Optimistic Concurrency Control in a Distributed NameNode Architecture for Hadoop Distributed File System

Qi Qi

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Manuel Antunes Veiga
Advisor: Prof. Jim Dowling

Examination Committee

Chairperson:	Prof. José Carlos Alves Pereira Monteiro
Supervisor:	Prof. Luís Manuel Antunes Veiga
Member of the Committee:	Prof. Nuno Manuel Ribeiro Preguiça

September 2014

Acknowledgments

The work presented is delivered as final thesis report at Instituto Superior Técnico - IST (Lisbon, Portugal). It is in partial fulfillment of the European Master in Distributed Computing - EMDC program 2012-2014. Royal Institute of Technology - KTH (Stockholm, Sweden) is the coordinator for this Erasmus Mundus master program. The study track has been composed of a first two semesters at IST, 3rd semester at KTH, and for this work and 4th semester, a degree project in Computer Systems Laboratory at Swedish Institute of Computer Science - SICS (Stockholm, Sweden).

Special thanks to my advisor Dr. Jim Dowling for his support throughout the project. With more than ten years' professional industry experience, Jim is always patient to help. He's the cool guy who gives answers faster than Google and StackOverFlow.

Thanks to Salman Niazi and Mahmoud Ismail for all the practical help. Without them I might have to spend quite a long time studying the code base of the precedent work.

I'm also grateful to my supervisor Prof. Luís Antunes Veiga for his continuous support and encouragement. When I was in IST, I liked staying in the classroom after his class and chatted with him for a while. Veiga was like a big brother there taking care of us.

I would like to thank the good friends I met in Portugal and Sweden, who leveled me up during these two years. Without you guys, this journey wouldn't have been such a legendary in my life.

I am truly thankful to my family for nursing me with all their affections and love.

Last, special appreciation to this young man, Qi Qi, who always has the guts to take any adventure in his life.

September 13, 2014, Stockholm

Qi Qi

Dedication

To my father, a man of integrity, who supports all my adventurous decisions so that I can live outside of the box.

Resumo

[To be added] Portuguese Abstract

Abstract

The *Hadoop Distributed File System* (HDFS) is the storage layer for Apache Hadoop ecosystem, persisting large data sets across multiple machines. However, the overall storage capacity is limited since the metadata is stored in-memory on a single server, called the *NameNode*. The heap size of the NameNode restricts the number of data files and addressable blocks persisted in the file system.

The *Hadoop Open Platform-as-a-service* (Hop) is an open platform-as-a-Service (PaaS) support of the Hadoop ecosystem on existing cloud platforms including Amazon Web Service and OpenStack. The storage layer of Hop, called the Hop-HDFS, is a highly available implementation of HDFS, based on storing the metadata in a distributed, in-memory, replicated database, called the *MySQL Cluster*. It aims to overcome the NameNode's limitation while maintaining the strong consistency semantics of HDFS so that applications written for HDFS can run on Hop-HDFS without modifications.

Precedent thesis works have contributed for a transaction model for Hop-HDFS. From system-level coarse grained locking to row-level fine grained locking, the strong consistency semantics have been ensured in Hop-HDFS, but the overall performance is restricted compared to the original HDFS.

In this thesis, we first analyze the limitation in HDFS NameNode implementation and provide an overview of Hop-HDFS illustrating how we overcome those problems. Then we give a systematic assessment on precedent works for Hop-HDFS comparing to HDFS, and also analyze the restriction when using pessimistic locking mechanisms to ensure the strong consistency semantics. Finally, based on the investigation of current shortcomings, we demonstrate how to improve the performance by designing a new model based on optimistic concurrency control with snapshot isolation as a proof of concept. The evaluation shows the significant improvement of this new model. The correctness of our implementation has been validated by 300+ Apache HDFS unit tests passing.

Palavras Chave
Keywords

Palavras Chave [To be corrected by native Portuguese speaker]

HDFS

MySQL Cluster

Controle de Concorrência

Snapshot Isolation

Transação

Vazão

Keywords

HDFS

MySQL Cluster

Concurrency Control

Snapshot Isolation

Transaction

Throughput

Index

List of Figures

List of Tables

I

Introduction and Background

1 Introduction

1.1 Motivation

1.1.1 The De Facto Industrial Standard in Big Data Era

The *Apache Hadoop* (Hadoop) ecosystem has become the de facto industrial standard to store, process and analyze large data sets in the big data era (Cloudera). It is widely used as a computational platform for a variety of areas including search engines, data warehousing, behavioral analysis, natural language processing, genomic analysis, image processing, etc (Shvachko 2011).

The *Hadoop Distributed File System* (HDFS) is the storage layer for Apache Hadoop, which enables petabytes of data to be persisted on clusters of commodity hardware at relatively low cost (Borthakur 2008). Inspired by the *Google File System* (GFS) (Ghemawat et al. 2003), the namespace, *metadata*, is decoupled from data and stored in-memory on a single server, called the *NameNode*. The file datasets are stored as sequences of blocks and replicated across potentially thousands of machines for fault tolerance.

1.1.2 Limits to growth in HDFS

Built upon the single namespace server, *the NameNode*, architecture, one well-known limitation of HDFS is the limitation to growth (Shvachko 2010). Since the metadata is kept in-memory for fast operation in NameNode, the number of file objects in the filesystem is limited by the amount of memory of the NameNode.

Approximately, the size of the metadata for a single file object having two blocks (replicated three times by default) is 600 bytes. As a rule of thumb, for one petabyte physical storage, it requires one gigabyte metadata in memory (Shvachko 2010). Table 1.1 gives an estimation of the memory requirement and its related physical storage capacity for different number of files.

Number of Files	Memory Requirement	Physical Storage
1 million	0.6 GB	0.6 PB
100 million	60 GB	60 PB
1 billion	600 GB	600 PB
2 billion	1200 GB	1200 PB

Table 1.1: Memory Requirement for Related Storage Capacity in HDFS

As HDFS runs in the *Java Virtual Machine* (JVM), due to interactive workloads, heap sizes larger than 60 GB is not considered practical (Shvachko 2010). Therefore, 100 million files will be the maximum storage capacity of HDFS.

1.1.3 Hop-HDFS and Its Limitation

The *Hadoop Open Platform-as-a-service* (Hop) (Dowling 2013) is an open platform-as-a-Service (PaaS) support of the Hadoop ecosystem on existing cloud platforms including Amazon Web Service and OpenStack. The storage layer of Hop, called the Hop-HDFS, is a highly available implementation of HDFS, based on storing the metadata in a distributed, in-memory, replicated database, called the *MySQL Cluster*. It aims to overcome the NameNode's limitation while maintaining the strong consistency semantics of HDFS so that applications written for HDFS can run on Hop-HDFS without modifications.

Precedent thesis works have contributed for a transaction model (Wasif 2012) (Peiro Sajjad & Hakimzadeh Harirbaf 2013) as well as a high availability multi-NameNode architecture (D'Souza 2013) for Hop-HDFS. It can store up to 4.1 billion files with 3TB MySQL Cluster support for metadata (Hakimzadeh et al. 2014).

However, in HDFS, the correctness and consistency of the namespace is ensured by atomic metadata mutation (Shvachko et al. 2010). In order to maintain the same level of strong consistency semantics, system-level coarse grained locking and row-level fine grained locking are adopted in precedent projects of Hop-HDFS, but the overall performance is heavily restricted compared to the original HDFS. Therefore, investigation for better concurrency control to improve the performance of Hop-HDFS is the main motivation.

1.2 Problem Statement

In HDFS, the NameNode's operations are categorized into *read* or *write* operations. To protect the metadata among parallel running threads, a global read/write lock (fsLock in *FSNamesystem - ReentrantReadWriteLock* in java language) is used to maintain the atomicity of the namespace. We call it *system-level lock*. Although ReentrantReadWriteLock (Oracle b) adopts a similar idea from *two-phase locking* (Berenson et al. 1995), it has other locking semantics including *fair mode, lock interruptions, condition support, etc,* which means that it is not totally equal to two-phase locking.

Concurrent threads to access shared object for read operations are allowed, but it restricts a single thread to access object for write operations. Therefore, all concurrent readers get the same view of the mutated data reflected by completed writes. We call it *Strong Consistency Semantics* in HDFS. This *single-writer-multiple-readers* concurrency model will not reduce the throughput much since the metadata is kept optimized data structures in-memory (Hakimzadeh et al. 2014) so the related operations on them are fast.

The first version of Hop-HDFS, called the KTHFS (Wasif 2012), adopts the system-level locking mechanism to serialize transactions. The strong consistency semantics is maintained, but due to the network latency from the external database architecture, each operation takes a long time lock on the filesystem. The performance is heavily degraded.

The second version of Hop-HDFS adopts a fine-grained row-level locking mechanism to improve the throughput (Hakimzadeh et al. 2014) (Peiro Sajjad & Hakimzadeh Harirbaf 2013) while maintaining the strong consistency semantics. Based on a hierarchical concurrency model, it builds a *directed acyclic graph* (DAG) for the namespace. Metadata operation that mutates the DAG either commit or abort (for partial failures) in a single transaction. *Implicit locking* (Gray et al. 1976) is used to take an explicit lock on the data row of the root of a subtree in a transaction, which implicitly acquires locks on all the descendants. However, this approach lowers the concurrency when multiple transactions try to mutate different descendants within the same subtree.

Besides the concurrency issue, there are challenges when implementing each HDFS operation as a single transaction. The storage engine, *NDB*, of MySQL Cluster supports only the *READ COMMITTED* transaction isolation level (MySQL d), the write results in transactions will be

exposed to read in different concurrent transactions. Without proper implementation, anomalies like *Lost Update, Fuzzy Read, Phantom, Read Skew and Write Skew* (Berenson et al. 1995) will generate incorrect results.

1.3 Contribution

In this thesis, we contribute to the following three ways:

- First, we analyzes the limitation of HDFS's NameNode implementation, with focus on the namespace locking mechanism.
- Second, we provide a systematic performance assessment of the distributed NameNode architecture in Hop-HDFS comparing to original HDFS while maintaining the strong consistency semantics.
- Third, we demonstrate how to improve the performance by designing a new model based on optimistic concurrency control with snapshot isolation as a proof of concept. The evaluation shows the significant improvement of this new model, and the correctness of our implementation has been validated by 300+ Apache HDFS unit tests passing.

1.4 Document Structure

[To be added after finishing the whole document.]

2 Background and Related Work

2.1 Distributed File Systems

Distributed File systems is the fundamental in big data era. They provide a high available storage service with fault tolerance for data corruption, which enable petabytes of data to be persisted across multiple low cost commodity machines reliably.

2.1.1 The Google File System

2.1.1.1 Design Principle

The Google File System (GFS) is a scalable distributed file system developed and widely used in *Google Incorporation* for large distributed data-intensive applications. With fault tolerance, it runs on clusters of inexpensive commodity hardware, which provides a storage layer for a large number of applications with high aggregate performance (Ghemawat et al. 2003). There are some design assumptions for the implementation of GFS:

- The system runs on top on inexpensive commodity hardware so component may often fails.
- Files stored on the system are fairly huge than the transitional standards, which means that Gigabyte files are common.
- There are three kinds of workloads in the system: large streaming reads, small random reads and large sequential writes which append data to files.
- Efficiently well-defined semantics for concurrent appends to the same file is needed.
- Data processing in bulk with high sustained bandwidth is more important than individual read or write with low latency.

2.1.1.2 The Architecture of GFS

The architecture of a GFS cluster consists of a single *master*, multiple *chunkservers*, and is accessed by multiple *clients* as shown in Figure 2.1.

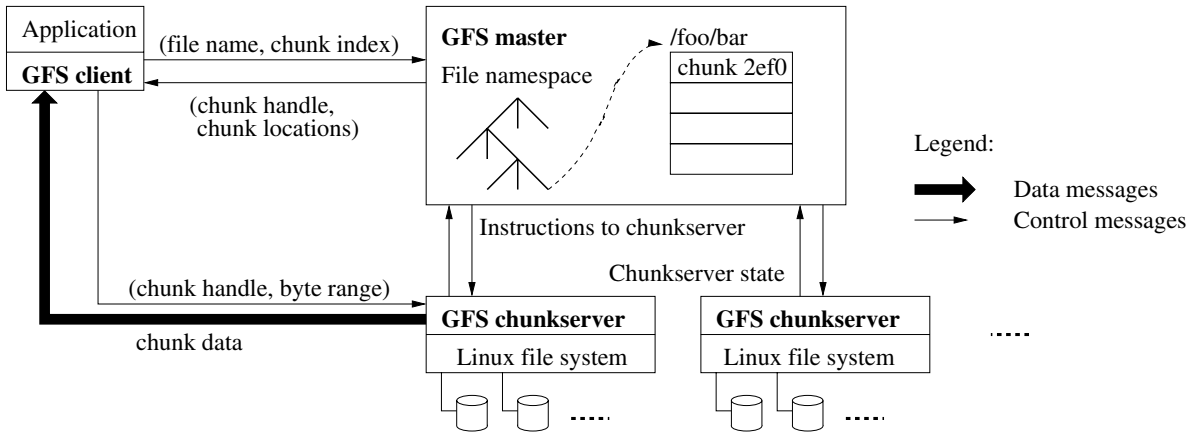


Figure 2.1: The Architecture of GFS (Ghemawat et al. 2003)

Files are divided into fixed size *chunks* stored in *chunkservers*. For fault tolerance, each chunk is replicated across multiple chunkservers and the default replication factor is three.

The *master* is a metadata server maintaining namespace, access control information, the file-chunk mappings and chunks' current locations. Besides, it is also responsible for system-wide activities including garbage collection, chunk lease management, chunk migration between chunkservers.

Although this single master server architecture simplifies the design of GFS, especially on complexed tasks like chunk placement and replication decisions using global knowledge, yet the master's involvement in reads and writes needs to be minimized otherwise it will become a bottleneck in the system.

2.1.2 The Hadoop Distributed File System

2.1.2.1 Design Principle

The *Hadoop Distributed File System* (HDFS) is inspired by the Google File System. Initially, HDFS is built for Hadoop Map-Reduce computational framework. With the development of Hadoop

ecosystem including HBase (HBase), Pig (Pig), Mahout (Mahout), Spark (Spark), etc, HDFS becomes the storage layer for other big data applications. Enabling petabytes of data to be persisted on clusters of commodity hardware at relatively low cost, HDFS aims to stream these large data sets at high bandwidth to user applications. Therefore, like GFS, HDFS is optimized for delivering a high throughput of data at the expense of latency (White 2012).

2.1.2.2 The Architecture of HDFS

Similar to GFS, HDFS stores metadata and file data separately. The architecture of a HDFS cluster consists of a single *NameNode*, multiple *DataNodes*, and is accessed by multiple *clients* as shown in Figure 2.2.

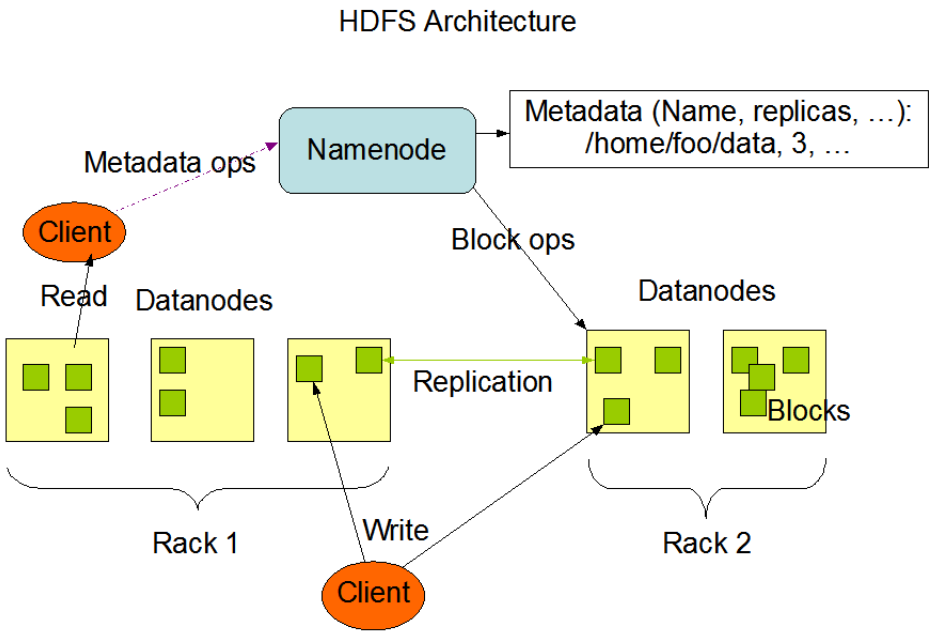


Figure 2.2: The Architecture of HDFS (Borthakur 2008)

Files in HDFS are split into smaller blocks stored in *DataNodes*. For fault tolerance, each block is replicated across multiple *DataNodes*.

The *NameNode* is a single dedicated metadata server maintaining the namespace, access control information, and file blocks mappings to DataNodes. The entire namespace is kept in-memory, called the *image*, of the *NameNode*. Its related persistent record, called the *checkpoint* is stored in the local physical file system. The modification, *editlogs*, of the *image*, called the *journal*, is also persisted in the local physical file system. Copies of the *checkpoints* and the *journals* can made

at other servers for durability. Therefore, the *NameNode* restores the namespace by loading the checkpoint and replaying the journal during its restart.

2.1.2.3 Single-Writer, Multiple-reader Model

Once a file is created, written with data and closed by the client application, the bytes written can not be modified. The file can only be reopened for append.

HDFS implements a *single-Writer, multiple-reader* model using lease management. A HDFS client opens a file for writing is granted a lease for the file and no other client can write to that file at the same time. The writing client needs to renew the lease periodically with the NameNode so that it can keep writing to the file. Otherwise, once the *soft limit* expires, other clients can preempt the lease. If the *hard limit* (one hour) expires and the client didn't renew the lease, HDFS will close the file on behalf of the writer and recover the lease.

HDFS allows a client to read a file which is open for writing, which means that the lease does not prevent other clients' reading. A file can have multiple concurrent readers.

2.2 Concurrency Control and Isolation Level in Transactional Systems

2.3 MySQL Cluster

2.3.1 Design Principle

MySQL Cluster is a highly available version of MySQL, an open source database management system, with high-redundancy adapted for the distributed computing environment. It integrates the standard MySQL server with an in-memory clustered storage engine called *NDB* (which stands for "Network DataBase"). MySQL Cluster is designed not to have any single point of failure a shared-nothing system running on inexpensive hardware (MySQL a).

2.3.2 The Architecture of MySQL Cluster

A MySQL Cluster consists of different processes, called the *nodes*. The communication between the nodes can be seen from Figure 2.3. *MySQL Servers* (mysqld, for query processing and NDB

data accessing) are the main nodes. *Data Nodes* (ndbd) serve as storage nodes. Besides, there will be one or more *NDB Management Servers* (ndb_mgmd).

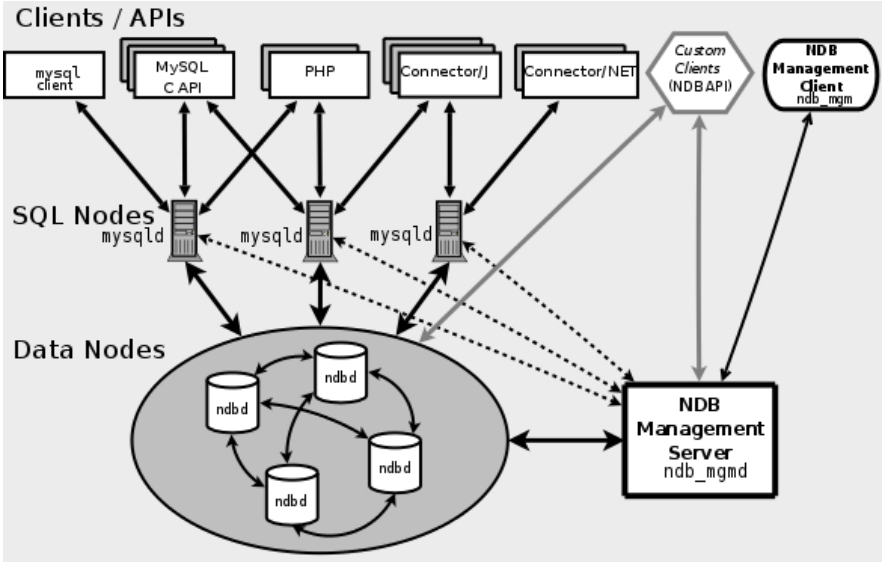


Figure 2.3: The Architecture of MySQL Cluster (MySQL a)

For fault tolerance, data in MySQL Cluster is replicated across multiple ndbds. Ndbds are divided into *node groups*. Each unit of data is called a *partition* stored by ndbd. The partitions of data are replicated within the same *node group*. The number of *node groups* is calculated as:

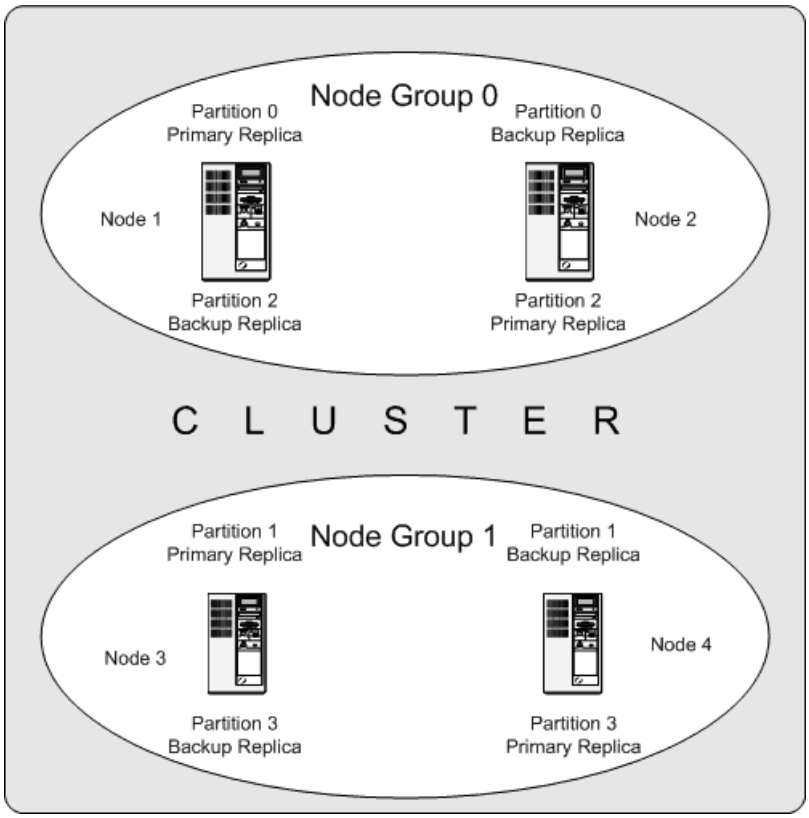
$$Number of NodeGroups = \frac{Number of DataNodes}{Number of Replicas}$$

For example, suppose that we have a cluster consisted with 4 data nodes with replication factor of 2, so there are 2 node groups as shown in Figure 2.4.

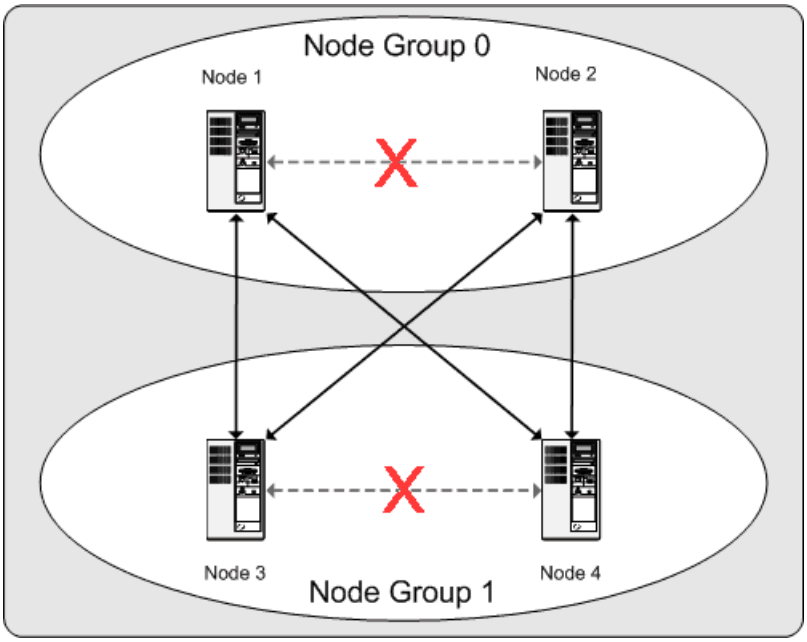
As we can see from Figure 2.4(a), the data stored in the cluster is divided into four partitions: *0, 1, 2, 3*. Each partition is stored within the same group with multiple replicas. So as long as each participating node group has at least one operating node, the cluster will have a complete copy of the data as shown in Figure 2.4(b). For example, suppose that *Node 2* and *Node 3* are operating, then partitions *0, 1, 2, 3* remain viable.

2.3.3 The Benchmark of MySQL Cluster

According to the white paper published by Oracle (MySQL 2012), MySQL Cluster can handle:



(a) Node Groups in MySQL Cluster



(b) Fault Tolerance in Node Groups

Figure 2.4: Node Groups in MySQL Cluster and Fault Tolerance (MySQL f)

- 4.3 Billion fully consistent reads per minute
- 1.2 Billion fully transactional writes per minute

They used an open source benchmarking tool, *FlexAsynch*, to test the performance and scalability of a MySQL Cluster running across 30 commodity Intel Xeon E5-equipped servers, comprised 15 node groups. The result for the write operation performance is shown in Figure 2.5.

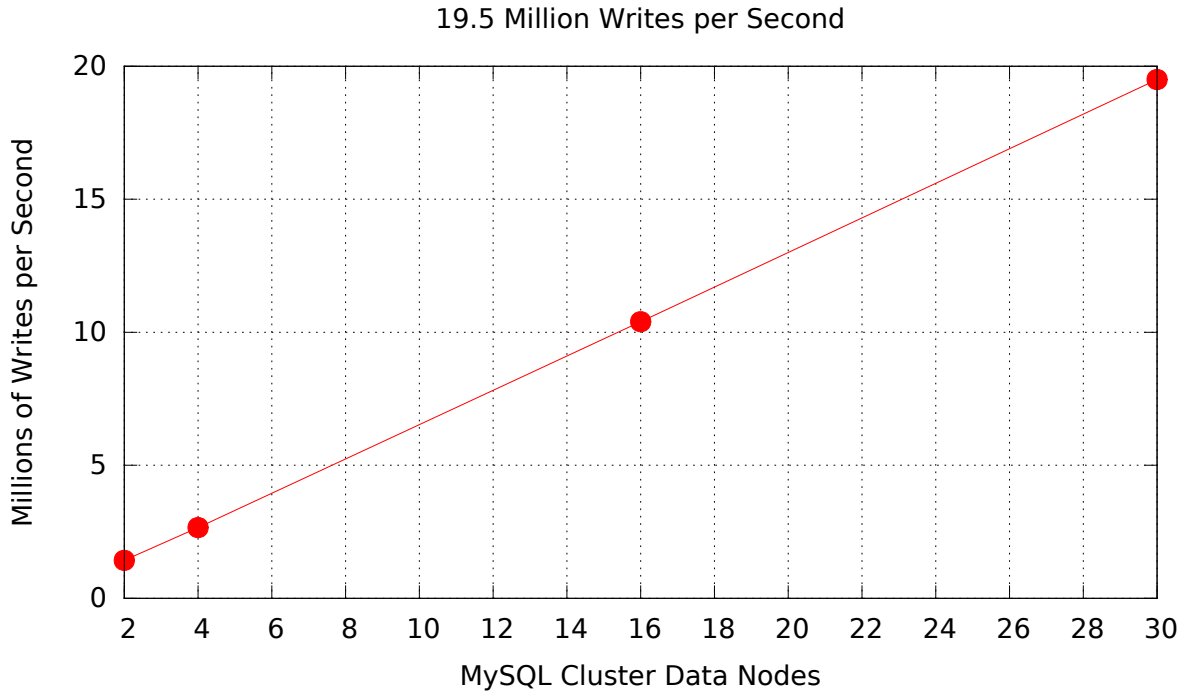


Figure 2.5: MySQL Cluster Scaling-out Writes Operations

Therefore, the high operations throughput, *72 million reads and 19.5 million writes operations per second*, of MySQL Cluster makes it the choice to be the distributed in-memory storage layer for the metadata in Hop-HDFS. But the trade off is that the NDB cluster storage engine supports only the *READ COMMITTED* transaction isolation level (MySQL e), which means that we need to put extra effort on the application layer to preclude anomalies in our implementation. See Chapter 5.

2.4 Hadoop Open Platform-as-a-service and Hop-HDFS

2.4.1 Hadoop Open Platform-as-a-service Design Purpose

The *Hadoop Open Platform-as-a-service* (Hop) (Dowling 2013) is an open platform-as-a-Service (PaaS) support of the Hadoop ecosystem on existing cloud platforms including Amazon Web Service and OpenStack. The goal is to automate the installation of both HDFS and Apache YARN so that unsophisticated users can deploy the stack on the cloud easily by a few clicks from our portal website.

2.4.2 Overcoming Limitations in HDFS NameNode Architecture

The storage layer of Hop, called the Hop-HDFS, is a new high available model for HDFS's metadata, aiming to overcome the major limitations of HDFS NameNode architecture:

- **The scalability of the namespace**: the memory size restricts the storage capacity in the system.
- **The throughput problem**: the throughput of the metadata operations is bounded by the ability of the single machine (NameNode)
- **The failure recovery**: it takes a long time for the NameNode to restart since it needs to load the checkpoint and replay the edit logs into the memory

2.4.3 The Architecture of Hop-HDFS

The architecture of Hop-HDFS consists of multiple *NameNodes*, multiple *DataNodes*, a *MySQL cluster* and is accessed by multiple *clients* as shown in Figure 2.6.

The design purpose for Hop-HDFS is to migrate the metadata from NameNode to an external distributed, in-memory, replicated database *MySQL Cluster*. Therefore, the size of the metadata is not limited by a single NameNode's heap size and the scalability problem is solved. We can now have multiple stateless NameNodes architecture so that multiple-writers-multiple-readers are allowed to operate on the namespace to improve the throughput.

Moreover, the fault tolerance of the metadata is handled by MySQL Cluster, which grantees high availability of 99.999%. The *checkpoint* and the *journal* for namespace is not needed, which

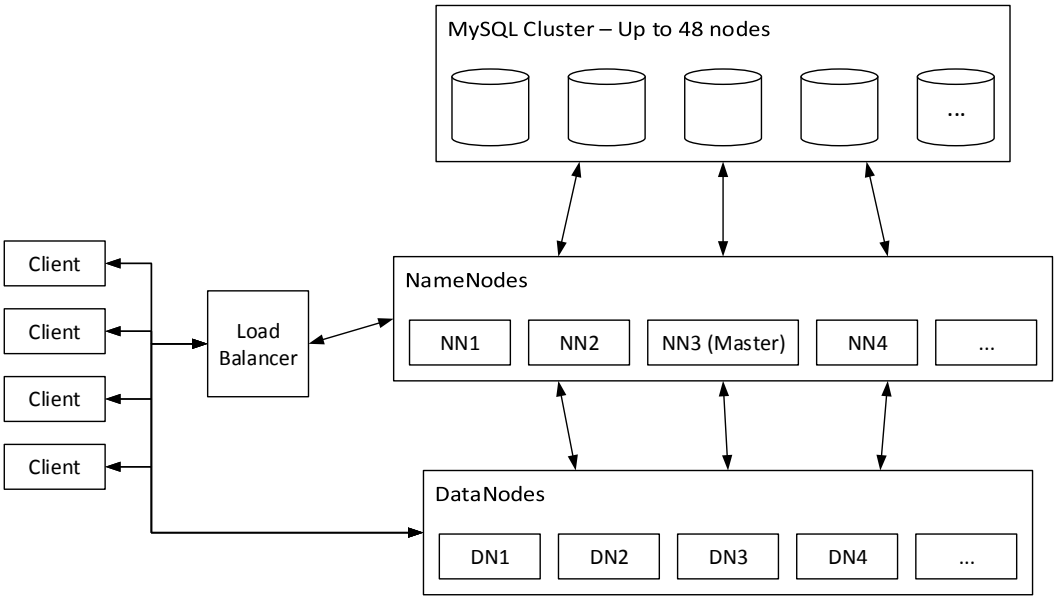


Figure 2.6: The Architecture of Hop-HDFS

save time from writing edit logs as well as restarting new NameNodes. Note that we have a leader election in this distributed NameNode architecture. The leader, *master*, will be responsible for tasks like block reporting and statistic functions.

As we discuss earlier, the size of the metadata for a single file object having two blocks (replicated three times by default) is 600 bytes. It requires 60 GB of RAM to store 100 million files in HDFS, 100 million files is also the maximum storage capacity for HDFS in practice. For MySQL Cluster, it supports up to 48 datanodes (MySQL b), which means that it can scale up to 12 TB in size with 256 GB RAM for each node in size. But conservatively, we assume that Cluster can support up to 3.072 TB for metadata with a replication of 2, which means that Hop-HDFS can store up to 4.1 billion files. A factor of 40 times increase (Hakimzadeh et al. 2014).

II

Namespace Concurrency Control and Assessment

3 Namespace Concurrency Control

3.1 Namespace Concurrency Control in GFS

3.1.1 Namespace Structure

Unlike traditional file systems, GFS doesn't have a per-directory data structure, which means that it doesn't support listing all files in a directory (i.e, *ls* in POSIX), nor aliasing for the same file or directory (i.e, hard or symbolic links). Instead, with prefix compression, GFS represents the namespace as a lookup table mapping full pathnames to metadata logically, which means that the full pathnames are similar to the hash keys in a hash table.

3.1.2 Namespace Concurrency Control

Each node (either an absolute directory name or an absolute file name) in the namespace tree will be associated a *read-write* lock. To prevent deadlock, locks are acquired in a *consistent total order*: first ordered by level, then ordered lexicographically within the same level (Ghemawat et al. 2003).

One benefit for the locking scheme in GFS is that it allows concurrent mutations for different files/directories within the same directory.

For example, suppose that we have a graphical tree representation for the namespace in GFS as shown in Figure 3.1. Concurrently, we have five operations involving files *f1*, *f2*, *f3*, *f4* and directory *d9*. As we can see from Table 3.1, there are no conflicting locks (*Read-Write and Write-Write*), all these five operations are all allowed to happen concurrently.

Since operations will be serialized properly when trying to obtain conflict locks(*Read-Write and Write-Write*), concurrent mutations on the same file/directory will be prevented.

For example, if there are another two concurrent operations. *Operation 1* wants to snapshot directory *d8* to be under directory *d3*, but *Operation 2* wants to create a new file *Qi.txt* under

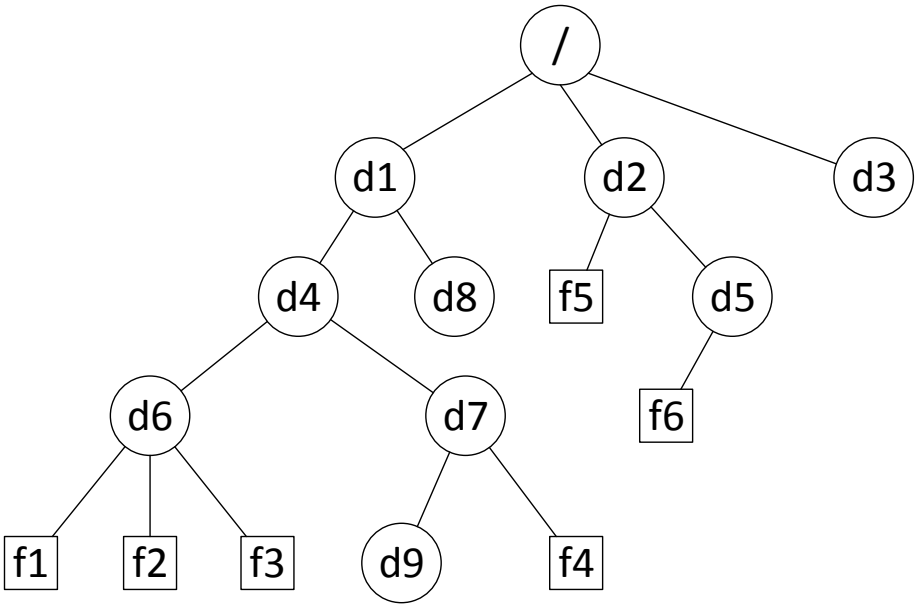


Figure 3.1: A Graphical Tree Representation for the Namespace in GFS

directory *d8*. Table 3.2 shows how conflict locks prevent the new file *Qi.txt* being created when directory *d8* is being snapshotting.

In sum, GFS trades off common file system requirements for this namespace locking scheme with nice concurrency control properties.

Total Order Locks	**Operation1**	**Operation2**	**Operation3**	**Operation4**	**Operation5**
/	Read1	Read2	Read3	Read4	Read5
/**d1**	Read1	Read2	Read3	Read4	Read5
/d1/**d4**	Read1	Read2	Read3	Read4	Read5
/d1/d4/**d6**	Read1	Read2	Read3		
/d1/d4/**d7**				Read4	Read5
/d1/d4/d6/**f1**	Write1				
/d1/d4/d6/**f2**		Write2			
/d1/d4/d6/**f3**			Write3		
/d1/d4/d7/**d9**				Write4	
/d1/d4/d7/**f4**					Write5

Table 3.1: Concurrent Mutations within for different files/directories and Related Read-Write Lock Sets

Total Order Locks	**Operation1**	**Operation2**
/	Read1	Read2
/**d1**	Read1	Read2
/**d3**	Read1	
/d1/**d8**	Write1	Read2 **(Conflicts with Write1)**
/d3/**d8**	Write1	
/d1/d8/**Qi.txt**		Write2

Table 3.2: Serialized Concurrent Mutations and Conflict Locks

3.2 Namespace Concurrency Control in HDFS

3.2.1 Namespace Structure

Unlike GFS, the interface to HDFS is patterned after UNIX, and it support POSIX like commands (e.g, *ls, mkdir, rm, cp, chown*) to the common file system. The namespace of HDFS is structured as a hierarchy of files and directories. Files and directories are represented on the NameNode by *INodes* with attributes like permissions, modification and access times, namespace and disk space quotas (Borthakur 2008). Each file is represented by an *INodeFile* object, each directory is represented by an *INodeDirectory*, and each symbolic link is represented by an *INodeSymlink object*. Figure 3.2 shows the Namespace INode Structure in UML diagram with major attributes.

3.2.2 Namespace Concurrency Control

The hierarchical INode structure makes HDFS not possible to adopt the namespace locking scheme from GFS. In order to support POSIX like operations (list files, set quotas, create symbolic links), INodeFiles, INodeDirectories and INodeSymlink objects are semantically related to each other, rather than just logical representation.

For example, suppose that HDFS adopts the namespace locking scheme in GFS. An INodeDirectory *D3* with quota *1* which only allows 1 more INode to be created inside it. Concurrently, there are four operations try to create an INodeFile inside *D3*. All of them put a read lock on *D3* first. Finding that the quota is 1, they then put a write lock on the file and create it under the directory. Finally four files are created under *D3* but it violates the quota. See Figure 3.3.

One way to solve this consistency problem is to synchronize all the related attributes among different threads under proper semantic group. However, it complicates the namespace design

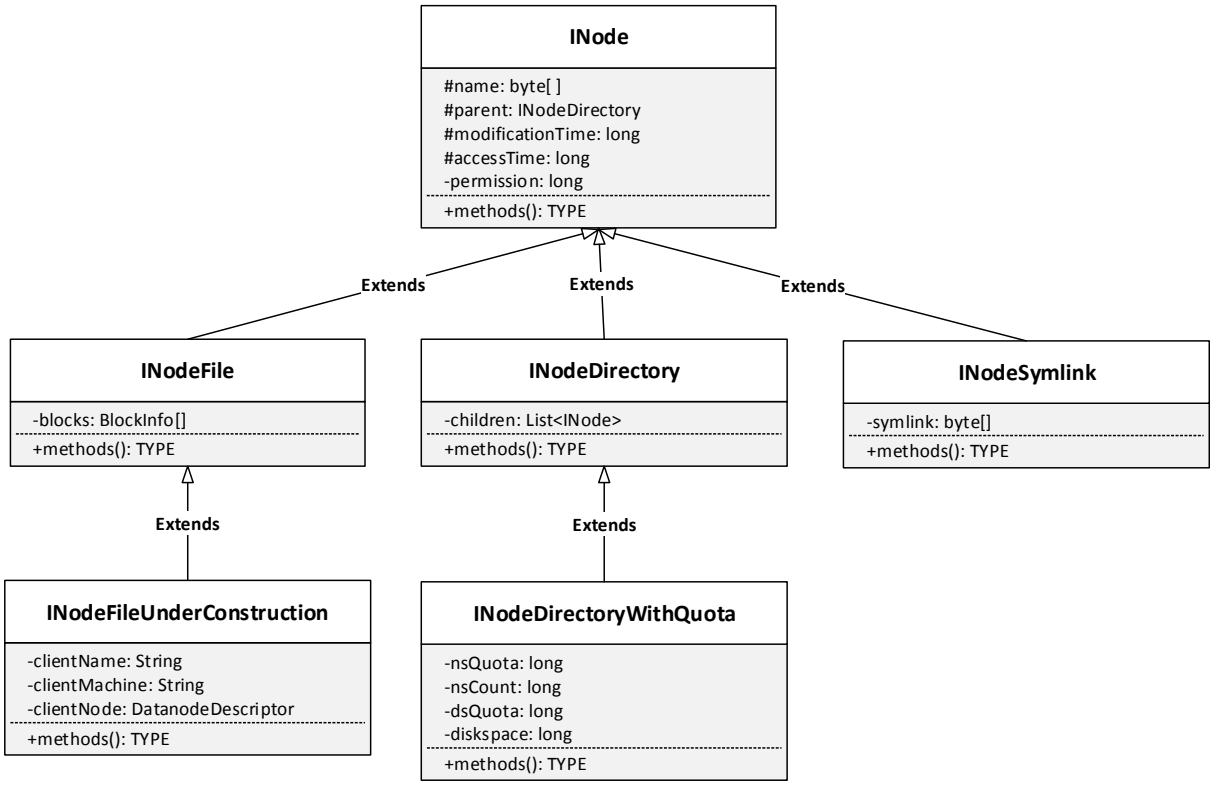


Figure 3.2: The Namespace INode Structure in HDFS

and is not realistic. Therefore, to protect the namespace among parallel running threads, a global read/write lock (fsLock in *FSNamesystem - ReentrantReadWriteLock* in java language) is used to maintain the atomicity of the namespace. We call it *system-level lock*.

HDFS categorizes the metadata operations into *read operations* and *write operations*. Concurrent threads to access the namespace for read operations are allowed, but it restricts a single thread to namespace for write operations. Therefore, all concurrent readers get the same view of the mutated data reflected by completed writes. We call it *Strong Consistency Semantics* in HDFS. (But it is still weaker than the standard POSIX consistency model since it trades some POSIX requirements for performance in terms of data coherency (White 2012))

3.2.3 Bottleneck

Although the namespace is kept in-memory for fast operations, the system-level lock is still the bottleneck in NameNode under high workload pressure. Here we analyze the Remote Procedure Call (RPC) for namespace operations between clients and NameNode. See Figure 3.4

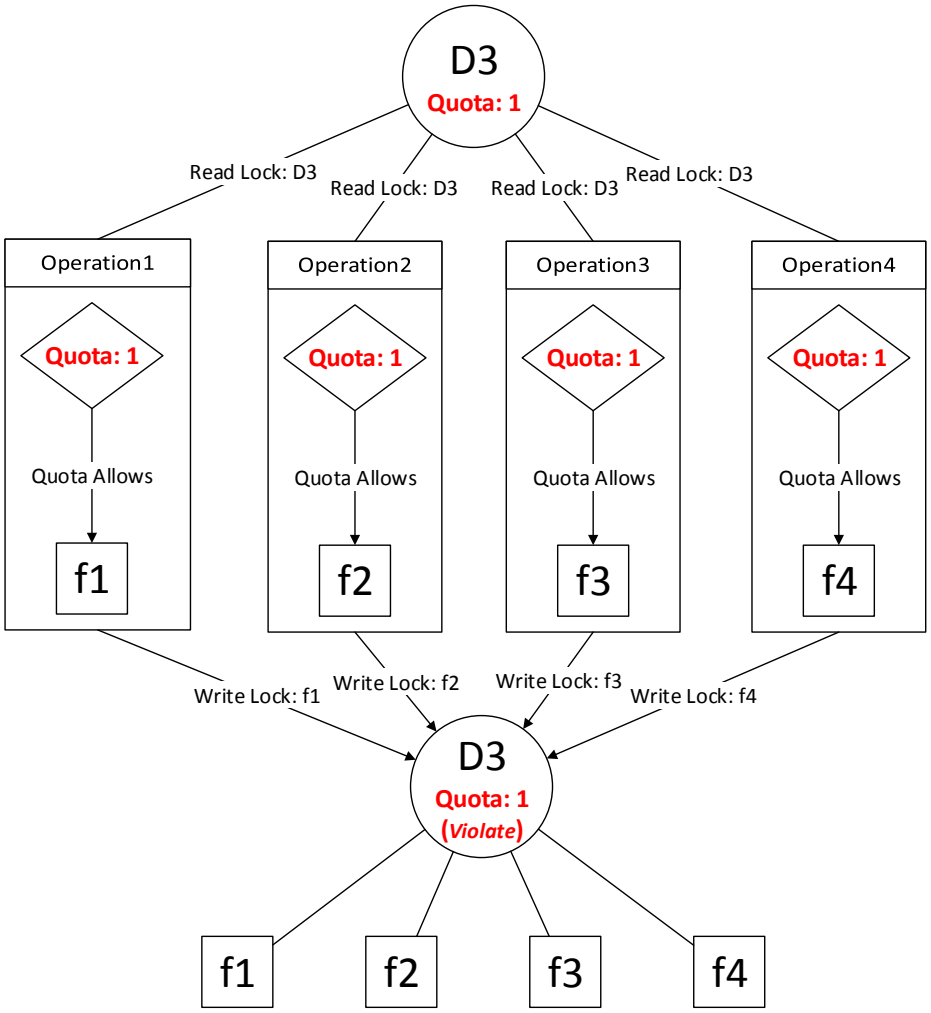


Figure 3.3: Violation in Quota Semantic

for the process:

1. Client makes an RPC request to NameNode RPC server, like *mkdir*.
2. The listener thread in NameNode RPC server accepts this request.
3. The Reader, child thread of Listener, processes the request and makes it as a Call object stored in the Call Queue, waiting for the handling.
4. One of the handlers gets a Call object (*mkdir*) from the queue. As *mkdir* belongs to write operation, the handler takes a write lock on the namespace.
5. After taking the write lock, a new directory will be created in the namespace within NameNode.
6. The modification record needs to be synchronized to the editlogs.
7. Release the write lock.

8. The callback is returned to the Responder thread.
9. The client get the result for this operation (either success or fail).

As we can see, any of the steps above may become the bottle. But in step 6, while the entire namespace is protected by the system-level lock, the modification record needs to be saved into the editlogs. Since the editlogs are written into the physical hard drives sequentially, the more syn edit to be handled, the slower it will be for the responder to return the callback. The system-level lock won't be released during this process, so the throughput will be decreased greatly during heavy workload.

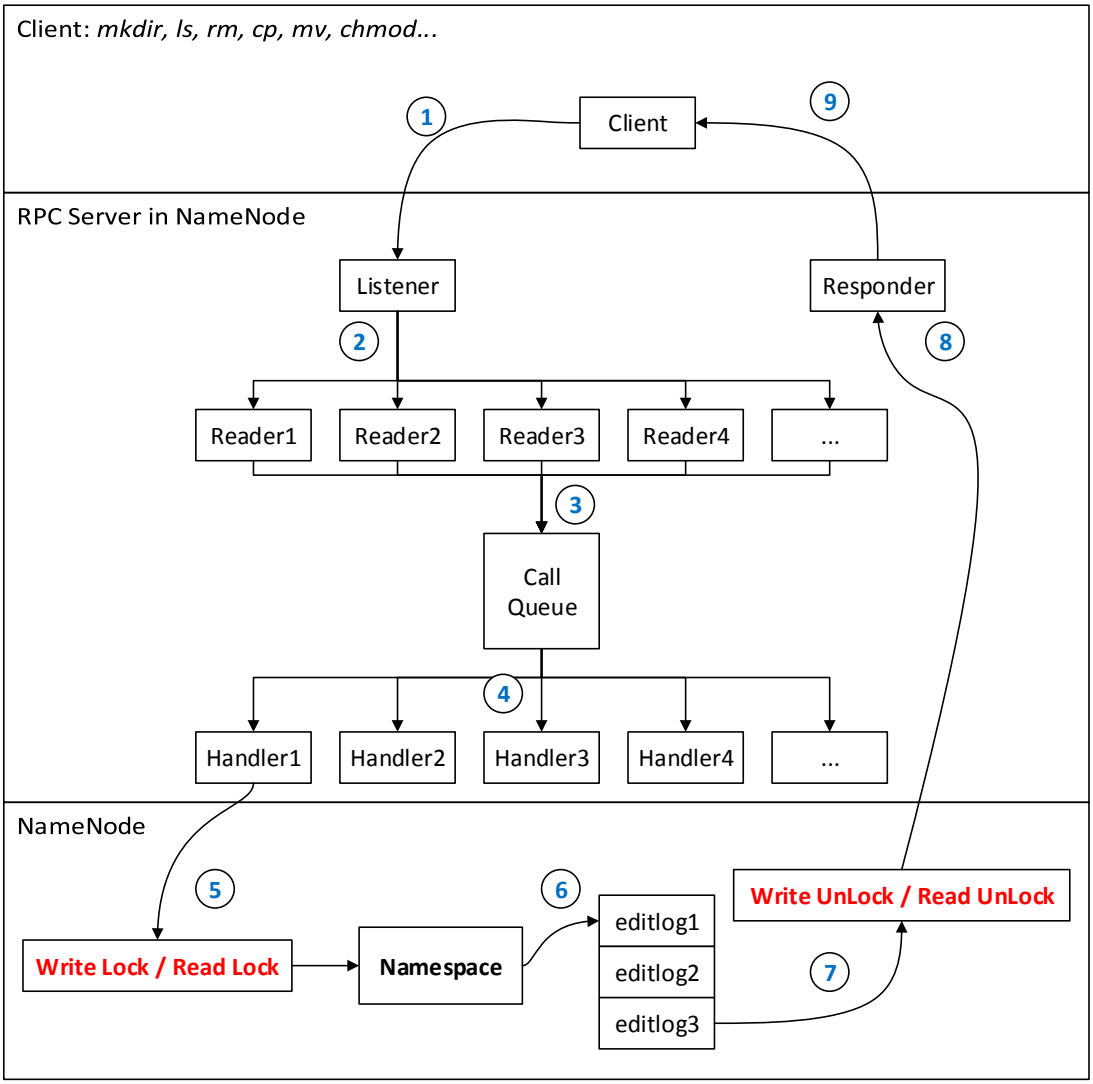


Figure 3.4: RPC between Clients and NameNode for Namespace Operations

3.3 Namespace Concurrency Control in Hop-HDFS

3.3.1 Namespace Structure

In HDFS, the namespace is kept in-memory as arrays and optimized data structure (like LinkedList) of objects with references for semantic constraints. Therefore, it has a *directed tree structure*, similar to Figure 3.1.

In Hop-HDFS, the namespace is stored into tables of MySQL Cluster database, so all INode objects are represented as individual row records in a single *inodes table*. In order to preserve the directed tree structure, we add an id column and a parent_id column to each row of in *inodes table*. Therefore, the graphical representation of the filesystem hierarchy for INodes is like Figure 3.5. The table representation in the database is like Table 3.3.

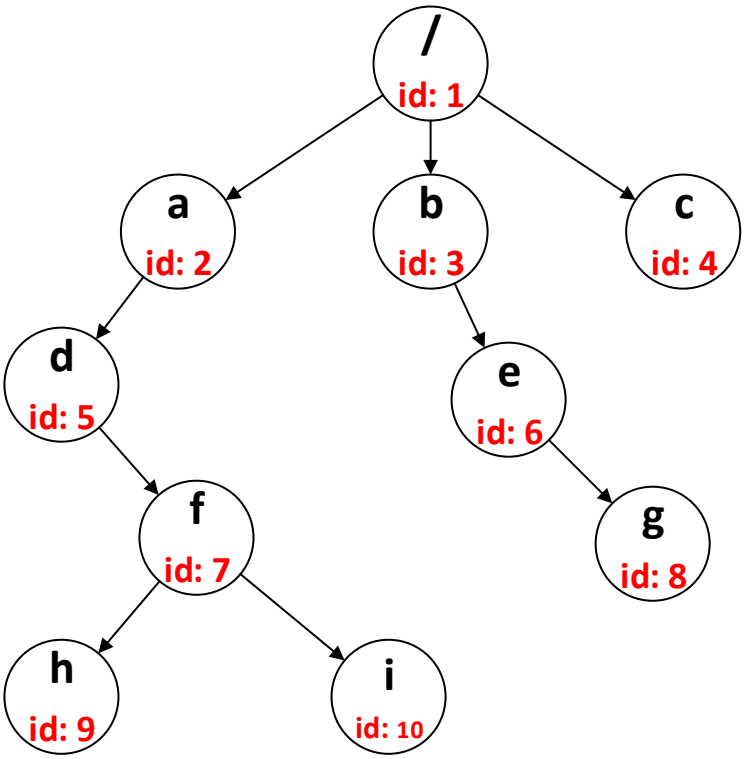


Figure 3.5: Filesystem Hierarchy with ID for INodes in Hop-HDFS

Since the *id* is unique and atomically generated for INodes in each new transaction, the *Primary Key* for the table is <name, parent_id> pair to avoid duplicated data rows during primary key lookup. Besides, the INode *id* is not known beforehand on the application side, but the <name, parent_id> pair is known since it is stored as a path string. So data rows can be looked up by

id	parent_id	name	other parameters...
1	0	/	...
2	1	a	...
3	1	b	...
4	1	c	...
5	2	d	...
6	3	e	...
7	5	f	...
8	6	g	...
9	7	h	...
10	7	i	...

Table 3.3: INode Table for Hop-HDFS

the <name, parent_id> pair *Primary Key* directly from database.

With the *id* and *parent_id* relationship, the hierarchy will be constructed correctly from the rows to be in-memory objects used by the name system.

3.3.2 Namespace Concurrency Control

4 Namespace Operation Performance Assessment

4.1 A

AAA

4.2 B

BBB

4.3 C

CCC

III Algorithmic Solution

5 Optimistic Concurrency Control with Snapshot Isolation on Semantic Related Group

The solution we propose to improve the throughput is based on the following four phases:

1. **Read Phase**: resolving the semantic related group and cache the snapshot copy within the handling transaction.
2. **Execution Phase**: transaction read/write operations are performed on its own snapshot and never fetch data from database.
3. **Validation Phase**: snapshot's related data rows are fetched from the database. If all their versions match with the original copy of the snapshot, go to update phase; else, abort and retry current transaction.
4. **Update Phase**: update related data in the database table. Abort and retry transaction if the instance already exists in the database for "new" data. Increase the versions of the modified rows by 1 if updated successfully.

The phases mentioned above will be illustrated detailedly in the following sections. The complete algorithm pseudocode can be found in Algorithm 1.

5.1 Resolving the Semantic Related Group

Resolving the semantic related group for each transaction is the fundamental step to preclude *anomalies* in our implementation. The *constraint violation* (Berenson et al. 1995) between individual data is formed within a semantic related group. In Hop-HDFS, each metadata operation is implemented as an individual transaction running by a worker thread. Any metadata operation related to the namespace will have one or two input parameters, called *Path*. Here's two examples for methods in the Filesystem API:

- boolean **mkdirs** (Path f): f is the path of the INodeDirectory to be created

- boolean **rename** (Path *src*, Path *dst*): *src* is the path to be renamed, *dst* is the new path after rename

Each *Path* object is related to a string representation of the "/" based absolute path name. For example, in Figure 3.5, the path for INode *h* is:

/a/d/f/h

Therefore, with the preservation of the *directed tree structure*, we can resolving a semantic related group for each INode along the edge of ancestors as a *LinkedList*. The semantic related group representation for INode *h* is:

h: {/->a->d->f}

In other words, when mutating INode *h*, all the semantic constraint can be found within INodes */*, *a*, *d*, *f*. With this knowledge, we can maintain the strong consistency semantics of original HDFS.

For each row in *inodes table*, the <name, parent_id> pair is the *Primary Key*. With the full path string, we can resolve its semantic related rows by primary key lookups directly from database as shown in Table 5.1.

	id	**parent_id**	**name**	**other parameters...**
Related *	1	0	/	...
Related *	2	1	a	...
	3	1	b	...
	4	1	c	...
Related *	5	2	d	...
	6	3	e	...
Related *	7	5	f	...
	8	6	g	...
Selected ✓	9	7	h	...
	10	7	i	...

Table 5.1: Table Representation for the Semantic Related Group

5.2 Per-Transaction Snapshot Isolation

As we mentioned before, MySQL Cluster supports only the READ COMMITTED transaction isolation level, which means that the committed results of write operations in transactions will

be exposed by reads in other transactions. Within a long running transaction, it could read two different versions of data, known as *fuzzy read*, and it could also get two different sets of results if the same query is issued twice, known as *phantom read*.

Snapshot isolation guarantees that all reads made within a transaction see a consistent view of at the database. At the beginning of the transaction, it reads data from a snapshot of the latest committed value. During transaction execution, reads and writes are performed on the this snapshot.

In commercial database management systems, like Microsoft SQL Server, Oracle, etc, *snapshot isolation* is implemented within multi version concurrency control (MVCC) (Berenson et al. 1995) on database server side. However, we need to implement snapshot isolation on the application side since MySQL Cluster supports only the READ COMMITTED isolation level.

After resolving the semantic related group, we take a snapshot on selected rows as well as all related rows of the committed values from database. This snapshot will be cached in-memory within its transaction. Each transaction will have its own copy of snapshot during the lifetime. All transaction operations will be performed on its own snapshot. Therefore, we called it Per-Transaction Snapshot Isolation.

Before validation phase, the transaction will never fetch any data from database since it has all the semantic related rows in the cache. Therefore, the snapshot provides a consistent view of data for each transaction from read phase until validation phase:

- *Fuzzy Read* is precluded by *snapshot isolation*: As we can see from Figure 5.1, the second read of Transaction 1 read from snapshot instead of database, not affected by the value committed by Transaction 2.
- *Phantom Read* is also precluded by *snapshot isolation with Semantic Related Group*: As we can see from Figure 5.2, Transaction 1 snapshot the semantic related group of x after the first count operation. So its second count operation is not affected by the value inserted by Transaction 2.

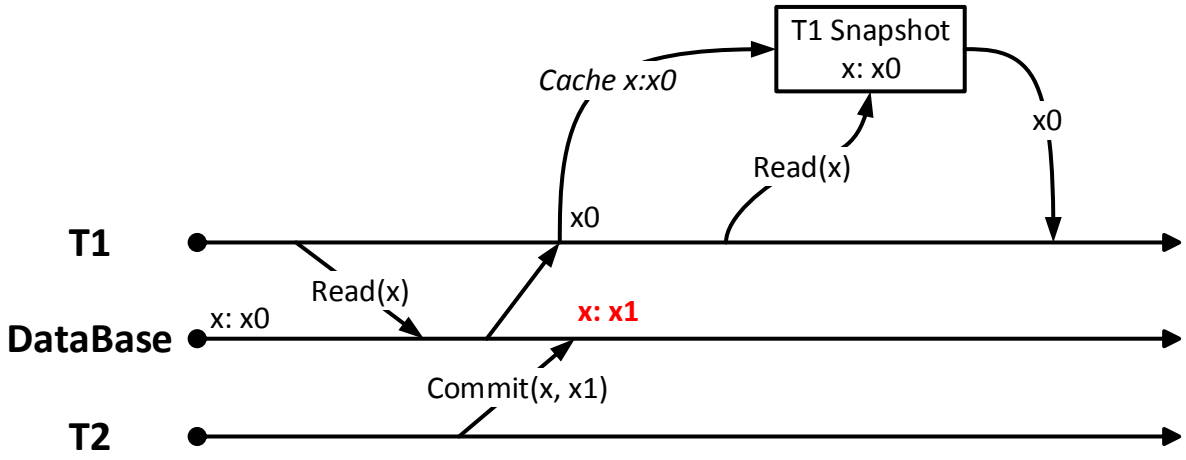


Figure 5.1: Snapshot Isolation Precludes Fuzzy Read

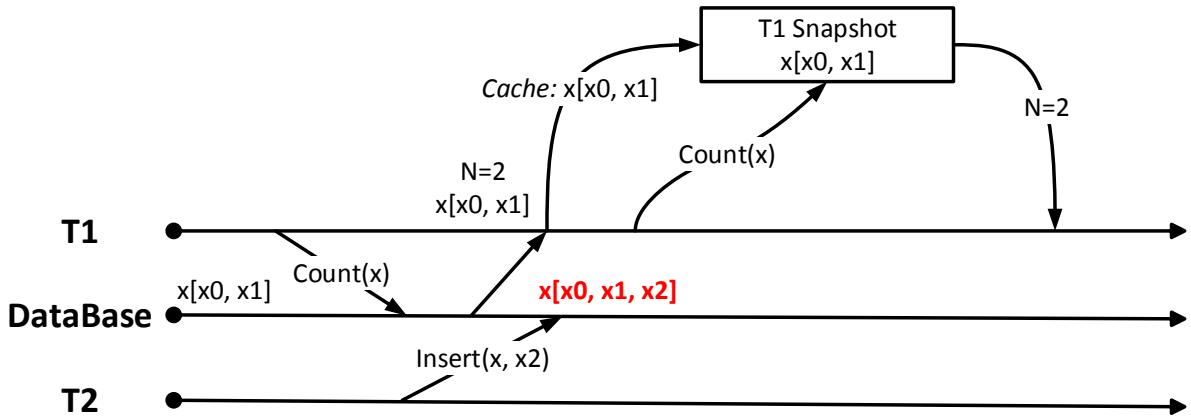


Figure 5.2: Snapshot Isolation with Semantic Related Group Precludes Phantom Read

5.3 ClusterJ and Lock Mode in MySQL Cluster

ClusterJ (MySQL c) is a Java connector based on object-relational mapping persistence frameworks to access data in MySQL Cluster. Since it uses a JNI bridge to the NDB API for direct access to NDB Cluster, it doesn't depend on the MySQL Server to access data in MySQL Cluster, which means that ClusterJ can perform some operations much more quickly since it communicates to the data nodes directly.

Therefore, with the mapping from java classes to database tables, we use ClusterJ to fetch and persist data in MySQL Cluster using primary key and unique key operations for single-table queries (not supporting multi-table operations though). If we recall the architecture of MySQL

Cluster from Figure 2.3, ClusterJ will be the Java Persistence API between Hop-HDFS and the *Data Nodes* (ndbd) , without going through *MySQL Servers* (mysqld).

Unlike *two-phase locking* (2PL) (Franklin 1997), there are three lock modes in MySQL Cluster:

1. **SHARED** (Read Lock, RL): Set a shared lock on rows
2. **EXCLUSIVE** (Write Lock, WL): Set an exclusive lock on rows
3. **READ_COMMITTED** (Read Committed, RC): Set no locks but read the most recent committed values

Shared and *Exclusive* locks have the same definition of those in Two-phase Locking. For *Read_Committed*, it is implemented for consistent nonlocking reads, which means that a fresh committed snapshot of data row is always presented to a query of database, regardless of whether Shared Lock or Exclusive Lock are taken on the current row or not. It is based on *Multiversion Concurrency Control* described by Oracle (Oracle a) for read consistency from a single point in time (*statement-level read consistency*). See Table 5.2 for the reference of the blocking effect.

We use *Read_Committed* for the *read phase* in our algorithm.

Lock Type	SHARED	EXCLUSIVE	READ_COMMITTED
SHARED	✓	Block	✓
EXCLUSIVE	Block	Block	✓
READ_COMMITTED	✓	✓	✓

Table 5.2: Locks Blocking Table in MySQL Cluster

5.4 Optimistic Concurrency Control

Our algorithm is based on *Optimistic Concurrency Control* (OCC) model to improve the overall read/write performance. Transactions are allowed to perform operations without blocking each other with optimistic methods. Concurrent transactions need to pass through a *validation phase* to before committing, so that the serializability is not violated. Transactions will abort and restart if they fail in the *validation phase*. OCC is the key approach to help remove the parent directory lock in Hop-HDFS so that transactions can operate under the same directory concurrently.

In *read phase*, transactions use *Read_Committed Lock* to fetch semantic related group as snapshots and cache them in-memory for their own use.

In *validation phase*, transactions will fetch the modified rows using *Exclusive Lock* and fetch the semantic related rows using *Shared Lock*. Then they compare the fetched values and the original copy of snapshot in the cache for their *versions*. If they are all the same, go to *update phase*. If not, abort current transaction, wait for a random milliseconds, and retry a new transaction from *read phase*.

Note that using *Shared Lock* to fetch semantic related rows can guarantee a consistent view in database until the transaction goes to the update phase, and also allows other Shared Locks taken on the same rows.

In order to avoid multiple database round trips, we will do the fetching in batch processing provided by *ClusterJ*.

Write skew anomaly is precluded by the validation phase on the snapshot of semantic related group in OCC, because constraint violation on all related data rows will be checked before transaction committed. See Figure 5.3 to see how optimistic concurrency control with snapshot isolation on semantic related group precludes *Write Skew*.

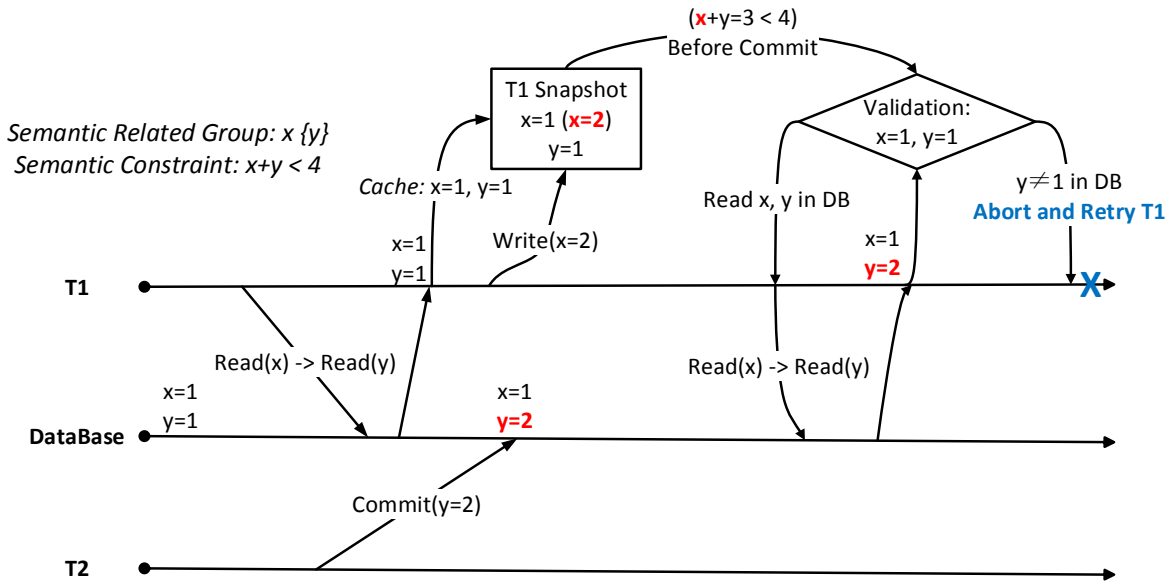


Figure 5.3: Optimistic Concurrency Control with Snapshot Isolation on Semantic Related Group Precludes Write Skew

Therefore, we use optimistic concurrency control with snapshot isolation on semantic related group to improve the throughput while the strong consistency semantics in original HDFS is maintained.

5.5 Total Order Update, Abort and Version Increase in Update Phase

We have a total order update rule in update phase so that dead lock will not occur due to lock cycle. If multiple rows are updated in a transaction during update phase, they will be sorted first by the *id value*, then they will be updated in ascending order according to their *ids*.

Since we can not take an Exclusive lock on the "new" row which not yet exists in the database beforehand, transactions may try to persist "new" with the same *Primary Key* and one will be overwritten by the other. As we have defined the <name, parent_id> pair to be the *Primary Key*, exception will be thrown when "duplicated" data is going to be persisted into the table by invoking *makePersistent()* function in ClusterJ rather than *savePersistent()* function.

The versions of the modified rows will be increased by 1 if they are successfully updated.

5.6 Pseudocode of the Complete Algorithm

Algorithm 1 Pseudocode of the Complete Algorithm - Optimistic Concurrency Control with Snapshot Isolation on Semantic Related Group

```
init: snapshot.clear, restart = true, try = 0, path = operation.src
while restart and try < 5 do
  restart = false
  try += 1

  tx.begin()
  /* 1. Read Phase */
  tx.lockMode(Read_Committed)
  tx.snapshot = resolve_semantic_related_group(path)
  tx.snapshot_copy = tx.snapshot

  /* 2. Execution Phase */
  operation_performTask(tx.snapshot) // HDFS Operation body performs only on snapshot

  /* 3. Validation Phase */
  tx.lockMode(Shared)
  relatedRows_DataBase = batchRead_Database(tx.snapshot_copy)
  tx.lockMode(Exclusive)
  modifiedRows_DataBase = batchRead_Database(tx.snapshot_copy)
  if versionCompare(relatedRows_DataBase, modifiedRows_DataBase, tx.snapshot_copy)
  == true then
    operation.modifiedRows.version+=1

    /* 4. Update Phase */
    total_order_sort(operation.modifiedRows)
    if batchPersist_Database(operation.modifiedRows) success then
      tx.commit()
      return SUCCESS // HDFS Operation Success and Return
    else
      tx.abort()
      retry = true
    end if
  else
    tx.abort()
    retry = true
  end if
end while
return FAIL // HDFS Operation Fails and Return
```

IV Evaluation and Conclusion

6 Evaluation

6.1 Experimental Setup

AAA

6.2 Performance Comparison

BBB

7 Conclusion and Future Work

7.1 Conclusion

AAA

7.2 Future Work

BBB

Bibliography

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil, & P. O'Neil (1995). A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, Volume 24, pp. 1–10. ACM.

Borthakur, D. (2008). Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*.

Cloudera. Hadoop and big data. http://www.cloudera.com/content/cloudera/en/about/hadoop-and-big-data.html.

Dowling, J. (2013). Hop: Hadoop open platform-as-a-service.

D'Souza, J. C. (2013). Kthfs–a highly available andscalable file system.

Franklin, M. J. (1997). Concurrency control and recovery.

Ghemawat, S., H. Gobioff, & S.-T. Leung (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, Volume 37, pp. 29–43. ACM.

Gray, J. N., R. A. Lorie, G. R. Putzolu, & I. L. Traiger (1976). Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pp. 365–394.

Hadoop, A. What is apache hadoop? http://hadoop.apache.org.

Hakimzadeh, K., H. P. Sajjad, & J. Dowling (2014). Scaling hdfs with a strongly consistent relational model for metadata. In *Distributed Applications and Interoperable Systems*, pp. 38–51. Springer.

HBase, A. Welcome to apache hbase. http://hbase.apache.org/.

Mahout, A. What is apache mahout? http://mahout.apache.org/.

MySQL. Chapter 18 mysql cluster ndb 7.3. http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster.html.

MySQL. Defining mysql cluster data nodes. http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-ndbd-definition.html.

MySQL. Java and mysql cluster. http://dev.mysql.com/doc/ndbapi/en/mccj-overview-java.html.

MySQL. Limits relating to transaction handling in mysql cluster. http://dev.mysql.com/doc/mysql-cluster-excerpt/5.1/en/mysql-cluster-limitations-transactions.html.

MySQL. Limits relating to transaction handling in mysql cluster. http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-limitations-transactions.html.

MySQL. Mysql cluster nodes, node groups, replicas, and partitions. http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-nodes-groups.html.

MySQL (2012, July). Mysql cluster benchmarks: Oracle and intel achieve 1 billion writes per minute.

Oracle. Data concurrency and consistency. http://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm.

Oracle. Java api documentation: Class reentrantreadwritelock. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html.

Peiro Sajjad, H. & M. Hakimzadeh Harirbaf (2013). Maintaining strong consistency semantics in a horizontally scalable and highly available implementation of hdfs.

Pig, A. Welcome to apache pig. http://pig.apache.org/.

Shvachko, K., H. Kuang, S. Radia, & R. Chansler (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10. IEEE.

Shvachko, K. V. (2010). Hdfs scalability: The limits to growth. *login 35*(2), 6–16.

Shvachko, K. V. (2011). Apache hadoop: The scalability update. *login: The Magazine of USENIX 36*, 7–13.

Spark, A. Apache spark welcome page. http://spark.apache.org/.

Wasif, M. (2012). A distributed namespace for a distributed file system.

White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

V Appendices

A Apache HDFS Unit Tests Passing List