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

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 9, 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* (Apache) 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) adopts a similar idea from *two-phase locking* (Berenson et al. 1995), it has far more 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), 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

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.

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.

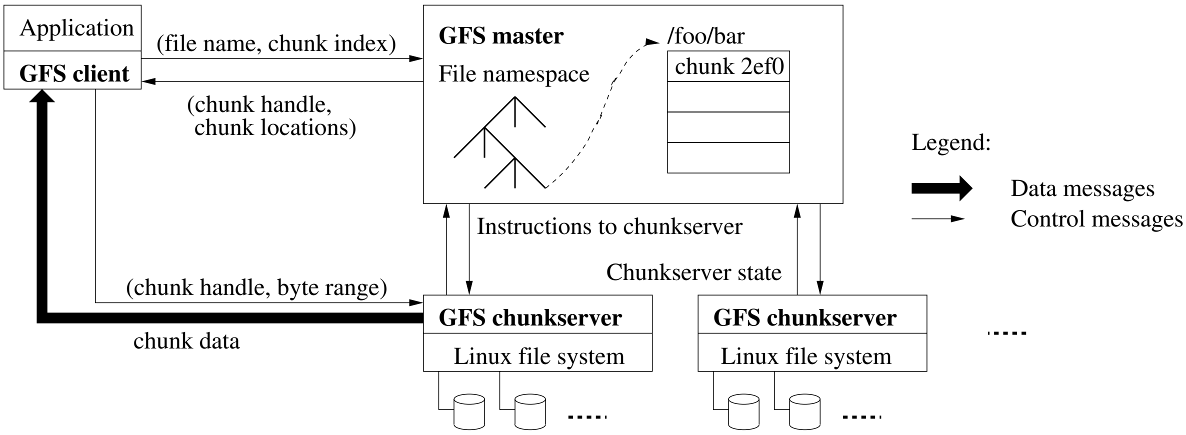

Figure 2.1: The Architecture of GFS

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.

2.2 Concurrency Control in Transactional Systems

BBB

2.3 Isolation Level in Transactional Systems

CCC

2.4 MySQL Cluster

DDD

II
Assessment in Hop-HDFS

3 Namespace Locking

3.0.1 The Namespace Locking in GFS

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. Therefore, each node 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.

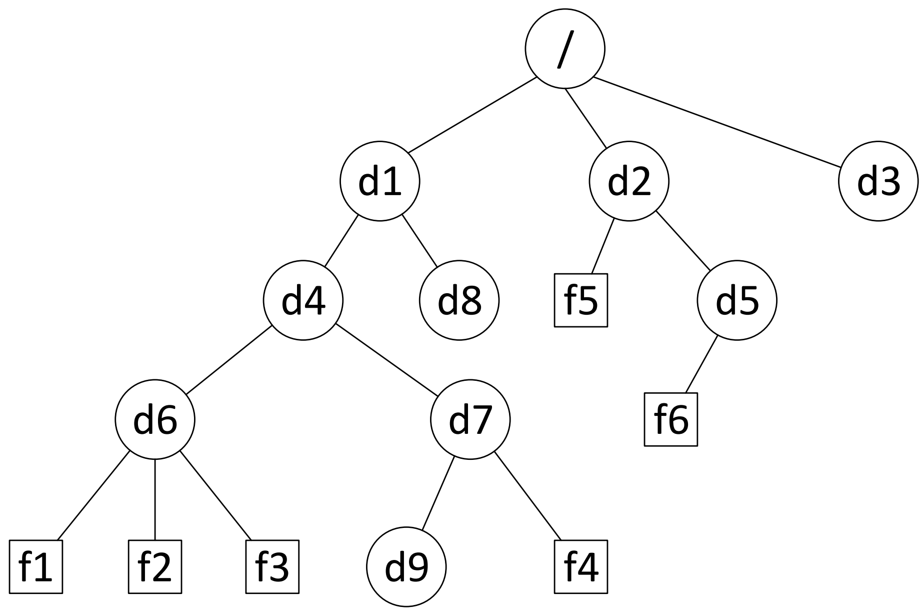

Figure 3.1: A Graphical Tree Representation for the Namespace in GFS

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

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	Write5
/d1/d4/d7/**f4**					

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

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.

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

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

3.1 A

3.2 B

3.3 C

CCC

3.4 D

DDD

4 Systematic Assessment of Hop-HDFS Performance

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

– Cerico

4.1 A

AAA

4.2 B

BBB

4.2.1 B1

BBB1

4.2.2 B2

BBB2

4.3 C

CCC

4.4 D

DDD

III Solution

5 Design

5.1 A

AAA

5.2 B

BBB

5.2.1 B1

BBB1

5.2.2 B2

BBB2

5.3 C

CCC

5.4 D

DDD

6 Implementation

6.1 A

AAA

6.2 B

BBB

6.2.1 B1

BBB1

6.2.2 B2

BBB2

6.3 C

CCC

6.4 D

DDD

IV Evaluation and Conclusion

7 Evaluation

7.1 A

AAA

7.2 B

BBB

7.2.1 B1

BBB1

7.2.2 B2

BBB2

7.3 C

CCC

7.4 D

DDD

8 Conclusion

8.1 A

AAA

8.2 B

BBB

8.2.1 B1

BBB1

8.2.2 B2

BBB2

8.3 C

CCC

8.4 D

DDD

Bibliography

Apache. What is apache hadoop? http://hadoop.apache.org.

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.

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.

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.

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.

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.

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.

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

V Appendices

A Apache HDFS Unit Tests Passing List