snapshots should be running at the same time – generally the answer is going to be “no”.*

To capture all the HFiles, we enable ‘backup mode’ for the table we are snapshotting and must remain enabled for at least the time to compact all the files in the taken snapshot into the backup directory. Tn practice, this is going to be quite a while – all the HFiles in the snapshot need to be compacted and there is really no good (efficient) way to do the automated checking. A better solution is to distcp the snapshot, where you actually copy the referenced files and then stop the hfile backup for the snapshotted table since you ensure that you got all the files in the snapshot. In practice, if you going to be taking regular snapshots it will probably easiest to just keep backup mode on and then periodically cleanup the HFiles in the archive/backup directory.

The HBaseAdmin then requests a snapshot for the table on the master. At this point, all the work for a snapshot preceeds remotely and the client only waits for confirmation that the snapshot completed. Once receiving the “snapshot completed” notification, the snapshot has been completely persisted to the filesystem and can be accessed via an external source (its considered immutable and durable, within the bounds of HDFS).

On the HMaster, when it receives a snapshot request, it passes off the request to the SnapshotManager. At this point the HMaster is done until it needs to return success or failure to the client.

Snapshot running has two distinct phases: (1) prepare, (2) commit (basic two-phase commit). The prepare phase is managed via a barrier node in zookeeper. The appearance of the barrier means that all the involved servers are going to prepare their snapshot and when ready, join the barrier. Once all the servers involved in the snaphot (hosting the table), the SnapshotSentinel tells the servers to commit via creating a complete node (essentially releasing the barrier).

The SnapshotManager first figures out which servers should be involved in the snapshot, so later it can check to see if everyone interested is ready to start (this is stored in the SnapshotSentinel as the expected set of servers). The SnapshotManager then creates the ‘snapshot znode’ for the given snapshot to propagate the notification to all the regionservers via the MasterZKSnapshotController. This node contains the serialized version of the SnapshotDescriptor.

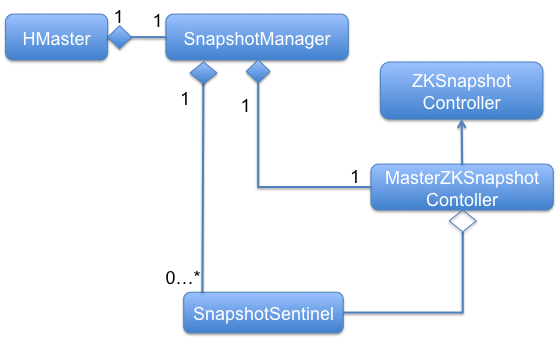
At this point, the layout of zookeeper is as follows:

/hbase/snapshot/start/

[snapshot-name] - snapshot desc/

/end/

/abort/

Where the name of the snapshot is the name of the znode and the data is the serialized SnapshotDescriptor. The class layout for snapshot related classes on the master is shown in Figure 1.

On each regionserver we have a RegionServerSnapshotHandler that has a RegionSnapshotZKController. This controller is listening for updates to /hbase/snapshot/start/and when an update occurs for the node, we consider it the start of the snapshot.

Figure - Snapshot classes used on the HMaster. This is almost exactly the class layout currently implemented. Incomplete and utility classes are not shown.

Each RS then reads the data from that node, which contains the table to snapshot.

*An optimization can be made here such that we store the snapshot descriptor under a znode with the table name, avoiding the stampede effect when a new snapshot is made, but this has postponed until it proves necessary.*

The RegionSnapshotZKController then updates the RegionServerSnapshotHandler (through the SnapshotListener interface) that it received the snapshot start indicator (SnapshotListener::startSnapshot()::boolean) and waits to see if it should update zookeeper that it is ready to snapshot (join the start barrier). If the server does not have any regions for the table, then we don’t need to update zookeeper that we are ready to snapshot.

*If we choose to pass the expected servers as part of the snapshot description, we could have all the servers listen all other servers to finish preparing their snapshot – joining the start barrier – and then proceed on their own, without waiting for master coordination. This is slightly more complex and has been tabled for the moment, but the zookeeper code is well encapsulated, so this change should not be too difficult.*

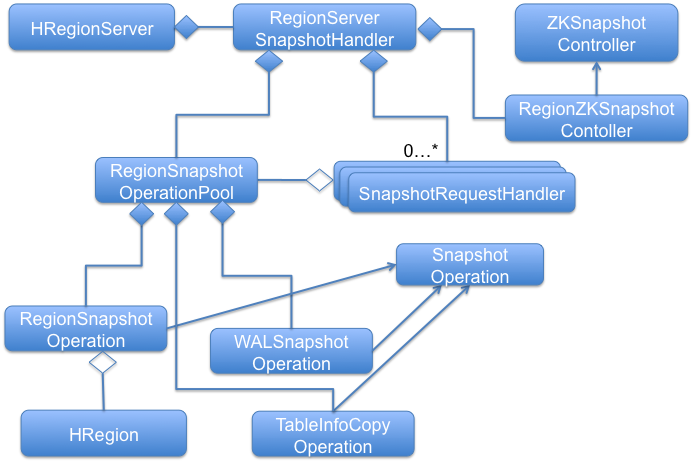
When receiving the notification to start the snapshot, the RegionServerSnapshotHandler then creates a SnapshotRequestHandler to handle the specific request. Included in this handler is the globalFailure indicator as well as the regions involved in the snapshot.

The SnapshotRequestHandler handles the meat of the request and has three main tasks:

1. Kick off a RegionSnapshotOperation for each region in the snapshot
   1. This calls HRegion::startSnapshot() internally and also monitors failure. We’ll get to what this does in a moment.
2. Start a WALSnapshotOperation, which internally will:
   1. Wait for the all associated regions to become stable (no more pending writes). This comes from the RegionSnapshotOperation.
   2. Write a ‘meta’ entry into the WAL for this snapshot
   3. Add a reference for all WAL files for the server currently present in the server’s .logs folder (.logs/[server name]). It is possible that not all WAL files are necessary, but this is cheaper than scanning the files to see which we need
3. Copy the .tableinfo into the .snapshot/[snapshot-name] directory
   1. This is a full copy of the .tableinfo, rather than just a reference creation, since the table info doesn’t inherently follow the copy-on-write semantics. However, this is a small file and shouldn’t be onerous to copy.

Figure 2 gives the general layout of the most of the regionserver specific classes discussed so far.

Figure - Snapshot operation classes to actually implement the parallelized snapshot functions, per regionserver.



All of these tasks are run asynchronously in a thread pool created just for snapshots, allowing us to parallelize the snapshot preparation. To monitor the status of the preparation, we get back a SnapshotStatusMonitor which internally has a SnapshotStatus (interface) for each of the operations that the SnapshotHandler started. Once all the statuses complete, we return true to the RegionSnapshotZKController, indicating that we are ready to start the snapshot. At this point, the zk structure looks like this:

/hbase/snapshot/start/

[snapshot-name] - snapshot desc/

/[regionservers done preparing snapshot]

...

/end/

/abort/

Remember how the WALSnapshotOperation waited on all the regions to become stable? Internally, the SnapshotRequestHandler actually waits on RegionSnapshotOperationStatus::regionsStable() to return true. At this point, the MVCC has been rolled forward on all the regions to a single point and we are blocking all writes to that region. Now we know the WAL has become stable, so we can put a snapshot edit in the WAL and create a reference to the WAL files. Note that we do not roll the WAL file, but just add a reference to the file. If we want to recover the snapshot and the WAL file is still ‘0 bytes’ in size, then we will need to roll the file on the server through the usual mechanisms. However, this rolling was considered too costly in terms of optimizing the latency of preparing a snapshot.

*The optimization here is that we can create references for all the current WAL files, then check once we reach stability that the WAL didn’t get rolled while we are waiting; this wasn’t implemented for simplicity at the moment, but is entirely feasible.*

The other key part of HRegion::startSnapshot() is creating a reference file for each of the HFiles that the region currently serves. Each region manages its own HFiles, so this was a natural factor of parallelization. We log the progress periodically for the number of HFiles collected, but we wait until all files have references added before considering the region ‘done’.

*In the current implementation we do reference counting for the included HFiles in .META. This ensures that we keep the HFiles around when doing cleanup. However, this code is likely going to be removed in favor of doing cleanup via checking the reference files in the snapshot directory, saving time to update .META. and allowing a snapshot preparation to complete faster. A similar method is currently used for the SnapshotLogCleaner, which will be discussed later.*

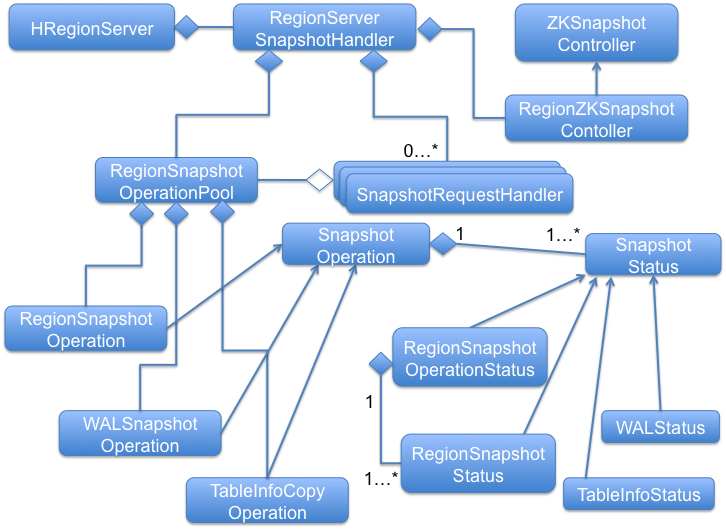
Keep in mind, while all this reference file creation is happening, we still have the write-lock on each of the involved regions. In fact, we cannot release the write lock until the snapshot is ready on *all regions on all servers for the table*. If we released the write lock, it would lead to an inconsistent point-in-time across the servers. Figure 3 gives a more detailed view at how all the different statuses and operations come together to run a snapshot.

Figure - Detailed view of the status and operation interaction while running a snapshot on the regionserver. Not pictured: error propagation between the snapshot controller and each of the statuses

Once the snapshot preparation has completed on all the regionservers, the master (more specifically the SnapshotSentinel) will notice that all the servers have ‘joined’ the start barrier for the snapshot. At this point, we release the snapshot barrier by creating the end barrier node for the snapshot. Specifically, the zookeeper structure now looks like:

/hbase/snapshot/start/

[snapshot-name] - snapshot desc/

/[regionservers done preparing snapshot]

...

/end/

[snapshot-name]/

/abort/

All the RegionZKSnapshotControllers are watching for children of the end barrier and then call RegionServerSnapshotHandler::finishSnapshot(snapshotName). If the server ran a snapshot with this name (note the requirement that all snapshot names are unique!), then it finishes the snapshot.

Internally, this just released the write lock and closes the region operation on each of each of the regions (all of this is handled by the RegionSnapshotOperation::finishSnapshot()). If we successfully complete the snapshot, then we add the server name to the end barrier and forget about the snapshot (similarly to how we notify the master that we are prepared to take the snapshot). ZooKeeper now looks like this:

/hbase/snapshot/start/

[snapshot-name] - snapshot desc/

/[regionservers done preparing snapshot]

...

/end/

[snapshot-name]/

/[regionservers that have committed]

...

/abort/

Once all the regions have successfully completed the snapshot, we consider the snapshot ‘done’ and the master can move it from the .snapshot/.tmp/[snapshot-name] directory to the .snapshot/[snapshot-name] directory and returns successfully to the client.

**Secondary Utilities**

There is slightly more to making sure that snapshots are consistent. As of right now, the only independent utility class (outside of the main ‘take a snapshot’ code) is the SnapshotLogCleaner.

HLogs are initially stored in the .logs/[server] directory under the server for which they logs edits. Eventually, as the HLogs are periodically rolled and the Memstore is flushed to HFiles the older HLogs are no longer necessary when recovering a region as these edits already exist in HFiles. Periodically, these old HFiles are archived to the .oldlogs directory, where they are periodically cleaned up (the default is after a certain amount of time the are deleted).

The LogCleaner periodically goes through the log files in the .oldlogs directory and asks each LogCleanerDelegate if it should delete the specified log file. The SnapshotLogCleaner keeps a cache of the current log files referenced in the snapshots on disk. If the log file is not found in the local cache, the SnapshotLogCleaner refreshes its cache by just rereading the file system state for the current snapshots and checking for the file name again.

A possible error situtation may occur if a WAL file that was referenced in the snapshot is moved to the .oldlogs directory and then cleaned up before the snapshot is completed. At this point the SnapshotLogCleaner would think that it is not referenced by any snapshot and consider the log valid to delete. First, this is highly unlikely because snapshot timeouts should be set fairly low (under 30 s) and log retention time fairly high (at least 6 hrs). However, even if these values are incorrectly set, this log is actually not storing any state that we care about and can safely ignore when rebuilding the table state from the snapshot (remember the log archiver only moves logs that hold edits that have all been written out to HFiles already). In the end, this is not an error, only a corner case for reestablishing table state from a snapshot.

A likely offshoot of HBASE-5547 is a mechanism similar to the logcleaner – a class that monitors the HFiles in the backup directory and periodically cleans them up. As such, it is trivial to implement a complementary cleanup tool for making sure that snapshot referenced HFiles are not deleted.

And that is basically it!

Well, not really, because in distributed systems we always worry about failures…

**Failure Situations**

Since snapshots are scaling across multiple servers and synchronize to a point in time are going to inherently be fragile. Further, as an initial implementation we allow this fragility as a trade-off for data consistency and implementation simplicity.

When each RegionServerSnapshotHandler gets a startSnapshot() notification it start a timer. If the timer exceeds the configurable snapshot timeout, the snapshot is considered ‘failed’; this mitigates against excessive impact on the cluster when taking a snapshot. If the snapshot doesn’t complete on the current server (both the prepare and complete phases), the snapshot is considered failed and the server locally releases its write lock. To ensure the snapshot fails globally, at the same time as failing locally, this error is propagated back to the other servers via writing this server’s name to zookeeper under the abort node.

For example, if server ‘region-server-A:1234’ timed out waiting for the complete notification (so it completed the prepare phase), zookeeper would look like:

/hbase/snapshot/start/

[snapshot-name] - snapshot desc/

/region-server-A:1234

...

/end/

/abort/

[snapshot-name]/

/region-server-A:1234

Since all the other regionservers, and the SnapshotSentinel on the master, are watching the abort znode for children, they will quickly see that abort notification and then propagate that failure back to their own snapshot attempts.

*Internally, we monitor this via the SnapshotFailureMonitor that keeps track of the globalFailureState, snapshotFailureState and the elapsed time. If any of these indicates breaks the monitor will throw an exception when SnapshotFailureMontior::checkFailure() is called. This will cause the snapshot to finish (releasing the region locks). If the error was cause internally, then it will propagate up the zookeeper and write the above discussed abort znode. If it came externally (another server failed the snapshot), then the snapshot is just aborted locally, again releasing the write locks. This up and down propagation and ensuring lock release has lead to some complexity around the progress and monitoring infrastructure – it will be refined later.*

Note that this abort notification does not need to be caused by a timeout; it can be caused by any internal failure in creating the snapshot – e.g. failure to write reference files or, in the current implementation, increasing the reference count for a file. Anything that causes a snapshot to not be complete is considered a failure for the snapshot, avoiding an incorrect view of the table state in the snapshot.

In addition to each snapshot having a failure indicator, we also have a global failure indicator that is inherently monitored for each snapshot running on each server. Global here is a slight misnomer – it’s global for the ‘world’ in which the snapshot operation is running: the region server. If the server dies/aborts, then we must inherently fail all the snapshots. This will cause some latency as we attempt to notify zookeeper to abort each of the snapshots, but if server cannot manage that before it dies (for instance a ‘kill -9’ on the process), then the snapshot will naturally timeout on the other servers.

As discussed in the theory section, this timeout mechanism also comes into play if we have excessive network latency or hardware failure. The cluster write throughput is only going to be impacted for a maximum of configurable snapshot timeout interval (less the JVM timer checking issues), even if we have multiple failures across the cluster.

**Conclusion**

At this point we have walked through the general mechanism for doing point-in-time snapshots in a distributed system. In short, a snapshot is a two-phase commit enabled across servers via a barrier node in zookeeper, where you:

1. freeze the world for a table for a *very short time* (within SLA boundaries),
2. create a bunch of soft-links for the files you care about
3. thaw the world
4. Any time files are deleted, you archive them instead of deleting them.

Only in full success situations do we inform the client that a snapshot was successful. We bound the error scenarios by using a configurable timeout that is monitored on each server. During the snapshot, any failures by involved servers – HRegionServers hosting regions of the snapshotted table or the HMaster – will cause a snapshot failure from the client’s perspective.

**Future directions**

The proposed snapshot implementation can also be used to do ‘soft’ snapshots - using a similar time-stamp based approach discussed in the introduction, but instead snapshotting ‘x seconds in the future’. This leads leading to minimal data copy and latency, as well as minimal blocking on each server, commensurate with a large batch of puts on regions. This enables offline data processing and backup with minimal impact on the running cluster.

Using the above approach for soft snapshots combined with a ‘timestamp oracle’ we can achieve **zero-latency, point-in-time** snapshots. Instead of using the common practice of just using timestamps assigned by the server for puts, you can use timestamps that are strictly increasing assigned from a “timestamp oracle” (see Percolator[[1]](#endnote--1)). At Google, they were able to create a timestamp oracle that generated 2 million timestamps a second, enough for most applications. A similar timestamp oracle can be created using HBase increment functionality in special counter table, with a timestamp row per-table.

1. “Large-scale Incremental Processing Using Distributed Transactions and Notifications”, Daniel Peng, Frank Dabek, *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010. http://research.google.com/pubs/pub36726.html [↑](#endnote-ref--1)