INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Lorem Ipsum Dolor sit Amet, sed do Eiusmod Tempor Incididunt ut Labore et Dolore Magna Aliqua

Author's Name

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Doutor escolher
Orientador:	Doutor Orientador
Vogais:	Doutor Um
	Doutor Dois
	Doutor Três
	Doutor Quatro

May 2007

Agradecimentos

Sed ut perspiciatis, unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam eaque ipsa, quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt, explicabo.

Nemo enim ipsam voluptatem, quia voluptas sit, aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos, qui ratione voluptatem sequi nesciunt, neque porro quisquam est, qui dolorem ipsum, quia dolor sit, amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt, ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur?

Quis autem vel eum iure reprehenderit, qui in ea voluptate velit esse, quam nihil molestiae consequatur, vel illum, qui dolorem eum fugiat, quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus, qui blanditiis praesentium voluptatum deleniti atque corrupti, quos dolores et quas molestias excepturi sint, obcaecati cupiditate non provident, similique sunt in culpa, qui officia deserunt mollitia animi, id est laborum et dolorum fuga.

Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio, cumque nihil impedit, quo minus id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae.

Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

Lisboa, August 14, 2014

Author's Name

Quo usque tandem abutere,
Catilina, patientia nostra? Quam
diu etiam furor iste tuus nos
eludet?

Resumo

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Palavras Chave
Keywords

Palavras Chave

Lorem ipsum

Facere Possimus

Omnis Iste Natus

Nihil Molestiae

Omnis Voluptas

Magna Aliqua

Keywords

Lorem ipsum

Facere Possimus

Omnis Iste Natus

Nihil Molestiae

Omnis Voluptas

Magna Aliqua

Índice

List of Figures

List of Tables

Lorem Ipsum

Minim Veniam

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

1 Apache Hadoop

Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...

– Cerico

1.1 Introduction

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.

1.2 Hadoop File System(HDFS)

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 in-

tended 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.

1.2.1 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. Figure1.1 shows the deployment of different nodes in HDFS.

1.2.2 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.

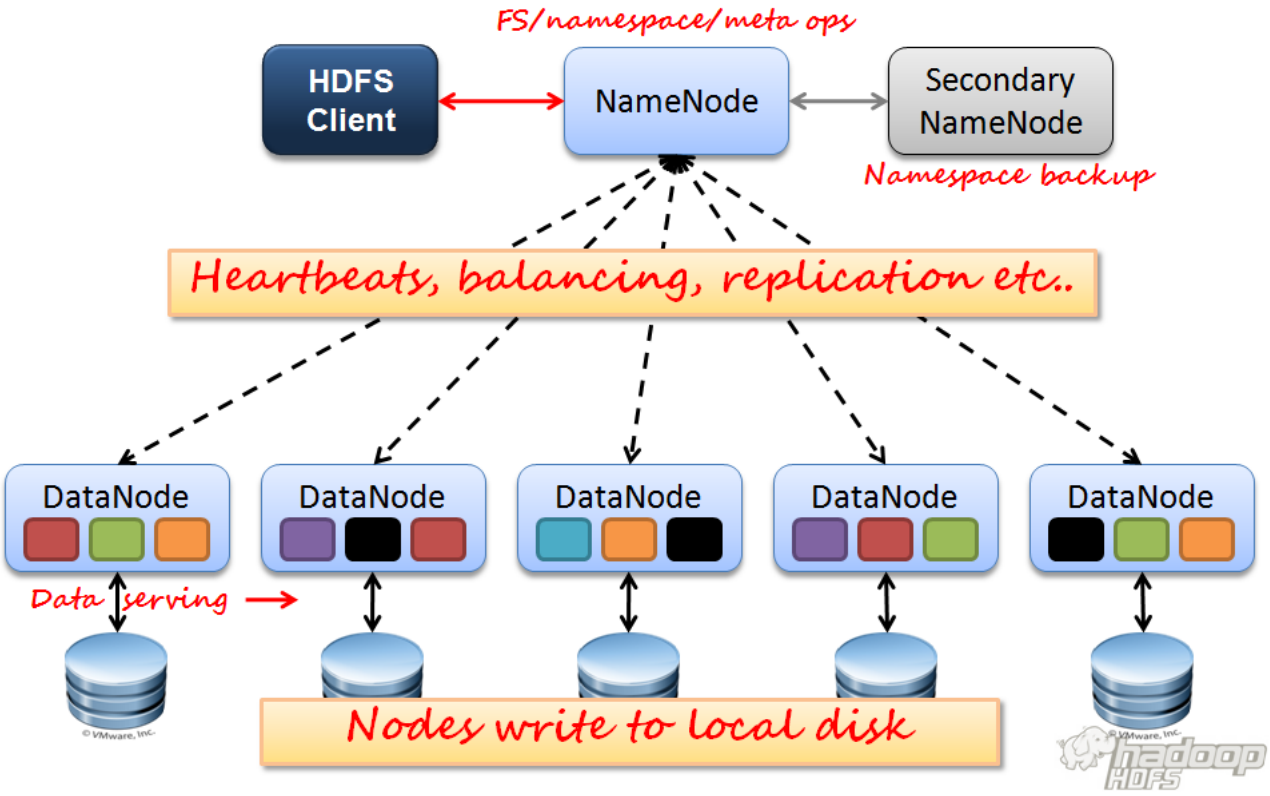

Figure 1.1: HDFS Architecture.

1.2.3 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 serializes 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

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
```

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.

1.2.4 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 Hadoop Open Platform as a service-HOP

2.1 Introduction

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)

2.2 HOP-HDFS

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

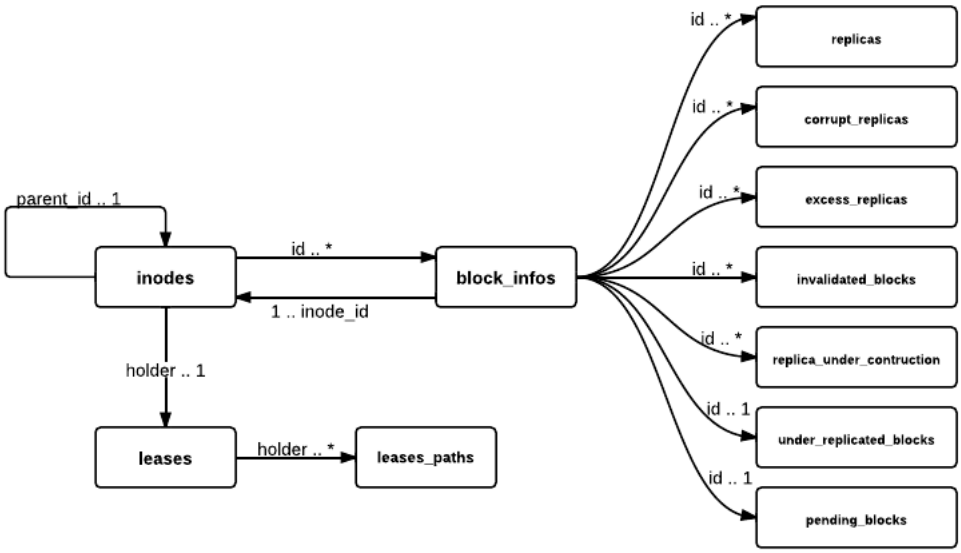

Figure 2.1: HOP-HDFS Table relations.

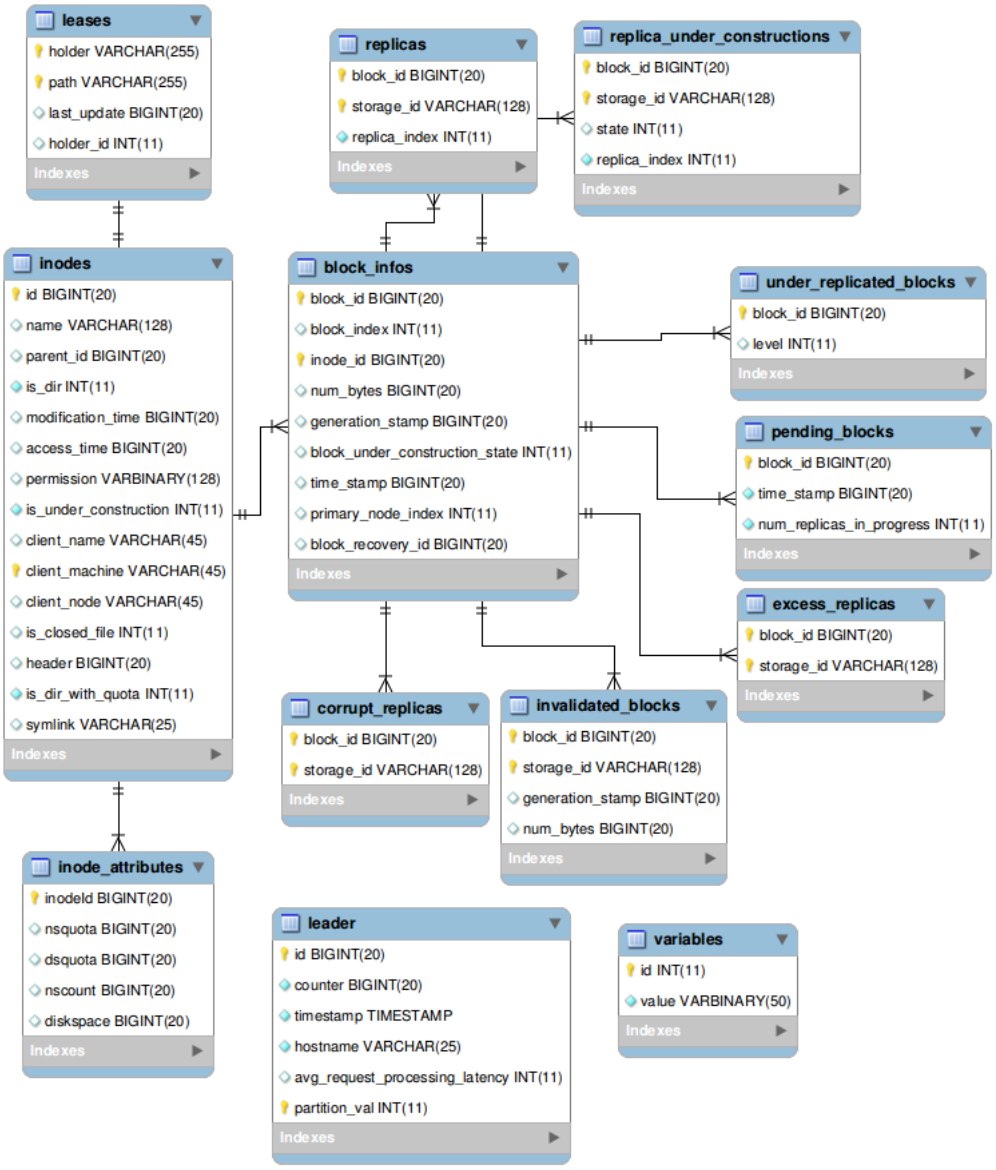

Figure 2.2: HOP-HDFS Schema

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.

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 non-empty 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

2.2.3 HOP-HDFS Implementation

n 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

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.

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 2.3. 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

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
```

are shown in figure 2.4. 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 2.3. 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.

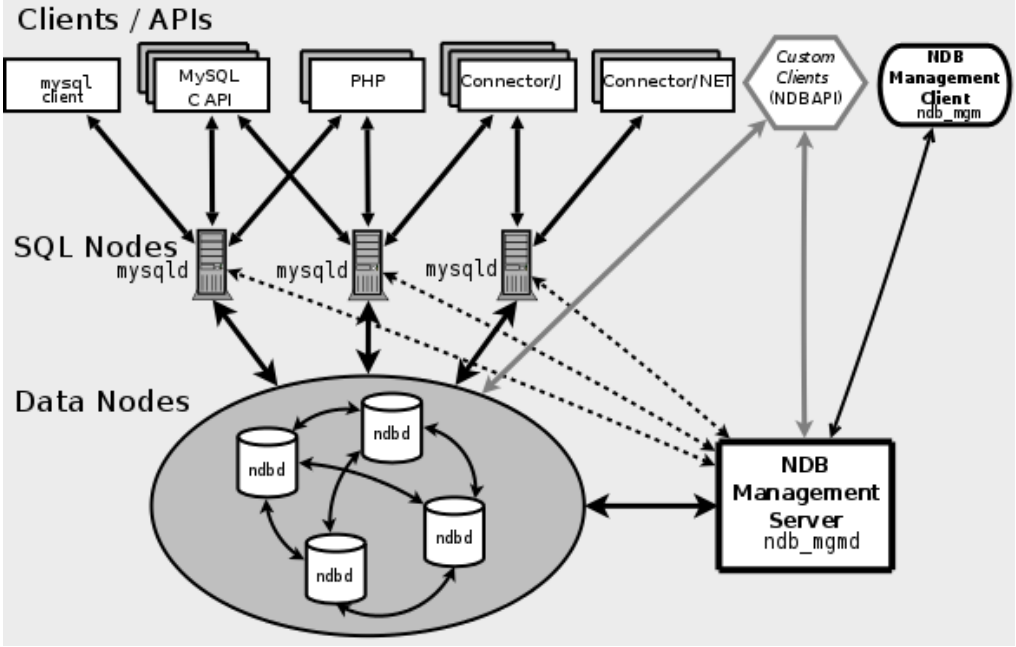

Figure 2.3: MySQL cluster

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

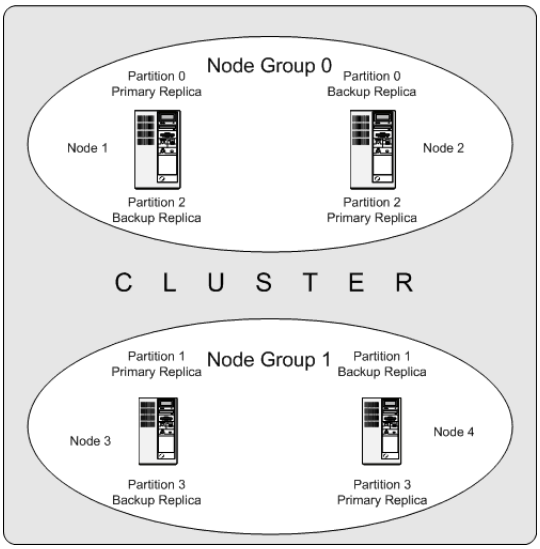

Figure 2.4: Node groups of MySQL cluster

indexes, scans, transactions and event handling. Clusterj connects directly to NDB Clusters 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.

II Problem Definition

3 Read-Only Nested Snapshots-Problem Definition

Users typically run experiments on the data they store in HDFS. Once an experiment overwrites or deletes existing files, it is not possible to revert to the previous state of data to run new experiments. In some cases users may like to take have different snapshots of data generated by different experiments. In some cases user may like to take snapshots and parent/child directories called nested snapshots.

Snapshots are point in time images of the file system. Snapshots should cover the following elements of the file system.

1. Snapshot of a subtree of the file system.
2. Snapshot of the entire file system.

3.1 UseCase Scenarios

1. **Protection against user errors:** Admin sets up a process to take RO 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.

3.2 Requirements and Operations to be supported

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

3.3 Related Work

3.3.1 Apache Hadoop Version 2

(Apache-Hadoop)

3.3.2 Hadoop at Facebook

(Facebook-Hadoop)

4 Read-Only Root Level Single Snapshot-Problem Definition

4.1 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.

4.1.1 Related Work

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.

III Solution

5 Read-Only Nested Snapshots

This section presents solution to Netsed Snapshots problem disussed in 3

5.1 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.

5.2 Modifications to the Schema

Following columns need to be added to the Inodes table described in the schema 2.2 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 2.2.

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.

5.3 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 childrens[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]

5.4 Listing children under a directory in a given Snapshot

5.5 Listing current children under a directory

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

Here two approaches two solve the issues are presented.

5.6.1 Approach 1:

Columns to be added to Inodes Table

1. Moved_ In/Created 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.

MovedPaths

Inode_ Id	Path	Time

Table 5.1: MovedPaths table

5.6.1.1 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. We take help of the table MovedPaths 5.1.Ex: we have /A/B/C/ as path, where A,B, and C are directories.Suppose we want to add a file in C. Now we need to check whether C is in any snapshot. On a first sight, we can think that checking whether any snapshots taken on A,B,C before C was created , but B or A may be moved from different place, for example B may be moved from /A/D/ path and there may be some snapshots taken on directory D even though no snapshots may be not taken on A and B. To handle that situation, whenever we move a file or directory, first we check whether it is in a snapshot, if yes then insert a row in the MovedPaths table containing original path before being moved.

Assume following path exist /U/V/W/X [U,V,W,X are directories, we are performing add/delete/modify on inodes(files or directories) in it]. Say we are performing operation on inode P whose path is /U/V/W/X/P.First, we need to find whether P is in any snapshot. We find that with below steps.If P is in any snapshot then we log the operation in corresponding table which are described above 5.2.

1. Check if any snapshots on U,V,W,X before present time.
2. If we not find any in step1 then check in the MovedPaths table recursively for U,V,W ,X if there any snapshots on any of the inodes in the MovedPaths.
3. If there any from step1 and step2 then check whether MovedIn/Created Time of X is less than any of them. If none is there means, there was no snapshot covering this directory after it created or moved in.

5.6.1.2 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.

5.6.1.3 Deleting logs

We follow the same approach as mentioned in the Approach1.

5.6.1.4 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. If user issued command to delete a directory, since a directory may contain files/directories which are moved into it from different directory on which snapshot was taken. we should not delete them permanently. We mark all the children with isDeleted=1 and the background delete thread will do check on inodes in it in a bottom-up fashion, deleting those not present in any snapshot permanently. We also delete any logs in D-List,MovedPaths associated with the inode that we want to delete permanently.

5.6.1.5 Deleting entries in MovedPaths Table

When a snashot on an Inode is deleted, we can check paths containing that Inode for any snapshot existing on Inodes in that path before the moved time. If there aren't any then we can delete them.If a file/directory is deleted then we can delete all entries related to that.

5.6.2 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.

5.6.2.1 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.

5.6.2.2 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

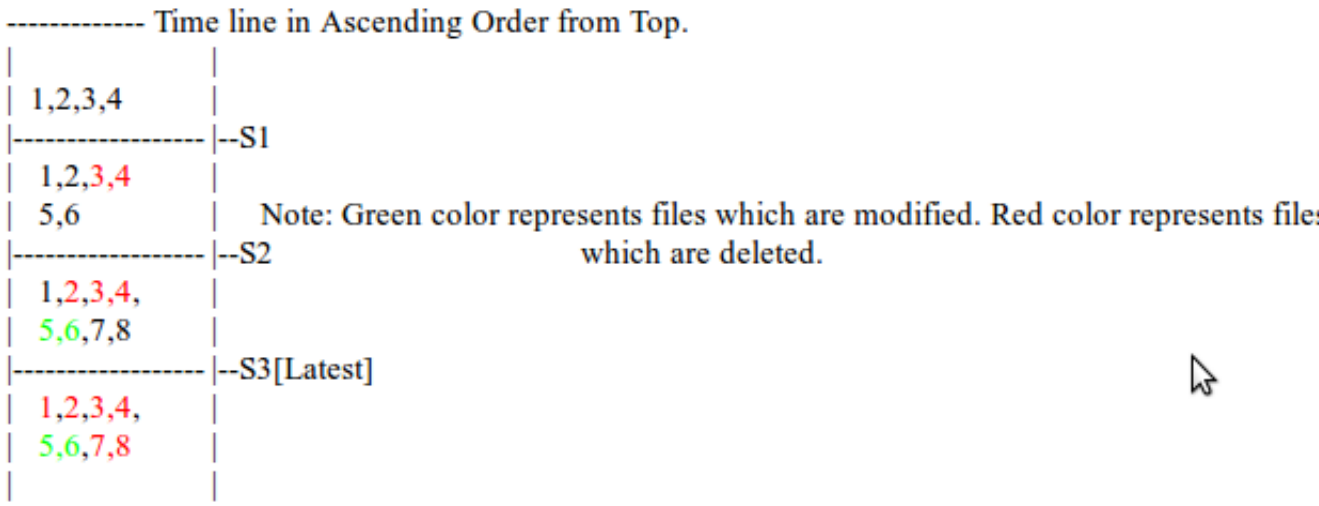

Figure 5.1: Deletion of Snapshot

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

5.6.2.3 Handling the replication factor change of a file

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.

5.6.2.4 Disadvantages:

1. Time for taking snapshot is O(n) where n is the number of descendants in directory

6 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.

6.1 Modifications to the Schema

Following columns need to be added to the Inodes table described in the schema 2.2 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 2.2.

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.

6.2 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.

```
void deleteWithSnapshotAtRootTaken(INode targetNode){

        Stack<INode> stck = new Stack<INode>();
        INode tempNode;
        List<INode> children;
        INode[] inodesTemp;
        INode removedInode;

        stck.add(targetNode);

        while (!stck.empty()) {
            tempNode = stck.pop();
            tempSts = tempNode.getStatus();
            tempStr = tempNode.getFullPathName();

            if (tempNode.getIsDeleted() == 1) {
                //This node is already marked deleted, so nothing to do.
                continue;
            }
            /*
             * This Inode can be a directory or file also it can be new or m
             */
            if (tempNode instanceof INodeDirectory) {
                children = ((INodeDirectory) tempNode).getChildren();
                if (children != null && !children.isEmpty()) {
                    stck.push(tempNode);
                    for (INode n : children) {
                        stck.push(n);
                    }
                } else {
                    if(tempNode.getStatus()==SnapShotConstants.New){
                        //delete completely this directory Inode
```

```
                        EntityManager.remove(tempNode);
                    }
                    else{
                    //Set isDeleted = 1.
                        tempNode.setIsDeletedNoPersistance(SnapShotConstants
                    }
                }
            } else if (tempNode instanceof INodeFile || tempNode instanceof
                if (tempSts == SnapShotConstants.New) {
                    //We can delete this file permanently and update the ans
                    //Remove the blocks associated with this file permanentl
                    }
                } else if (tempSts == SnapShotConstants.Original || tempSts
                    //Set isDeleted = 1.
                    tempNode.setIsDeletedNoPersistance(1);
                }
            }
        }// 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.
}
```

6.3 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
4. Delete from INodes 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

IV

Implementation and Evaluation

7 Read-Only Nested Snapshots Implementation and Evaluation

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.

7.1 Modifications to the Schema

Following columns need to be added to the Inodes table described in the schema 2.2 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 2.2.

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.

8 Read-Only Root Level Single Snapshot Implementation and Evaluation

8.1 Evaluation

The roll back algorithm mentioned in 6.3 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.

The results with evaluation at NDB cluster are shown below.

UnModified inodes that are Deleted	UnModified inodes that are not Deleted	Modified inodes	New Inodes	Time

Table 8.1: Roll Back Time with MySql Server

UnModified inodes that are Deleted	UnModified inodes that are not Deleted	Modified inodes	New Inodes	Time

Table 8.2: Roll Back Time with ClusterJ

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.

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.

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

V Appendices

A Commodo Consequat

Figure A.1: Soluta nobis est eligendi optio.