HDFS Snapshots

October 31, 2012

Contents

1 Overview

1.1 Motivation

Users of file systems want to be able to back up and restore their most important data. Doing so is currently difficult in HDFS because of the volume of data stored in a large HDFS instance and the inability to create a consistent backup of a particular moment in time.

1.2 Specific Use Cases

1. HDFS snapshots will be useful to get a consistent state to use with distcp. Currently, distcp users have to worry about corner cases like files being deleted or moved while theyre in the process of being synchronized between sites.

2. Some organizations might also want to make monthly or weekly backups of their data. Snapshots would allow those backups to be made on a more flexible schedule. For example, one could create a snapshot at the precise end of the month, and then make the backup for the month of September on September 4th based on the snapshot, rather than having to do it immediately at the beginning of the month.

3. HDFS snapshots can be used to implement HBase snapshots efficiently than HDFS currently allows for given the interface exposed to users of HDFS.

4. HDFS snapshots can protect against oops! moments where a system administrator deletes files she doesnt intend to. Although the trash feature of HDFS serves some of the same purpose, the trash can be defeated by configuration problems. Files which are deleted by code calling `FileSystem#delete` or a related API also do not go into the trash, but are directly deleted.

5. Snapshots can provide organizations with metrics for what data changed over time. Some organizations have to retain data for a certain amount of time for legal or process reasons, and snapshots can help make this more efficient and easier.

6. Snapshots can be used to provide a point in time view of the file system for auditing/compliance purposes.

1.3 Requirements

1.3.1 Goals for HDFS Snapshots

(In no particular order)

1. Read-only snapshots

2. Consistent (to be defined below) snapshots with respect to both NN and DN state

3. Sufficient for DR backup purposes

4. Sufficient for producing consistent HBase snapshots

5. Support maintaining/surfacing of multiple (10s) of separate snapshots

6. Low resource overhead at the NN to maintain snapshots
7. Constant-time or otherwise very quick to create a new snapshot
8. Low or non-noticeable impact to workloads running on HDFS while a snapshot is being created.
9. Support all current configurations of running HDFS (security, HA, Federation, etc.)
10. Easy for users and administrators to browse and restore the contents of snapshots. Surfacing snapshots as regular directories is well-known and natural to users of existing file systems which support snapshots.
11. subtree snapshots, i.e. it should be possible to create a snapshot of just a subdirectory of the file system tree.
12. Deletion of snapshots, with support for deleting snapshots in a different order from when they were taken.
13. Support for allowing users (non-super users) to create snapshots of specific subtrees of the file system, configurable by the administrator.

1.3.2 Explicit Non-goals for HDFS Snapshots

1. Read-write snapshots

2 User Experience and API

2.1 How snapshots will be created

Before a snapshot can be created for a given subtree of the file system, the top level directory of that subtree must be marked as *snapshottable.* A snapshot can only be created for directories which are marked *snapshottable.* To simplify the administration of snapshots, a *snapshottable directory* must not have any descendent that is itself a *snapshottable directory.*

2.2 User commands to work with snapshots

2.2.1 Commands to manage snapshots generally

The following commands are used to manage snapshots in the file system as a whole, and are available only to the super user.

- To mark a directory as snapshottable:

  ```
  $ hdfs dfsadmin -allowSnapshots <path>
  ```

- To mark a directory as being no longer snapshottable:

  ```
  $ hdfs dfsadmin -disallowSnapshots <path>
  ```

  Note: This command will return an error if there are any snapshots currently present for this *snapshottable directory*

- To allow an arbitrary user to create snapshots of a given snapshottable directory:

```
$ hdfs dfsadmin -addSnapshotUser <user> <path>
```

- To disallow a particular user to create snapshots for a given snapshottable directory:

```
$ hdfs dfsadmin -removeSnapshotUser <user> <path>
```

- To list all *snapshottable directories* in the system:

```
$ hdfs dfsadmin -listSnapshottableDirs
```

2.2.2 Commands to manage individual snapshots

The following commands are used to manage individual snapshots. These commands are available to the super user or those users who have been added by the super user as being able to create snapshots for a given snapshottable directory using the *dfsadmin -addSnapshotUser* command previously mentioned.

- To list snapshot information (users allowed to create for a given *snapshottable directory*, how many snapshots currently exist of the *snapshottable directory*, the unique numeric IDs of those snapshots, etc.):

```
$ hdfs dfsadmin -snapshotInfo <path>
```

- To create a new snapshot of a *snapshottable directory*:

```
$ hdfs dfsadmin -createSnapshot <path>
```

- To delete a previously-created snapshot of a *snapshottable directory*:

```
$ hdfs dfsadmin -deleteSnapshot <path> <snapshot ID>
```

For all of the shell commands described above, corresponding API calls will be added to the `HdfsAdmin` class. Any command which takes a path to a *snapshottable directory* as an argument will return an error if the path provided was not in fact previously marked as a *snapshottable directory*.

2.3 How clients will access snapshots

The primary concern with materializing snapshotted data should be ease of use for the user. To this end, we propose for HDFS a user experience which is familiar to many users, by mimicking the user experience of WAFL in this regard.[1] In WAFL, access to the snapshots of a given directory is available directly from that directory via a special directory called *".snapshot"* A *.snapshot* directory is available for every directory under a volume which has been snapshotted. This directory entry is not listed in the results of calls to `readdir()`, so a listing of a directory will not show the *.snapshot* directory. However, operations on paths which explicitly include the *.snapshot* directory in the path will succeed. This behavior is desirable for several reasons. When a user is exploring a directory present in the file system and wishes to see what snapshots exist of the contents of that directory, the available snapshots of that directory can easily be located at a well-known path without needing to move up the directory hierarchy to locate a snapshot root directory. It is desirable to not include the special *.snapshot* directory in the regular directory listings so that tools which recursively traverse the file system do not need to specifically exclude snapshots.

[1] http://media.netapp.com/documents/wp_3002.pdf

We propose we do something similar for HDFS snapshots, wherein a call to `'hadoop fs -ls <path>'` will not include in its results a *.snapshot* directory. However, when a path is explored which contains a *.snapshot* directory in the path, the NameNode will identify this and modify the returned results appropriately. For example, assuming that */user* were marked as a *snapshottable directory* with two snapshots created of it, the following results might be returned:

```
$ hadoop fs -ls /user/atm/
Found 4 items
drw-r--r-- 1 atm atm    0 2012-10-08 14:17 .staging
drw-r--r-- 1 atm atm    0 2012-10-19 10:04 data-set-1
drw-r--r-- 1 atm atm    0 2012-10-19 10:06 data-set-2
-rw-r--r-- 1 atm atm 3338 2012-10-19 11:10 logs-2012-10-15.txt

$ hadoop fs -ls /user/atm/.snapshot/
Found 2 items
drw-r--r-- 1 atm atm 0 2012-10-19 10:00 /user/atm/.snapshot/000001
drw-r--r-- 1 atm atm 0 2012-10-19 12:00 /user/atm/.snapshot/000002

$ hadoop fs -ls /user/atm/.snapshot/000001/
Found 1 items
drw-r--r-- 1 atm atm 0 2012-10-08 14:17 .staging

$ hadoop fs -ls /user/atm/.snapshot/000002/
Found 4 items
drw-r--r-- 1 atm atm    0 2012-10-08 14:17 .staging
drw-r--r-- 1 atm atm    0 2012-10-19 10:04 data-set-1
drw-r--r-- 1 atm atm    0 2012-10-19 10:06 data-set-2
-rw-r--r-- 1 atm atm 3338 2012-10-19 11:10 logs-2012-10-15.txt
```

3 Technical Approach

From a technical perspective, there are three main questions that need to be decided. The first question is how we should represent snapshots on the NameNode. The second question is how snapshots should be materialized to clients. The third quest is how snapshot creation should deal with open files being written.

3.1 Assumptions

This design relies on the following behaviors of HDFS:

1. After being written and closed, files in HDFS may only be reopened for read or append.
2. Files in HDFS cannot be truncated except by deletion and recreation of a file with the same name.

3.2 Representing snapshots at the NameNode

If snapshots are read-only, we can give each snapshot a monotonically-increasing number which represents its position in the file system history. We can then tag all inodes with a `start_snap` field, representing

the first snapshot in which that inode was present, and an `end_snap` field, representing the last snapshot in which that inode was present. The special value PRESENT represents the latest revision which is still in existence. At any point in time, the global variable `next_snap` represents the number of the next snapshot which will be taken.

When any inode is deleted, moved, or otherwise modified, we examine the `start_snap` field of that inode. If it is `next_snap`, then we simply perform the modification in place, as usual. If it is older than `next_snap`, that means that the inode which we modified was present in an older snapshot. In that case, we change the inodes `end_snap` field to be `next_snap`, indicating that the inode is no longer relevant to snapshots beginning at `next_snap`. Then, we create a new inode with `start_snap` = `next_snap`, and `end_snap` = PRESENT. This inode contains the information about the file or directory which will be applicable to snapshots at `next_snap` and possibly beyond.

Taking a snapshot is easy in this scheme. To take a snapshot, you simply increment `next_snap`. No other work is required. Taking a snapshot is hence O(1). Memory and disk space consumption are also O(1) at snapshot creation time under this scheme. This design does require that we copy an inode when it is modified (file length increased, permissions changed, etc) since we do need to create a new copy of the inode at that time. This is not a major burden, however.As mentioned earlier, we only have to do this when there is an older, snapshotted copy of the inode.

In most cases, all of the different versions of an INodeFile can share a copy of the same array of BlockInfo structures.We simply store the visible length along with each version. (The exception to this is when someone creates a file with the same name as an existing file, causing the existing file to be truncated.)

Similarly, in most cases, all of the different versions of an INodeDirectory can share a copy of the same array of children. The key insight here is that if someone examines the file system hierarchy at snap S, additional entries in the array with `start_snap` ¿ S will not affect his view of the filesystem.

How many simultaneous snapshots will be possible under this scheme? It depends on how many bytes we want to give to `start_snap` and `end_snap`. If we give them 8 bytes, then approximately 2^{64} simultaneous snapshots will be possible, though this may be overkill. Using 4 bytes each will likely be sufficient. Assuming we start at 0 and go to Javas `Integer.MAX_VALUE`, 2^{31} snapshots could be taken. This would allow a very zealous snapshotter to create one snapshot of the file system every second for the next 68 years. Much more likely would be a user taking snapshots once per minute or hour.

In order to support subtree snapshots, we introduce a new class `INodeDirectorySnapshottable` to represent the aforementioned *snapshottable directory* in the NameNode. The data for which snapshots exist, what the value of `next_snap` is, etc. will be stored in each `INodeDirectorySnapshottable`.

3.3 Materializing snapshots to clients

To implement the client view of snapshots, we explicitly do not modify the result of `ClientProtocol#getListing` to include *.snapshot* directories in the results when listing regular directories. When a *.snapshot* directory is listed, the NameNode will modify the results to include an entry for each snapshot available. These directories will be named *"X"* where X is the numeric snapshot ID of that snapshot. The `mtime` of these directories will be the time the snapshot was taken. When a *.snapshot/X* directory is listed, the directory contents at the time the snapshot was created will be returned.

3.4 Dealing with open files being written

Most of the time, when a snapshot is taken, there will be some files which are being written to by various clients. How should these files appear in the snapshot?

Unless we properly deal with open files being written, restoring from an HDFS snapshot may corrupt systems built on top of HDFS, such as HBase. HBase maintains a write-ahead log, stored as an HDFS file which is periodically hflushed.This write-ahead log contains the details of HBase operations that have been committed. Some of those HBase operations refer to HFiles, which are also stored as HDFS files. We need to maintain the ordering of those operations from the region servers point of view.

In other words, if an HBase operation is present in the snapshots view of the HBase WAL, the other HDFS files it references must also be present.Similarly, if an HDFS file has been deleted in a snapshot, the HBase operation marking its deletion must also appear in the snapshots view of the WAL.It is not too difficult to achieve this goal for NameNode operations - the NameNode serializes all operations into its own write-ahead log, the edit log. However, writing to an HDFS file usually only involves talking to the DataNodes, making serialization more difficult.

There are two approaches we consider here, which we will henceforth call the *super-sync* approach, and the *viewstamped* approach.

3.4.1 Open files option 1: super-sync

In the first approach, we add a *super-sync* option to the `hflush()` and `hsync()` operations of the `DFSOutputStream` class. If *super-sync* is set, the visible length of the file being synced will be updated on the NameNode. When a snapshot is taken, it will contain all files that are open for write. The length of those super-synced files will be one of the lengths that was super-synced. The length of other files being written which have not been super-synced will be the length of the file as of the last block boundary. This can be implemented simply by just adding a length field to the `ClientProtocol#fsync()` operation, which makes an RPC to the NameNode. Ordinarily this only needs to be called when new blocks have been allocated for the `DFSOutputStream` since the last `hflush` or `hsync` calls, but when the super-sync flag is set, this call will be made to the NameNode regardless, so as to update the length.

The main intention behind adding this operation is to allow clients which deliberately interleave namespace modifications and data flushes to have that history correctly represented in snapshots, without imposing overhead on less critical open files like MapReduce temporary files. For example, HBase could super-sync its WAL. Essentially, this solves the problem of serializing DataNode-only operations by making super-synced flushes also be NameNode operations.

3.4.2 Open files option 2: viewstamps

The second approach, the *viewstamped* approach, takes advantage of the monotonically increasing transaction IDs which the NameNode assigns to operations. Whenever the `DFSClient` performs a NameNode operation, the NameNode can return an 8-byte ID to the client, corresponding to the edit log transaction ID of the completed operation. Then, when the `DFSClient` performs a write to a file on a DataNode, it passes along this transaction ID to the DN.

For each currently open file on a DataNode, it maintains a small map of recent transaction IDs to offsets within the file. The DataNodes send this information back to the NameNode in their block reports and heartbeats. Since the number of open files for write at any given time is small, and the number of clients which interleave data writes with namespace modifications is even smaller, the overhead should be slim. During snapshot creation, the snapshot is associated with a transaction ID, and will not honor any visible lengths that are associated with a greater transaction ID.

This ensures that snapshots contain a consistent view of the world from the perspective of a single DFSClient.

This does imply, however, a bit of coordination between the DNs and the NN during snapshot creation. Since heartbeats may be delayed for some period of time, and some of the file lengths for open files may not have reached the NN when the command to create the snapshot is run, the process of creating a snapshot must wait to receive all of the appropriate (for the transaction ID of the snapshot) lengths for files being written from the DataNodes. Note that the RPC to begin snapshot creation can return immediately, while the creation of the snapshot can then continue asynchronously on the NN. The snapshot will not be made available to clients until it is complete, i.e. the proper lengths for all files being written have been received from the NN.

3.4.3 Replication factor

When a file which shares a block list across several snapshots has several different values for replication factor between those snapshots, which replication factor the NameNode should attempt to maintain for the blocks of that file is not immediately obvious. Here, we consider three options:

1. Each snapshot of the file maintains its replication factor separately, like all other per-file attributes. The actual replication factor maintained by the NameNode is the max of all of these replication factors.
2. Each snapshot of the file maintains its replication factor separately, like all other per-file attributes. The actual replication factor maintained by the NameNode is the current replication factor of the file in the file system (not in a snapshot) or the replication factor in the most recent snapshot if the file has been removed from the current file system.
3. Replication factor is treated differently from the other file attributes. The snapshots of the file share a single replication factor across all snapshots. When this replication factor is updated for the current file, this change appears across all snapshots.

While all of these options seem potentially valid, we propose we go with option #3 for the following reasons:

1. It seems desirable to allow a user that wishes to lower their DFS used space to lower the replication factor of a file and have that actually affect their space consumption. If we go with option #1, this is not possible.
2. In some sense the target replication factor is an attribute of the blocks, not of the file. Since the blocks are shared between all instances of a file across snapshots, this attribute should be shared as well. Unlike other file attributes like permissions, file length, etc. the target replication factor of a file should be transparent to clients, and is really just a hint as to how the blocks should be stored. This further suggests we should go with option #3.

3.4.4 Other technical considerations

For being-written files, we need to make sure that file `mtime` is updated appropriately when a block is added to a file or file length is otherwise updated on the NameNode.

Since snapshots are immutable, `atime` for files, even when enabled in HDFS, will not be updated for files in a snapshot.

3.4.5 Future directions

Read/write snapshots, if we ever choose to implement them, will require a different system for dealing with the NN snapshot metadata. An upgrade operation could easily handle the fsimage format conversion if necessary.

3.5 Development stages and milestones

It seems likely that the three broad aspects of the technical direction described above (representation of snapshots at the NN, materializing snapshots to clients, and dealing with open files being written) can be implemented in parallel. These will likely represent the long poles of implementing snapshots. Other distinct stages that will be necessary to implement will be amending the `DFSAdmin` administrative commands to support snapshots and dealing with replication factor properly at the NameNode.

Regarding the two options for NN/DN consistency, it seems reasonable to implement the *super-sync* option (Option 1) first, and then to later implement the *viewstamped* option (Option 2) if we decide its desirable. If Option 2 is implemented after Option 1, we can just make the hflush/hsync operations ignore the *super-sync* flag, since the correct file length will be included in snapshots without needing to explicitly set the *super-sync* flag.

3.6 Compatibility

This work will be targeted for trunk / Hadoop 3.x, which means that we are allowed to break wire compatibility and require a NameNode metadata upgrade. We should try to maintain wire compatibility wherever possible, though, and with the use of protobufs for RPC serialization this should be possible.

3.7 Prior Work

The most directly related prior work is Snapshots in Hadoop Distributed File System by Sameer Agarwal, Dhruba Borthakur, and Ion Stoica (http://www.cs.berkeley.edu/ sameerag/hdfs_snapshots_ucb_tr.pdf). This design uses a reference counting system for tracking how many snapshots a particular file or directory is referenced from. This design uses a technique the authors call Selective Copy-on-Append for tracking the appropriate length of a file for a given snapshot, and to reduce the memory overhead of storing snapshots in the NameNode. Because this design requires updating a reference count for each file/directory whenever a snapshot is created, snapshot creation time is linear in the number of files that are being snapshotted.

4 Revision history

```
commit 979e189165810d999e69c7a14f546848f48c2e9a
Author: Aaron T. Myers <atm@cloudera.com>
Date:   Wed Oct 31 18:35:11 2012 -0700

    Initial commit in TeX format. Fleshes out some more details and includes
    concept of snapshottable directories.
```