Snapshotting in Hadoop Distributed File System for Hadoop Open Platform as Service

Pushparaj Moatamari

Instituto Superior Tecnico

Abstract. The amount of data stored in modern data centres is growing rapidly nowadays. Large-scale distributed file systems, that maintain the massive data sets in data centres, are designed to work with commodity hardware. Due to the quality and quantity of the hardware components in such systems, failures are considered normal events and, as such, distributed file systems are designed to be highly fault-tolerant.
A concrete implementation of such a file system is the Hadoop Distributed File System (HDFS).Snapshot means capturing the state of the storage system at an exact point in time and is used to provide full recovery of data when lost.Operational as well as analytical applications manipulate the data in the distributed file system on behalf of the user or the administrator. Application-level errors or even inadvertent user errors can mistakenly delete data or modify data in an unexpected way. In this case, snapshots can be used to recover to a known, well-defined state.Snapshots can be used in Model Training, Managing Real-time Data Analysis and also to produce backups on the fly(Hot Backups).We designed nested snapshots which enables multiple snapshots on any directory. We designed and implemented root level single snapshot by which roll-back during software up-gradation can be made.We evaluate our designs and algorithms, and we show that time to take snapshot is constant and roll-back time is proportional to the changes since snapshot.

Keywords: Hadoop,HDFS,Distributed FileSystem,Snapshots,HOPS

1 Introduction

The need to maintain and analyse a rapidly growing amount of data, which is often referred to as big data, is increasing vastly. Nowadays, not only the big internet companies such as Google, Facebook and Yahoo! are applying methods to analyse such data , but more and more enterprises at all. This trend was already underlined by a study from The Data Warehousing Institute (TDWI) conducted across multiple sectors in 2011. The study (Russom 2011) revealedthat 34% of the surveyed companies were applying methods of big data analytics, where as 70% thought of big data as an opportunity.

A common approach to handle massive data sets is executing a distributed file system such as the Google File System (GFS) or the Hadoop Distributed File System (HDFS) on data centres with hundreds to thousands of nodes storing petabytes of data. Popular examples of such data centres are the ones from Google, Facebook and Yahoo! with respectively 1000 to 7000 , 3000 and 3500 nodes providing storage capacities from 9.8 (Yahoo!) to hundreds of petabytes (Facebook).

Many a times users using those big data sets would like to run experiments or analysis which may overwrite or delete the existing data.One option is to save the data before runninganalytics, since the data size is very large it is not a feasible option. The underlying file system has to support features to snapshot the data which enables users to run experiments and rollback to previous state of data, if something does not work.

Following scenarios demand the need of utility to take snapshots on file system.

1. **Protection against user errors:** Admin sets up a process to take RO(Read-Only) snapshots periodically in a rolling manner so that there are always x number of RO snapshots on HDFS. If a user accidentally deletes a file, the file can be restored from the latest RO snapshot.
2. **Backup:** Admin wants to do a backup of a dataset. Depending on the requirements, admin takes a read-only (henceforth referred to as RO) snapshot in HDFS. This RO snapshot is then read and data is sent across to the remote backup location.
3. **Experimental/Test setups:** A user wants to test an application against the main dataset. Normally, without doing a full copy of the dataset, this is a very risky proposition since the test setups can corrupt/overwrite production data. Admin creates a read-write (henceforth referred to as RW) snapshot of the production dataset and assigns the RW snapshot to the user to be used for experiment. Changes done to the RW snapshot will not be reflected on the production dataset.

4. **Model Training** Machine-learning frameworks such as Mahout can use snapshots to enable a reproducible and audit-able model training process. Snapshots allow the training process to work against a preserved image of the training data from a precise moment in time.
5. **Managing Real-time Data Analysis** By using snapshots, query engines like Apache Drill can produce precise synchronic summaries of data sources subject to constant updates such as sensor data or social media streams. Using a snapshot for such analyses allows very precise comparisons to be done across multiple ever-changing data sources without having to stop real-time data ingestion.

Following goals are desired from the solution to snapshotting on FileSystem

1. Able to take multiple snapshots and nested snapshots on directories and files
2. Able to support quick rollback in case of software upgradation failure.
3. Time to take snapshot should be constant, should not dependant on the size of the fileSystem.

Contribution and structure

The contributions of thesis work are
1. Design and algorithms for implementing Read-Only Nested Snapshots in HDFS of HOP. The design enables the users to take snapshot in constant amount of time since all oper- ations on snapshotted directories are logged in an efficient manner to retrieve the meta- data of the filesystem to the state before snapshot.
2. Design and implementation of Single Snapshot which facilitates roll-back in case of software upgradataion failures.The rollback algorithm is implemented and evaluated against MySql server and clusterJ to find the efficient mechanism to perform it.

The thesis is organized in a number chapters. In Chapter 2, following, we study and analyze the relevant related work in the literature on the thesis topics.In chapter 3, we describe the related work by Apache Hadoop Version 2 Snapshots and Facebook-Hadoop. In Chapter 4, we describe the main insights of our proposed solution, highlighting relevant aspects regarding architecture, algorithms. In Chapter 4, we evaluate the performance of our solution. Chapter 5 closes this document with some conclusions and future work.

2 Background

2.1 Hadoop Distributed File System(HDFS)

HDFS[8] represents both directories and files as inodes (INode) in metadata. Directory inodes contain a list of file inodes, and files inodes are made up a number of blocks, stored in a BlockInfo object. A block, in its turn, is replicated on a number of different data nodes in the system (default 3). Each replica is a Replica object in meta- data. As blocks in HDFS are large, typically 64-512 MB in size, and stored on remote DataNodes, metadata is used to keep track of the state of blocks. A block being written is a PendingBlock, while a block can be under-replicated if a DataNode fails (UnderReplicatedBlock) or over-replicated (ExcessReplica) if that DataNode recovers after the block has been re-replicated. Blocks can also be in an InvalidatedBlock state. Similarly, replicas (of blocks) can be in Repli- caUnderConstruction and CorruptedReplica states. Finally, a Lease is a mutual grant for a number of files being mutated by a single client while LeasePath is an exclusive lock regarding a single file and a single client.

Filesystem operations in HDFS, such as file open/close/read/write/append, are blocking operations that are implemented internally as a sequence of metadata operations and block operations orchestrated by the client. First, the client queries or mutates meta- data at the NameNode, then blocks are queried or mutated at DataNodes, and this process may repeat until the filesystem operation returns control to the client. The consistency of filesystem operations is maintained across metadata and block operations using leases stored in the NameNode. If there is a failure at the client and the client doesn't recover, any leases held by the client will eventually expire and their resources will be freed. If there is a failure in the NameNode or a DataNode during a filesystem operation, the client may be able to retry the operation to make progress or if it cannot make progress it will try to release the leases and return an error (the NameNode needs to be contactable to release leases).

2.2 Hadoop Open Platform as Service (HOPS)-Hadoop Distributed File System

HOPS- HDFS[3][4][5], replaces the Primary-Secondary metadata model with shared, transactional memory, implemented using a distributed, in-memory, shared-nothing database, MySQL Cluster (NDB). MySQL Cluster is a real-time, ACID-complaint transactional database, with no single point of failure, that has predictable, millisecond response times with the ability to service millions of operations per second. By leveraging MySQL Cluster to store HDFS' metadata, the size of HDFS' metadata is no longer limited to the amount of memory that can be managed on the heap of JVM on a single node. It stores the metadata in a replicated, distributed, in-memory database that can scale up to several tens of nodes, all while maintaining the consistency semantics of HDFS. It maintains the consistency of the metadata, while providing high performance, all within a multiple-writer, multiple-reader concurrency model. Multiple concurrent writers are now supported for the filesystem as a whole, but single-writer concurrency is enforced at the inode level. It guarantees freedom from deadlock and progress by logically organizing inodes (and their constituent blocks and replicas) into a hierarchy and having transactions defining on a global order for transactions acquiring both explicit locks and implicit locks on subtrees in the hierarchy. The HOPS-HDFS schema is describe in the below diagram.

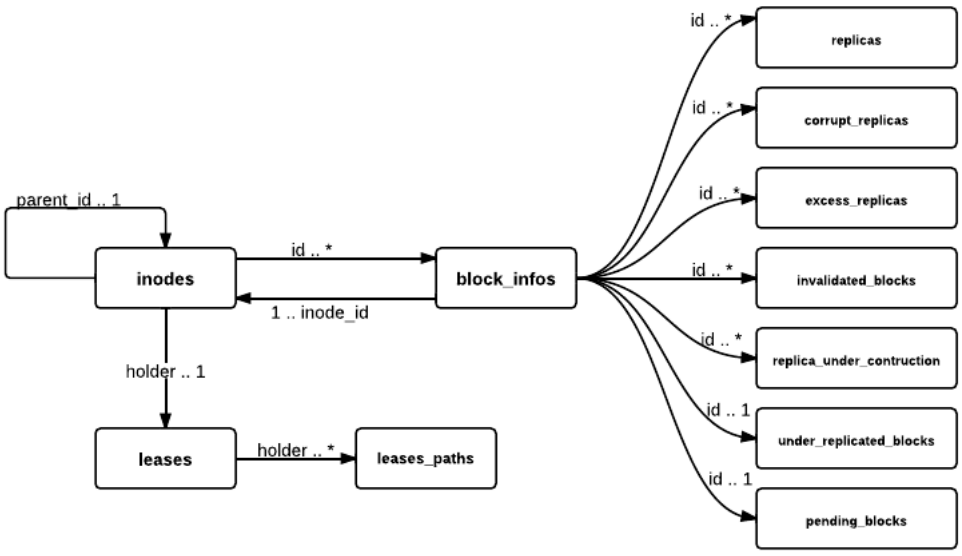

Fig. 1: HOP-HDFS Table relations.

3 Related work

Snapshots in Apache HDFS Version-2

Apache Hadoop implemented snapshots[1] in their latest version (Apache-Hadoop) , which sup- ports nested snapshots and constant time for the creation of snapshots. Since metadata and data are separated it supports strong consistency for metadata snapshot but not for data.In case of file being written, unless the datanode notifies the namenode or client notifies namenode of latest length of last block via hsync, namenode is unaware of current length of the file when snapshot was being taken.The solution also couldnt address the case of replication change of half-filled last block after taking snapshot, where it is appended after taking snapshot.

Apache Hadoop distribution provides single snapshot mechanism to protected file system meta-data and storage-data from software upgrades. The snapshot mechanism lets adminis- trators persistently save the current state of the filesystem, so that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace andstorage state as they were at the time of the snapshot.

The snapshot (only one can exist) is created at the cluster administrators option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and

merges them in memory. Then it writes the new checkpoint and the empty journal to a new location, so that the old checkpoint and journal remain unchanged.

During handshake the NameNode instructs DataNodes whether to create a local snapshot.The local snapshot on the DataNode cannot be created by replicating the directories containing the data files as this would require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories.

3.1 Snapshots in Facebook-HDFS

Facebook has implemented a solution to Snapshots [2]. The solution scales linearly with the number of inodes(file or directories) in the filesystem.It uses 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.The solution has space overhead since Whenever a snap- shot is taken, a node is created in the snapshot tree that keeps track of all the files in the namespace by maintaining a list of its blocks IDs along with their unique generation timestamps. If a snapshot was taken while file is being written, after the write is finished and data node notifies namenode, it saves the location of the file from where it started writing not exactly saving the location in file when snapshot was taken.

4 Read-Only Root Level Single Snapshot

This section explain the solution to single snapshot that can be taken on root which can be rolled back.This solution can be used for roll-backing during software upgrades.Following conditions are applied to the solution

1. Creation of directories with Quota, either name-space quota or disk-space quota is not allowed.
2. Each file consists of blocks. Each blocks-size is typically 64 MB but can be set to any value.Blocks should be written completely.

Following columns need to be added to the Inodes table.

1. isDeleted

Value	Summary
0	Indicates that this Inode is not deleted.
1	Indicates that this Inode deleted after Root Level snapshot was taken.

2. status

Value	Summary
0	Indicates that this Inode was created before taking Root Level Snapshot.
2	Indicates that this Inode created before taking Root Level snapshot but modified after that.
3	Indicates that this Inode was created after taking Root Level snapshot.

Following Columns should be added to BlockInfos table described in the schema

1. status

Value	Summary
0	Indicates that this Block was created before taking Root Level Snapshot.
2	Indicates that this Block created before taking Root Level snapshot but modified after that.
3	Indicates that this Block was created after taking Root Level snapshot.

4.1 Rules for Modifying the fileSystem meta-data

Following rules apply when client issues operations after root level snapshot had been taken.

1. If an inode(file or directory) is created after taking root level snapshot, its status is set to 3.

2. If an inode row is modified and its status is 0, then, a back-up of current row is saved with id = -(current id), parent_ id=-(current parent_ id)[To prevent sql query retrieving the back-up rows while 'ls' command issued, parent id is set to negative of original].The status of current row is changed to 2.
3. If a block is created after taking root level snapshot,its status is set to 3.
4. If a block is modified by appending data to it and its status is 0, then, a back-up of current row is saved with block_ id = -(current block_ id) and inode_ id = -(current inode_ id)[since two block info rows can't have same block index id when retrieved with a parent id].The status of current row is changed to 2.
5. Deletion of a directory or file after root level snapshot was taken. Children of the INode to be deleted are examined in depth-first manner.All the files which are created after snapshot was taken are permanently deleted.The directory's isDeleted flag is set to true.

Roll Back Following algorithm is used to roll back the file-system to the state at the time when Root Level Snapshot was taken.

For INodes:

1. Delete from INodes where status=2 or status=3
2. Update INodes set isDeleted=0 where id>0 and isDeleted=1
3. Update INodes set id = -id, parent_ id = -parent_ id where id<0

For Blocks:

1. Delete from Block_ Info where status=2 or status=3
2. Update Block_ Info set block_ id = -block_ id, inode_ id = -inode_ id where id<0
3. Delete from Block_ Info where block_ id<0

5 Read-Only Nested Snapshots

In this section we describe the solution which can enable taking multiple snapshots and nested snapshots on the file System.

Following columns need to be added to the Inodes table.

1. isDeleted

Value	Summary
0	Indicates that this Inode is not deleted.
1	Indicates that this Inode deleted after snapshot was taken[on its ancestors].

2. isSnapshottableDirectory

Value	Summary
0	Indicates that snapshots can't be taken on this directory.
1	Indicates that snapshots can be taken on this directory.

Following tables need to be added to the schema.

1. **SNAPS**

Inode_ Id	User	SnapShot_ Id	Time

Stores the Indode Id and corresponding snapshots taken on that directory. Time can be a physical clock or logical clock(Global) whose value always increase.

2. **C-List**

Inode_ Id	Time	Created_ Inode_ Id

Stores the id's of children(files or directories) of directory on which snapshot was taken.

3. **D-List**

Inode_ Id	Time	Deleted_ Inode_ Id

Stores the files or directories deleted in a directory on which snapshot was taken. But the rows are not deleted from Inode table, it is an indication to say that these rows were deleted after taking snapshots.

4. **M-List**
Inode_ Id	Time	Modified_ Inode_ Id	Original Row
   After taking a Snapshot if the columns of a particular row are modified then before modifying the row , we copy the original row and store it in this table. When we want to get back to the snapshot, just replace the existing inode row with this original row.
5. **MV-List**
Inode_ Id	Time	Moved_ Inode_ Id	Original Row
   When an inode[either file or directory] is moved, its parentId changes to moved-into directory. In order to get the moved directory when ls command issued at the snapshot after which this inode was moved, we put that row here.
6. **MV-IN-List**
Inode_ Id	Time	Moved_ In_ Inode_ Id
   When a directory or file is moved into this directory(with inode_ id) from other directory.
7. **Block-Info-C-List**
Inode_ Id	Block_ Id	Time
   Stores the blocks that are created in a file after the snapshot was taken on the directory in which this file exist.
8. **Block-Info-M-List**
Inode_ Id	Block_ Id	Time	Original_ Row
   Stores the blocks that are modified in a file after the snapshot was taken on the directory in which this file exist.This is typically for last blocks which are not complete at the time of snapshot.

5.1 Rules for Operations

1. When we create a new file or directory put an entry in c-list.
2. When an inode is modified [rename, touch] it is just put in the M-List. It is not put in the D-List.
3. When you delete a file , put it in the dlist. And set isInodeDeleted to true.
4. Deleting a directory When an directory is deleted, first we will check whether it is in a snapshot[explained later], if yes then we will set isDeleted=1 and also for all of its children[recursive]. We only put the directory in D-List of its parent and we do not put children in D-List.
5. When an inode is moved to some other directory we put it in MV-List of parent directory. We place it in MV-IN-list of destination directory.[the parent_ id is set to the destination directory]

5.2 Listing children under a directory in a given Snapshot

```
void ls_at_snapshot(int id, int stime){

    children={Get children whose parentId=id};
    children = childen - { children deleted before stime} -{ children created after stime}
                       - {children moved_in  after stime};
    children = children + {children moved-out after stime};
    modifiedChildren = { children modified after stime};

    for(child chd : children){
         child inode;
        if(modifiedChildren.contains(chd)){
            modChd = modifiedChildren.get(chd.getId());
            if(chd.movedTime>modChd.modifiedTime){
                #This child modified first then moved.So return the row stored M-List with modChd.mod
                inode = modChd;
            }
            else{
               inode = chd;.
            }
        }
        else{
              inode = chd;
        }
```

```
        if(inode is directory){
             ls+at_snapshot(inode.getId(),stime);
        }
        else{
            print(inode);
        }
    }
}
```

5.3 Determining Which Snapshots an Inode belongs

Columns to be added to Inodes Table
1. Moved_ In/Created Time(Join Time): When an Inode is created we put that time in that column. When we move an Inode from one directory to another we note that time in that.We refer it as join time meaning the time this inode joined in its present directory.

InodeSnapshotMap

Inode_ Id	BelongedTo_ inode_ id	BeginTime	EndTime

Table 1: InodeSnapshotMap table

When to Log
When we add/ deleted/modify directories or files in a directory then we log changes under that directory. We log only when this directory is in any snapshot[which is taken on this directory or one-of its ancestors]. So before performing any operation in this directory we check whether this directory is in any snapshot or not.Consider directory path /A/B/C/, we want to add a file to directory C.We can log this in C-List if C is in a snapshot.
1. First Check any snapshots taken on C.If yes then log.
2. If there are any snapshots on B after JoinTime(C) or if any Snapshots on A after JT(B) and JT(C) then log.
3. If there is any entry for C in InodeSnapshotMap then log.

Moving an Inode
Consider directory paths /A/B/ and /U/V/W/, move directory W to B.

1. Get the list of snapshots taken on U, after JoinTime(V), snapshots taken on V after JoinTime(W). In the form like {U,Time of First Snapshot after Join Time, Time of Last Snapshot after Join Time}.
2. Since some inodes under W may have join times greater than W's join time, we need to check for each inode in sub-tree on which snapshots of U,V is present.After determining which snapshots it is in then insert row in InodeSnapshotMap table.In this way we capture in which snapshots a particular inode is in when it is being moved.

Logging modifications of files and blocks:
When we change any columns corresponding to the file in Inode table those were handled as mentioned above. If we append new blocks to or modify existing blocks then we should check whether to log them or not. This depends on whether this file is in any snapshot or not. So we follow the similar procedure as mentioned above.

Deletion of a file/or directory
When the issuer issues command to delete a file, first , we check whether it is in any snapshot, if not then we permanently delete it.When deleting a directory, if it is in snapshot,we mark all the children with isDeleted=1 and the background delete thread will do check on inodes , deleting those not present in any snapshot permanently.Consider /A/B/C/ suppose want to delete C. First get the list of snapshots on A,B as explained above then for each child in sub-tree rooted at C set isDeleted=1, also inserting rows in inodesMap table based on the join time of child with the list of snapshots on A,B.

Deleting entries in MovedPaths Table
When we delete a snapshot on an inode, we check for entries in InodeSnapshotMap with that can be deleted.

6 Implementation

We implemented the Read-Only root level single snapshot' roll back algorithm using clusterJ.The implementation guarantees progress even if the name-node crashes.
1. Take the write lock on root so that no subtree operations under root can be allowed after rollBack command issued.
2.Take read lock on all the inodes of the fileSystem, to make sure that there are no operations going on them while roll-back command issued.
3.Execute the four tasks, with a thread pool for each task with batch size of which can be configured, defaulted to 100,000 rows.
Task1: Delete the inodes with status=2 or status=3 in batch-size.
Task2: Set isDeleted=0 from inodes where isDeleted=1 in batch-sizes.
Task3: Set id=-id and parent_ id=-parent_ id where id¡0 in batch-sizes.
4.**Failure Handling** For Task1, Task2 and Task3 , when a NameNode starts executing it will insert a row in Transaction with Task Id , NameNode Id,status. If status is InProgress and NameNode is dead, then the leader will direct other namnode to execute that task. There is a progress after each failure followed taking over by another namenode.

7 Evaluation

The algorithm to list subtree of directory at a given snapshot was implemented in SQl and executed against MySql NDB cluster. In this benchmark Fig 2, a single directory with 1,000,000 files is used as test directory. Snapshot is taken when vector clock is 5000 i.e after completion 5000 operation. As we can see from 2 that time of execution of query scales linearly.This is the expected result for the algorithm to retrieve elements in a snapshot, which should be proportional to the number of changes since snapshot.

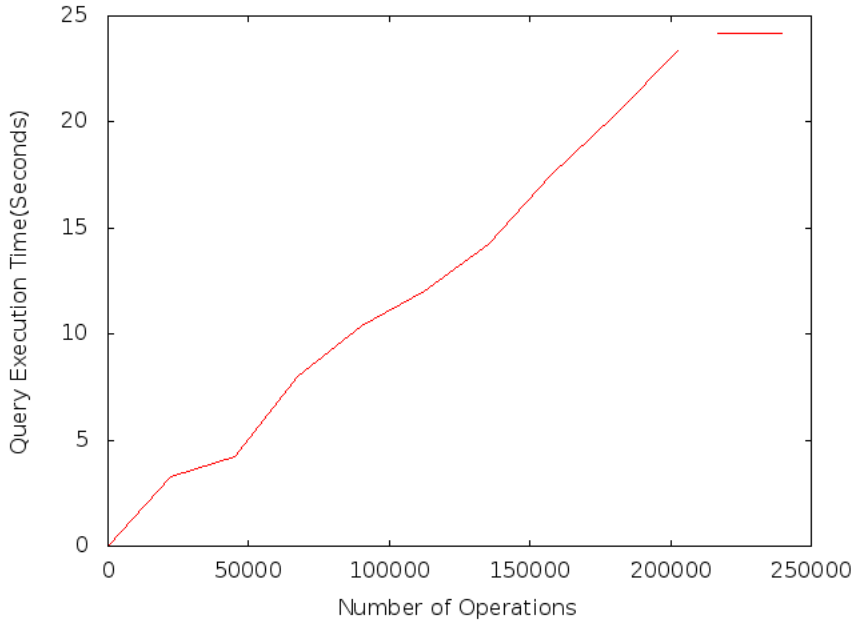

Fig. 2: Benchmark on Single Directory

The time to take snapshot is the time to insert a row in SNAPS table which is constant irrespective of size of file system. The time-overhead of taking snapshots on Hadoop version developed by Facebook Fig 3 is directionally proportional to the number of inodes(files/directories) in the fileSystem.

The roll back algorithm for read only root level single snapshot is implemented using a thread pool, where each thread processes given number of table records [100,000]. The implementation is executed at MySql server[7] as well as via directly connecting NDB-Cluster with ClusterJ[6].

As we can see that execution of RollBack-Algorithm at MySql Fig 5 taking much time because of update of column which is a primary key, in this case id,which we are changing from its negative value

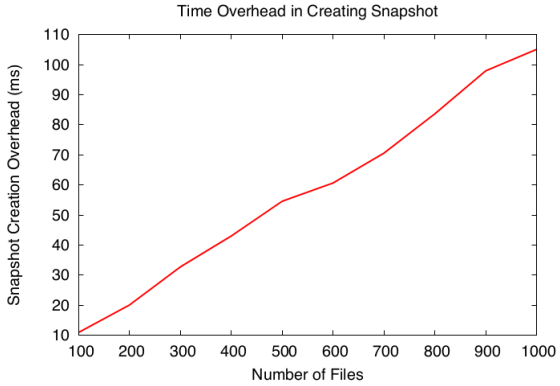

Fig. 3: Time Overheads in HDFS@Facebook

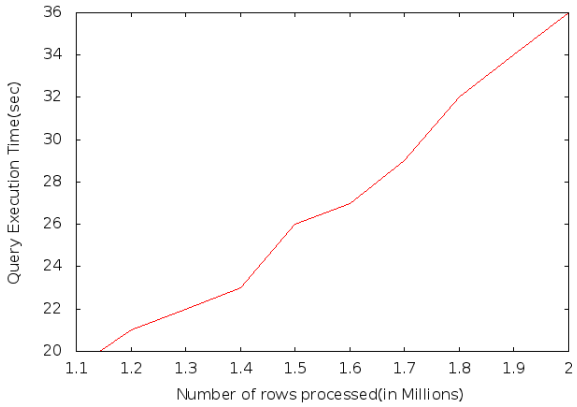

Fig. 4: Benchmark-Graph with ClusterJ

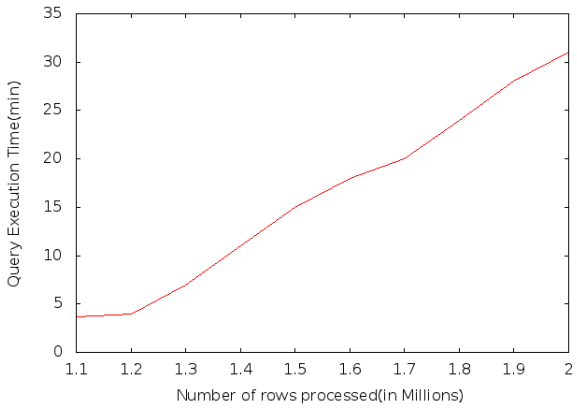

Fig. 5: Benchmark-Graph on MySqlServer

to positive value.In evaluations with clusterJ 4 batch size of rows were read, for each row a new row with id negative of the former row's id is inserted hence it is taking less time than MySQl Server.

8 Conclusion

We presented design and benchmark of algorithms for Read-Only Nested snapshots. Time to take snapshot is constant. Time to retrieve subtree in one of the snapshots at a directory is directly proportional to the number of operations executed in that directory after taking snapshot.We presented and implemented algorithms for Read-Only root level snapshot which is used in case of software upgrades

Future work

Following tasks are to be implemented and executed.

1. **Implementing Nested Snapshots:** Present work completely implemented Read only single snapshot and roll-back of it.More details analysis of code and algorithms presented for Read only nested snapshots has to be done to implement them.
2. **Integrating Read-Only Root Level SingleSnapshot and Read-Only Netsted Snpashot solutions:** We presented independent solutions for Read-Only Nested Snapshots and Read-Only Root Level Snapshots.It is effective to integrate both solutions by analysing and designing new algorithms.
3. **Roll-Backing to a particular snapshot:** We discussed how to retrieve subtree at a particular snpashot but didn't propose method to roll back to particular snapshot.At present we have some ideas to explore upon. We need to evaluate them by benchmarking.
4. **Length of file being written** The solution to Read only root level snapshot need to be enhanced to support snapshotting of files that are being written while snapshotting and writing to file takes place at the same time.

References

1. Apache-Hadoop."apche hadoop version 2". Accessed August 11, 2014.`http://hadoop.apache.org/docs/r2.0.6-alpha/hadoop-project-dist/hadoop-common/releasenotes.html`.
2. Facebook-Hadoop. "snapshots in hadoop distributed file system". Accessed August 11, 2014.`http://www.cs.berkeley.edu/~sameerag/hdfs_snapshots_ucb_tr.pdf`.
3. Hakimzadeh, K., H. Peiro Sajjad, & J. Dowling (2014). Scaling hdfs with a strongly consistent relational model for metadata. In K. Magoutis & P. Pietzuch (Eds.), "Distributed Applications and Interoperable Systems", Lecture Notes in Computer Science, pp. 38–51.Springer Berlin Heidelberg.
4. HopStart."hadoop open paas".Accessed August 2, 2014.`http://www.hopstart.org`.
5. Malik, W. R. (2012). "a distributed namespace for a distributed file system". Master's thesis, KTH.
6. Oracle-ClusterJ. "mysql clusterj overview". Accessed August 11, 2014.`http://dev.mysql.com/doc/ndbapi/en/mccj-using-clusterj.html`.
7. Oracle-MySql. "mysql cluster overview". Accessed August 2, 2014.`http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-overview.html`.
8. White, T. (2009). Hadoop: The Definitive Guide. OReily Media.