KTH Royal Institute of Technology

Master Thesis

Maintaining Strong Consistency Semantics in a Horizontally Scalable and Highly Available Implementation of HDFS

Authors: Hooman Peiro Sajjad
Mahmoud Hakimzadeh Harirbaf

Supervisor: Dr. Jim Dowling

Program of Study: Master of Software Engineering of Distributed System

TRITA-ICT-EX-2013:79
Abstract

The Hadoop Distributed Filesystem (HDFS) is the storage layer of Hadoop, scaling to support tens of petabytes of data at companies such as Facebook and Yahoo. One well-known limitation of HDFS is that its metadata has been stored in-memory on a single node, called the NameNode. To overcome NameNode’s limitations, a distributed file system concept based on HDFS, called KTHFS, was proposed in which NameNode’s metadata are stored on an in-memory replicated distributed database, MySQL Cluster.

In this thesis, we show how to store the meta-data of HDFS NameNode in an external distributed database while maintaining strong consistency semantics of HDFS for both filesystem operations and primitive HDFS operations. Our implementation supports MySQL Cluster, to store the meta-data, although it only supports a read-committed transaction isolation model. As a read-committed isolation model cannot guarantee strong consistency, we needed to carefully design how meta-data is read and written in MySQL Cluster to ensure our system preserves HDFS’s consistency model and is both deadlock free and highly performant. We developed a transaction model based on taking meta-data snapshotting and the careful ordering of database operations. Our model is general enough to support any database providing at least read-committed isolation level. We evaluate our model and show how HDFS can scale, while maintaining strong consistency, to terabytes of meta-data.
Acknowledgements

First we would like to express the deepest appreciation to our supervisor Dr. Jim Dowling who has been a great help and vision for us during this thesis. We also thank the staff of Swedish Institute of Computer Science for providing us an appropriate environment to work smoothly.

Finally, an honorable mention goes to our families for their unconditional support and encouragement. Without their help, we would face many difficulties during this.
# Table of Content

Chapter 1 - Introduction........................................................................................................9
Chapter 2 - Related Distributed FileSystems.................................................................11
  2.1 Google FileSystem.................................................................................................11
  2.2 TidyFS................................................................................................................12
  2.3 HDFS Federation...............................................................................................13
Chapter 3 - Background.................................................................................................14
  3.1 HDFS..................................................................................................................14
     3.1.1 HDFS Architecture....................................................................................14
     3.1.2 HDFS NameNode.....................................................................................14
     3.1.3 HDFS Capacity.........................................................................................16
     3.1.4 HDFS NameNode’s Data-structures.........................................................16
     4.1.5 NameNode Operations.............................................................................17
     3.1.6 POSIX compliant filesystem....................................................................19
  3.2 Mysql Cluster......................................................................................................20
     3.2.1 Concurrency Control in NDBCluster.....................................................22
     3.2.2 Clusterj....................................................................................................22
  3.3 KTHFS................................................................................................................23
Chapter 4 - Problem Definition......................................................................................25
  4.1 System-level lock is a throughput bottleneck....................................................25
  4.2 Ensuring Strongly Consistent Semantics............................................................25
     4.2.1 HDFS consistency model.........................................................................26
     4.2.2 Challenges in implementing HDFS Operations as Transactions..............26
     4.2.3 Consistency and Transaction Isolation Levels........................................27
  4.3 Reducing the Number of Round-Trips to the Database.......................................28
Chapter 5 - Solution.......................................................................................................30
  5.1 Snapshot isolation...............................................................................................30
     5.1.1 Basic schema.............................................................................................30
  5.2 Avoid modification conflicts..............................................................................30
     5.2.1 Row level locking as complement of snapshot isolation..........................31
  5.3 Semantically related mutations..........................................................................31
     5.3.1 Phantom read happens in semantically related operations.....................33
     5.3.2 Parent lock to avoid semantically related mutations..............................33
  5.4 Deadlock prone scenarios..................................................................................33
     5.4.1 Total order locking...................................................................................34
  5.5 Multi Transaction Operations.............................................................................35
  5.6 Solution at the concrete level.............................................................................36
     5.6.1 HDFS Total ordering challenges.............................................................36
     5.6.2 Total ordering categories.........................................................................39
  5.7 Solution’s safety and liveness.............................................................................42
Chapter 6 - Implementation............................................................................................44
  6.1 Transaction context.............................................................................................44
  6.2 Persistence layer..................................................................................................45
Chapter 7- Evaluation and Experiments.........................................................................47
7.1 The impact on the database roundtrips.................................................................47
7.2 Row-level lock in Mysql Cluster............................................................................47
7.3 System Lock vs Row Level Lock...........................................................................49

Chapter 8 - Conclusion and future work....................................................................52
  8.1 Conclusion............................................................................................................52
  8.2 Future Works........................................................................................................53
    8.2.1 Relaxing Locks...............................................................................................53
    8.2.2 Inter Transaction Cache................................................................................53
    8.2.3 Optimistic Transaction Conflict Resolution..................................................53
    8.2.4 NsCount and DsCount in INodeDirectory....................................................53

References..................................................................................................................54
Appendix......................................................................................................................55
List of Figures

Figure 1: GFS Architecture..................................................................................................................11
Figure 2: TidyFS Architecture...............................................................................................................12
Figure 3: Federation Architecture..........................................................................................................13
Figure 4: HDFS Architecture................................................................................................................15
Figure 5: HDFS Data Structure...............................................................................................................17
Figure 6: A Filesystem operation is a composite of NameNode primitive operations..................19
Figure 7: Mysql Cluster.........................................................................................................................20
Figure 8: Node groups of Mysql Cluster...............................................................................................21
Figure 9: Update throughput of Mysql Cluster......................................................................................22
Figure 10: KTHFS Architecture............................................................................................................23
Figure 11: Fuzzy read in KTHFS...........................................................................................................27
Figure 12: Phantom read in KTHFS.......................................................................................................27
Figure 13: Violation of filesystem semantics........................................................................................32
Figure 14: Locking in a conflicting order..............................................................................................34
Figure 15: Lock upgrade runs into deadlock.........................................................................................34
Figure 16: Locking just all component of given path...........................................................................40
Figure 17: Locking all component of given path plus immediate children of last directory.............40
Figure 18: Locking all component of given path plus all recursive children of last directory...........41
Figure 19: Locking just all existing component of given path while last existing component is unspecified.................................................................41
Figure 20: Main classes of transaction context.....................................................................................44
Figure 21: Main classes of persistence layer..........................................................................................46
Figure 22: Impact of transaction context on reducing number of database roundtrips.................47
Figure 23: The namespace design to benchmark concurrency control performance in Mysql Cluster.................................................................48
Figure 24: Concurrency control performance in Mysql Cluster..........................................................49
Figure 25: Namespace design for GeneralNNThroughputBenchmark.............................................50
Figure 26: The comparison between NameNode with system level lock and the NameNode with row level lock for Create operation..........................................................50
Figure 27: The comparison between NameNode with system level lock and the NameNode with row level lock for Open operation.................................................................51
List of Tables

Table 1: HDFS Limitations.......................................................................................................16
Table 2: NameNode’s Operations............................................................................................18
Table 3: Isolation levels............................................................................................................28
Table 4: HDFS total ordering challenges and their solution.....................................................38
Table 5: Basic order of lock acquiring......................................................................................39
Table 6: Multi transaction operation when just block-id is given..............................................42
Table 7: Multi transaction operation when just lease is given..................................................42
Table 8: Complete NameNode Operations..............................................................................55
List of Algorithms

Algorithm 1: System-level locking schema in HDFS..............................................................15
Algorithm 2: Snapshot-isolation schema.................................................................................31
Algorithm 3: Snapshot-isolation with row-level lock schema..................................................32
Algorithm 4: Snapshot-isolation with total ordered row-level lock schema.............................35
Algorithm 5: Multiple transactions schema..............................................................................37
Chapter 1 - Introduction

The Hadoop Distributed File System is part of a Apache Hadoop Core project which is designed to run on commodity hardware [HDFS-0], scaling to support tens of petabytes of data at companies such as Facebook and Yahoo. HDFS is a highly fault-tolerant distributed file system that provides high throughput access to application data and is suitable for applications with large data sets. A few POSIX requirements are relaxed in HDFS in order to enable streaming access to file system data.

One well-known limitation of HDFS is that its metadata has been stored in-memory on a single node, called the NameNode. In terms of storage, The number of objects stored by HDFS is limited to the memory size of the NameNode. Indeed, a single NameNode is a single point of failure. Beside that, it is limited in terms of handling clients requests while managing internal loads[HDFS-1].

To overcome NameNode's limitations, a distributed file system concept based on HDFS, called KTHFS, was proposed [KTHFS]. The idea is to store the NameNode's metadata on an in-memory replicated distributed database, MySQL Cluster and to have stateless NameNodes with the aim of reaching more storage capacity than HDFS, highly available NameNodes, scaling out the NameNode's throughput and preserving HDFS API’s semantics. A partial implementation of the idea is also available. In this implementation, the data persisted in the database are defined. The NameNodes are categorised in two different roles: Reader and Writer. Reader NameNodes are responsible for the operations that are read-only and Writer NameNode can do all the operations.

In this thesis, we show how to store the meta-data of HDFS' NameNode in an external distributed database, supporting an isolation level of at least read-committed, while maintaining HDFS' existing consistency semantics for filesystem metadata. HDFS supports strong consistency in metadata level. As we already introduced, HDFS metadata is managed by NameNode, it protects concurrent accesses to metadata by means of system-level read/write lock. However, there are a number of challenges that we need to address to ensure that our filesystem maintains the strong consistency semantics of HDFS for multiple writers concurrent with other writers/readers. In addition to the problem of maintaining HDFS consistency, we also address the problem of how to reduce an excessive number of round-trips to the database.

Although our solution is not bound to any specific database, we chose Mysql Cluster as the metadata storage. Mysql-cluster does not support more than read-committed isolation level. If we just rely on the database isolation level the solution can break the consistency level of HDFS. We introduce an implementation of snapshot isolation level to assure the HDFS’s consistency. Indeed, we removed HDFS’s system level lock and use the database’s row level lock in our snapshot implementation. This allows us to define a more fine grain locking for operations and introducing more parallelization for both write and read operations. We defined mechanisms which prove our solution does not have common pessimistic concurrency control problems, as deadlock while keeps the metadata consistent.
In the end, we evaluate our work with the system level lock implementation and also show how the number of the database round-trips are reduced in our solution.
Chapter 2 - Related Distributed FileSystems

In this section, we introduce some of the distributed filesystems well known in this area. We describe their goals and their architecture briefly. A distributed filesystem is a file-service system whose components including users, servers and storage devices are distributed on different machines. There are multiple storage devices involved and the components communicate over network [DFS].

2.1 Google FileSystem

Google File System (GFS) is a scalable distributed file system for the large data-intensive applications developed in Google company. The system is built from many inexpensive commodity hardwares [GFS-1]. There are some assumptions in implementation of GFS:

- The system is built from inexpensive commodity hardwares that often fails.
- The system stores fairly large files, multi Gigabyte files are common.
- The system workloads consist of: large streaming reads, small random reads and many large, sequential writes that append to the same file.
- A well defined semantic for multiple clients appending concurrently to the same file.
- Having a high sustained bandwidth to process data in bulk is more important than low latency.

A GFS cluster consists of a single master and multiple chunk servers and multiple clients to access the file system. Its architecture is shown in figure 1.

![GFS Architecture](image)

File are divided to fixed size chunks which are stored in chunk servers. For reliability each chunk is replicated over multiple chunk servers. The replication factor by default is three but the user can designate different replication levels for different regions of the file namespace.

The master is a metadata server maintaining the namespace, access control information,
the mapping from files to chunks of data, and the current locations of chunks. It is also responsible for system-wide activities such as chuck lease management, garbage collection of orphaned chunks, and chunk migrations between chunkservers.

As it can be seen, GFS has a single master which simplifies the design. It also enables master to make sophisticated chunk placement and replication decisions using global knowledge. However, it can become bottleneck if it involves it all read and write operations. In order to avoid this, clients read and write the chunks directly to the chunkservers and only ask the master for the appropriate chunkservers to contact.

2.2 TidyFS

In recent years, an explosion of interest in computing using clusters of commodity and shared nothing computers emerged. Following this trend, Microsoft introduced their distributed file system called Tidy FS [TidyFS]. It is a simpler file system comparing to HDFS and GFS. The system avoids complex replication properties of the workload such as the absence of concurrent writes to a file by multiple clients, and the existence of end-to-end fault tolerance in the execution engine. The requirements for having such a file system is so similar to GFS. The Tidy FS system architecture is depicted in figure 2.

![Figure 2: TidyFS Architecture](image)

The TidyFS storage system is composed of three components: a metadata server; a node service that performs housekeeping tasks running on each storage computers; and the TidyFS Explorer, a graphical user interface which allows users to view the state of the system.

The metadata server is similar to NameNode in HDFS and is responsible for storing the mapping of stream names to sequences of parts, the per-stream replication factor, the location of each part replica, and the state of each storage computer in the system, among other information. Since the metadata server has a vital role in TidyFS, it is a replicated component using the Autopilot Replicated State Library to replicate the metadata and operations on that metadata using the Paxos algorithm.

Node services are responsible for a set of maintenance tasks that must be carried out on a routine basis. They are Windows services that run continuously on each storage computer in
the cluster. The tasks include periodic reporting of the amount of free space on the storage computer disk devices to the metadata server, garbage collection, part replication, and part validation.

2.3 HDFS Federation

HDFS Federation was developed in HDFS-1052 branch in order to address namespace scalability and isolation problem of HDFS. HDFS does not provide isolation by itself. It means that many different applications access a HDFS cluster in parallel and if one of them overloads the namenode it will slow down the whole cluster. Indeed, HDFS cluster is horizontally scalable with the addition of data-nodes but the namespace is not. It is only possible to scale the namespace vertically by adding better hardwares.

HDFS Federation uses multiple independent namenodes/namespace The namenodes are independent and don’t require coordination with each other. The data-nodes are used as common storage for blocks by all the namenodes. Each data-node registers with all the namenodes in the cluster. The architecture is shown in figure 3.

A namespace and its block pool together are called Namespace Volume. It is a self-contained unit of management and when a namenode/namespace is deleted, the corresponding block at the datanode is deleted.
Chapter 3 - Background

This chapter gives an overview about the technologies we are using which are Hadoop Distributed File System and Mysql-Cluster. We go through their design, architecture, and applications first, then we jump into their characteristics from point of view of the problem that this thesis is trying to address and how well they are doing in terms of scalability and throughput. At the end, we start describing the idea behind KTHFS, its architecture drivers and the amount of work had been done up to the point this thesis started.

3.1 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 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 highly 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.

3.1.1 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. Figure 4 shows the deployment of different nodes in HDFS [HDFS-2].

3.1.2 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 algorithm 1.

![HDFS Architecture Diagram](image)

**Algorithm 1**: System-level locking schema in HDFS

```java
operation lock
    if op.type = 'write' then ns.acquireWriteLock() else ns.acquireReadLock()

operation performTask
    // Operation body

operation unlock
    if op.type = 'write' then ns.releaseWriteLock() else ns.releaseReadLock()
```

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.

### 3.1.3 HDFS Capacity

Hadoop Distributed File System was developed based on four main scale requirements [HDFS-1] which are storage capacity, number of nodes, number of clients and number of files. The largest HDFS cluster has been deployed in Facebook, according to a blog posted by Dhruba Borthakur [FB]. Indeed, The second biggest HDFS cluster has been used in Yahoo. The details of HDFS target scale requirements and those one deployed in Facebook and Yahoo are shown in table 1.

<table>
<thead>
<tr>
<th>Table 1: HDFS Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
</tr>
<tr>
<td><strong>Capacity</strong></td>
</tr>
<tr>
<td><strong>Nodes</strong></td>
</tr>
<tr>
<td><strong>Clients</strong></td>
</tr>
<tr>
<td><strong>Files</strong></td>
</tr>
</tbody>
</table>

### 3.1.4 HDFS NameNode’s Data-structures

The persistent data structures of KTHFS 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.
2. **block_infos**: 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 under-construction or completed. All under-construction 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. **leases_path**: Each lease path represents an under-construction 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 data-node, sometime we refer
to replicas as blocks. All the replicas of the same block points to the same block-info.

6. **corrupted_replicas**: A replica become corrupted in the copy operations or due to the storage damages. Name-node realizes this by comparing checksum in the report of the replica’s data-node with the checksum of the original block.

7. **excess_replicas**: A block could become over replicated because of an already dead data-node coming alive again and contain some replicas which has been removed meanwhile from name-node. So distinguishing that, name-node 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. **replica_under_construction**: Replications of a block which are being streamed by client into data-nodes.

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.

Figure 5: HDFS Data Structure
4.1.5 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.

Five of the most common primitive operations are shown in the table 2. The full list of the primitive operations can be found in table 8 in Appendix section.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKDIR</td>
<td>Creates a directory recursively, it requires a no lock on all the existing components of the path but write lock on the last existing. This operation has P#1, P#2 and P#9.</td>
</tr>
</tbody>
</table>
| START_FILE                 | 1. If file does not exist, It creates inodes for all the non-existent directories and new file, writes owner of the lease and creates new lease-path.  
                                2. If file already exists first removes the file, its blocks and dependencies, lease and lease path, then it does the first scenario.  
                                3. This operation has P#1, P#2, P#3, P#9. |
| GET_ADDITIONAL_BLOCK       | In the middle of writing a file, this is the client’s mean of noticing name-node that the already being written block is finished while it is asking for the locations of next block. NameNode removes all the replica-under-constructions of last block, it also changes type of block-info from under-construction to completed one. It has P#4 and P#5. |
| COMPLETE                   | Like get_additional_block, it happens for last block of the file, NameNode just removes the replica-under-constructions and changes type of block-info from under-construction to completed. |
| GET_BLOCK_LOCATIONS        | Given path of the file, it returns location if its blocks and updates access-time of the file_inode. |

An example of HDFS operation and its primitive operations are shown in figure 6. As it can be seen, user enters command copyFromLocal using HDFS Client. Under the hood, this operation includes multiple primitive operations as sending startFile request to the NameNode and after that multiple getAdditionalBlock operations to the NameNode to add
blocks to the created file. For every getAdditionalBlock, it receives a list of DataNodes to write
the block to. Indeed, DataNodes after receiving the block data from the client they send a
blockReceivedAndDeleted request to the NameNode. In the end, HDFS Client send a
complete request to the NameNode to close the file.

![Diagram of filesystem operations](image)

**Figure 6: A Filesystem operation is a composite of NameNode primitive operations**

### 3.1.6 POSIX compliant filesystem

POSIX-fs is file system part of POSIX operating system. It has been being a standard for
designing file-systems. It is about naming, hard-links, 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
POSIX-compliant file system is increased performance for data throughput and support for
non-POSIX 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.

3.2 Mysql Cluster

Mysql Cluster is a Database Management System (DBMS) that integrates the standard Mysql Server with an in-memory clustered storage engine called NDB Cluster (which stands for “Network Database”) [MYSQL-3]. It provides a shared-nothing 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 7.

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:
A simple cluster of 4 datanodes with replication factor of 2 and consequently 2 node groups are shown in figure 8.

![Node groups of Mysql Cluster](image)

Figure 8: Node groups of Mysql Cluster

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 ndbds 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], 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 9.

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.
3.2.1 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-1]. 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 real-time services, that are, operations expected to complete in bounded time. As such, NDB enables the construction of services that can fail-over, 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.

3.2.2 Clusterj

Clusterj is Java connector implementation of NDB Cluster, Mysql Cluster’s storage engine, [CLUSTERJ]. 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 instead of connecting to mysql. 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.
3.3 KTHFS

KTHFS is a distributed file system concept based on Hadoop Distributed File System with the aim of reaching the following goals:

- Going beyond the maximum storage capacity of HDFS
- Scaling out the NameNode’s throughput
- A Highly available NameNode
- Preserving HDFS API’s semantics, so that applications using HDFS can also use KTHFS

The initial idea of KTHFS was to transfer the metadata in NameNode to a scalable distributed database as Mysql Cluster and to have stateless NameNodes. The KTHFS design is shown in figure 10.

As we discussed earlier, HDFS NameNode keeps all metadata in memory. According to [HDFS-2I], estimates show that each metadata object in NameNode takes fewer than 200 bytes memory. Indeed, a file on average has 1.5 block which makes the total metadata size for a file to become 600 bytes (1 file object + 2 blocks object). In order to store 100 million files in HDFS, we need at least 60 GB of RAM. HDFS in practice could have up to 64 GB of memory so the size of metadata could be maximum 64 GB. However, Mysql Cluster can have up to 48 datanodes [Mysql-4] and store up to 4 TB of data. So by using Mysql Cluster as an external metadata storage we can store 30 times more metadata.

The high availability of NameNodes in KTHFS comes from having stateless NameNodes.
This means that if any of the NameNodes crash other NameNodes can continue serving clients. Indeed, Mysql Cluster is also a shared nothing database that is fault tolerant in case of having at least a replication factor of 2. It guarantees high availability of 99.999%.

Mysql Cluster can handle more than 1 billions read and write requests per minute. By having multiple NameNodes handling clients’ requests on top of it we can go beyond the limits of one NameNode in HDFS in handling client’s requests.

There is a partial implementation of the idea available [KTHFS]. In this implementation, the data persisted in the database are defined. The NameNodes are categorised in two different roles: Reader and Writer. Reader NameNodes are responsible for the operations that are read-only and Writer NameNode can do all the operations.
Chapter 4 - Problem Definition

As in KTHFS, the NameNodes' meta-data will be stored in NDB. The first version of KTHFS supported concurrent reads and a single writer. However, system level lock became unusable and an obstacle to scale out throughput. Moreover, there are a number of challenges that we need to address to ensure that our filesystem maintains the strong consistency semantics of HDFS for multiple concurrent writers. In addition, we also address the problem of how to reduce an excessive number of round-trips to the database. Too many round-trips cause filesystem operations to take an unreasonably long amount of time, and reduce the scalability of the filesystem by increasing load on the database.

4.1 System-level lock is a throughput bottleneck

HDFS uses read/write locks to protect parallel threads from running into each other’s data. It categorizes all the File-System operations into read or write operations. Read-read and write-write operations are mutually exclusive but read-read operations are allowed to access the shared objects in parallel. Consequently, operations concurrent to other write operations working with any object are serialized.

In HDFS, in order to keep the implementation of locks simple, it locks the whole namesystem whenever a write operation needs to access it. This won't reduce the throughput in HDFS due to keeping the metadata in memory. However, in external database architecture, every operation takes the lock on the namesystem for longer time because of the network latency. This would make the sequential process of operations more expensive. It is not possible to scale out parallelization factor of the independent read-write or write-write operations so system-level lock happens to be a bottleneck in storage based implementation.

As a matter of fact, in the multiple NameNode implementation of HDFS, there is not any distributed synchronization between all the NameNodes’ in memory lock. Hence, practically system level lock became useless by losing its protection functionality for filesystems metadata.

4.2 Ensuring Strongly Consistent Semantics

Consistency in filesystems defines rules of concurrent access to shared objects both data and metadata, reading an object while being written and the other way round. As in HDFS, consistency model is definable from two perspectives, primitive NameNode operations and filesystem operations. In this study, we try to maintain the consistency semantics in primitive NameNode operations, as a result of so, the filesystem operations in KTHFS will support consistency semantics while each filesystem operation is a composition of primitive operations.
This study is interested in two levels of consistencies which are defined as following:

1. **Strong consistency**: In the distributed processing models, each process could be a reader or writer of data at a time. Data is strongly consistent if all the readers accessing data simultaneously get the the same view in which result of all the completed write operations are reflected.

2. **Weak consistency**: In weak consistency model, simultaneous readers could get different version of data and results of write operations not necessarily available to read immediately after write operations but eventually data goes into consistent state.

### 4.2.1 HDFS consistency model

1. **File System 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 [CAP] 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 non-atomic write operation. HDFS serializes read/write operations just at the primitive operations' level not the files block-data. 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.

### 4.2.2 Challenges in implementing HDFS Operations as Transactions

To begin with, we implemented each HDFS operation as a single transaction, where we began the transaction, read and wrote the necessary meta-data from NDB, and then either committed the transaction, or in the case of failure, the transaction was aborted and then possibly re-tried.
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*.

![Figure 11: Fuzzy read in KTHFS](image1)

![Figure 12: Phantom read in KTHFS](image2)

### 4.2.3 Consistency and Transaction Isolation Levels

NDB Cluster engine supports transactions. In order to discuss further about the type of transactions and limitations of NDB Cluster, first we define some common concepts about transactions.

A database transaction must provide ACID properties. ACID means:

- **Atomic**: Either all or none of the operations in a transaction succeed.
- **Consistent**: Any transaction will bring the database from a valid state to another valid state.
- **Isolated**: In the strongest case, it ensures the concurrent execution of transactions results in a system state that would be achieved if transactions were executed serially.
- **Durable**: Once the transaction has been committed, the changes will be persisted.

The isolation property [oracle] of transaction is defined to prevent three phenomenon which could appear in concurrent execution of transactions:

---

27
- **Dirty reads**: A transaction reads an uncommitted version of a data written by another transaction.
- **Nonrepeatable (fuzzy) reads**: A transaction reads again a previously read data and finds that the data is modified or deleted by another transaction.
- **Phantom reads (or phantoms)**: A transaction re-runs a query with a search condition and finds that another committed transaction has inserted a new row which satisfies the condition.

There are four levels of isolation that permit a transaction running at a particular isolation level to experience some of those aforementioned phenomenon. This is shown in table 3.

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read committed</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>

Read-uncommitted is the lowest isolation level which allows dirty reads. This means a transaction may see a change on a data which is not yet committed. Read-committed isolation level guarantees a transaction only see the last committed state of the data. In Repeatable-read isolation level, reads are repeatable. It means two read operations in the same transaction would return the same value of the data. However, it cannot guarantee that range queries would give the same result set. This is ensured in Serializable which is the highest isolation level.

One of the limitations of NDB Cluster storage engine is that it only supports Read Committed transaction isolation level [MYSQL-2]. This means that, in NDB Cluster, concurrent transactions could experience both fuzzy reads and phantoms.
4.3 Reducing the Number of Round-Trips to the Database

Introducing external database onto a system which is used to work with in memory objects, puts loads of network latencies into each query for accessing data. Converting each old-fashioned data access routine to a database query may result in excessively unnecessary network roundtrips. This is the case that the same query is being run more than once inside the same query which is not required. Therefore issuing queries should be performed carefully to reduce amount of roundtrips to database.
Chapter 5 - Solution

In this chapter we try to motivate the ideas behind our solution for assuring correctness of strong consistency model in primitive operations of HDFS NameNode. First we go through all parts of solution one by one at the semantic level, then we describe which problems being solved by each solution. In the scope of each solution new challenges appeared and so we had to deal with them by modifying and extending our solution, in order to overcome them.

5.1 Snapshot isolation

As it was already introduced, Mysql-cluster does not support more than read-committed isolation level. If we just rely on database isolation level the solution may manage to do fuzzy or/and phantom reads. If we could implement repeatable-read(snapshot) isolation level we can at least get rid of nonrepeatable(fuzzy) reads.

To do so we need to understand semantic of repeatable-read isolation level. Snapshot isolation as its name implies, works like this: transaction takes a snapshot of required data first, then it applies all the modifications on it and all the intermediate find operations just hit the snapshot not real data, so further on when operation is finished it flushes all the changes available in snapshot into database if there was not any parallel conflicting mutation.

5.1.1 Basic schema

The idea here is to use read-committed isolation level provided by Mysql-Cluster and try to implement snapshot isolation at the application side, so our schema for snapshot isolation looks like algorithm 2.

5.2 Avoid modification conflicts

Even though at first glance it seems implementing snapshot isolation is trivial, this schema is not a complete solution yet. The issue is preventing concurrent transaction to apply parallel mutations on same rows. There are two approaches to ensure mutual exclusion for concurrent updates:

1. Optimistic (Conflict Resolution): It proceeds commit stage of the transaction, if any conflict was found out, transaction is aborted otherwise it is committed. It is useful in the applications where conflicting operations are unlikely.

2. Pessimistic (Conflict Prevention): Instead of letting all concurrent modifiers precede, each transaction locks the required rows and prevents other transaction to access them before current transaction moves on the commit step.
Both approaches are applicable in the proposed snapshot isolation schema. Optimistic approach requires a validation phase in commit step while pessimistic approach should be handled in snapshotting function at the time of reading data. The pessimistic option is implemented in this thesis and optimistic is a great spot for future works.

Algorithm 2: Snapshot-isolation schema

```
initially: snapshot.clear;

operation doOperation
    tx.begin
    snapshotting()
    performTask()
    tx.commit

operation snapshotting
    foreach x in op do
        snapshot <- tx.find(x.query)

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

5.2.1 Row level locking as complement of snapshot isolation
To avoid fuzzy reads phenomena in snapshotting section, we have to make sure the read rows of data are not being changed anymore after first read. For this reason we introduce row level locking in our solution. In this solution database queries need to be categorized into read or write, shared lock is acquired for read queries and mutual lock is acquired for write ones. In this way whatever is being read in a transaction is guaranteed to be unmodifiable during the transaction life-cycle, so the snapshotting process is not being affected by fuzzy reads anymore. Algorithm 3 shows the lock update in the schema.

5.3 Semantically related mutations
Although row level locking prevents concurrent mutations on mutual data, it has no sense of the rows or operations which are semantically dependent. For instance, if namespace quota limit for “/d1/d2” is 1, while “d2” is empty, following concurrent transactions may manage to exceed the quota beyond its specified limit.
t1

checkQuota("/d1/d2")
addlnode("/d1/d2", "foo.mp3")

Exceeded quota limit!!!

Algorithm 3: Snapshot-isolation with row-level lock schema

initially: snapshot.clear;

operation doOperation
  tx.begin
  snapshotting()
  performTask()
  tx.commit

operation snapshotting
  foreach x in op do
    tx.lockLevel(x.lockType)
    snapshot <- tx.find(x.query)

operation performTask
  //Operation Body, referring to cache for data

As another case in point, two parallel transactions creating a file with a same name could ended up creating two files with same name inside the same directory which is against the filesystem semantic.
5.3.1 Phantom read happens in semantically related operations

As analysing all HDFS operations, all the phantom read cases could happen for the operations which are semantically dependent. Therefore, solution which is proposed for avoiding semantically related mutations could solve the phantom read in HDFS also. As an illustration counting number of blocks of a file could be pointed out:

\[
\begin{align*}
\text{t1} & \quad \text{t2} \\
\text{countBlocks("/d1/d2/foo.flv")} & \quad \text{addBlock("/d1/d2/foo.flv")} \\
\text{....} & \\
\text{countBlocks("/d1/d2/foo.flv")} &
\end{align*}
\]

Above example depicts that the first transaction gets two different numbers because the second transaction has added a new block meanwhile the two \text{countBlocks} in former. So we call \text{countBlocks} and \text{addBlock} for same file are semantically related though they are working on different rows.

5.3.2 Parent lock to avoid semantically related mutations

Parent lock is a notion introduced in this work to serialize semantically related operations. By knowing which operations are dependent we can specify a parent lock for them to be acquired at the as guarantee that just one of them is being performed at a time. Parent lock could be defined as new data-type, or lock on the table supported by database, or it could be a logical lock on a parent data-type. Given HDFS operations, there are always a logical dependencies between different data-types. For instance, \text{BlockInfos} are always dependent on the \text{file-inode}, meaning in all of the operations dealing with \text{BlockInfo} as data-type, they always resolve the corresponding Inode in the same transaction as well. Taking this dependency as granted, we could designate corresponding \text{Inode} as locking object of \text{BlockInfo}. If we could acquire a logical group lock for each set of objects, then we can serialize semantically related updates.

5.4 Deadlock prone scenarios

Locking concurrency control has it own shortcomings which have to be taken into account in our solution, different deadlock occasions are listed below.

1. **Conflicting lock order**
First, if two transactions are going to update the same two rows but in opposite orders then deadlock happens:

Figure 14: Locking in a conflicting order

2. **Lock upgrade**
   Second, deadlock could happen because of lock upgrades. If multiple threads acquire a read lock on the same row and try to upgrade the lock to the write lock, none of them will be able to achieve the write lock on the row. Because, it’s only possible to acquire the write lock when all read locks are released.

Figure 15: Lock upgrade runs into deadlock

5.4.1 Total order locking

To avoid transactions acquiring locks in conflicting orders, transactions should reach agreement on a total ordering on how data is read. That is, 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**:

\[
\forall (w_i, w_j) \text{ if } X_{w_i} \cap X_{w_j} \neq \emptyset \text{ then transactions of } (w_i, w_j) \text{ must be serialized}
\]

\[
\forall (r_i, w_j) \text{ if } X_{r_i} \cap X_{w_j} \neq \emptyset \text{ then transactions of } (r_i, w_j) \text{ must be serialized}
\]

To serialize operations all the operations should follow **total order rule**:

\[
\forall (x_n, x_m) \in X_{n} \cap X_{o} \text{ if } O_i Locks[x_n, x_m] \text{ then } O_j must Lock[x_n, x_m]
\]

In this way there is always one operation either \(O_i\) or \(O_j\) which can get the lock on \(x_n\) first and move on the rest. This operation is determined that not any other operation already took the lock on \(x_n\) and it is awaiting \(x_n\).
The original idea is borrowed from [GFS-1], though it has been extended to some extent. The main difference between GFS and HDFS is the way they modeled the metadata of the file-system. GFS stores full paths in a hash table while HDFS keeps data in a old fashioned inode data structure. For instance in case of file creation, GFS does not concern about keeping write lock on parents, because there is no directory or inode-like data structures while KTHFS takes all the directory ancestors into account. The novelty of our solution is that KTHFS deals with multiple tables, blocks and so on which higher up the difficulty of the lock solution.

All the locking operations happen while transaction take a snapshot of required data in memory. Total order lock acquisition is essential to ensure not any two transactions get stuck in any deadlock. Lock upgrades is also prevented by demanding the strongest type of lock required during the whole transaction on each row. Algorithm 4 shows updates in the schema.

Algorithm 4: Snapshot-isolation with total ordered row-level lock schema

5.5 Multi Transaction Operations

For some filesystem operations, the parameters supplied are not enough to ensure a total order on acquiring resources.

For example there might be a data-type \( y \) for which \( x \) is parent, and lock on \( x \) should be acquired to access \( y \). Since \( y \) is not supplied in the operations' parameters we need to add a preliminary state to find out \( y \) first, then start the main transaction by locking \( y \) then \( x \). In our first transaction we do not mutate state, so even after it commits, we can safely rollback.
All of the state mutations happen in the second transaction. In this way there is a risk that the state of the data is changed meanwhile between two transactions. Hence, the second transaction always need to validate the state of the data before start the operation, if it was safe it can go ahead and perform the operation otherwise the whole process should be restart from first transaction. We update our solution schema for multiple transaction in algorithm 5.

5.6 Solution at the concrete level

5.6.1 HDFS Total ordering challenges

This gives an overview about the kinds of locks that should be acquired for each operation on different tables, and reference to the kind of problem being faced for our locking solution. All of this problems with their proposed solution have been described in next section.
Algorithm 5: Multiple transactions schema

Initially: snapshot.clear, restart = true, try = 0;

Operation doOperation
    While restart and try < 3
        restart = false
        try = try + 1
        If op.should_resolve_parent Then
            tx1.begin
            resolve_parent(op.param)
            tx1.commit
        End If
    End While

Operation resolve_parent(y)
    tx.lockLevel("read_committed")
    tx.find(y.parent_query)

Operation snapshotting
    S = total_order_sort(op.X)
    For each x in S Do
        If x is a parent Then level = x.parent_level_lock
        Else level = x.strongest_lock_type
        tx.lockLevel(level)
        cache <-> tx.find(x.query)
    End For

Operation performTask
    //Operation Body, referring to cache for data
<table>
<thead>
<tr>
<th>#</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>How to know what inode is the last existing directory in the user-supplied path in order to acquire the write lock on it?</td>
<td>We acquire the existing path with the read-committed and then for the last existing inode (parent), then again the write lock should be acquired.</td>
</tr>
<tr>
<td>2</td>
<td>Redundant inodes with same name being created simultaneously, because name is not table's primary-key</td>
<td>Write lock on the parent inode-directory</td>
</tr>
<tr>
<td>3</td>
<td>Two opposite order for resolving lease exist, file-&gt;lease and lease-path&gt;lease</td>
<td>Supposing lease which is acquired by inode-file is the same as the Lease which is resolved by lease-path, then remove acquiring the lease by the lease-path and replace it with acquiring lease with the holder given in the inode-file.</td>
</tr>
<tr>
<td>4</td>
<td>Read order issue for reading multiple corrupted-replicas or excess-replicas in case we support three finders block-id, storage-id and primary-key.</td>
<td>Block could be shared lock for corrupted- replica and excess- replica, always read by block-id</td>
</tr>
<tr>
<td>5</td>
<td>Order issue for reading multiple block-infos of a file and deadlock problem.</td>
<td>We always take write lock of the belonging file of those blocks</td>
</tr>
<tr>
<td>6</td>
<td>Ordering issue of block-infos while reading multiple block-infos not related to the same file by block-id.</td>
<td>Since all the blocks not need to be processed together, in a priori no locking transaction first read all block-infos, then start out new per block transaction and resolve each block after its inode.</td>
</tr>
<tr>
<td>7</td>
<td>Ordering issue of replicas not necessarily belong to the same block when we get bunch of storage-ids in block-report.</td>
<td>In two priori transactions first read replicas by storage-id and resolve its block-id then in second transaction read each block and find its file's inode-id then it starts out new transaction per replica going with the basic order from file to replica.</td>
</tr>
<tr>
<td>8</td>
<td>Accessing multiple replica-under-constructions by storage-id</td>
<td>We could have got in-memory lock on storage-ids to avoid deadlocks coming from other operations given storage-id, though most of them has been changed to total order, and for this operation it seems not important.</td>
</tr>
<tr>
<td>9</td>
<td>HDFS updates the quota eagerly which means whatever update</td>
<td>We calculate quota lazily when required, since directories by default are non-quota enabled if</td>
</tr>
</tbody>
</table>
happens in a directory that directory and all of its ancestors should be updated in terms of occupied amount of both namespace and disc space. This requires write lock to be acquired on root all the way down to that path, it degrades the parallelization factor of KTHFS.

does not affect performance too much.

5.6.2 Total ordering categories

Our consistent total order solution has been categorized into the following sections:

1. **Basic order:**
   There is a natural ordering for resolving rows of tables in majority of operations. These operations are among the following set, the operations are micro level operations which are introduced in HDFS part of introduction.

   COMPLETE, GET_FILE_INFO, GET_CONTACT_SUMMARY, RECOVER_LEASE, HANDLE_HEARTBEAT, RENEW_LEASE, SET_OWNER, CONCAT, SET_TIMES, SET_REPLICATION, APPEND_FILE, ABANDON_BLOCK, SET_QUOTA, GET_MISSING_BLOCKS_COUNT, GET_BLOCKS_TOTAL, GET_NUMBER_OF_MISSING_BLOCKS, GET_PENDING_DELETION_BLOCKS_COUNT, GET_EXCESS_BLOCKS_COUNT, GENERATE_HEALTH_REPORT

   This order is shown in table 5. There are some steps for acquiring locks, all the steps should be executed in order. For tables fallen into the same step lock could be acquired in parallel though. In other words no cyclic dependencies has been found in any operation between the tables in the same step.

<table>
<thead>
<tr>
<th>Step</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Directory#1(root)</td>
</tr>
<tr>
<td>2</td>
<td>Directory#2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>Directory#n</td>
</tr>
<tr>
<td>n+1</td>
<td>File</td>
</tr>
<tr>
<td>n+2</td>
<td>Block-Infos, Leases</td>
</tr>
<tr>
<td>n+3</td>
<td>Lease-paths, Replicas, Corrupted-replicas, Under-construction-replicas, Under-replicated-block, Invalidated-block</td>
</tr>
</tbody>
</table>

Table 5: Basic order of lock acquiring
Basic order itself could have four variations for resolving inodes:

**a. Only path:**
This is for scenarios we just need to resolve the one path which is supplied by client, it is started from root inode to file or directory inode. All the path components are supposed to be available so it deterministically knows which inodes should be resolved from inode table e.g. `setPermission("dir1/dir2/f1.txt")`. Operations follow only path pattern are as following.

GET_ADDITIONAL_BLOCK, GET_BLOCK_LOCATIONS, SET_PERMISSION, GET_PREFERRED_BLOCK_SIZE, FSYNC

Example: r/d1/c1

![Figure 16: Locking just all component of given path](image)

**b. Immediate children:**
This happens when for last component of the specified path we need to lock all of its direct children.

GET_LISTING

Example: r/d1

![Figure 17: Locking all component of given path plus immediate children of last directory](image)

**c. Recursive children:**
For requests like “DELETE” all the children as of last component should be resolved recursively.

DELETE
d. **Unknown head:**
For those scenarios like “mkdir recursive” that we don’t know up till which part of the path is available and which part is not. Since lock level on last existing node should be different than the rest. These scenarios are:

MKDIR, START_FILE, GET_ADDITIONAL_DATANODE, CREATE_SYM_LINK

Example: r/d1/d2/d3/c1

![](image)

**Figure 19:** Locking just all existing component of given path while last existing component is unspecified.

2. **By Block:**
In the Basic order scenarios, the path is always given, while in some other operations the only given parameter is block-id. To make these scenarios adherent with basic ordering we have come up with the idea to read blockInfo with read-committed lock, then by having inode id, we start operation with the basic ordering. Following operations are among those:

BLOCK_RECEIVED_AND_DELETED, REPLICATION_MONITOR, ACTIVATION, COMMIT_BLOCK_SYNCHRONIZATION, UPDATE_BLOCK_FOR_PIPELINE, UPDATE_PIPELINE, FIND_AND_MARK_BLOCKS_AS_CORRUPT, GET_INODE, HEARTBEAT_MONITOR, DECOMMISION_MONITOR, REGISTER_DATANODE, REMOVE_DATANODE

The only flaw with this solution is that before starting basic ordering, data might have changed meanwhile, so always a validation is required if any update happens that changes data consistency, then operation should be failed.

Another case is when operation has list of block-ids like “BLOCK_RECEIVED_AND_DELETED”,

Example: r/d1
in this case we need to read all the corresponding blocks in preliminary read-committed phase then start out new transaction of basic ordering for each.

Table 6: Multi transaction operation when just block-id is given

<table>
<thead>
<tr>
<th>Phase</th>
<th>Step</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1: Read-Committed</td>
<td>1</td>
<td>given-blocks</td>
</tr>
<tr>
<td>t2: Basic-Order</td>
<td>1</td>
<td>file</td>
</tr>
<tr>
<td>t2: Basic-Order</td>
<td>2</td>
<td>block</td>
</tr>
<tr>
<td>t2: Basic-Order</td>
<td>3</td>
<td>replicas, corrupted-replicas, excess-replicas, under-replicated-block, pending-block, replicas-under-construction, invalidated-blocks</td>
</tr>
</tbody>
</table>

3. **By Lease:**
Since in the basic order “lease” always comes after path and file, in other scenarios getting started by “lease” they take locks exactly in the opposite order which is “lease”, “lease-path” and “file”. Since we have to use the same ordering to avoid dead-lock, we need to realize which files are involved in each scenario and start off the main scenario with files' lock acquired. There are two these kind of operations:

LEASE_MANAGER_MONITOR, ACTIVATION

Table 7: Multi transaction operation when just lease is given

<table>
<thead>
<tr>
<th>Phase</th>
<th>Step</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1: Read-Committed</td>
<td>1</td>
<td>given-leases</td>
</tr>
<tr>
<td>t1: Read-Committed</td>
<td>2</td>
<td>lease-paths</td>
</tr>
<tr>
<td>t2: Basic-Order</td>
<td>1</td>
<td>file</td>
</tr>
<tr>
<td>t2: Basic-Order</td>
<td>2</td>
<td>block</td>
</tr>
<tr>
<td>t2: Basic-Order</td>
<td>3</td>
<td>replicas, corrupted-replicas, excess-replicas, under-replicated-block, pending-block, replicas-under-construction, invalidated-blocks</td>
</tr>
</tbody>
</table>

5.7 Solution’s safety and liveness

The solution we provided, for either single or multi-transaction, serializes primitive HDFS operations as HDFS does, it is deadlock free and it doesn't change semantics of HDFS.
In terms of serialization, although it has higher parallelization factor than HDFS, it never runs dependent operations in parallel. The snapshot isolation guarantees that each transaction gets the latest committed consistent view of the data, besides pessimistic locking protects the current snapshotted data from mutations.

Solution is deadlock free because total order locking guarantees no circular dependencies are existed in the contention transactions. Since we get the strongest required lock for each object there is no risk of deadlock because of lock upgrades.

We protect side effects of concurrent writes to change semantic of the HDFS operations, for example parallel data insertions by forcing transactions to acquire parent locks for each metadata object if required.

Multi-transaction operations are also correct, because first transaction just reads the data and modifications are applied in the second transaction. The correctness of the data passed from first to second transaction is always validated so it never runs into invalidated data, in case any invalidity is found out in data, transaction is aborted and operations is restarted from start. Multi-transaction operations are also progressing, because restarting the operation just happens for a reasonably limited number of time if it cannot progress it throws an exception.
Chapter 6 - Implementation

6.1 Transaction context:

To avoid incorrect updates by stale data all the manipulations inside the same transaction are better to be applied on the same instance of each persistence object from very start point of the transaction till its very end. This solution helps out to avoid the throughput flaw by keeping track of all the read database rows and avoid doing this over and over per transaction. Figure 20 is the design of the transaction context:

![Diagram of Transaction Context]

The idea is to read each row just once in transaction's lifetime, get all the manipulations done on same attached objects, write all the new or updated objects altogether. As figure 20 shows, the main objects of transaction context are as following:

- **EntityManager**: Is the main interface of persistence layer exposed to rest part of the code for accessing persistent objects. It maintains a list of TransactionContext per
thread, indeed each thread just works on a single operation and transaction at a time, therefore each request coming into EntityManager being forward to the TransactionContext belonging to its thread. EntityManager creates a new context for each thread at the time of first request, that moment on the same TransactionContext is used for later requests.

- **TransactionContext**: It is the context for each transaction. It contains a context per each Entity-Type, so it dispatch operations coming for each Entity-Type into its own context. It is also responsible for managing life-cycle of transactions by beginning a transaction, ask EntityContext to prepare its modification for commit or ask for cleaning its status at the point of rollback.

- **EntityContext**: It has number of implementations equals to the number of entity-types. For instance BlockInfoContext is the EntityContext for BlockInfo, it does sort of caching for objects of each entity in the lifetime of the transaction. It means it always checks out its in memory cache for resolving new object if it gets a cache hit then it uses the same object otherwise it queries database only once. Correct instance management is the most important task of the EntityContext, because otherwise in case of having two instances of the same object, one might be ended up being persisted at the time of commit while it is not latest version. For that sake, it needs to synchronize instances of its entities when they are being find by another relation. As an illustration, in case of BlockInfo, it is really essential that INodeFile keeps the same instances of the BlockInfo as the instances already been found individually.

### 6.2 Persistence layer

To study how effective could be different database solutions in the Fail-Recovery model of the KTHFS Name-Node, we abstract out persistence layer. Indeed, to run the test classes for development purpose does not require doing operations on a real Mysql-cluster. One solution was to rewrite over 500 test classes using mocking libraries. It wasn't feasible for us to do so with the given time. The other solution was to use a lightweight embedded database. So for running the test classes and development purpose we chose Apache Derby database. By switching persistence into Derby, the development machine is offload of getting connected to MySql-cluster so we got faster development. Class diagram of persistence level is shown in figure 21.
The main classes of persistence layer consist of:

- **StorageFactory**
  
The factory class which builds all the storage implementation layer based on the current configuration. For instance if configuration is set up on MySql-Cluster then it builds its connector and for each entity-type its Clusterj data-access implementation.

- **StorageConnector**
  
  It is in charge of taking care of connections and sessions to database. It has connection pool, keeps all open till end and for each connections it manages its session per transaction. It does all the table creation and cleanup operations as well.

- **EntityDataAccess**
  
  Meaning accessor of data for each entity-type. It knows how to query database, how to do conversion of result-sets into entity-level objects. It has prepare method given removed, new and modified objects from its counterpart in transaction-context layer, all conversions of entity-objects into database level queries should be done and it runs queries. Prepare is called on all the data-access level objects by TransactionContext as first step of commit operation.
Chapter 7- Evaluation and Experiments

In this chapter, we provide an evaluation and comparison of JHDFS with its previous implementation. We discuss the impact of the snapshot on reducing the number of roundtrips to the database. The performance of Mysql Cluster on handling row level locks. And in the end the effect of implementation of row level lock in the NameNode’s performance.

All the Mysql Cluster nodes and the NameNode runs on machines each with 2 AMD 6 core CPUs (2.5 GHz clock speed, 512 KB cache size) connected with 1 GB ethernet. Indeed, we use JVM 1.6 and Clusterj 7.1.15a as Mysql Cluster connector.

7.1 The impact on the database roundtrips

As it is mentioned previously, the transaction context provides a snapshot isolation while caching the data retrieved from the database in the local memory. This reduces the number of roundtrips to the database and consequently the operation time. The effect of transaction context is shown in figure 22.

![Impact of Transaction Context](image)

Figure 22: Impact of transaction context on reducing number of database roundtrips

As it can be seen, GET_BLOCK_LOCATION is improved by more than 60 percent. This elevates the whole throughput of the system regarding reading files. For START_FILE operation this improvement is more than 40%. Indeed, operation COMPLETE is improved around 50%.
7.2 Row-level lock in Mysql Cluster

As it is mentioned before, we have been using Mysql Cluster as the metadata storage. The version of NDB Cluster is 7.2.8. Our setup of NDB Cluster has 4 DataNodes, each running on a different machine.

Since an external database is being used to store the metadata information, performance of the database could directly affect performance of the whole filesystem. It is important to evaluate the performance of row level lock mechanism in Mysql Cluster and to know the different lock modes’ overhead in Mysql Cluster. Because there is not any official benchmark related to concurrency control of NDBCluster, we implemented our version of it naming RowLevelLockMicroBenchmark. In this benchmark, enables us to measure the amount of time it takes for some specific number of threads to read some rows of data with a specific lock type. We chose inodes table having an already existing namespace data in it for the benchmark. This way would be more similar to the way Mysql Cluster’s row level lock is used in our implementation of Distributed File System.

We are interested to measure the effect of number of concurrent transactions taking different lock types on the same row. In this experiment, we increase the number of threads exponentially with base 2 and run it for each of the shared, exclusive and read-committed lock types. The namespace structure is shown in [namespace-1]. It contains of a root and a parent shared between all threads and a specific file per thread. All threads read root with no lock (read-committed) and then the shared parent with the lock specified in the benchmark.

![Diagram](image.png)

Figure 23: The namespace design to benchmark concurrency control performance in Mysql Cluster

The results for 10,000 operations are depicted in the diagram [bench-1].
As the number of threads is being increased, the time of doing 10,000 operations is decreased almost linearly for reading with shared lock or read commit. However, for write lock, we can see that the total time is halved for more than one thread but it cannot get lower than that. This is because only one thread can acquire write lock on the parent so rest of the threads have to wait in the queue on Mysql Cluster. However, multiple threads do better than one in reading with write lock. This is because the threads can read the root and only wait for the parent until the write lock on the parent is released. So this halves the total time in reading with write lock.

### 7.3 System Lock vs Row Level Lock

In order to compare NameNode using System Lock with NameNode using Row Level Lock, we implemented a NameNode benchmark which is influenced by NNThroughputBenchmark but with different namespace architecture. We call it GeneralINNThroughputBenchmark. In this benchmark, we want to measure the throughput of open and create operations on two different locking mechanism with a constant 64 number of threads while increasing number of directories (decreasing number of files per directory). The namespace design is as in figure 25. As it can be seen, the number of concurrent threads on each parent directory is a function of number of directories. In order to benchmark, we ran each experiment with the same settings for 3 times and then calculate the average.
The result of create operation for 16384 (2^{14}) files are depicted in figure 26.

As diagram shows, the throughput for creating files under a directory are the same for both system level lock and row level lock. This is because all 64 threads try to acquire a write lock.
on the same directory in NameNode with row level lock. In other words, this is almost as taking a system level write lock on FSNameSystem. The advantage of row level locking is revealed as we increase the number of directories. Increasing number of directories, we see that the throughput of NameNode with row level lock ramps up while for NameNode using System level locking the throughput is almost the same.

Since in Open operation, the number of directories does not affect to parallelization, we are only interested in general throughput of the two different locking mechanisms. To evaluate it, we ran the GeneralNNThroughputBenchmark for $2^{14}$ files with one directories. This means all the threads will take a read lock on the file in NameNode with row level lock and in NameNode using system level lock, all threads will acquire the read lock on FSNameSystem. The results are shown in figure 27.

![Figure 27: The comparison between NameNode with system level lock and the NameNode with row level lock for Open operation](image)

NameNode’s Open operation throughput is almost the same in both types of lockings. The throughput for row level lock is slightly lower than system level lock which could be due to the overhead of acquiring row level read locks in Mysql Cluster.
Chapter 8 - Conclusion and future work

8.1 Conclusion

In this thesis, we discussed some of the most important challenges in having a horizontally scalable HDFS NameNode using an external distributed datastore. We have implemented our solution on an ongoing project called KTHFS.

We defined the consistency problems of KTHFS and the challenges to ensure the consistency model of HDFS. We mentioned that HDFS is strongly consistent in the metadata level and having multiple NameNodes reading and writing, in parallel, would make the NameNode’s metadata inconsistent. Indeed, HDFS NameNode guarantees the metadata consistency using system level locks. However, this system level locks are so inefficient when we use an external datastore as a metadata storage. On the other hand by removing the system level lock in NameNode, even a single NameNode would lead to an inconsistent metadata. This inconsistency is due to the isolation level of Mysql Cluster. We shown that a metadata storage providing a read-committed isolation level, as in Mysql Cluster, is not enough to guarantee the HDFS consistency semantics.

To ensure the strong consistency semantic of HDFS NameNode, we introduced a client side (NameNode) snapshotting schema using pessimistic concurrency control (row level lock supported by the datastore). And digged through all the operations and defined the lock levels each resource requires for every operations.

Another challenge was to avoid deadlock problems appearing because of using locks. To solve the deadlock problem, we defined a total order on all of the persisting metadata. By ensuring the total order on all operations and avoiding lock upgrades we ensure that there will be no deadlock in the system.

However, to maintain HDFS operations semantics, snapshot isolation and locking resources are not enough. We solved this problem by defining parent locks. It means, for every group of resources, there is a single resource which is called parent. And all concurrent transactions are required to take a lock on the parent resource before accessing the group.

In some of the operations it is not possible to follow the total order we defined to avoid deadlocks. This is due to the input that the operations receives in the beginning. To solve this issue, we break the operation in multiple transactions in order to find all the resources required for acquiring the locks in the total order. We ensure the correctness of the operation by verifying any change on the data between each transaction. If the data meanwhile is changed, it will restart the whole operation for a specific number of times. An issue with this is the liveness problem. If the metadata is being changed continuously, this will cause these type of operations fail.

One of the other challenges of KTHFS was to reduce the excessive number of roundtrips to the datastore in NameNode. Our snapshotting solution, as we shown in evaluation, solved
the round trip problem by caching the read data in memory and pushing all the changes to the datastore when committing a transaction.

8.2 Future Works

8.2.1 Relaxing Locks

Currently, to ensure that the metadata is consistent and to preserve the semantics of HDFS operations we defined the lock levels required for each row of data strictly to avoid any decision based on the gut feeling. However, by further analysis operations and their relations some of the locks may become relaxer. This will increase concurrency of the NameNode and consequently will increase the total throughput of the filesystem.

8.2.2 Inter Transaction Cache

Snapshotting reduces the number of roundtrips to the database for every individual transaction by caching the read data in memory. But it does not avoid the round trips for two transactions reaching the same resources. For example, there can be a large batch operation all writing under a specific path. Caching the path components in memory could increase reduce the database queries and increase the performance.

8.2.3 Optimistic Transaction Conflict Resolution

To implement snapshotting, we used pessimistic concurrency control using locks to guarantee that the snapshot will not be modified by the concurrent transactions. The other approach is optimistic approach. In optimistic approach, after taking snapshot other concurrent transactions could modify the resources. However, there should be a verification phase in the end to abort the transaction if any of the data in the snapshot to detect conflicts. Either of the solutions, optimistic or pessimistic, have their pros and cons. For example, in the applications where concurrent transactions do not conflict frequently, we expect to have a better performance in optimistic implementation. However, frequent conflicts in optimistic approach would cause many transactions to abort in the verification phase.

8.2.4 NsCount and DsCount in INodeDirectory

Every INodeDirectory in HDFS has two variables called NsCount and DsCount which keep track of number of files/directories it has and amount of disks it uses. To update these informations in INodeDirectories, HDFS updates from child to the root. To do so in our implementation would be too expensive because it needs to take a write on the whole path. One way to solve this problem is to update NsCount and DsCount. There can be other solutions for this problem which requires further investigation.
References


[MySQL] Oracle July 2012


### Appendix

**Table 8: Complete NameNode Operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKDIR</td>
<td>Creates a directory recursively, it requires a no lock on all the existing components of the path but write lock on the last existing. This operation has P#1, P#2 and P#9.</td>
</tr>
</tbody>
</table>
| START_FILE(create)      | 1. If file does not exist, It creates inodes for all the non-existent directories and new file, writes owner of the lease and creates new lease-path.  
2. If file already exists first removes the file, its blocks and dependencies, lease and lease path, then it does the first scenario.  
3. This operation has P#1, P#2, P#3, P#9.                                                          |
<p>| GET_ADDITIONAL_BLOCK    | In the middle of writing a file, this is the client's mean of noticing name-node that the already being written block is finished while it is asking for the locations of next block. Name-node removes all the replica-under-constructions of last block, it also changes type of block-info from under-construction to completed one. It has P#4 and P#5. |
| COMPLETE                | Like get_additional_block, it happens for last block of the file, name-node just removes the replica-under-constructions and changes type of block-info from under-construction to completed.                                      |
| BLOCK_RECEIVED_AND_DELETE | Data-nodes notifies name-node about blocks it has been received or deleted as of last report. It has P#6.                                                                                               |
| DELETE                  | Client could issue this command either for a file or directory. In case of file, name-node removes all the blocks and replicas of that file and creates new invalidated-block for each replica. In case of directory, it removes all the subdirectories and files recursively. P#9 |
| PROCESS_REPORT(block report) | When data-node periodically sends list of available replicas to name-node to get its status updated in name-node and name-node goes through all the blocks and updates any possible non-updated status of a block, if it is becoming under-replicated or over-replicated or it happens to turn from under-replicated into enoughly replicated. This operations has P#7. |
| GET_FILE_INFO | This operation is to read namely information of a file from its inode like size etc and the real information on disc like size of its blocks. |
| GET_CONTENTE_SUMMARY | Client expects name-space and disc-space information. Given a directory it returns both quota limitation on name-space and disc-space as well as available amounts. In case of given a file, it just returns disc-space information. |
| GET_BLOCK_LOCATIONS | Given path of the file, it returns location if its blocks and updates access-time of the file_inode. |
| LEASE_MANAGER_MONITOR | A background operation happening often in name-node, since each lease has soft-limit and hard-limit, it goes through all the available leases, it recovers leases which have met the soft limit and removes leases have touched the hard-limit. By default soft-limit is 60 seconds and hard-limit is 60 minutes. |
| RECOVERLEASE | It is a facility for a client waiting for a lease which is held by another client to take over lease, write hold, of that file. If all the anticipated blocks of the file are completed, it closes the file and changes it from under-construction to completed. If the last block is under construction, the block recovery will be started for that block and lease is recovered. |
| HEARTBEAT_MONITOR | Periodically goes through list of data-nodes, and picks a dead data-node. For that it removes its replicas, and fixes status of the block. P#4 &amp; P#7. |
| HANDLE_HEARTBEAT | It just considers if there is any under-replicated block in the data-nodes descriptor, returns the block-recovery command to data-node. |
| REPLICATION_MONITOR | Checks if there are any pending replications which are timed-out to add them to the under replicated lists in order to be replicated again. Computing data-nodes work contains computing replication work and computing invalidate work. |
| DECOMMISSION_MONITOR | It checks if replication is in progress for the replicas of the datanode or not. If replication is finished then decommissions the datanode. |
| INITIALIZE | Name-node is started for the first time then adds the root directory if name-node is restarted then just reads the root. |
| ACTIVATION | Begins LeaseManager and Resource Monitor threads. |
| RENEW_LEASE | Client asks name-node to renew it lease time to |</p>
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_ADDITIONAL_DATANODE</td>
<td>Client who is not holder of the file asks for more data-node.</td>
</tr>
<tr>
<td>META_SAVE</td>
<td>This operation is used for original HDFS, it dumps all metadata into a file. It is not being used anymore.</td>
</tr>
<tr>
<td>SET_PERMISSION</td>
<td>Changes permission of a file while logs available blocks of the file.</td>
</tr>
<tr>
<td>GET_OWNER</td>
<td>Changes owner of a file while logs available blocks of the file.</td>
</tr>
<tr>
<td>GET_STATS</td>
<td>Gets the status of DataNodes and their replicas.</td>
</tr>
<tr>
<td>CONCAT</td>
<td>It concatenates contents of two files into one. In HDFS it is only possible to concat files located in the same directory.</td>
</tr>
<tr>
<td>SET_TIMES</td>
<td>Updates access-time and modification-time of a given file.</td>
</tr>
<tr>
<td>CREATE_SYM_LINK</td>
<td>Creates symbolic link pointing to the target path without resolving the target path.</td>
</tr>
<tr>
<td>GET_PREFERED_BLOCK_SIZE</td>
<td>Return the block size of a specified file.</td>
</tr>
<tr>
<td>SET_REPLICATION</td>
<td>Sets up new replication factor for the given file.</td>
</tr>
<tr>
<td>APPEND_FILE</td>
<td>A client wants to append at the end of a given file, if lease of the file is already taken by another client and its soft limit is expired then its lease will be recovered for new client.</td>
</tr>
<tr>
<td>ABANDON_BLOCK</td>
<td>Removes a block from the pending creates list.</td>
</tr>
<tr>
<td>SET_QUOTA</td>
<td>Sets up or changes namespace and disk space limitation.</td>
</tr>
<tr>
<td>RENAME_TO</td>
<td>Renames a file or directory or moves from a source to a destination. P#3, P#9</td>
</tr>
<tr>
<td>FSYNC</td>
<td>It used to do checkpointing of in memory data structures into disc. Though it does not do anything special in KTHFS since the blocks are all persisted in the database when they are created.</td>
</tr>
<tr>
<td>COMMIT_BLOCK_SYNCRONIZATION</td>
<td>Updates block’s size and generation stamp.</td>
</tr>
<tr>
<td>GET_LISTING</td>
<td>Returns list of available files and directories in the given path.</td>
</tr>
<tr>
<td>REGISTER_DATANODE</td>
<td>Registers a datanode to the NameNode.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GET_MISSING_BLOCK_COUNT</td>
<td>Returns number of the blocks which are under replicated with specific level, it just needs to count under-replicated-blocks table.</td>
</tr>
<tr>
<td>GET_BLOCKS_TOTAL</td>
<td>Returns total number of blocks in the system.</td>
</tr>
<tr>
<td>UPDATE_BLOCK_FOR_PIPELINE</td>
<td>Pipeline is a sequence of clients followed by some data-nodes for writing replicas of a block. For any reason a client might ask for a new generation stamp and secret token to build a new pipeline for a specific block to be written.</td>
</tr>
<tr>
<td>UPDATE_PIPELINE</td>
<td>Updates the pipeline given an old block, new block and list of new data-nodes.</td>
</tr>
<tr>
<td>LIST_CORRUPT_FILE_BLOCKS</td>
<td>Returns a list of corrupted files/blocks.</td>
</tr>
<tr>
<td>GET_NUMBER_OF_MISSING_BLOCKS</td>
<td>Returns number of the blocks which are under replicated, it just needs to count under-replicated-blocks table.</td>
</tr>
<tr>
<td>GET_PENDING_DELETION_BLOCKS_COUNT</td>
<td>Returns number of invalidated-blocks subject to be deleted.</td>
</tr>
<tr>
<td>GET_EXCESS_BLOCK_COUNT</td>
<td>Returns number of over-replicated-blocks.</td>
</tr>
<tr>
<td>FIND_AND_MARK_BLOCKS_AS_CORRUPT</td>
<td>Given a block-id, client or data-node report a block as corrupted by this method.</td>
</tr>
<tr>
<td>REMOVE_DATANODE</td>
<td>It removes a data-node from system meaning all the data related to that data-node needed to be removed. P#6</td>
</tr>
<tr>
<td>GET_DELEGATION_TOKEN</td>
<td>NameNode requires to authenticate clients before processing their requests. To do so, NameNode uses Kerberos authentication. However, in order to prevent high load on Kerberos KDC (Key Distribution Center), NameNode generates a delegation token for every client after authenticating them via Kerberos for the first time and identify the clients by delegation token in the later requests.</td>
</tr>
</tbody>
</table>