INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

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

Pushparaj Motamari

Dissertation for the degree of Master of Science in Distributed Computing

Jury

Presidente:	Prof. José Carlos Alves Pereira Monteiro
Orientador:	Prof. Luís Manuel Antunes Veiga
Vogal:	Prof. João Manuel dos Santos Lourenço

September 2014

European Master in Distributed Computing (EMDC)

This thesis is a part of the curricula of the European Master in Distributed Computing, a cooperation between KTH Royal Institute of Technology in Sweden,Instituto Superior Tecnico (IST) in Portugal and Universitat Politecnica de Catalunya (UPC) in Spain. This double degree master program is supported by Education,Audiovisual and Culture Executive Agency (EACEA) of the European Union.

My Study track during the master studies of the two years is as follows:

First Year: Instituto Superiot Tecnico,Universidade de Lisboa
Third Semester:KTH Royal Institute of Technology
Fourth Semester: SICS (Swedish Institute of Computer Science) and Instituto Superior Tecnico, Universidade de Lisboa

Acknowledgements

I would like to express my deepest gratitude to my supervisors Dr. Luis Veiga and Dr. Jim Dowling, who were a great source of inspiration and motivation. Also, I would like to thank my advisers Mahmoud Ismail and Salman Niazi for countless hours of support.

I would like to thank Instituto Superior Technico for providing me opportunity to learn and excel in distributed computing.Finally, I would like to thank Swedish Institute of Computer Science (SICS) for providing me with a nice working environment, all necessary compute resources, and great colleagues who always willing to exchange ideas.

Lisboa, October 3, 2014

Pushparaj Motamari

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 and implemented nested snapshots which enables multiple snapshots on any directory. We designed and implemented root level single snapshot by which roll-back during software upgrade 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 the snapshot was taken.

Resumo

A quantidade de dados armazenados em centros de dados modernos cresce rapidamente hoje em dia. Sistemas distribuídos de larga escala, que mantêm grandes conjuntos de dados em centros de dados, são projetados para trabalhar com hardware comum. Devido à qualidade e quantidade dos componentes de hardware nesses sistemas, as faltas são consideradas eventos normais e, como tal, os sistemas distribuídos de ficheiros são projetados para ser altamente tolerante a faltas.

Uma realização concreta de um tal sistema é o Hadoop Distributed File System (HDFS) . Um snapshot consiste em capturar o estado do sistema de armazenamento num ponto exacto no tempo e pode ser utilizado para permitir a recuperação total dos dados quando ocorre uma falha. As aplicações manipulam os dados no sistema de ficheiros distribuído em nome de utilizadores ou administradores. Erros ao nível aplicacional ou até memso dos utilizadores podem remover informação por engano ou modificar dados de uma forma inesperada. Neste caso, os snapshots podem ser utilizados posteriormente para recuperar o sistema com o estado de um ponto anterior. Estes podem ser usados no treino de modelos, em anaálise de dados em tempo real, e também para backups rápidos (Hot Backups) .

Desenhámos e realizámos um mecanismso de snaphsots aninhados que permite vários snapshots em qualquer pasta. O snapshot de nível raiz permite o roll-back durante a actualização de software. Avaliámos os nossos mecanismos e algoritmos, demonstrando tempo para tirar um snapshot é constante e o tempo de roll-back é proporcional à quantidade de modifcações desde o snapshot.

Keywords

Hadoop

HDFS

Distributed FileSystem

Snapshots

HOPS

Palavras Chave

Hadoop

HDFS

sistema de ficheiros distribuído

Snapshots

HOPS

Index

List of Figures

List of Tables

1 Introduction

1.1 Overview

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) revealed that 34% of the surveyed companies were applying methods of big data analytics, whereas 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 running analytics, 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 roll back to previous state of data, if something does not work.

1.2 Problem Definition

During software upgrades the possibility of corrupting the filesystem due to software bugs or human mistakes increases. This invites a solution to minimize potential damage to the data stored in the system during upgrades.We need to take a snapshot at root of the file system before proceeding with the upgrade.If upgrade didn't work then administrator can roll back the system to the snapshot.The snapshot can be taken at any time and can be rolled-back to at any time. Following scenarios also 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.

1.3 Goals

Following goals are desired from the solution

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.

1.4 Contributions

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 operations on snapshotted directories are logged in an efficient manner to retrieve the metadata 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.

1.5 Structure of the thesis

The remaining of this 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 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 and Related Work

2.1 Hadoop File System(HDFS)

Apache Hadoop (White 2009) is an open-source software framework for large-scale data processing. It includes a distributed file system called the Hadoop Distributed File System (HDFS) and a framework for MapReduce. Many data analysis, data warehousing and machine learning solutions have been built on top of it. The most commonly known extensions of Hadoop are Apache Pig (Foundation c), Apache Hive (A.Thusoo et al. 2009), Apache HBase (Foundation a), Apache Zookeeper (Foundation d) and Apache Mahout (Foundation b). Recent version of Hadoop also include a resource negotiator called Yet-Another- Resource-Negotiator (YARN), often also referred to as NextGen MapReduce or short MRv2. YARN is, inter alia, used to execute MapReduce jobs. The design and concepts used by Hadoop are inspired by the Google papers about GFS and MapReduce (White 2009, p. 9). Similar to MapReduce on GFS, Hadoop is exploiting data locality for MapReduce jobs by trying to execute map jobs on a DataNode which hosts the data. If not possible, the framework will attempt to execute the job on a node close to the location of data, for instance on the same rack. This can greatly improve the overall performance [28] and reduces the network bandwidth requirements.// HDFS is Hadoop's distributed file system which has been designed after Google File System.It was initially created to be used in a Map-Reduce computational framework of Hadoop by Apache though later on it started to be used for other big data applications as a storage which can support massive amount of data on commodity machines.Hadoop File System were intended to be distributed for being accessed and used inside by distributed processing machines of Hadoop with a short response time and maximum parallel streaming factor. On the other hand, in order for HDFS to be used as a storage of immutable data for applications like Facebook, the high availability is a key requirement besides the throughput and response time.Moreover, as a file system to be compliant to the common file system standard, it provides posix like interface in terms of operations, however it has a weaker consistency model than posix which is being discussed

later on in this section.

2.1.1 HDFS Architecture

HDFS splits up each file into smaller blocks and replicates each block on a different random machine. Machines storing replicas of the blocks called DataNode. On the other hand since it needs to have namespace metadata accessible altogether, there is a dedicated metadata machine called NameNode. For having fast access to metadata, NameNode stores metadata in memory. Accessing to HDFS happens through its clients, each client asks NameNode about namespace information, or location of blocks to be read or written, then it connects to DataNodes for reading or writing file data. Figure2.1 shows the deployment of different nodes in HDFS.

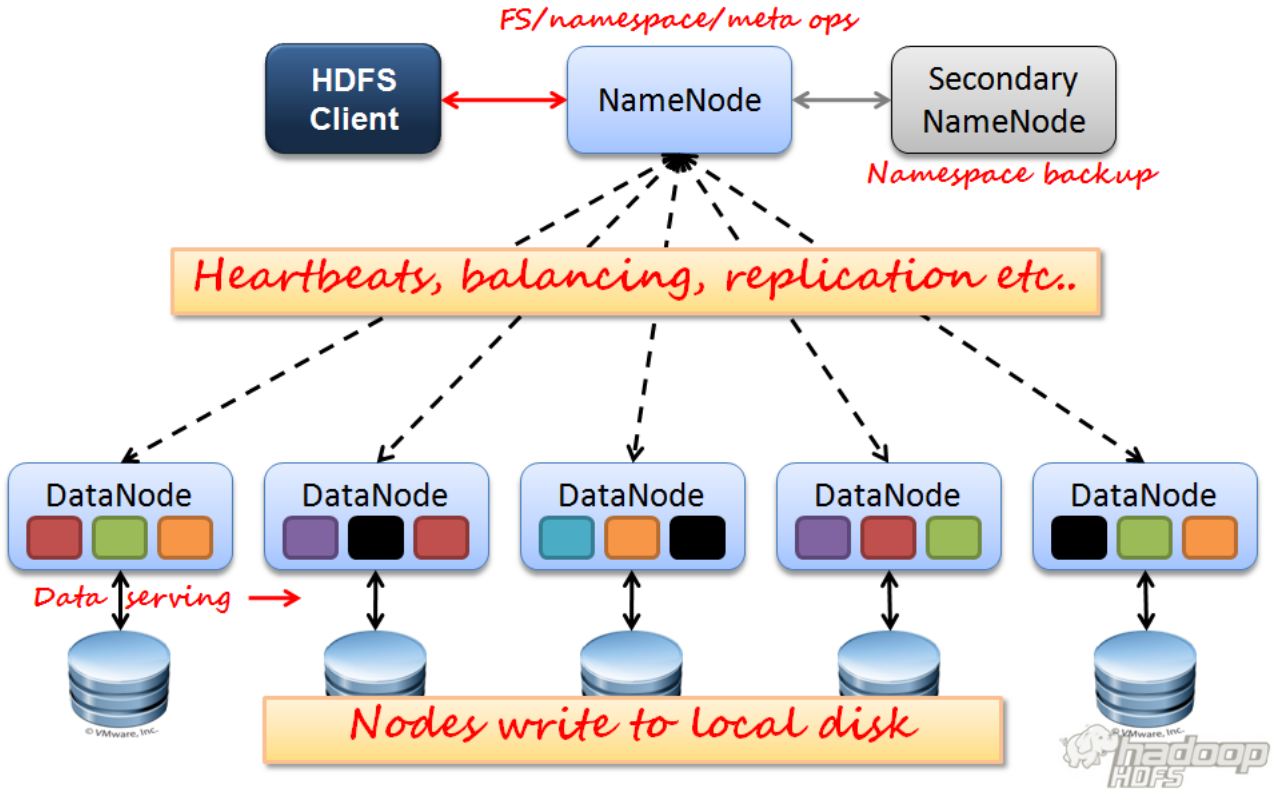

Figure 2.1: HDFS Architecture.

2.1.2 HDFS NameNode

NameNode is known as metadata server of HDFS. Its multithreaded server in which size of the thread pool is configurable. It keeps all metadata information in memory which is described

in the next section. The way NameNode protects race condition for metadata modification is based on read/write lock. It splits all operations into read or write operations.Its procedure is shown in alogithm 1.In this way multiple read operations could be run in parallel though they are serialized with each single write operation. Other than serving client's requests, NameNode has been serving part for DataNodes, via this service. DataNodes notice NameNode about receiving or deletion of blocks or they send over list of their replicas periodically. Moreover, NameNode has one still running thread namely ReplicationMonitor to get under-replication and over-replication under its radar and plans for deletion/replication accordingly. Moreover, LeaseMonitor controls the time limit that each client holds the write operation of files. So it walks through all leases and inspect their soft-limit/hard-limit and decides to recover or revoke an expired lease.

Algorithm 1 System-Level locking schema in HDFS

```
Operation lock
if op.type = write then
    ns.acquireWriteLock()
else
    ns.acquireReadLock()
end if

Operation perform Task
//Operation body

Operation unlock
if op.type = write then
    ns.releaseWriteLock()
else
    ns.releaseReadLock()
end if
```

2.1.3 HDFS consistency model

1. **FileSystem Operations**

   In general most of the distributed file systems like GFS and HDFS have a relaxed version of consistency because of the impossibility result of CAP theorem (Gilbert & Lynch 2002) which limits scalability of file system. Even though some works refer to HDFS as sequential consistent file system for data and from filesystem operations point of view, it does not certainly have sequential consistency due to nonatomic write operation. HDFS seri-

alizes read/write operations just at the primitive operations' level not the files blockdata. As each write operations consists of multiple micro addBlock operations which makes it unsortable when multiple parallel reads are being performed with one write. Though it protects multiple writes by means of a persistable mechanism called lease.

2. **Primitive NameNode Operations**
   From primitive operations point of view, HDFS is strongly consistent in both data and metadata level. From data level it is strongly consistent because each file's block is not available for read unless it gets completely replicated. It means write operation should be completely finished first, then readers will all get the same version of that block and there is not case of version mismatch in the replicas read by two different readers. From metadata level, as already been mentioned, system level lock serializes all the write operations which results in mutated state of all writes bing available for all readers.

2.1.4 POSIX compliant filesystem

POSIXfs is file system part of POSIX operating system. It has been being a standard for designing filesystems. It is about naming, hardlinks, access control, time stamping and standard folder hierarchy. Under POSIX standards, almost all file operations shall be linearized. Specifically all read operations should have effects of all previous write operations. HDFS is not fully POSIX compliant, because the requirements for a POSIX file system differ from the target goals for a Hadoop application. The tradeoff of not having a fully POSIXcompliant file system is increased performance for data throughput and support for nonPOSIX operations such as Append. Moreover, HDFS consistency model is weaker than POSIX. HDFS is strongly consistent from primitive HDFS operations while from filesystem operations it has a relaxed version of consistency, on the other hand, POSIX filesystem operations are linearizable which is the highest level of consistency.

2.2 Hadoop Open Platform as Service(HOP)-HDFS

Hadoop Open Platform as a service (HOP) (HopStart) is a Hadoop distribution based on Apache Hadoop . It provides namespace scalability through the support of multiple NameNodes, platform as a service support for creating and managing clusters, and a dashboard for

simplified administration. HOP is developed in cooperation of KTH and SICS (SICS) HOP-HDFS(Malik 2012) (Sajjad 2013) is a fork of HDFS and part of HOP. It aims on providing high availability and scalability for HDFS. This is achieved by making the NameNode stateless and thereby adding support for the use of multiple NameNodes at the same time. Instead of storing any state in the NameNode, the state is stored in a distributed database offering high-availability and high-redundancy. Therefore, the current implementation uses MySQL Cluster (Oracle-MySql), which utilizes NDB Cluster as an underlying storage engine. HOP-HDFS is a promising approach that could make HDFS similar to Colos-sus, while overcoming the scalability and availability limitations of the current Hadoop implementation. Through its support for larger amounts of metadata, it could also make the use of block sizes smaller than 64 megabytes efficient, what might be useful for many applications.

2.2.1 HOP-HDFS Architecture

The persistent data structures of HOP-HDFS (here after referred as HDFS) are defined as 11 database tables. These tables contain all the information about namespace,metadata,block locations and many other information that name-node in HDFS stores in FSImage and keeps in memory.

1. **inodes:** The table representing inode data structure in HDFS which contains the namespace and metadata of the files and directories. inodes are related together by their parent_id and resembles a hierarchical namespace as in the HDFS. Each row has a unique id which is the primary key.

2. **block_inofs:** Block is a primitive of HDFS storing a chunk of a file, block-info is its metadata keeping a reference to its file-inode, the list of block's replica which are scattered among multiple data-nodes.

3. **leases:** Basically each file in HDFS is either underconstruction or completed. All underconstruction files are assigned a sort of write lock to them, this lock is persisted in database. Each lease corresponds to just one client machine, each client could be writing multiple files at a time.

4. **lease_path:** Each lease path represents an underconstruction file, it holds full path of that file and points to the lease as its holder.

5. **replicas:** A copy of a Block which is persisted in one datanode, sometime we refer to replicas as blocks. All the replicas of the same block points to the same blockinfo.

6. **corrupted_ replicas:** A replica become corrupted in the copy operations or due to the storage damages. Namenode realizes this by comparing checksum in the report of the replica's datanode with the checksum of the original block.

7. **excess_ replicas:** A block could become over replicated because of an already dead datanode coming alive again and contain some replicas which has been removed meanwhile from namenode. So distinguishing that, namenode marks marks some replicas to be removed later on.

8. **invalidated_ blocks:** For every datanode keeps a list of blocks that are going to be invalidated(removed) on that datanode due to any reason.

9. **replicas_ under_ construction:** Replications of a block which are being streamed by client into datanodes.

10. **under_ replicated_ blocks:** Keeps track of the blocks which has been under replicated, it realizes the priority of under replications as follow. Priority 0 is the highest priority. Blocks having only one replica or having only decommissioned replicas are assigned priority 0. Blocks having expected number of replicas but not enough racks are assigned with priority 3. If the number of replicas of a block are 3 times less than expected number of replicas then the priority is assigned to 1. The rest of low replication cases are assigned priority 2. Blocks having zero number of replicas but also zero number of decommissioned replicas are assigned priority 4 as corrupted blocks.

11. **pending_ blocks:** Represents a blocks that are being replicated.

The figure 2.2 illustrates the relation between tables. The figure 2.3 gives the columns stored in each table.

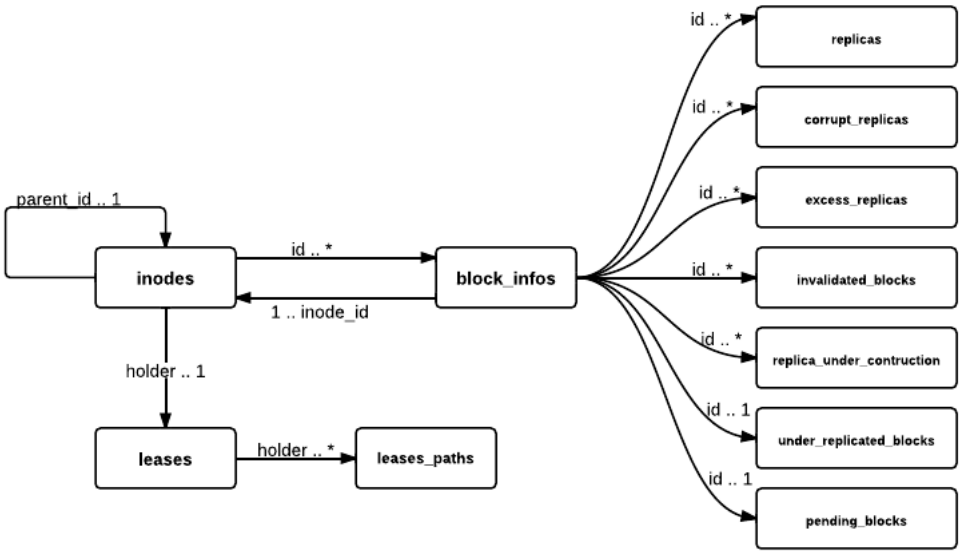

Figure 2.2: HOP-HDFS Table relations.

2.2.2 NameNode Operations

Every operation defined in the HDFS client API (such as createFile, open, etc) maps onto one or more of the following primitive HDFS operations. Each operation defined in the primitive HDFS API maps onto a protocol message (where each protocol message contains request, reply, and exception parts) sent between the NameNode, client, and DataNodes. Some common primitive operations are shown in the table 2.1. The full list of the primitive operations can be found in Thesis report (Sajjad 2013) Appendix section.

2.2.3 HOP-HDFS Implementation

In HOP-HDFS each HDFS opeartion is implemented as a single transaction,where after transaction began, read and write the necessary meta-data from NDB , and then either commit the transaction, or in case of failure, the transaction was aborted and then possibly retried. However, the default isolation level of NDB is read committed, which allows the results of write operations in transactions to be exposed to read operations in different concurrent transactions. This means that a relatively long running read transaction could read two different versions of data within the same transaction, known as a fuzzy read, or it could get different sets of results if the same query is issued twice within the same transaction this is known as a phantom read. In report(Sajjad 2013) and paper (Hakimzadeh et al. 2014) they proposed and implemented

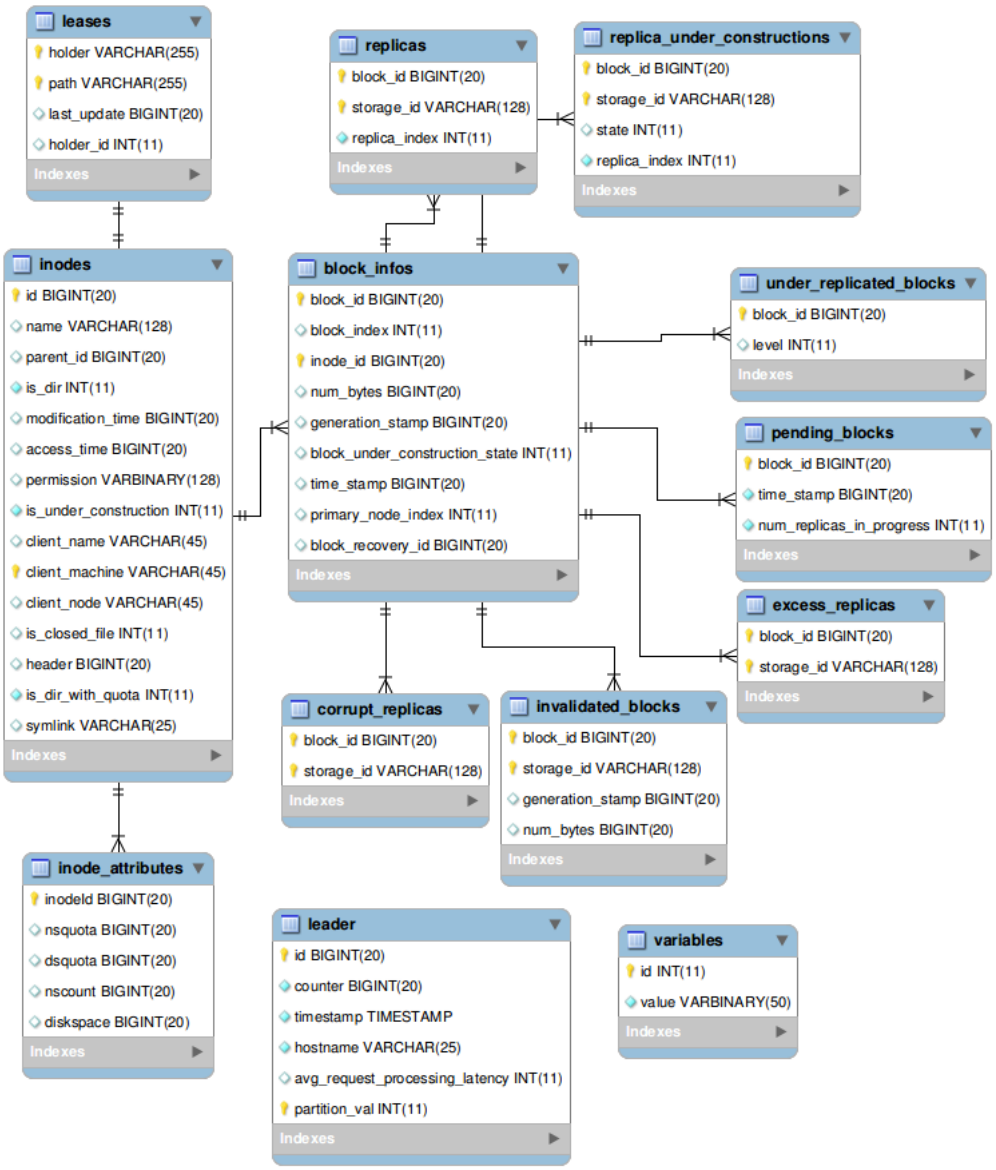

Figure 2.3: HOP-HDFS Schema

OPERATION	SUMMARY
MKDIR	Creates a directory recursively, it requires a no lock on all the existing components of the path but write lock on the last existing.
START_FILE	1. If file does not exist, It creates inodes for all the nonexistent directories and new file, writes owner of the lease and creates new leasepath. 2. If file already exists first removes the file, its blocks and dependencies, lease and lease path, then it does the first scenario.
GET_ADDITIONAL_BLOCK	In the middle of writing a file, this is the client's mean of noticing namenode that the already being written block is finished while it is asking for the locations of next block. NameNode removes all the replicaunderconstructions of last block, it also changes type of blockinfo from underconstruction to completed one.
COMPLETE	Like get_additional_block, it happens for last block of the file, NameNode just removes the replicaunderconstructions and changes type of blockinfo from underconstruction to completed.
GET_BLOCK_LOCATIONS	Given path of the file, it returns location if its blocks and updates accesstime of the fileinode.
DELETE	Delete the given file or directory from the file system.
RENAME	Renames gives SRC to DST. Without OVERWRITE option, rename fails if the dst already exists. With OVERWRITE option, rename overwrites the dst, if it is a file or an empty directory. Rename fails if dst is a nonempty directory. The rename opeartion is atomic.
APPEND	Append to the end of the file.It retuns the partially completed last block if any.

Table 2.1: NameNode's Operations

the snapshot-isolation method 2 which pessimistically locks the rows of data preventing other transactions from accessing. Transactions that contain both a read and a modify filesystem operation for the same shared metadata object should be serialized based on the serialization rule:

$-\forall(w_i, w_j) \quad if \quad X_{wi} \cap X_{wj} \neq \phi$ then transactions of (w_i, w_j) must be serialized;
$-\forall(r_i, w_j) \quad if \quad X_{ri} \cap X_{wj} \neq \phi$ then transactions of (r_i, w_j) must be serialized.

First, the hierarchy of the file system to define a partial ordering over inodes. Transactions follow this partial ordering when taking locks, ensuring that the circular wait condition for deadlock never holds. Similarly, the partial ordering ensures that if a transaction takes an exclusive lock on a directory inode, subsequent transactions will be prevented from accessing

the directory's subtree until the lock on the directory's lock is released. Implicit locks are required for operations such as creating files, where concurrent metadata operations could return success even though only one of actually succeeded. For operations such as deleting a directory, explicit locks on all child nodes are required.

Algorithm 2 Snapshotting taking locks in a total order

```
snapshot.clear

Operation doOperation

tx.begin
create-snapshot()
performTask()
tx.commit

Operation create-snapshot

S = total_ order_ sort(op.X)
for all x in S do
    if x is a parent then
        level = x.parent_ level_ lock
    else
        level = x.strongest_ lock_ type
        tx.lockLevel(level)
        snapshot <- tx.find(x.query)
    end if
end for

Operation performTask
//Operation Body,referring to transaction cache for data
```

2.3 MySQL Cluster

Mysql Cluster is a Database Management System (DBMS) that integrates the standard Mysql Server with an inmemory clustered storage engine called NDB Cluster (which stands for "Network DataBase") . It provides a sharednothing system with no single point of failure.

Mysql Cluster is a compound of different processes called **nodes**. The main nodes are Mysql Servers (mysqld, for accessing NDB data), data nodes (ndbd, as the data storage), one or more management servers (ndb_ mgmd). The relationship between these nodes are shown in figure 2.4. The data in Mysql Cluster is replicated over multiple ndbds so this makes the

database to be available in case of node failures. Ndbds are divided into **node groups**. Each unit of data stored by ndbd is called a **partition**. The partitions of data are replicated into ndbds of the same node group while node groups are calculated indirectly as following:

$$Numberof Nodegroups = \frac{numberofdatanodes}{numberofreplicas}$$

A simple cluster of 4 datanodes with replication factor of 2 and consequently 2 node groups are shown in figure 2.5. As it can be seen, the data stored in the database are divided into 4 partitions. There are two replicas of each partition into ndbds of the same node group. So even if one of the ndbds in each of the node groups are failed, the whole data in the cluster will be available. However, if both ndbs in a node group become unavailable then those partitions stored by the failed ndbs also will become unavailable. According to a white paper published by Oracle , Mysql Cluster can handle 4.3 billion fully consistent reads and 1.2 fully transactional writes per minute. They used an open source benchmark called flexAsynch and a Mysql Cluster of 30 data nodes, comprised 15 node groups. The detail of their system configuration is available in the referenced white paper. The results for write operations are shown in figure 2.4. The 72 million reads and 19.5 million write operations per second of Mysql Cluster shows that it has a high throughput for simple read and write operations.

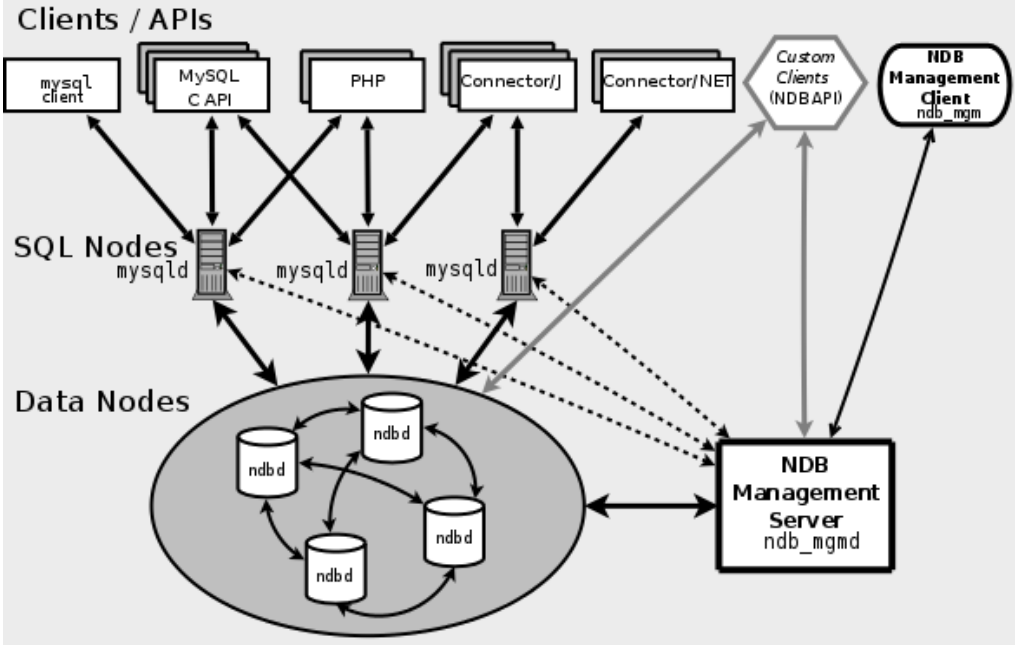

Figure 2.4: MySQL cluster

2.3.1 Concurrency Control in NDBCluster

NDB supports pessimistic concurrency control based on locking. It supports row level locking. NDB throws a timeout error if a requested lock cannot be acquired within a specified time (MySql). Concurrent transactions, requested by parallel threads or applications, reaching the same row could end up with deadlock. So, it is up to applications to handle deadlocks gracefully. This means that the timed out transaction should be rolled back and restarted. Transactions in NDB are expected to complete within a short period of time, by default 2 seconds. This enables NDB to support realtime services, that are, operations expected to complete in bounded time. As such, NDB enables the construction of services that can failover, on node failures, within a few seconds ongoing transactions on the node that dies timeout within a couple of seconds, and its transactions can be restarted on another node in the system.

2.3.2 ClusterJ

Clusterj is Java connector implementation of NDB Cluster, Mysql Cluster's storage engine, (Oracle-MySql). Clusterj uses a JNI bridge to the NDB API to have a direct access to NDB Cluster. The NDB API is an application programming interface for Mysql Cluster that implements indexes, scans, transactions and event handling. Clusterj connects directly to NDB Clusters

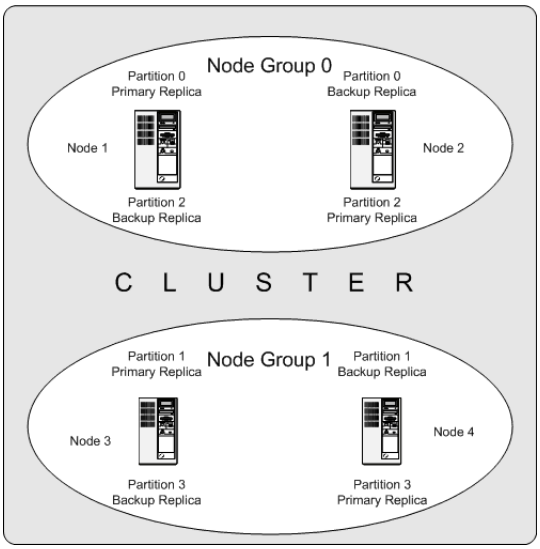

Figure 2.5: Node groups of MySQL cluster

instead of connecting to mysqld. It is a persistence framework in the style of Java Persistence API. It provides a data mapper mapping java classes to database tables which separates the data from business logic.

2.4 Related Work

2.4.1 Snapshots in Apache Hadoop Version2

Apache Hadoop implemented snapshots in their latest version (Apache-Hadoop) , which supports 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 couldn't 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 administrators 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 and storage state as they were at the time of the snapshot.

The snapshot (only one can exist) is created at the cluster administrator's 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.

The cluster administrator can choose to roll back HDFS to the snapshot state when restarting the system. The NameNode recovers the checkpoint saved when the snapshot was created. DataNodes restore the previously renamed directories and initiate a background process to delete block replicas created after the snapshot was made. Having chosen to roll back, there is no provision to roll forward. The cluster administrator can recover the storage occupied by the snapshot by commanding the system to abandon the snapshot; for snapshots created during upgrade, this finalizes the software upgrade.

System evolution may lead to a change in the format of the NameNode's checkpoint and journal files, or in the data representation of block replica files on DataNodes. The layout version identifies the data representation formats, and is persistently stored in the NameNode's and the DataNodes' storage directories. During startup each node compares the layout version of the current software with the version stored in its storage directories and automatically converts data from older formats to the newer ones. The conversion requires the mandatory creation of a snapshot when the system restarts with the new software layout version.

2.4.2 Snapshots in Hadoop at Facebook

Facebook has implemented a solution to Snapshots (Facebook-Hadoop). 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 lim-

its 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.

3 Solution

3.1 Operations to Support

1. All file system operations are allowed on the snapshotted file/directories including: Rename is allowed both within and across snapshottable directory boundaries.
2. Any modification to the current files or directories are not reflected in the snapshots. The snapshot is read-only.
   The modification include length change, renaming of the file name, permission changes or any other attribute changes such as replication factor etc.
3. The snapshot files and directories can be only be read, not modified in any way (except that the entire snapshot can be deleted). This means the replication factor, permission or any other attributes of file or directory cannot be changed. The snapshots are truly read-only.
4. Nested snapshots are allowed.
5. Access time is not tracked for snapshots.
6. Snapshots should be created very fast.
7. Block data is not copied for snapshots

3.2 Read-Only Nested Snapshots

3.2.1 Snapshottable Directories

These are directories that are configured by the system administrator to allow snapshots. A snapshot can be created only at these snapshot roots instead of at arbitrary directories. Directories that are marked snapshottable cannot be deleted until all the snapshots under that

directory are deleted. Similarly, another directory cannot be renamed to an existing a snapshottable directory that has snapshots (since rename involves deletion of the rename target). The above restrictions simplify the design by not having to deal with how to mange a snapshot when the snapshottable directory is deleted and no longer exists or worst still a new directory with the same name is created in its place.

3.2.2 Modifications to the Schema

Following columns need to be added to the Inodes table described in the schema 2.3 of HOP File System.

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 2.3.

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.

3.2.3 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]

Example

Consider the a small file-system tree.

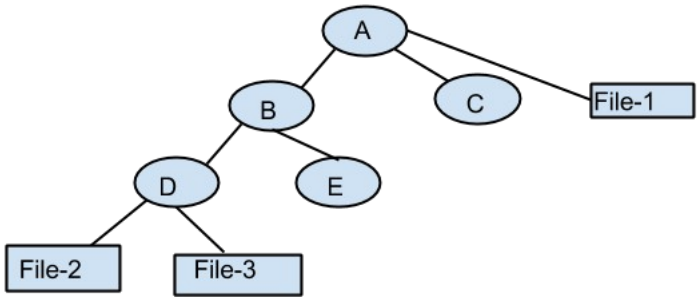

Figure 3.1: Sample File System Tree

ACTIONS

Directory	Opeartion	Time	Comments
A	Take Snapshot on A	1	
E	Create file F4	2	Insert row in Inodes table for F4 with parent E
E	Create file F5	3	Insert row in Inodes table for F5 with parent E.
D	Delete File F2	4	1)Insert row in d-list. 2)set isDeleted=1 on Inodes table.
A	Take Snapshot on A	5	
D	Create File F6	6	Insert row in Inodes table for F6 with parent D
D	Move File F1 to C	7	1)Insert row in MV-LIST. 2)Insert row in MV-IN-LIST Table. 3)Change the parent_id to 3 for F4 in inodes table.

Table 3.1: Operations

SNAPS

Inode_ Id	User	Snapshot_ id	Time
A	Admin	SA1	1
A	Admin	SA2	5

Table 3.2: List of Snpashots taken

C-List

Inode_ Id	Time	Created_ Node_ Id
E	2	F4
E	3	F5
D	6	F6

Table 3.3: C-List

D-List

Inode_ Id	Time	Deleted_ Node_ Id
D	4	F2

Table 3.4: Dlist

MV-IN-List

Inode_ Id	Time	MovedIn_ Node_ Id
C	7	F1

MV-List

Inode_ Id	Time	Moved_ Node_ Id	Original Row
A	7	F1	Orignal Row before moving

Table 3.5: MV List

3.2.4 Listing children under a directory in a given Snapshot

To get the subtree under a directory at a particular snapshot, we use the following pseduo algorithm called with directory id and the time at which snapshot was taken. The algorithm presented here gives a general idea, the implementation in sql(Snapshot-ls) addresses/solves the fine grained issues, for example, inode created after snapshot, moved, then moved-in, like those cases.

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

    children={Get children whose parentId=id};
    children = childen - { children deleted before stime} -{ children created af
                       - {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
                inode = modChd;
            }
            else{
               inode = chd;.
            }
        }
```

```
        else{
              inode = chd;
        }

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

3.2.5 Listing current children under a directory

For listing current subtree under a directory, we select children which are not deleted.

```
void ls_current(int id){
    children={Get children whose parentId=id};
    children = childen - { children isDeleted=1};

    for(child chd : children){
         if(inode is directory){
             ls_current(inode.getId());
        }
        else{
            print(inode);
        }
    }
}
```

3.2.6 Logging, Removing logs and Deleting inodes which are not referred by any snapshot

Here two approaches two solve the issues are presented.

3.2.6.1 Approach 1:

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 3.6: 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.

3.2.6.2 Approach :2

When snapshot is taken we place the inodes under the snapshot in below table.

Inode_ Id	Snapshot_ time

if an inode with isDeleted=true and there is no entry in the above table , then we can remove that file from HDFS permanently.

The tables M-List, D-List, C-list , MV-list and MV-IN-List are populated for a directory when it

is in a snapshot means an entry can be found in the above table.

Cleaning the logs when a Snapshot is Deleted

When a snapshot is deleted, all inodes under that snapshot can delete their logs on a criteria explained shortly. 1,2..numbers represent files in a directory P. S1,S2,S3 represent snapshots taken in increasing chronological manner. As per the query to list files at a certain snapshot, say S2 , we get all inodes from inodes table whose parent is P then remove all the files created after taking snapshot and adjust those which are moved out , modified. Then remove files deleted before taking snapshot

to get files at S2 for directory P ==> {1,2,3,4,5,6,7,8} - {3,4}-{7,8} ={1,2,5,6} It means, a snapshot at time T1requires logs in C-List, M-List, MV-List, Mv-In-List after time T1 and logs in D-List before T1.

When we delete a snapshot S , then for each inode under it we execute following algorithm.

ALGORITHM

if(S is first snapshot in which this inode is present when all snapshots in which it is present are arranged in chronological manner) then;
delete D-List Logs before S. delete logs in C-List, M-List, MV-List, Mv-In-List until next snapshot.

For example: If we delete S1, then delete logs in D-List before S1, and logs in C-List, M-List, MV-List, MV-IN-List in between S1 and S2.

Deleting an file/Inode

If the file is with isDeleted=1 and there is no entry for it in above table then it can be deleted permanently. After deleting file, we will check in D-List, to see if there are any logs with this Inode, if there then we will delete them.The same applies for directory. When user issues a command to delete a directory then mark isDeleted=1 for all of its children.Each children is an Inode, so we If the Inode is with isDeleted=1 and there is no entry for it in above table then it can be deleted permanently. After deleting Inode, we will check in D-List, to see if there are any logs with this Inode, if there then we will delete them

Handling the replication factor change of a file

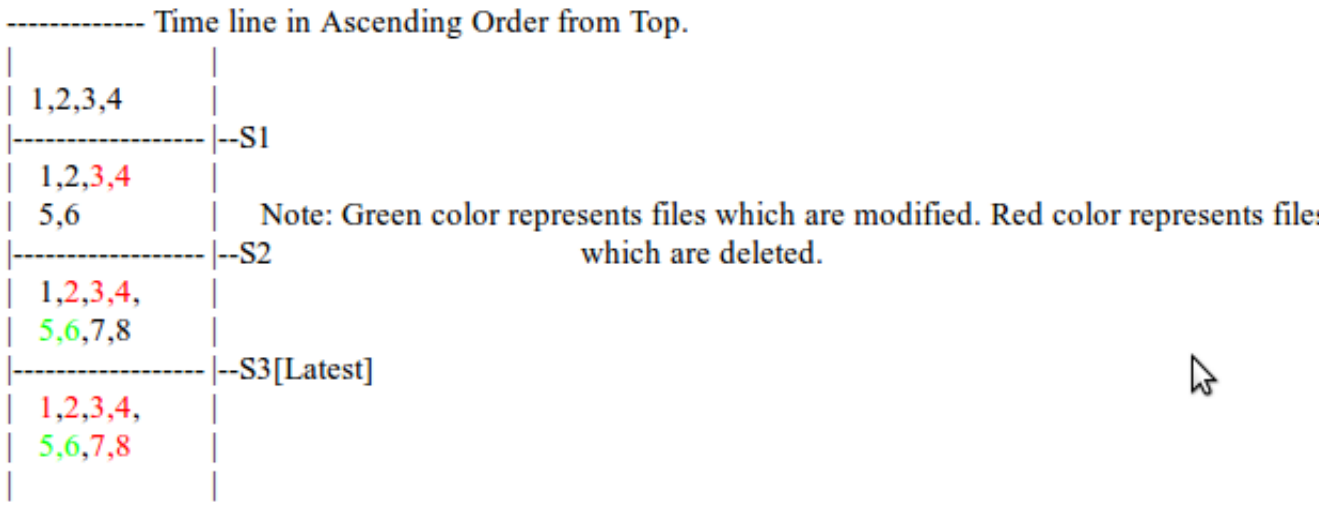

Figure 3.2: Deletion of Snapshot

Since we keep latest information about an Inode in Inodes' table, we need to a mechanism to handle the case of replication factor changes. For example, in S1 the replication factor is 3 , then changed to 6 and S2 was taken , then changed to 9 , then S3 was taken, then changed to 2. We find value 2 in Inodes' table. The replication factor= Max(Current value, Max(values in M-List for this Inode)). In this case we will find it as 9 and we expect block report mentioning replication factor of 9 for each block.Suppose S3 was deleted , then the row with value 9 is deleted and we find maximum value 6.

Disadvantages:

1. Time for taking snapshot is O(n) where n is the number of descendants in directory
2. The space overhead on taking snapshot is o(n) where n is the number of descendants in directory.

3.2.7 Length of file being Written

This section explains the case when snapshot was taken while writing to file is in progress.In HDFS file System metadata and actual data are separated and store at different places.Various options to know the length of the file are:

1. Length of all completed/finalized blocks and zero length considered from block under construction.

2. Length when block is closed after snapshot creation occurred.
3. Length as seen at or around the time of snapshot creation in NameNode.
    - Either, DataNode reports the length in heartbeat or other commands
    - Or, NameNode queries the length
4. Namenode and Datanodes could have communication to establish the snapshot length after snapshot command is received by the namenode. But this is very complicated to implement because of the distributed nature and the failure modes.
5. Client that care about the length of a file being written, could update length to the namenode as part of hflush/hsync. This is the length that gets recorded in the snapshot.
6. "Precise length" of the file at the time of snapshot creation.

Clearly (6) is not possible given namespace and data block storage separation. Any guarantees in this regard will not be useful for the application, given that an application will need this consistency across many files that are part of the snapshot.
(1) Does not capture any changes that has occurred in the file from the time of new block creation to the time of snapshot command. This is a good choice only if an application is willing to close the files before creating snapshots. That means application must have control over the data being written to the files and also snapshot creation. This may not be useful for HBase.
(2) will not be acceptable for many applications. Some reasons being - a file could be slowly written to and may not close for a long time. What is the length of such files in snapshot? Secondly, length of the file changes post snapshot - what guarantee is HDFS providing in terms of snapshot creation time?
(5)has the advantage that only applications that care about the correct length to be recorded in a snapshot use the hflush/hsync operation to report the length to the NN.

In both Read only Root Level Snapshot and Read only Nested Snapshots, (5) is chosen to implement.

3.3 Read-Only Root Level Single Snapshot

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.

3.3.1 Modifications to the Schema

Following columns need to be added to the Inodes table described in the schema 2.3 of HOP File System.

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 2.3.

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.

3.3.2 Rules for Modifying the fileSystem meta-data

Following rules apply when client issues operations described on 2.1 after root level snapshot had been taken.
HOP-HDFS as well as Apache HDFS allow only appends at the end of file.Both allow over-writing of an existing file.

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.

```
void deleteWithSnapshotAtRootTaken(INode targetNode){

        Stack<INode> stck = new Stack<INode>();
        //parentStck is used to track completion of processing of a director
        Stack<INode> parentStck = new Stack<INode>();
        INode tempNode;
        List<INode> children;
        INode[] inodesTemp;
        INode removedInode;

        stck.add(targetNode);
        atomic{
           targetNode.setIsDeleted=1;
        }
        while (!stck.empty()) {
            tempNode = stck.pop();
            tempSts = tempNode.getStatus();
            tempStr = tempNode.getFullPathName();


            /*
             * This Inode can be a directory or file also it can be new or
             modified or original.
             */
            if (tempNode instanceof INodeDirectory) {
                if(parentStack.top().equals(tempNode)){
                    //Processing children is completed
                    parentStack.pop();
                    if(tempNode.getStatus()==SnapShotConstants.New){
                         //delete completely this directory Inode
                        EntityManager.remove(tempNode);
                    }
```

```
                }

                else{
                    parentStack.push(tempNode);
                    children = ((INodeDirectory) tempNode).getChildren();
                    if (children != null && !children.isEmpty()) {
                        stck.push(tempNode);
                        for (INode n : children) {
                            stck.push(n);
                        }
                    }
                }
            } else if (tempNode instanceof INodeFile ||
                         tempNode instanceof INodeSymlink) {

                if (tempSts == SnapShotConstants.New) {
                    atomic(In Single-Transaction){
                     //We can delete this file permanently and
                     //update the ancestors about changes in the quota.
                     //Remove the blocks associated with this file
                     // permanently.
                     }
                }
              }

       }// End of while loop



    }//End of method
```

6. Renaming/Moving an INode(File or Directory)

```
void renameINode(INode src, INode dst){
        1. Update the modification time of parent of src.
        2. Update the modification time of parent of dst.
        3. deletWithSnapshotAtRootTaken(dst).
        4. Change the parent\_ id of src to dst.parent\_ id.
}
```

3.3.3 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

3.4 Implementation Details

3.4.1 RollBack Algorithm Implementation

With ClusterJ it is implemented as below.

1. Take the write lock on root so that no subtree operations under root can be allowed after

rollBack command issued.

1.1 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.

```
DeleteRunnable(int start, int end) {
    void run(){
        //delete from inodes where status=2 or status=3 and id>=start
        and id<=end;
        }
}

UpdateRunnableForCoulmnIsDeletd(int start, int end){
    void run(){

        //update inodes set isDeleted=0 where id>=start and
        id<=end  and isDeleted=1;
        }
}

UpdateRunnableForColumnId(int start, int end,Lock lock){
void run(){
    List<INode> oldRows =  select * from inodes where id<=end and id>=start;
    List<INode> newRows = new ArrayList<INode>(oldRows.size());
        for(INode row: oldRows){
            INode updatedRow = session.instanceOf(INode.dto);
            updateRow.setId(-row.getId());
            updateRow.setParentId(-row.getParentId());
            //similarly other columns.
        }
         Synchronized(lock){
            session.savePersistentAll(newRows);
        }
       //delete from inodes where id>=start and id<=end;
```

```
    }//end of run method
}

ThreadPool pool = new ThreadPool();
BatchSize=100,000;

Task1:
int maxId = select max(id) from inodes where status=2 or status=3;
int minId = select min(id) from inodes where status =2 or status=3;
int count=minId;

while(count<=maxId){
    pool.add(new DeleteRunnable(count,count+BlockSize);
    count = count+BlockSize;
}
//wait for completion of task.

Task2:

maxId = select max(id) from inodes where isDeleted=1 and id>0;
minId = select min(id) from inodes where  isDeleted=1 and id>0;
count=minId;

while(count<=maxId){
    pool.add(new UpdateRunnableForCoulmnIsDeletd(count,count+BlockSize);
    count = count+BlockSize;
}

//wait for completion of task.
Task3:

maxId = select max(id) from inodes where id<0;
```

```
minId = select min(id) from inodes where id<0;
count=minId;
Lock lock = new Lock();

while(count<=maxId){
    pool.add(new UpdateRunnableForColumnId(count,count+BlockSize,lock);
    count = count+BlockSize;
}

//wait for completion of task.
```

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.

4 Evaluation

4.1 Read-Only Nested Snapshots Implementation Evaluation

The algorithm to list subtree of directory at a given snapshot was implemented in SQl and executed against MySql NDB cluster.

4.1.1 Benchmark for measuring query execution time

In this benchmark, 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. Following operations are executed on the test directory.

1. **ADD :** Adding new files to the directory
2. **DEL:** Deleting exisiting files in the directory
3. **RENAME:** Renaming exisiting files in the directory
4. **MOV:** Moving file from this directory to another directory.
5. **MOV_IN:** Moving back the files that were moved out in MOV operation.
6. **Time:** Time taken for executing listing files in the directory at snapshot taken at time 5000. These measurements are taken on ndb cluster while the query executed at MySql server instead of clusterj.

The benchmark4.2 4.1 shows that the execution time scales linearly with the number of operations. The time to take snapshot is constant which just requires inserting a row into the SNAPS table, where as overhead to take snapshot with the design implemented by Facebook (Facebook-Hadoop) grows linearly 4.3 with the number of files/inodes in the fileSysetm.

ADD ops	DEL ops	MOV ops	MovIN ops	RENAME ops	Total Ops	New Time
5000	5000	5000	2500	5000	22,500	3.72 sec
5000*2	5000*2	5000*2	2500*2	5000*2	45,000	4.25 sec
5000*3	5000*3	5000*3	2500*3	5000*3	67500	7.98sec
5000*4	5000*4	5000*4	2500*4	5000*4	90,000	10.4 sec
5000*5	5000*5	5000*5	2500*5	5000*5	112,500	12.03 sec
5000*6	5000*6	5000*6	2500*6	5000*6	135,000	14.21 sec
5000*7	5000*7	5000*7	2500*7	5000*7	157,500	17.48 sec
5000*8	5000*8	5000*8	2500*8	5000*8	180,000	20.25 sec
5000*9	5000*9	5000*9	2500*9	5000*9	202,500	23.35 sec

Figure 4.1: Benchmark on Single Directory

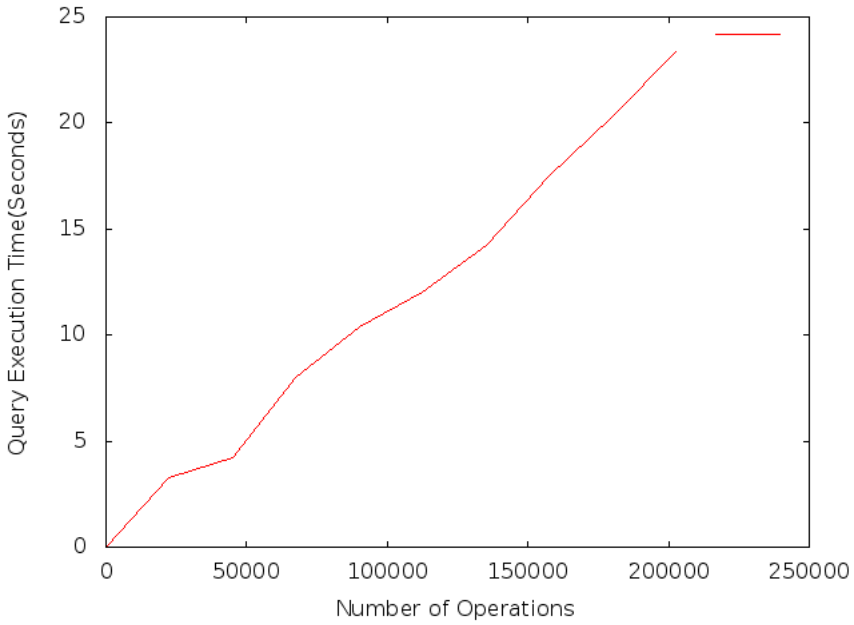

Figure 4.2: Benchmark on Single Directory Graph

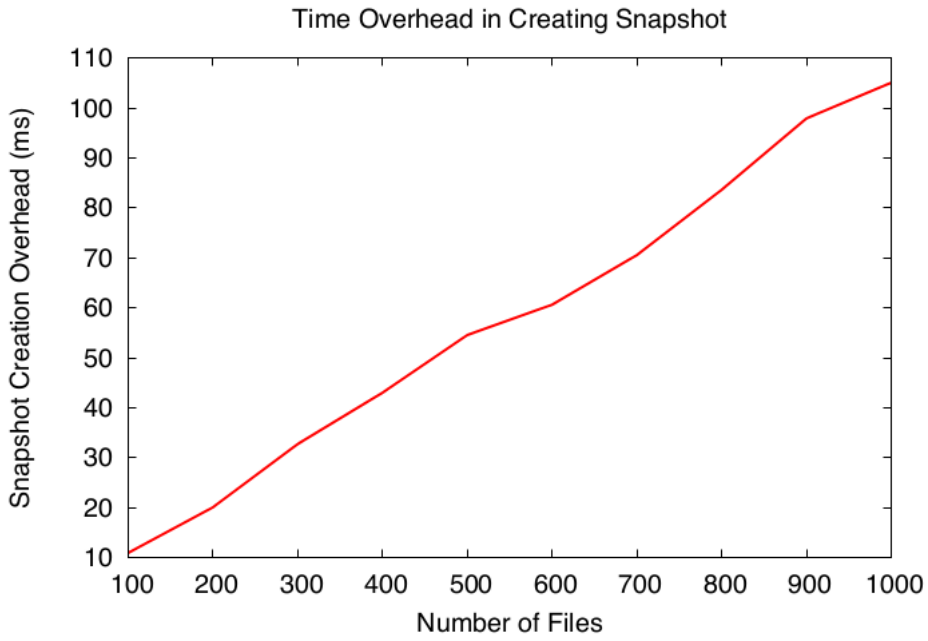

Figure 4.3: Time Overheads in HDFS@Facebook

4.2 Read-Only Root Level Single Snapshot Implementation Evaluation

4.2.1 Evaluation of RollBack

Cluster Configuration

MySql Cluster with 6 data nodes and 1 management server is used for evaluation.The management server which also host mysql server deamon.

Management Server& Data Nodes: 40GB RAM, 24 Intel(R) Xeon(R) CPU X5660 @ 2.80GHz each with 6 Cores.

The roll back algorithm mentioned in 3.3.3 is implemented using a thread pool, where each thread processes given number of table records [100,000]. The implementation is executed at MySql server as well as via directly connecting NDB-Cluster(Oracle-MySql) with ClusterJ(Oracle-ClusterJ).

The results with evaluation at MySql Server are shown below.

UnModified inodes that are Deleted	UnModified inodes that are not Deleted	Modified inodes	New Inodes	Total rows Processed	Time
450,000	450,000	100,000	100,000	1,200,000	4 min 35 sec
400,000	400,000	200,000	100,000	1,300,000	7 min 37sec
350,000	350,000	300,000	100,000	1,400,000	11min 25 sec
300,000	300,000	400,000	100,000	1,500,000	14min 40 sec
250,000	250,000	500,000	100,000	1,600,000	18 min 10sec
200,000	200,000	600,000	100,000	1,700,000	20 min 5 sec
150,000	150,000	700,000	100,000	1,800,000	23 min 50sec
100,000	100,000	800,000	100,000	1,900,000	27min 40sec
50,000	50,000	900,000	100,000	2,000,000	31 min 2 sec

Figure 4.4: Benchmark on MySqlServer

The high execution time with MySql server is because of update of column which is a primary key, in this case id,which we are changing from its negative value to positive value. In evaluations with clusterJ, batch size of rows were read, for each row a new row with id negative of the former row's id is inserted.

Evalaution with ClusterJ

The rollback algorithm explained in section 3.4.1 is evalauted with clusteJ and follwing results were obtained.

As we can infer from above graphs4.7 and 4.5 that roll-back with clusterJ is efficient and fast.

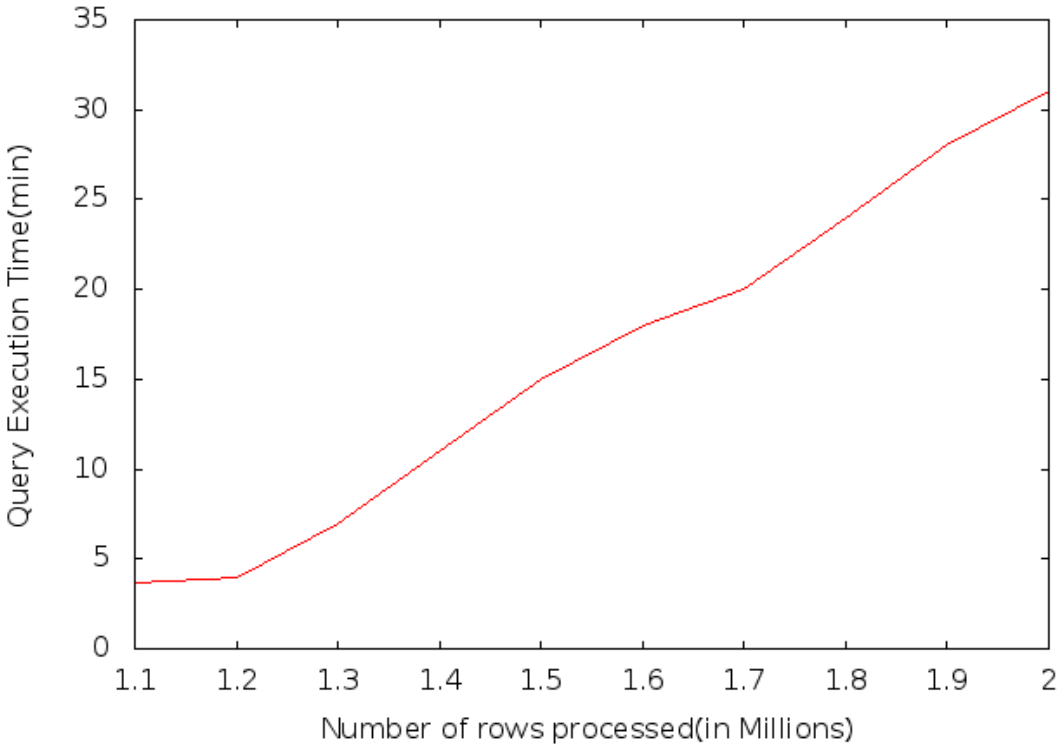

Figure 4.5: Benchmark-Graph on MySqlServer

UnModified inodes that are Deleted	UnModified inodes that are not Deleted	Modified inodes	New Inodes	Total rows Processed	Time
450,000	450,000	100,000	100,000	1,200,000	21 sec
400,000	400,000	200,000	100,000	1,300,000	22 sec
350,000	350,000	300,000	100,000	1,400,000	23 sec
300,000	300,000	400,000	100,000	1,500,000	26sec
250,000	250,000	500,000	100,000	1,600,000	27sec
200,000	200,000	600,000	100,000	1,700,000	29sec
150,000	150,000	700,000	100,000	1,800,000	32 sec
100,000	100,000	800,000	100,000	1,900,000	34 sec
50,000	50,000	900,000	100,000	2,000,000	36 sec

Figure 4.6: Benchmark with ClusterJ

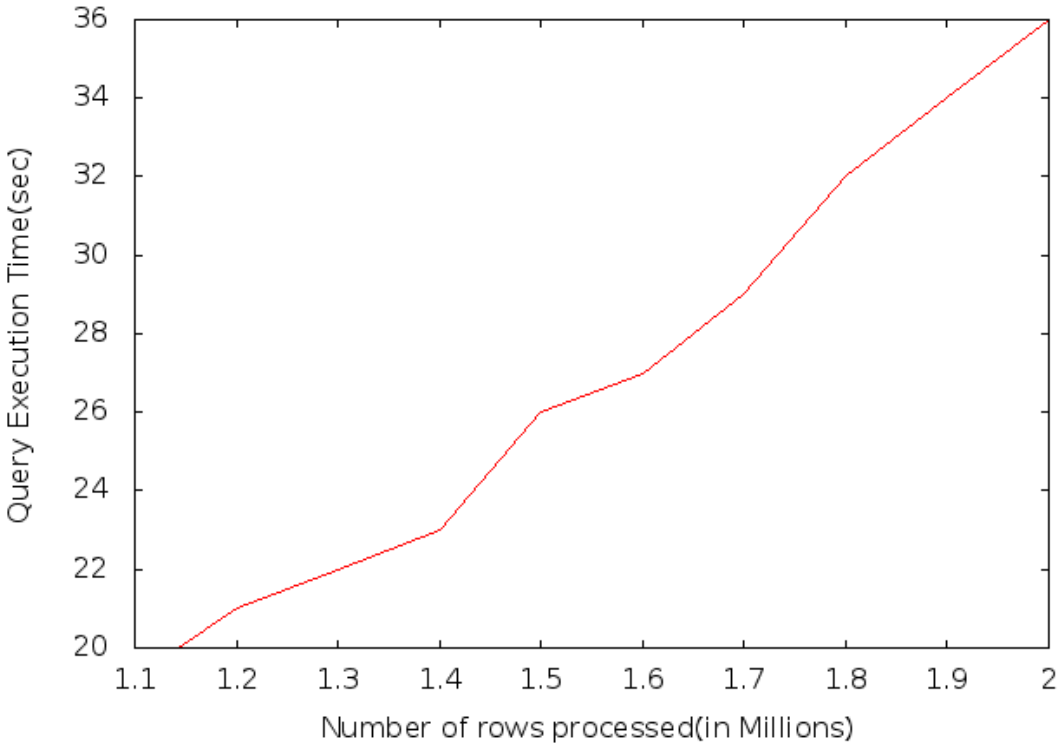

Figure 4.7: Benchmark-Graph with ClusterJ

5 Conclusions

5.1 Conclusions

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.

5.2 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. **Evaluating the two approaches of logging:** In section 3.2.6 we discussed two approaches to log the operations and deleting the file that are not referred by any snapshot. Those two approaches need to evaluated by benchmarking.

3. **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.

4. **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.

5. **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.

Bibliography

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.

A.Thusoo, J.Sarma, N.Jain, Z.Shao, P.Chakka, S.Anthony, H.Liu, P.Wyckoff, & R.Murthy (2009). Hive: a warehousing solution over a map reduce framework. In *Proceedings of the VLDB Endowment*, Volume 2, pp. 1626–1629.

Facebook-Hadoop. "snapshots in hadoop distributed file system". Accessed August 11, 2014.http://www.cs.berkeley.edu/~sameerag/hdfs_snapshots_ucb_tr.pdf.

Foundation, A. S. "apache hbase". Accessed August 2, 2014.http://hbase.apache.org.

Foundation, A. S. "apache mahout". Accessed August 2, 2014.http://mahout.apache.org.

Foundation, A. S. "apache pig". Accessed August 2, 2014.http://pig.apache.org.

Foundation, A. S. "apache zookeeper". Accessed August 2, 2014.http://zookeeper.apache.org.

Gilbert, S. & N. Lynch (2002, June). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*(2), 51–59.

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.

HopStart. ”hadoop open paas”. Accessed August 2, 2014.http://www.hopstart.org.

Malik, W. R. (2012). ”a distributed namespace for a distributed file system”. Master’s thesis, KTH.

MySql, D. Z. ”mysql :: Mysql cluster api developer guide :: 1.3.3.2 ndb record structure”. Accessed August 2, 2014.http://http://dev.mysql.com/doc/mysqlclusterexcerpt/5.1/en/mysqlclusterlimitationstransactions.html.

Oracle-ClusterJ. ”mysql clusterj overview”. Accessed August 11, 2014.http://dev.mysql.com/doc/ndbapi/en/mccj-using-clusterj.html.

Oracle-MySql. ”mysql cluster overview”. Accessed August 2, 2014.http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-overview.html.

Russom, P. (2011). Big data analytics. Technical report, TDWI Best Practices Report.

Sajjad, M. H. H. H. P. (2013). ”maintaining strong consistency semantics in a horizontally scalable and highly available implementation of hdfs ”. Master’s thesis, KTH.

SICS. ”swedish ict sics”. Accessed August 2, 2014.http://www.sics.se.

Snapshot-ls. ”list subtree in a snapshot”. Accessed August 2, 2014.https://github.com/pushparajxa/Snapshots/blob/master/Final_Short.sql.

White, T. (2009). *Hadoop: The Definitive Guide*. OReily Media.