HBase Snapshot
Detailed Design

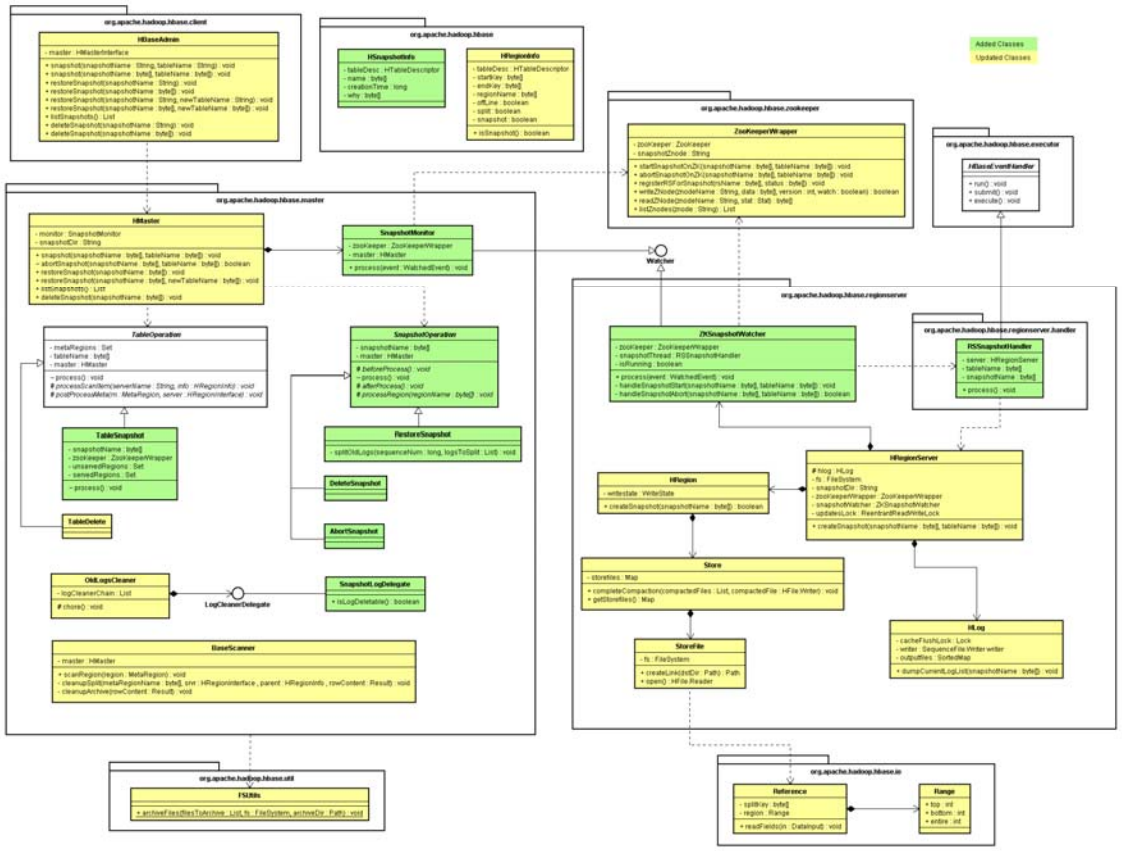
Li Chongxin

1 Introduction

Snapshot of HBase[1] takes a copy of the specified table at a particular point in time. A design document has been provided on how snapshot should be performed and existing functions should be modified to support snapshots. In this document, we will describe the implementation details for snapshot, including all the classes and methods that are to be created and updated based on the design in previous documentation. To better understand this document, we strongly recommend you to go over the design document[2] first.

2 Class Diagram

Following class diagram illustrates most of the major changes that are to be made to existing system. (Click to view)



[1] https://issues.apache.org/jira/browse/HBASE-50

[2] https://issues.apache.org/jira/secure/attachment/12446965/HBase+Snapshot+Design+Report+V3.pdf

In the above diagram, the green ones are classes to be created, while the yellow ones are existing classes which need to be modified to support snapshot. Because of space limit, for each class in the diagram, only the methods and attributes which would be affected and involved in snapshot are displayed. Although class/method signatures might vary a little during the programming, the whole implementation plan won't change a lot.

3 Implementation Description

In the following sections, snapshot implementation details are described according to the design document. Section 3.1, 3.2, 3.3 each corresponds with chapter 5 (Snapshot Creation), chapter 6 (Snapshot Maintenance) and chapter 7 (Snapshot Operations) of the design document. Chapter 4 (Message Passing) of design document is not described in a separate section but is touched on in all three sections.

3.1 Snapshot Creation

This section describes how a snapshot would be created from the client request.

- **HBaseAdmin** [updated]

Class Description:

On client side, all the snapshot operations (create, list, delete, restore) start in this class.

Methods:

- [added] `public void snapshot(String snapshotName, String tableName)`
  Description: Create a snapshot <snapshotName> for table <tableName>. This method will just convert the parameters type and then call below method.

- [added] `public void snapshot(byte[] snapshotName, byte[] tableName)`
  Description: This method will call `HMaster.snapshot` to perform the operation.

- **HMaster** [updated]

Class Description:

To keep client thin, master will orchestrate the whole snapshot process (as Todd suggested). Snapshot is both started and aborted in master.

Attributes:

`SnapshotMonitor monitor`: `Watcher` that monitors the snapshot progress via ZooKeeper. Register this watcher as a listener for the `ZooKeeperWrapper` instance associated with `HMaster`.

Methods:

- [added] `public void snapshot(byte[] snapshotName, byte[] tableName)`
  Description: Create `TableSnapshot` instance and then process. Snapshot will be started in `TableSnapshot` just as other `TableOperations`.

- [added] `boolean abortSnapshot(byte[] snapshotName, byte[] tableName)`
  Description: Abort a snapshot which is in progress. An `AbortSnapshot` (which is a subclass of `SnapshotOperation`) instance is created to abort the snapshot
  A snapshot should be aborted under several failure scenarios, such as region server breaking down (other failure scenarios are still under analysis). `SnapshotMonitor` detects the failure via ZooKeeper and then call this method. This method is not public so that snapshot could not be aborted by the client manually.

- **ZooKeeperWrapper** [updated]

Class Description:

All the messages between master and region server are passed via ZooKeeper. This is a helper class for ZK.

Attributes:

`String snapshotZnode`: ZNode for snapshot, which is usually /hbase/snapshot. This is the parent node of all intermediate znodes for snapshot.

Methods:

- [added] `public void startSnapshotOnZK(byte[] snapshotName, byte[] tableName)`

  Description: Set data for znode "snapshotZnode" to notify all the region servers to start the snapshot on each RS. Call `ZooKeeperWrapper.writeZNode` to set the znode data.

- [added] `public void abortSnapshotOnZK(byte[] snapshotName, byte[] tableName)`

  Description: Set data for znode "snapshotZnode" to notify all the region servers to abort snapshot on each RS. Call `ZooKeeperWrapper.writeZNode` to set the znode data.

- [added] `public void registerRSForSnapshot(byte[] rsName, byte[] status)`

  Description: Register RS for a specified status. A znode named by this RS will be created under the status znode. There are two statuses for RS during snapshot: *ready* and *finish*.

- **TableSnapshot** [created]

Class Description:

Subclass of `TableOperation`. This class performs the snapshot on master side.

Attributes:

`TreeSet<HRegionInfo> unservedRegions`: regions that are offline

`TreeSet<HRegionInfo> servedRegions`: regions that are online

Methods:

- `void process()`

  Description: Override `process()` method in super class. This method works in three steps:

  a) In the first step, metadata will be scanned as what it does right now.

  b) After all the regions are scanned, this method will decide whether to start the snapshot across the cluster or just by the master.
  If all the regions are online, call `ZooKeeperWrapper.startSnapshotOnZK` to start the snapshot via ZK and register the master as the watcher so that `SnapshotMonitor` will start monitor the snapshot progress.
  If all the regions are offline, we can create the snapshot here on the master side. If the table is partial open, an exception will be thrown.

  c) At last, call `dumpSnapshotInfo` method to dump the snapshot meta information to file system.

- `public void processScanItem(String serverName, HRegionInfo info)`

  Description: If a region is online, put it in `servedRegions`. Otherwise, put it in `unservedRegions`.

- `public void postProcessMeta(MetaRegion m, HRegionInterface server)`

  Description: There is nothing to do for this method.

- `private void dumpSnapshotInfo()`

  Description: Dump the snapshot meta information into a file ".snapshotinfo" under snapshot dir.

- **SnapshotMonitor** [created]

Class Description:

This class implements the `Watcher` interface of ZooKeeper and monitors snapshot znode and all its children nodes.

Methods:

- `public void process(WatchedEvent event)`

  Description: Process all kinds of events for snapshot.

  a. RS register under *start* status
  b. RS register under *finish* status: record RS name. When all RS have finished, snapshot is done.
  c. RS znode under *start* status disappear: Because znodes under start status are ephemeral, this event indicates the corresponding RS is down. Snapshot should be aborted by calling method `HMaster.abortSnapshot`. It will abort the snapshot and do the clean up work.

- **ZKSnapshotWatcher** [created]

Class Description:

This class also implements the `Watcher` interface of ZooKeeper. Different from `SnapshotMonitor` on the master side, this class works on each region server.

Attributes:

`RSSnapshotHandler snapshotThread`: the thread which performs snapshot on RS.

`boolean isRunning`: indicate whether the snapshot is running on this RS.

Methods:

- `public void process(WatchedEvent event)`

  Description: This method processes two kinds of events for snapshot.

  a. Start is set on snapshot znode: call method `handleSnapshotStart`
  b. Abort is set on snapshot znode: call method `handleSnapshotAbort`

- `private void handleSnapshotStart(byte[] snapshotName, byte[] tableName)`

  Description: handle snapshot start event. To start a snapshot on region server, an `RSSnapshotHandler` is created and submitted.

- `private void handleSnapshotAbort(byte[] snapshotName, byte[] tableName)`

Description: handle snapshot abort event. If a snapshot is still running on this region server, just interrupt the snapshot thread. The master will do the clean up work in .META. and HDFS.

- **RSSnapshotHandler** [created]

Class Description:

This class is a subclass of `HBaseEventHandler` and is executed on region server (However, `HBaseEventHandler` is not implemented yet). This handler can be executed either synchronously or asynchronously in a thread pool. This is the thread where snapshot is performed on region server.

Attributes:

`HRegionServer server`: associated region server.

Methods:

- `public void process()`

Description: This method will call `HRegionServer.createSnapshot` to create the snapshot. If any exception is caught here (which indicates the snapshot fails on this region server), remove the corresponding RS znode under the start status. This will notify the `SnapshotMonitor` to abort the snapshot. Otherwise, create a RS znode under finish status to report snapshot finished on this RS.

- **HRegionServer** [updated]

Attributes:

`ZKSnapshotWatcher snapshotWatcher`: ZK watcher for snapshot.

Methods:

- [added] `public void createSnapshot(byte[] snapshotName, byte[] tableName)`

Description: This is main method to perform snapshot on RS. This method works in two steps:

a) Call `HLog.dumpCurrentLogList` to dump a file that keeps the log sequence number as well as a list of log files that are currently used.

b) For each region of this RS, if it belongs to the snapshot table. Call `HRegion.createSnapshot` to perform snapshot for this region.

- **HLog** [updated]

Attributes:

`Lock cacheFlushLock`: lock for log rolling and cache flushing

`SequenceFile.Writer writer`: current log file

`SortedMap<Long, Path> outputfiles`: map of log files but the current one

Methods:

- [added] `public void dumpCurrentLogList(byte[] snapshotName)`

Description: Dump a file that contains the log sequence number and current log files list. These log information will be used to reconstruct the memstore content when restore. Lock should be obtained on "`cacheFlushLock`" so that log rolling and cache flushing is suspended. This won't lock the system for a long time because dumping the log file list is pretty quick.

- **HRegion** [updated]

Class Description:

Region data (including metadata and `HFiles`) are backed up in this class

Attributes:

`WriteState writestate:` object that keeps the current status of the region

Methods:

- [added] `public boolean createSnapshot(byte[] snapshotName)`

  Description: This method works in four steps:

  a) First obtain lock on `writestate` and check the region status. If a flush or compaction is going, return false because region is writing `HFiles` right now. Otherwise, set `writestate.writesEnabled` to false to suspend compact and flush during the process of snapshot.

  b) Dump the metadata of this region from .META. into file ".regioninfo" under corresponding snapshot directory.

  c) For each `StoreFile` of this region, create a reference file for this `StoreFile` under corresponding snapshot directory. And then Increase the reference count by one for each file of this region. See Appendix A.

  d) Set `writestate.writesEnabled` back to true to enable compact and flush.

- **StoreFile** [updated]

Methods:

- [added] `public Path createLink(Path dstDir)`

  Description: write out a reference file for this `StoreFile`.

- [updated] `protected HFile.Reader open()`

  Description: If a `StoreFile` is a half reference file after split, open it with a `HalfHFileReader`. If it is an `entire` reference file, open it with an ordinary reader.

- **Reference** [updated]

Class Description:

Utility class for file system operation

Attributes:

`Range region:` add a new reference type "`entire`", which indicate this is a reference to an entire file.

3.2 Snapshot Maintenance

This section describes how existing functions would be modified to support snapshot.

- **Store** [updated]

Methods:

- [updated] `public void completeCompaction(List compactedFiles, HFile.Writer compactedFile)`

  Description: Instead of deleting the old store files, call `FSUtils.archiveFiles` to archive them into the archive directory.

- **TableDelete** [updated]

Class Description:

To delete a table, old files should be archived instead of deleting directly.

Methods:

- [updated] `public void postProcessMeta(MetaRegion m, HRegionInterface server)`

  Description: This method works in two steps:

  a) Remove the region from .META.

  b) Check reference count information for this region in .META. Call method `FSUtils.archiveFiles` to archive the files whose reference count is not zero.

- **OldLogsCleaner** [updated]

Class Description:

Take snapshot into consideration in Log cleaner

Attributes:

`List<LogsCleanerDelegate> logCleanerChain`: these log cleaners will check whether to delete a log in chain.

- **SnapshotLogDelegate** [created]

Class Description:

This class implements the `LogsCleanerDelegate` interface.

Methods:

- `public boolean isLogDeletable()`

  Description: Check the ".oldlogs" file for each snapshot under the snapshot dir. If the log is in the log list, then this log is not deletable. If the specified log is not in any of the log list, then this log is deletable.

- **BaseScanner** [updated]

Class Description:

This class scans region metadata in .META. table. Because reference count information is also saved in META, these snapshot metadata should also be taken into account when scanning.

Methods:

- [updated] `protected void scanRegion(MetaRegion region)`

  Description: There are two types of metadata for a region right now. If the row key is prefixed with ".SNAPSHOT.", it is snapshot metadata and call `cleanupArchive` to clean up the archived files if necessary. Otherwise, it is ordinary metadata and should be processed as before.

- [updated] `private void cleanupSplit(byte [] metaRegionName, HRegionInterface srvr, HRegionInfo parent, Result rowContent)`

  Description: When a split parent region is not used any more, check the reference count information for this region. If the reference count information for this region is empty, this region can be deleted from HDFS. Otherwise, call `FSUtils.archiveFiles` to archive the region files.

- [added] `private void cleanupArchive(Result rowContent)`

  Description: Check the reference count information of this region, if the reference count for a file is zero, this file can be deleted from the archived directory.

- **HRegionInfo** [updated]

Attributes:

`boolean snapshot:` indicate whether this region is used in snapshots

- **FSUtils** [updated]

Class Description:

Utility class for file system operation

Methods:

- [added] `public static void archiveFiles(List filesToArchive, FileSystem fs, Path archiveDir)`

  Description: Move all the files to the archive directory.

3.3 Snapshot Operations

This section describes how snapshot operations would be performed.

- **HBaseAdmin** [updated]

Class Description:

On the client side, Snapshot operations start here.

Methods:

- [added] `public void restoreSnapshot(String snapshotName)`

  Description: restore the given snapshot into a table with the original table name. This method just converts the parameter type and calls below method.

- [added] `public void restoreSnapshot(byte[] snapshotName)`

  Description: restore the given snapshot into a table with the original table name. Call `HMaster.restoreSnapshot`.

- [added] `public void restoreSnapshot(String snapshotName, String newTableName)`

  Description: restore the given snapshot into a table with a new name. This method just converts the parameter type and calls below method. (Low Priority)

- [added] `public void restoreSnapshot(byte[] snapshotName, byte[] newTableName)`

  Description: restore the given snapshot into a table with a new name. Call `HMaster.restoreSnapshot`. (Low Priority)

- [added] `public List listSnapshots()`

  Description: List all the snapshots in the system. Call `HMaster.listSnapshots`

- [added] `public void deleteSnapshot(String snapshotName)`

  Description: Delete a specified snapshot. This method just converts the parameter type and calls below method.

- [added] `public void deleteSnapshot(byte[] snapshotName)`

  Description: Delete a specified snapshot. Call `HMaster.deleteSnapshot`.

- **HMaster** [updated]

Class Description:

Because snapshots are not active, all the snapshot related operations can be finished on master side.

Methods:

- [added] `public void restoreSnapshot(byte[] snapshotName)`

  Description: Just like table operations, a `RestoreSnapshot` instance is created and processed to perform the operation.

- [added] `public void restoreSnapshot(byte[] snapshotName, byte[] newTableName)`

  Description: TBD (Low Priority)

- [added] `public List<HSnapshotInfo> listSnapshots()`

  Description: Get ".snapshotinfo" under each snapshot dir. (`HSnapshotInfo` should extend `VersionedWritable`, right?)

- [added] `public void deleteSnapshot(byte[] snapshotName)`

  Description: A `DeleteSnapshot` instance is created and processed to perform the operation.

- **SnapshotOperation** [created]

Class Description:

This is the abstract class for operations of a single snapshot.

Attributes:

`String snapshotName:`

Methods:

- [added] `void process()`

  Description: For each region of this snapshot, call `processRegion`

- [added] `protected abstract void beforeProcess()`

  Description: work to do before processing regions.

- [added] `protected abstract void processRegion(byte[] regionName)`

  Description: work to do for each region.

- [added] `protected abstract void afterProcess()`

  Description: work to do after all regions have been processed.

- **AbortSnapshot** [created]

Class Description:

Abort a snapshot that is being created.

Methods:

- [added] `protected void beforeProcess()`

  Description: Call `ZooKeeperWrapper.abortSnapshotOnZK` to notify all the region servers to abort the snapshot if it is still in process. (probably should sleep for a while waiting for snapshot thread is interrupted on RS)

- [added] `protected void processRegion(byte[] regionName)`
  Description: Snapshot might be partially created. Thus for the files that have been created in this snapshot region, decrease the reference count for this file in .META. by one.

- [added] `protected void afterProcess()`
  Description: Delete the whole directory for this snapshot on HDFS.

- **RestoreSnapshot** [created]

Class Description:

Restore snapshot into a table

Methods:

- [added] `protected void beforeProcess()`
  Description: Check whether the table already exists. If not, create table dir under /hbase and copy the file .oldlogs. Otherwise, an exception will be thrown.

- [added] `protected void processRegion(byte[] regionName)`
  Description: There are two things to do for each region:
  a) Copy all the files for this region from snapshot dir into the corresponding places under table dir.
  b) Put the region metadata in ".regioninfo" into .META. However, the region info should be set as *offline* at this time.

- [added] `protected void afterProcess()`
  Description: This method works in two steps:
  a) Call method `splitLogs` to split the logs listed in ".oldlogs". After the logs are split successfully, delete the file ".oldlogs".
  b) Update .META. to enable the table. The table regions will be assigned to RS in MetaScanner.

- [added] `private void splitLogs(List logsToSplit, long sequenceNum)`
  Description: Split the logs up to the `sequenceNum`.

- **DeleteSnapshot** [created]

Description:

Delete a snapshot

Methods:

- [added] `protected abstract void beforeProcess()`
  Description: nothing to do

- [added] `protected abstract void processRegion(byte[] regionName)`
  Description: For each file in this snapshot region, update the reference count information in .META. to decrease by one.

- [added] `protected abstract void afterProcess()`
  Description: Delete the whole directory for this snapshot.

Appendix A. Reference Count in .META.

A new column family "snapshot" is added to the .META. to keep the snapshot metadata, which is reference count information of HFiles, no matter it is still in use or archived. The qualifier is the file name of HFile while the value is the number of snapshots that have a reference to this file. The schema of .META. is shown as below:

Row Key	*Column Family* Info		*Column Family* historian		*Column Family* snapshot	
Encoded-Region Name	*Qualifier*	*Value*	*Qualifier*	*Value*	*Qualifier*	*Value*
	regioninfo	[HRegionInfo]			[filename1]	[Long]
	server	[String]			[filename2]	[Long]
	serverstartcode	[Long]			[filename3]	[Long]
	splitA	[HRegionInfo]			[filename4]	[Long]
	splitB	[HRegionInfo]			...	...

For each region of a table, there is a row in .META. to keep the metadata of this region. Because the meta row for a table region could be deleted after a split, the reference count information should not be kept in the original meta row. And also because the reference count information is related to all the snapshots, it is not appropriate to keep it in the metadata of one snapshot.

Once a region is backed up in a snapshot, a *new* row is inserted into .META. to keep the reference count information for this region. To differentiate it from the original meta row, a prefix *".SNAPSHOT."* is added to the original region name. So the row keys for these snapshot regions start with *".SNAPSHOT."* (Of course it is encoded afterwards). It's like a virtual table ".SNAPSHOT." which keeps all the regions that are used by snapshots, even though they actually belong to different user tables. Removing the original region meta in .META. does not impact these snapshot meta.