Snapshots in Hadoop Distributed File System

Sameer Agarwal
UC Berkeley

Dhruba Borthakur
Facebook Inc.

Ion Stoica
UC Berkeley

Abstract

The ability to take snapshots is an essential functionality of any file system, as snapshots enable system administrators to perform data backup and recovery in case of failure. We present a low-overhead snapshot solution for HDFS, a popular distributed file system for large clusters of commodity servers. Our solution obviates the need for complex distributed snapshot algorithms, by taking advantage of the centralized architecture of the HDFS control plane which stores all file metadata on a single node, and alleviates the need for expensive copy-on write operations by taking advantage of the HDFS limited interface that restricts the write operations to append and truncate only. Furthermore, our solution employs new snapshot data structures to address the inherent challenges related to data replication and distribution in HDFS. In this paper, we have designed, implemented and evaluated a fast and efficient snapshot solution based on selective-copy-on-appends that is specifically suited for HDFS like distributed file systems.

1 Introduction

File systems supporting critical large scale distributed data-intensive applications require frequent automated system backups with zero or minimal application downtime. As a result, the ability to take snapshots has emerged as an essential feature of any file system. Not surprisingly, developing such snapshots schemes and mechanisms has received considerable attention in the past [9, 7, 8, 6].

However, the previously proposed snapshot solutions are not a good match for the recently developed distributed file systems such as GFS [6] and HDFS [2], as these file systems are different in several important aspects form traditional file systems.

First, file systems such as GFS and HDFS logically separate the file system metadata and the data itself. A master node (known as the Namenode in HDFS) tracks the file metadata and each file is replicated and divided into fixed sized data blocks which are spread across the cluster nodes. While presence of the metadata on a single node considerably simplifies the snapshot solution, tracking the state of multiple data blocks and their replicas requires more evolved data structures in addition to the snapshot tree.

Second, these file systems target applications that read and write large volumes of data, such as MapReduce [5, 1], and BigTable [4]. As a result, these file systems are optimized for sequential disk access, and expose a very limited interface: the write operations are restricted to appends and truncates only. Given these constraints, using the standard copy-on-write snapshot solutions results in unnecessary overhead. We will show that unless an append follows a truncate operation, file appends and truncates do not result in overwriting existing data and their state across different snapshots could be tracked using data pointers. This forms the basis of our selective-copy-on-appends snapshot scheme.

While many of the techniques described in this paper can be generalized to most of the existing file systems, we believe that our key contribution lies in highlighting how can we design very low overhead file system features by taking into account the specific architecture and constraints recently proposed distributed file systems for data intensive applications.

The rest of the paper focuses on the design and implementation of snapshot support in HDFS and is outlined as follows. In Section 2, we present the details of the current HDFS architecture, and in Section 3 we describe our snapshot solution. Finally, we present the evaluation our solution in Section 4.

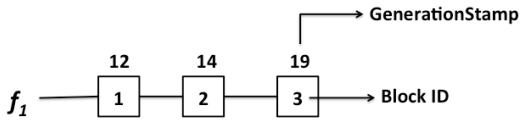


Figure 1: HDFS Files

2 Background: HDFS Design

Similarly to Google File System [6], Hadoop Distributed File System (HDFS) [2] is a fault tolerant distributed file system designed to run on large commodity clusters, where the storage is attached to the compute nodes. HDFS employs a master-slave architecture [3] where the master (or the Namenode) manages the file system namespace and access permissions. Additionally, there are a large number of slaves (or the Datanodes) which manage the storage attached to the physical nodes on which they run. Each file in HDFS is split into a number of blocks which are replicated and stored on a set of Datanodes. The Namenode manages the file system namespace as well as the metadata about file and block(s) associations as shown in Figure 1. Each data block is identified by a Block ID which specifies its position in the file and a unique, monotonically increasing Generation Timestamp. Since these are assigned by the Namenode, no two HDFS blocks can ever have the same Generation Timestamp. Another distinctive aspect of HDFS is that it relaxes a few POSIX requirements (disabling writes/locks anywhere other than the tail of the file) to enable high speed data streaming access. Being designed for batch processing applications as opposed to interactive usage, it chooses high throughput access to application data over low latency.

2.1 File Creations and Appends

Figure 2 shows the sequence of steps involved in creating/appending a file in HDFS. When the client invokes the API, it requests a new block storage from the Namenode. Depending on the size of the file, a client can make many such requests serially. Upon receiving the request, the Namenode updates its namespace and assigns a datanode, block ID and generation timestamp to the block and sends it back to the client. Then the client directly updates the metadata on the assigned datanode and streams the rest of the data through a cache. The datanode generally replicates this data to a fixed number of other datanodes for reliability and efficiency. Finally, it sends an ACK back to the namenode on successful transfer and it updates its metadata to reflect the same. In case of appends, the namenode returns the metadata associated with the last block of the file to the client and assigns subsequent blocks once the last block has been completely written. A key observation here is that unless an append operation follows a truncation, file appends never result in overwriting existing data.

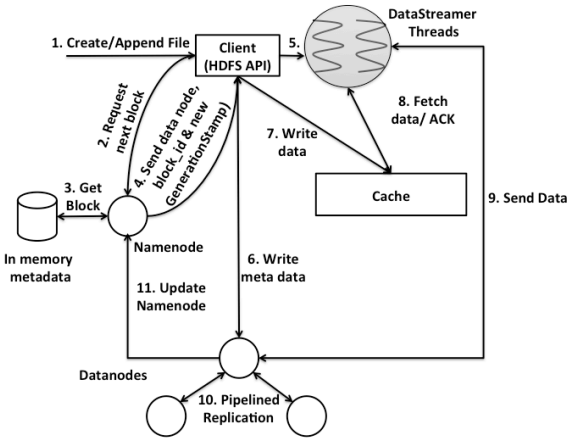


Figure 2: HDFS File Create and Appends

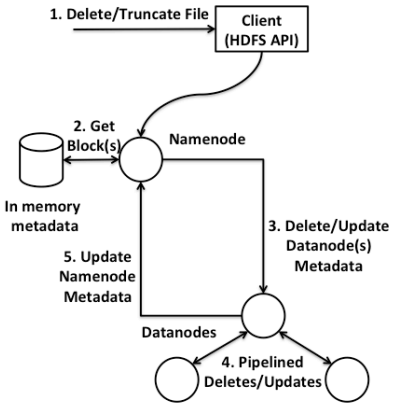


Figure 3: HDFS File Deletes and Truncations

2.2 File Deletions and Truncations

Figure 3 shows the sequence of steps involved in deleting and/or truncating a file in HDFS. When the client invokes the API, the namenode removes the file entry from the namespace and directs all the corresponding datan-

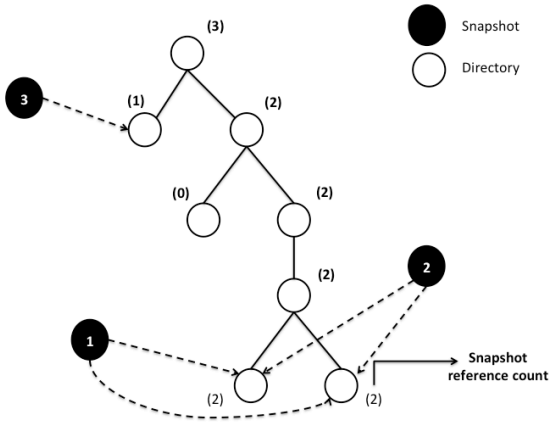


Figure 4: Directory/Snapshot Tree

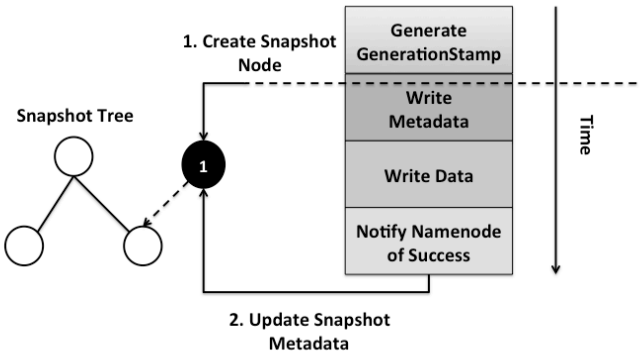


Figure 5: Snapshot Workflow

odes to delete their corresponding metadata. In case of file truncations, the namenode directs the corresponding datanode(s) to update their block size metadata to reflect the change. Since file truncations only result in updating the metadata in datanodes, they never result in overwriting existing data.

3 Snapshots in HDFS

This section describes our snapshot solution in detail. To track all the files referenced by different snapshots, we maintain a fault tolerant in-memory snapshot tree as shown in Figure 4. Each node corresponds to a file or a directory in HDFS which are referenced by zero or more system snapshots. Further, each file or directory is associated with an integer SNAPSHOT_REF_COUNT which denotes the number of snapshots that are pointing to it or its children in the hierarchy tree.

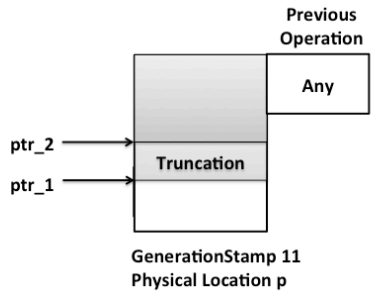
In order to manage the file system namespace and prevent the garbage collector from permanently deleting or overwriting blocks, we modified the existing *Trash* functionality in HDFS. *Trash* is a pluggable module in HDFS which is analogous to a recycle bin. When a file is deleted, the data still remains intact and only the namespace reflects the deletion. This allowed us to easily manage the file system metadata and the garbage collector functionality.

3.1 Design Details

This section describes our low-overhead snapshot solution for HDFS. HDFS differs from traditional file systems in two important aspects. First, HDFS separates the metadata and the data. While the metadata is stored at a single node, the actual data is replicated and spread throughout the cluster. Using a single node to manage the metadata enables us to use a single logical global clock, thereby obviating the need for more complex vector clock based snapshot solutions [**?**]. Second, targeting primarily data intensive frameworks such as MapReduce, HDFS like file systems expose a limited interface, as data writes are restricted to append and truncate operations. Given these constraint, using standard copy-on-write snapshot solutions result in unnecessary overhead. As we observed in Section 2, unless an append follows a truncate operation, file appends and truncates do not result in overwriting existing data and their state across different snapshots could be tracked using data pointers. This observation represents the basis of our selective-copy-on-appends snapshot scheme. Whenever a snapshot is taken, a node is created in the snapshot tree (as shown in in Figure 4) that keeps track of all the files in the namespace by maintaining a list of its blocks IDs along with their unique generation timestamps. The following subsections discuss how the snapshot tree is updated during various file system operations.

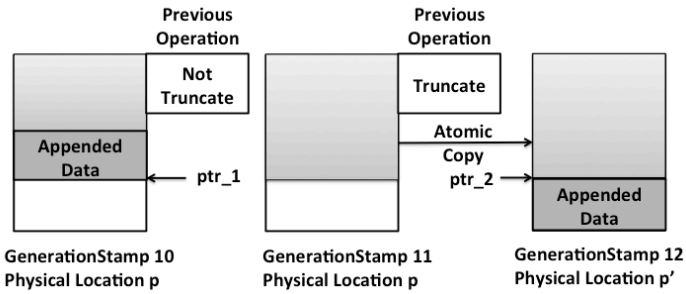
3.2 Selective Copy-on-Append

In this section, we describe our selective copy on appends snapshot scheme. As long as the data does not get overwritten, we need only to store the end pointers to the data in each block over consecutive updates. Thus, when a series of zero or more consecutive appends are followed by a series of zero or more truncates on a single block, a pointer based solution will suffice. Only when there are one or more append operations after one or more truncates, data gets overwritten and we make



Block ID	GenerationStamp	Physical Location
X	10	p, ptr_1
X	11	p, ptr_2

Figure 6: Truncate Semantics



Block ID	GenerationStamp	Physical Location
X	10	p, ptr_1
X	11	p, ptr_1
X	12	p', ptr_2

Figure 7: Append Semantics

a new copy of the block (known as copy on write). In summary, we use pointers as long as there are only appends (AAA), only truncates (TTT) or appends followed by truncates (AAATTT), and employ copy-on-write to create two copies of the original block when an append follows one or more truncates (TTTA).

We discuss the detailed of our solution next. As shown in Figure 4, a creation or update operation in HDFS consists of four stages: (1) Namenode generates a Generation Timestamp, (2) the client writes metadata to the datanode (3), the client streams data to the datanode (4), and the datanode(s) notify the namenode on successful operation. If a snapshot is taken during the time one or more data block(s) are being updated, the snapshot tree can be atomically updated as soon as the namenode is notified of successful operation without affecting any existing operations. We have implemented the selected copy-on-append functionality as a HDFS wrapper.

3.2.1 File Truncates

As described in Figure 6, a truncate operation can never result in overwriting existing data. For every file truncation operation, we create a mapping between the block's generation timestamp, length and existing physical location.

3.2.2 File Appends

As described in Figure 7, an append operation can only result in overwriting existing data if directly follows a file truncate operation. In this case, we make a new copy of the existing block atomically and create a mapping between the block's generation timestamp, length and its physical location. In all other cases, we just create a mapping between the block's generation timestamp, length and existing physical location.

4 Evaluation

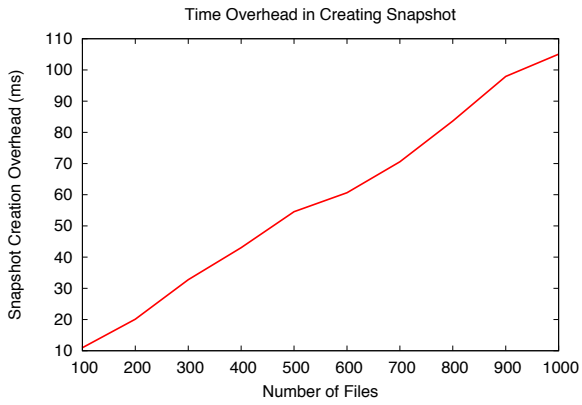


Figure 8: Time Overheads

We implemented the entire snapshot support for HDFS in about $1,200$ lines of code. The snapshot tree and garbage collection is managed by the Namenode and the rest of the code involves writing wrappers for the HDFS interface to implement selective copy on appends and modifying the *Trash* functionality. Being designed for a production system which requires frequent snapshots,

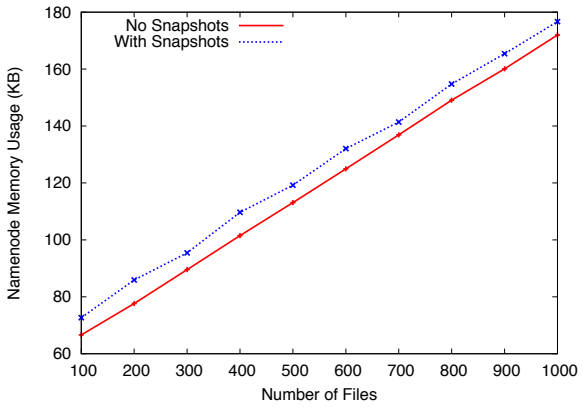


Figure 9: Memory Overheads

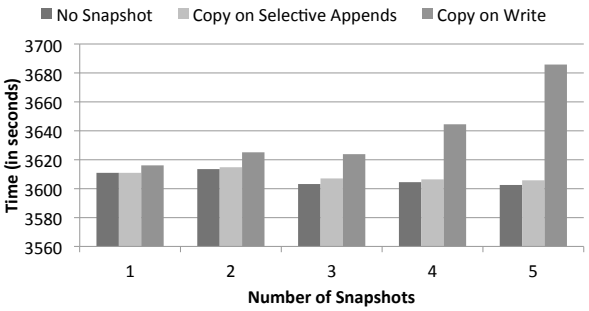


Figure 10: Hive Overheads

aiming for very low overheads were a fundamental aspects of our design. In order to evaluate our design, we conducted various experiments on an Amazon EC2 cluster consisting of 100 small instances and measured scalability in terms of time and memory overheads. The file trace for our experiments came from a $3,000$ node Facebook cluster. Figure 8 shows that the snapshot creation overhead is as low as 0.1 second for about $1,000$ files and scales almost linearly with the number of files in the system.

Figure 9 plots the size of the in memory metadata at the Namenode versus the number of files. We observe that the additional memory overhead is virtually constant (i.e., around 5 KB) for the Facebook trace we used. This is because the additional overhead is primarily due to append and truncate operations, which are not only infrequent but are also limited to a very small number of files.

To evaluate the efficiency gain of our selective-copy-on-append solution over the traditional copy-on-write solutions, we have mounted the Hive transaction log on HDFS. This log exhibits frequent appends, as every transaction generates an append. We ran five instances of approximately an hour long Hive workload trace on our cluster and measured the time overhead between our solution and a modified solution using always copy-on-write respectively. Figure 10 plots the median time taken by the trace to complete using both solutions as a function of number of snapshots taken. As expected, the selective-copy-on-append solution incurs a much lower overhead than the copy-on-write solution.

5 Conclusion

In this paper, we have designed, implemented and evaluated a low overhead snapshot solution for the Hadoop Distributed File System. Our solution has linear time complexity for creating snapshots with respect to the number of files, and has negligible namenode memory overhead. Our solution relies on using a selective copy-on-append scheme that minimizes the number of copy-on-write operations. This optimization is made possible by taking advantage of the restricted interface exposed by HDFS, which limits the write operations to appends and truncates only.

References

[1] Apache hadoop mapreduce. `http://hadoop.apache.org/mapreduce/`.

[2] Hadoop distributed file system. `http://hadoop.apache.org/hdfs/`.

[3] BORTHAKUR, D. The hadoop distributed file system: Architecture and design. `http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf`.

[4] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data. In *OSDI* (2006), pp. 205–218.

[5] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[6] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[7] PAWLOWSKI, B., JUSZCZAK, C., STAUBACH, P., SMITH, C., LEBEL, D., AND HITZ, D. Nfs version 3: Design and implementation. In *USENIX Summer* (1994), pp. 137–152.

[8] RODEH, O., AND TEPERMAN, A. zfs - a scalable distributed file system using object disks. In *IEEE Symposium on Mass Storage Systems* (2003), pp. 207–218.

[9] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI* (2006), USENIX Association, pp. 307–320.