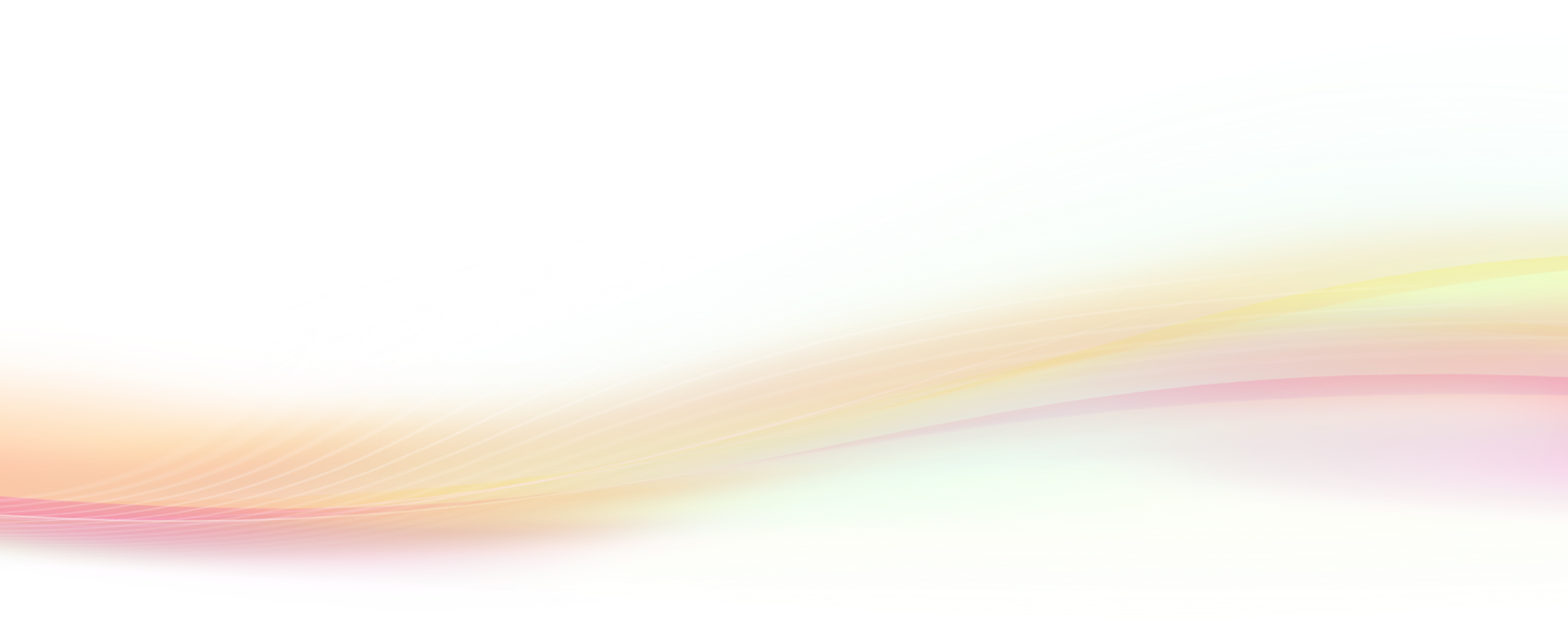
Offline Snapshots

Revision 0.1

Jesse Yates – 11/30/12



# Introduction

HBase is based around two main data storage locations, the MemStore in region and the immutable HFiles. Further, all mutations are time-stamped to provide a per-row ordering. Together, these two facts make it easily conceivable to coordinate a snapshot of the data stored in a given table, even though it is distributed across many servers. However, history has shown that this is not entirely a simple task within the HBase architecture, leading to many rewrites (but no committed code…yet) of snapshots. Further, because of the consistency guarantees in HBase – only on a single row – there are also multiple interpretations of what a ‘snapshot’ in HBase means. To avoid the complications of these interpretations, we will focus only on taking snapshots of an offline table and how restoring a table from a snapshot.

# History

Snapshots were originally proposed in HBASE-50, but the work there went unfinished until the discussion was revived in HBASE-6055. The sizeable amount of code from HBASE-50 was so far behind the current state of HBase that nothing could be preserved and we had to essentially start again from scratch. Originally, HBASE-6055 started as one mega JIRA consisting of taking and restoring offline and online snapshots. Given the widespread interest in the JIRA the work was then split into ‘taking a snapshot’ which remained as and ‘restoring a snapshot’, with HBASE-6055 as the overarching issue.

However, after implementation the tasks quickly needed to be far more granular. Thus, we have the current state of subtasks to HBASE-6055, offline snapshots (HBASE-6863), snapshot file cleaners (HBASE-6865), shell support (HBASE-6353), export snapshot (HBASE-6802), restore snapshot (HBASE-6230) and online snapshots (HBASE-7206, HBASE-7212, HBASE-6866, HBASE-6867), along with several other helper JIRAs.

Currently, full offline snapshots, restore and shell support has been completed for HBase 0.96. There are still a few rough edges to iron out, but most of these have follow-on JIRAs already filed (and in some cases patches) and they don’t affect the correctness in any way.

# Offline Snapshots

Offline snapshots have a very easy consistency model – the table is offline, so we just keep track of the current data and don’t need to worry about any new information. Further we attempt to do this with minimal data copy, leveraging HFile and HLog references and file archiving to manage a majority of the work. Snapshots are guaranteed to always complete in a configurable amount of time, ensuring minimal downtime for the table; this is more important for the online snapshot case, but is still nice to have for the offline case as well.

## Client side

All snapshots are initiated from the HBaseAdmin class and are serialized into a TakeSnapshotRequest. Most snapshot requests go through the synchronous interface, which leverages the async snapshot interface and extracts the max amount of time a snapshot can take from that response. The client then divides this max time by the number of retries to determine a frequency to check for the snapshot to complete. This allows us to check to see if the snapshot is done with a reasonable interval with an exponential ramp up; the snapshot is more likely to complete, rather than less as time progresses, so the standard exponential back-off mechanism doesn’t make as much sense. The restore methods (HBaseAdmin::restoreSnapshot() and ::cloneSnapshot()) also follow a similar algorithm.

## Server side

The current snapshot/restore mechanisms are all coordinated entirely on the primary HMaster. If the HMaster fails in the middle of the operation, there is no attempt to recover and continue on a backup HMaster (its conceivable, but a bit more complicated). When a snapshot/restore request is received on the master, we first check to another request with the same name has already been received or is in process. After that, the operation is ‘registered’ with the SnapshotManager and run in a background thread.

## Taking a snapshot

Currently, we only support taking a single snapshot at a time. This is a simpler first pass, though there has been discussion to enable multiple, concurrent snapshots (and the design is such that is *should* be easy to enable multiple at the same time). The SnapshotManager acts as the single synchronization point for determining if snapshot can be run. To prepare a snapshot its ensures that

* no existing snapshot with the same snapshot name exists
* no other snapshot is also running
* that we aren’t restoring the table at the same time as taking a snapshot of it

After that, as part of the preparation for taking the snapshot, the SnapshotManager also

* removes any old working directory for snapshots of the same name
  + they must have been failures, so we can delete them
* creates a new working directory for the snapshot
  + /hbase/.snapshot/.tmp/[snapshotname]

Finally, after verifying that the snapshot can be taken, it passes the snapshot request onto a DisabledTableSnapshotHandler – an EventHandler- and submits that to the HMaster’s execution pool. If we get an exception here, the SnapshotManager is also responsible for cleaning up the working directory.

The DisabledTableSnapshot handler has a couple primary roles:

1. Timing the snapshot
2. Capturing all relevant information
3. Verfying the snapshot is correct
4. Cleaning up the working directory, if there is a failure

All snapshots are required to run in a configurable amount of time (hbase.snapshot.disabled.master.timeout), after which they are required to fail. This ensures that the timing SLAs can be met on both the client (see above) and the on the server.

The actual ‘snapshotting’ is broken in a series of subtasks that the handler executes (currently this is a serial process, but it can be easily parallelized in the future):

1. Reference all the HFiles
2. Copy the current table info
3. Copy all the all the region infos
4. Copy the pending recovered edits

These are all tasks that need to be executed in the distributed and centralized snaphot use cases, so they are broken into their own classes and can be executed in an embarrassingly parallel manner.

**NOTE 1:** we don’t actually copy the HFiles we just note down their names. This minimizes the amount of data copied, but still preserves correct semantics. We can continue to read from HFiles on a running table (even if they are compacted away on the original table) because we are using FileLinks, which are location agnostic. This will be covered more closely in the ‘Snapshot Restore’ section.

**NOTE 2:** It is not necessary to also capture the WALs in the snapshot because when a table goes offline, it flushes all the memstores to disk. It is possible the table went down uncleanly in the past and therefore has some recovered edits from log splitting, so we need to capture those. Therefore, all we need to keep track of are (1) the HFiles and (2) the recovered.edits, to capture all the data in the table.

Once all the tasks have completed, the handler then verifies the snapshot. The goal is a best-effort check of the contents – that the .XXXinfo files are readable, that all the referenced HFiles and WALs exist, etc. If we attempted to do much more verification, we will essentially just be taking the snapshot over again. The verification step is included in the snapshot timing – a snapshot is not considered ‘done’ until it has all the necessary files and verified.

## Snapshot FileSystem Layout

The snapshot of a table is designed to look very similar to the original table to help ease restoration from the snapshot and to make it easier to understand when admins are browsing the filesystem. All completed snapshots are stored under the /hbase/.snapshot/ directory, each under its own directory named for the name of the snapshot (so, for instance a snapshot named ‘Tuesday\_at 5\_PM’ would live under the /hbase/.snapshot/ Tuesday\_at 5\_PM directory). In progress snapshots are again stored under their name, but under the /hbase/.snapshot/.tmp directory (so the previous example would be run under /hbase/.snapshot/ .tmp/Tuesday\_at 5\_PM) and then moved out of the temporary directory when it is finished.

Lets suppose we have an HBase installation with a table named ‘test’. After we put some data into the table, the filesystem will look something like:

|--ROOT-/

|----.tableinfo.0000000001

|----.tmp/

|----70236052/

|-------.oldlogs/

|----------hlog.1354232849848

|-------.regioninfo

|-------info/

|----------d9795819c1544c9d9a45854539aa1b43

|-.META./

|----.tableinfo.0000000001

|----.tmp/

|----1028785192/

|-------.oldlogs/|----------hlog.1354232850017

|-------.regioninfo

|-------info/

|-.logs/

|----192.168.1.133,55844,1354232849686/

|-------192.168.1.133%2C55844%2C1354232849686.1354232860496

|----192.168.1.133,55845,1354232849719/

|-------192.168.1.133%2C55845%2C1354232849719.1354232860496

|-.oldlogs/

|-hbase.id

|-hbase.version

|-test/

|----.tableinfo.0000000001

|----.tmp/

|----750f1d346fad2924318fa84d6d67dd01/

|-------.regioninfo

|-------.tmp/

|-------fam/

|----------01fffce27f2e47f2af9925c98e28cc13

|----------520b568dc91b4d93a9dd69fa57983b5a

|----------5845710958ab402684a06b3c249e4ff8

|----------62c78e8eb6024d9f823ce75501b7ec3f

|----------63b48fd501a345ebbeb8cd7dddd2c05f

If we then offline table ‘test’ and take a snapshot named ‘offlineTableSnapshot’, we will find there is an new directory containing the snapshot, looking like:

|-.snapshot/

|----.tmp/

|----offlineTableSnapshot/

|-------.snapshotinfo

|-------.tableinfo.0000000001

|-------.tmp/

|-------750f1d346fad2924318fa84d6d67dd01/

|----------.regioninfo

|----------.tmp/

|----------fam/

|-------------01fffce27f2e47f2af9925c98e28cc13

|-------------520b568dc91b4d93a9dd69fa57983b5a

|-------------5845710958ab402684a06b3c249e4ff8

|-------------62c78e8eb6024d9f823ce75501b7ec3f

|-------------63b48fd501a345ebbeb8cd7dddd2c05f

Under the under completed snapshot we have a layout that looks strikingly similar to our original table. The only differences are:

1. Addition of .snapshotinfo
   1. a serialized version of the SnapshotDescription that, well describes the snapshot – name, table, creation time, type
2. All the files under 750f1d346fad2924318fa84d6d67dd01/fam are empty files
   1. All HFiles have a unique name and are still stored under the same region directory, so we can easily and uniquely find the HFile under its original table or in the .archive directory

## Namenode Load Concerns

The process outline above creates a rather large number files, which has caused come concern. At 5-10 HFiles per region, and possibly thousands of regions we can easily hit the low millions of files created per snapshot. This is a non-trivial amount of load on the NameNode and on its own is a fairly lengthy process. For this first cut implementation, this was an acceptable tradeoff; tables are generally disabled during non-peak hours, which means the NameNode will be light loaded during that time (also, its just easier to thing about this way).

There are plans to move the “one file= one reference file” mechanism to a more succinct ‘snapshot manifest’ that encapsulates all the necessary HFiles and HLogs for the given snapshot in a single file. This will dramatically cut down on the load to the NameNode, though it does present some challenges for the distributed snapshot use-case. A follow on JIRA is planned, though not yet filed.

## Cleanup Concerns

There exists the possibility of superfluous snapshots remaining in the working snapshot directory (/hbase/.snapshot/.tmp) as previous HMasters may have been hard-stopped without allowing proper cleanup (i.e. kill -9). We consider this a rare case and therefore do no provide for the clean up, though HBASE-7240 has been filed to address this issue.

# Restoring From a Snapshot

There are three things you can do once you have a snapshot:

1. Clone - make a new table with the snapshot
2. Restore - restore an existing table to an older snapshot
3. Export - copy the snapshot

On the client, all the restore operations follow a similar pattern to the take a snapshot interface: they call an async version of the method and then periodically (exponential backoff to a max) check to see if the operation has completed.

## Cloning a Snapshot

Before we even start creating a new table, the HMaster checks to see if the snapshot exists and that the table we are attempting to create does not exist. After that we ask the SnapshotManager to clone the snapshot.

The SnapshotManager begins by ensuring that we aren’t already restoring or snapshotting the table we are attempting to create (it’s a corner case, but it could happen; generally, you are a bad admin if this is ever an issue). Once we pass all these checks we start a CloneSnapshotHandler – a new CreateTableHandler.

The CloneSnapshotHandler acts just a like a CreateTableHandler, except:

1. HTableDescriptor is loaded from the snapshot directory
2. It creates regions from the snapshot
   1. Regioninfos are loaded to describe the region
   2. HFileLinks are used to populate the expected families
3. Link the WALs
   1. Offline snapshots don’t need do this step – there are no WALs to copy
4. Copy over the recovered.edits

**NOTE:** Because we have full control over the region boundaries, the linked HFiles fit perfectly into the new table without any compactions.

Once all necessary files have been copied or linked, we can finish up creating the table just like any other table.

## Restore a Table

Here we have a table that we want to rollback to a previous snapshot. First, the table has to be *already offline* before starting the restore. Then on the client, a snapshot of the existing table is made to ensure we can rollback to it, if necessary (i.e. the restore fails) or if the user desires it.

Once we have the snapshot of the table, we then pass the restore request onto the HMaster and SnapshotManager, which go through the same checks as the clone use case.

In fact, the RestoreSnapshotHandler looks almost exactly the same as the CloneSnapshotHandler except it modifies the existing table, rather than creating a new one. In addition to all the work the CloneSnapshotHandler does above, the RestoreHandler also attempts to reuse the existing state of the table. For instance, if some of the snapshot regions still exist, then we leave them there. Or if some of the families are the same, then we just keep them on the filesystem as-is. On the flip side, we may also need to remove some of the existing regions (using the HFileArchiver, so we don’t actually *delete* the HFiles), since the snapshot didn’t have them.

In the end, we have a table that has the same layout as the table when that snapshot was taken. Some of the HFiles may actually be HFileLinks rather than real HFiles, but otherwise it, for all intents and purposes, is a fully functioning read/write table. And keep in mind, you can always restore back again to the original table state!

## FileSystem Space Considerations

When cloning a snapshot (and to a lesser degree restoring) the filesystem only grows minimally at first. You have to copy over a handful of files, but all the HFiles are empty files. This means that initially, the table has a very small footprint. This is until you write some data to the table and cause a compaction. During the compaction the original HFiles are scanned and rewritten into a new single HFile.

Lets go back to the example above, where we took a snapshot of a table. Suppose after taking a snapshot of the table that we then cloned it to another table named ‘test\_2’. This table will look exactly the same as the original table – test – except for the HFileLinks. Let suppose that ‘test’ has 10GB of data in the family ‘fam’. The overall filesystem is probably around 10.5 GB after the snapshot has been restored (probably an over-exaggeration, even including -ROOT- and .META.). Lets say we write 2 GB of data to ‘test\_2’ and cause a compaction of ‘fam’. All of sudden the size of the filesystem jumps from 12.5 (10.5 originally + the 2 GB we just wrote) to 22.5 GB. In reality we were only getting a temporary discount on space – if the table had been a real table, this jump wouldn’t even have been noticed.

The interesting aspect here is that we get essentially zero-cost read-only tables. There is a little overhead in the extra files to copy over (.tableinfo, .regioninfos), but otherwise the only cost is network use to the Data Nodes hosting those HFiles.