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 the 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 there 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.

15 September 2014, Stockholm

Qi Qi

European Master in Distributed Computing (EMDC)

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

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

- First Year (Portugal): Instituto Superior Técnico, Universidade de Lisboa
- Third Semester (Sweden): KTH Royal Institute of Technology
- Fourth Semester (Sweden): Computer Systems Laboratory at Swedish Institute of Computer Science (SICS) / Instituto Superior Técnico, Universidade de Lisboa

Dedication

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

Resumo

O Hadoop Distributed File System (HDFS) é a camada de armazenamento para o ecossistema Apache Hadoop, armazenando grandes conjuntos de dados em várias máquinas. No entanto, a capacidade de armazenamento total é limitada uma vez que os metadados são armazenados na memória de um único servidor, chamado NameNode. O tamanho de heap do NameNode restringe o volume de dados, ficheiros e blocos endereçáveis no sistema.

A plataforma Hadoop Open (Hop) é uma plataforma-como-serviço (PaaS) para o ecossistema Hadoop em plataformasde nuvem existentes, incluindo Amazon Web Service e OpenStack. A camada de armazenamento Hop, Hop-HDFS, é uma implementação de alta disponibilidade do HDFS, armazenando os metadados numa base de dados replicada em memória distribuída, MySQL Cluster. O objetivo é superar as limitações do NameNode, mantendo a semântica de consistência forte do HDFS para que as aplicações escritas para HDFS podem ser executados em Hop-HDFS sem modificações.

Trabalhos anteriores têm contribuído para a adopção de um modelo transaccional para o Hop-HDFS. De granularidade lata de nível sistema até a mais fina, com trincos sobre registos, as semânticas de consistência forte foram mantidas no Hop-HDFS, mas com desempenho muito restrito comparado com o HDFS orginal.

Nesta tese, analisamos primeiro as limitações na implementação actual do HDFS e fornecemos uma visão geral do Hop-HDFS ilustrando como superámos essas limitações. Em seguida, fazemos uma avaliação sistemática dos trabalhos anteriores para o Hop-HDFS comparando com o HDFS, e também analisamos as restrições ao utilizar mecanismos de sincronização pessimista para garantir consistência forte. Finalmente, a partir da investigação de deficiências atuais, demonstramos como melhorar o desempenho através da concepção de um novo modelo baseado no controle de concorrência optimista com snapshot isolation como prova de conceito. A avaliação mostra melhoria significativa do desempenho com novo modelo. A nossa implementação foi validada por mais de 300 testes de unidade ao Apache HDFS.

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 provide a solution for Hop-HDFS based on optimistic concurrency control with snapshot isolation on semantic related group to improve the operation throughput while maintaining the strong consistency semantics in HDFS. 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

HDFS

Cluster MySQL

Controlo de Concorrência

Snapshot Isolation

Débito

Keywords

HDFS

MySQL Cluster

Concurrency Control

Snapshot Isolation

Throughput

Index

Lista de Figuras

Lista de Tabelas

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 shortcoming 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 a single machine.

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

Tabela 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. Hop-HDFS can store up to 4.1 billion files with 3TB MySQL Cluster support for metadata (Hakimzadeh et al. 2014), a factor of 40 increase over Shvachko's estimate (Shvachko 2010) for HDFS from 2010.

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 methods to improve the performance of Hop-HDFS is the main motivation of this thesis.

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.

The NameNode in HDFS allows concurrent threads to access shared object for read operations, 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. The concurrency limitation of this *single-writer-multiple-readers* model is compensated by fast in-memory metadata operations.

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, 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 aiming 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 lock on 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 implementing each HDFS operation as a single transaction. Because 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 reads in different concurrent transactions. Without proper implementation, anomalies like *Fuzzy Read, Phantom, and Write Skew* (Berenson et al. 1995) will produce incorrect

results.

1.3 Contribution

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

- First, we discuss the architectures of related distributed file systems, including Google File System, HDFS and Hop-HDFS. With focus on their namespace concurrency control schemes, we analyzes the limitation of HDFS's NameNode implementation.
- Second, we provide an overview of Hop-HDFS illustrating how it overcomes limitations in HDFS. With a systematic performance assessment between Hop-HDFS and HDFS, we discuss the current shortcomings in Hop-HDFS, which motivates this thesis for a better concurrency control scheme.
- Third, we provide a solution for Hop-HDFS based on optimistic concurrency control with snapshot isolation on semantic related group to improve the operation throughput while maintaining the strong consistency semantics in HDFS. 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.

1.4 Document Structure

The thesis is organized as follows. Chapter 2 gives the architecture overview of Google File System (GFS), Hadoop Distributed File System (HDFS), MySQL Cluster, Hop-HDFS and knowledge on concurrency control in database management systems. In Chapter 3, we further discuss the namespace concurrency control scheme on GFS, HDFS and Hop-HDFS, and related limitations, following by a systematic performance assessment between HDFS and Hop-HDFS. In Chapter 5, we provide a solution based on *Optimistic Concurrency Control with Snapshot Isolation on Semantic Related Group* and demonstrate how we overcome the shortcomings in Hop-HDFS and improve the operation throughput, while maintaining the strong consistency semantics in HDFS. In Chapter 6, we give a detailed evaluation of our solution and shows the significant performance improvement. Finally, Chapter 7 gives the conclusion and future work of this thesis.

2 Background and Related Work

2.1 Distributed File Systems

Distributed File system is the fundamental storage layer 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.
- 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 low latency read or write.

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.

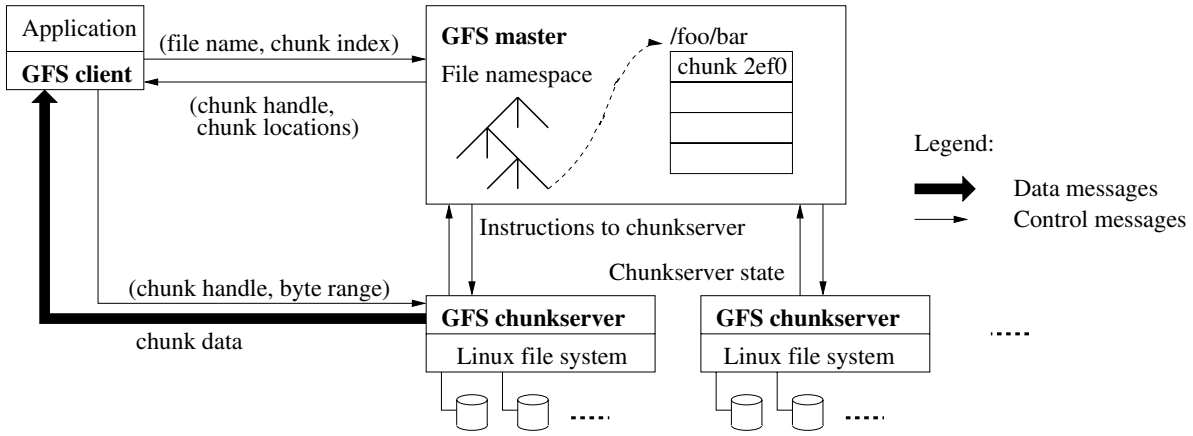


Figura 2.1: The Architecture of GFS (Ghemawat et al. 2003)

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 and chunk migrations.

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 all these big data applications. While 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.

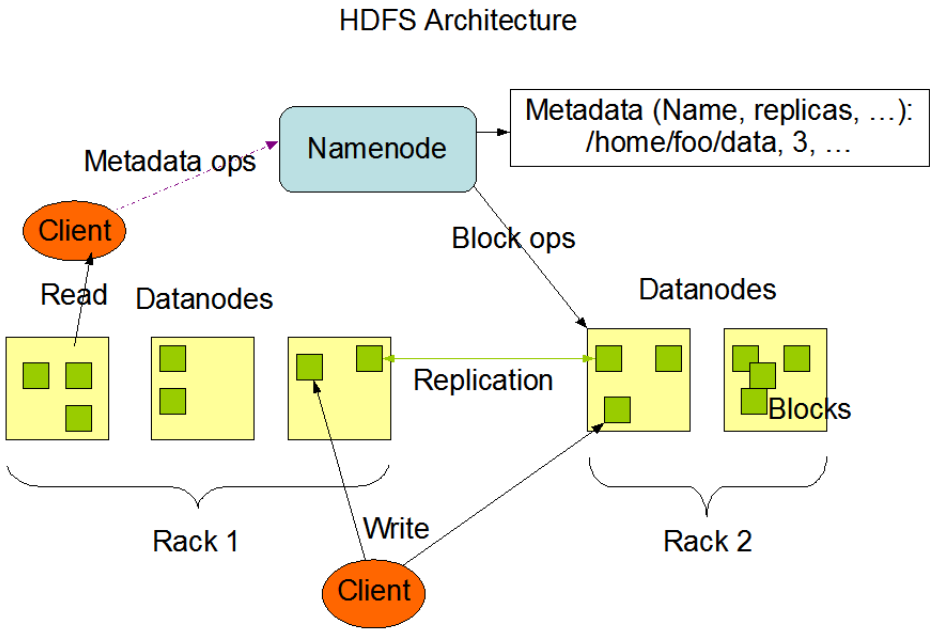


Figura 2.2: The Architecture of HDFS (Borthakur 2008)

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*. The persistent record of *image*, called the *checkpoint*, is stored in the local physical file system. The modification of the namespace (*image*), called the *journal*, is also persisted in the local physical file system. Copies of the *checkpoints* and the *journals* can be stored 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 a client application, the written bytes can not be modified. The file can only be reopened for append.

HDFS implements a *single-Writer, multiple-reader* model by using lease management. A HDFS client opens a file for writing is granted a lease of the file and no other clients 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 that 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

2.2.1 Concurrency Control in Database Management System

In a multiuser database management system, *Concurrency Control* permits concurrent users to access a database while preserving the illusion that each user is executing along on a dedicated system (Bernstein & Goodman 1981). The main idea is to ensure individual users see consistent states of the database even though users' operations may be interleaved in the shared data.

In general, there are three concurrency control methods (Franklin 1997):

1. **Pessimistic Concurrency Control** (PCC): depends on *Two-Phase Locking* (2PL) to block transactions so that interference does not occur when transactions run on to the shared data.
2. **Optimistic Concurrency Control** (OCC): depends on validation to ensure serializability. Before transactions commit, the reads and writes should not conflict with other concurrent transactions. If not, the validatation phase will abort and retry the current transaction.
3. **Multi Version Concurrency Control** (MVCC): when data items are updating by transactions, their previous versions will retain so that other read-only transactions can be provi-

ded with these older versions instead of blocking to read new data. It allows a consistent snapshot of the database to be presented.

2.2.2 Isolation Level for Concurrent Transactions

Complete isolation of concurrent running transactions might make one transaction not possible to perform an update into a database table being queried by another transaction. Therefore, real-world databases will provide different levels of transaction isolation so that data consistency will be compromised for performance.

Therefore, there are four isolation levels are defined in ANSI/ISO SQL-92 specifications (Ansi 1992): (1) *Read Uncommitted* (2) *Read Committed* (3) *Repeatable Read* (4) *Serializable*. But a new isolation level, *Snapshot Isolation* and the *anomalies* characterized by isolation types is re-defined in the paper *A Critique of ANSI SQL Isolation Levels* (Berenson et al. 1995).

Isolation Level	Dirty Write	Dirty Read	Cursor Lost Update	Lost Update	Fuzzy Read	Phantom	Read Skew	Write Skew
Read Uncommitted	X	✓	✓	✓	✓	✓	✓	✓
Read Committed	X	X	✓	✓	✓	✓	✓	✓
Cursor Stability	X	X	X	sometimes	sometimes	✓	✓	sometimes
Repeatable Read	X	X	X	X	X	✓	X	X
Snapshot	X	X	X	X	X	sometimes	X	✓
Serializable	X	X	X	X	X	X	X	X

Tabela 2.1: Isolation Types Characterized by Possible Anomalies Allowed (Berenson et al. 1995)

Full definitions of all the anomalies mentioned in Table 2.1 can be found in the paper *A Critique of ANSI SQL Isolation Levels*. We will discuss how we preclude *fuzzy read, phantom and write skew* in our solution, which is based on *Snapshot Isolation* level in Chapter 5.

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 as 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).

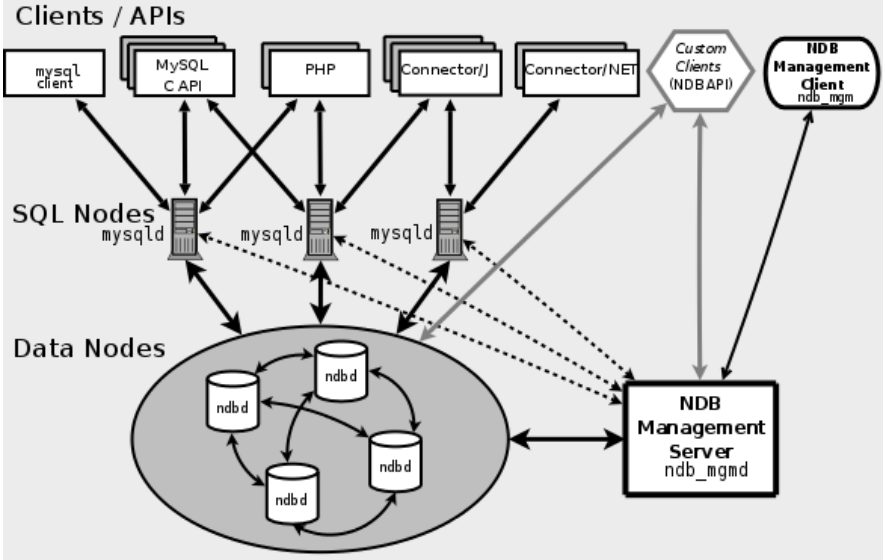


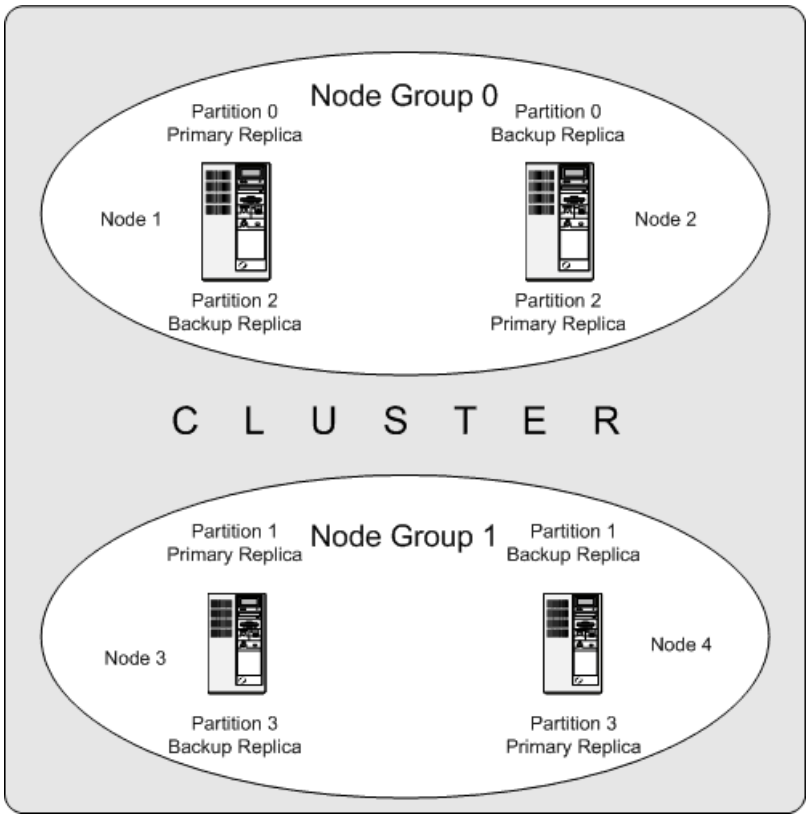
Figura 2.3: The Architecture of MySQL Cluster (MySQL a)

2.3.3 Fault Tolerance in MySQL Cluster

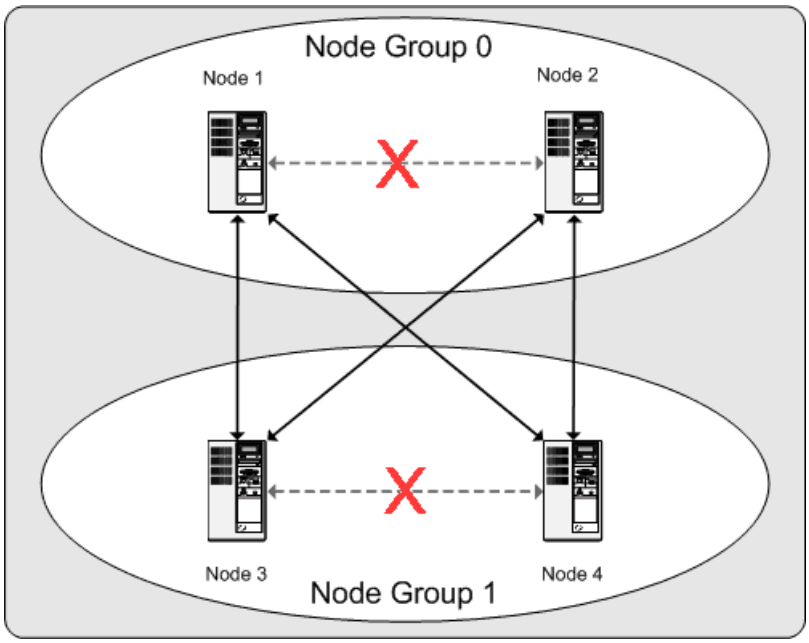
For fault tolerance, data in MySQL Cluster is replicated across multiple ndbds. Ndbds are divided into *node groups*. Each unit of data is called a *partition* stored by ndbd. The partitions of data are replicated within the same *node group*. The number of *node groups* is calculated as:

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

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.



(a) Node Groups in MySQL Cluster



(b) Fault Tolerance in Node Groups

Figura 2.4: Node Groups in MySQL Cluster and Fault Tolerance (MySQL e)

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.

2.3.4 The Benchmark of MySQL Cluster

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

- 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.

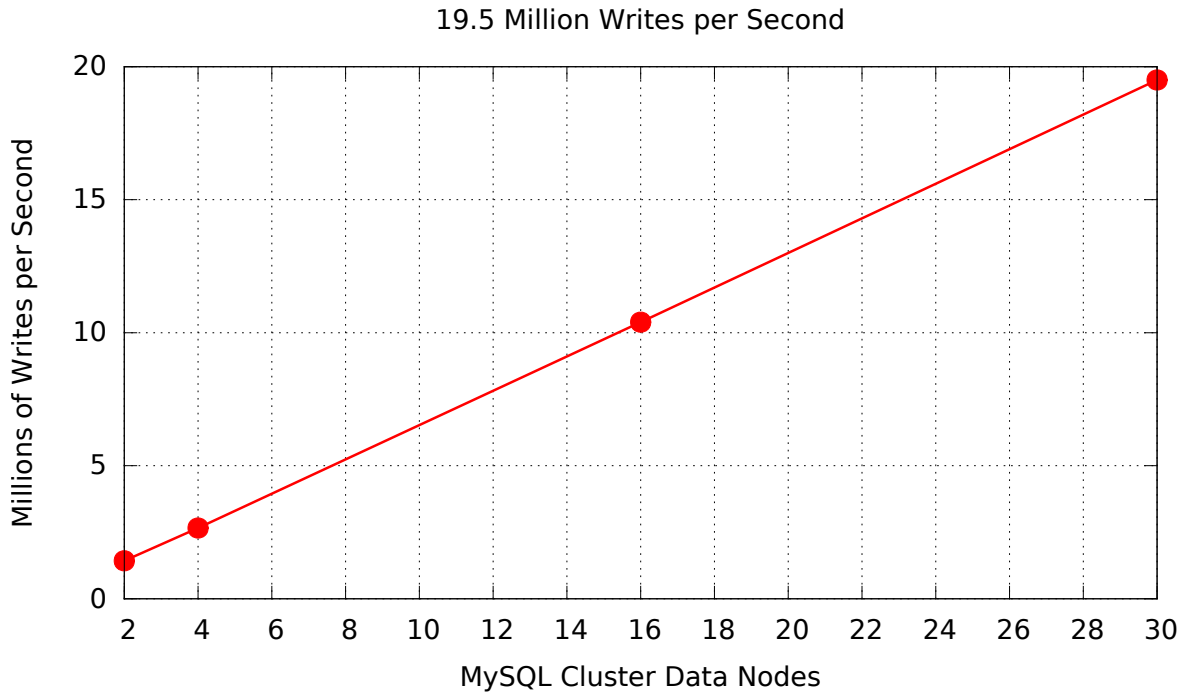


Figura 2.5: MySQL Cluster Scaling-out Writes Operations

Therefore, the high operation 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 d), 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 from the journal 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

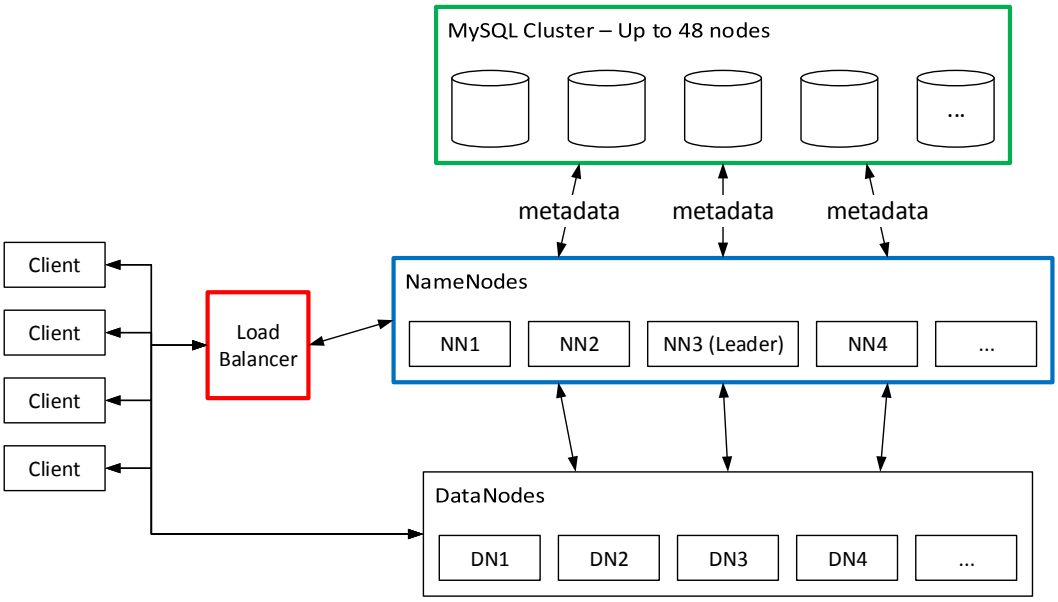


Figura 2.6: The Architecture of Hop-HDFS

is not limited by a single NameNode's heap size so that the scalability problem can be solved. In Hop-HDFS, we have this *multiple stateless NameNodes* architecture so that multiple-writers and 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 removed as a result, which reduces the time on writing edit logs as well as restarting new NameNodes on namespace recovery. Note that we have a leader election process 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 MySQL Cluster can support up to 3.072 TB for metadata with a data replication of 2, which means that Hop-HDFS can store up to 4.1 billion files. A factor of 40 times increase over Shvachko's estimate (Shvachko 2010) for HDFS from 2010.

Summary

In this chapter, we discussed the architectures of relevant distributed file systems GFS and HDFS as well as an in-memory, replicated, distributed database management system MySQL Cluster. We presented the related knowledge on concurrency control and Isolation level in multi-user database management system so that we know the risks when we use MySQL Cluster to be the storage layer. Finally, we introduce Hadoop Open Platform-as-a-service and Hop-HDFS. We illustrated how we overcome the shortcomings in HDFS with a distributed NameNode architecture developed in Hop-HDFS.

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 operations like 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.

Besides, 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

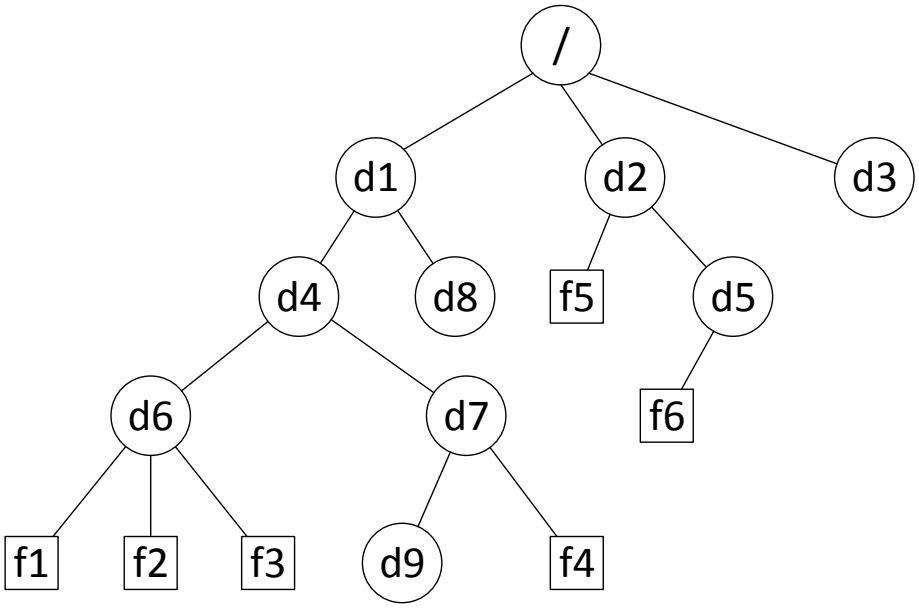


Figura 3.1: A Graphical Tree Representation for the Namespace in GFS

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

3.1.3 Limitations

The concurrency for the namespace is maximized in GFS, but it trades off common file system functions for those 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

Tabela 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

Tabela 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, so it supports operations like *ls, mkdir, rm, cp, chown* in POSIX standards. 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 with semantic related INodes 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 being 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 semantic related attributes with proper order. However, it is not realistic because it complicates the namespace design.

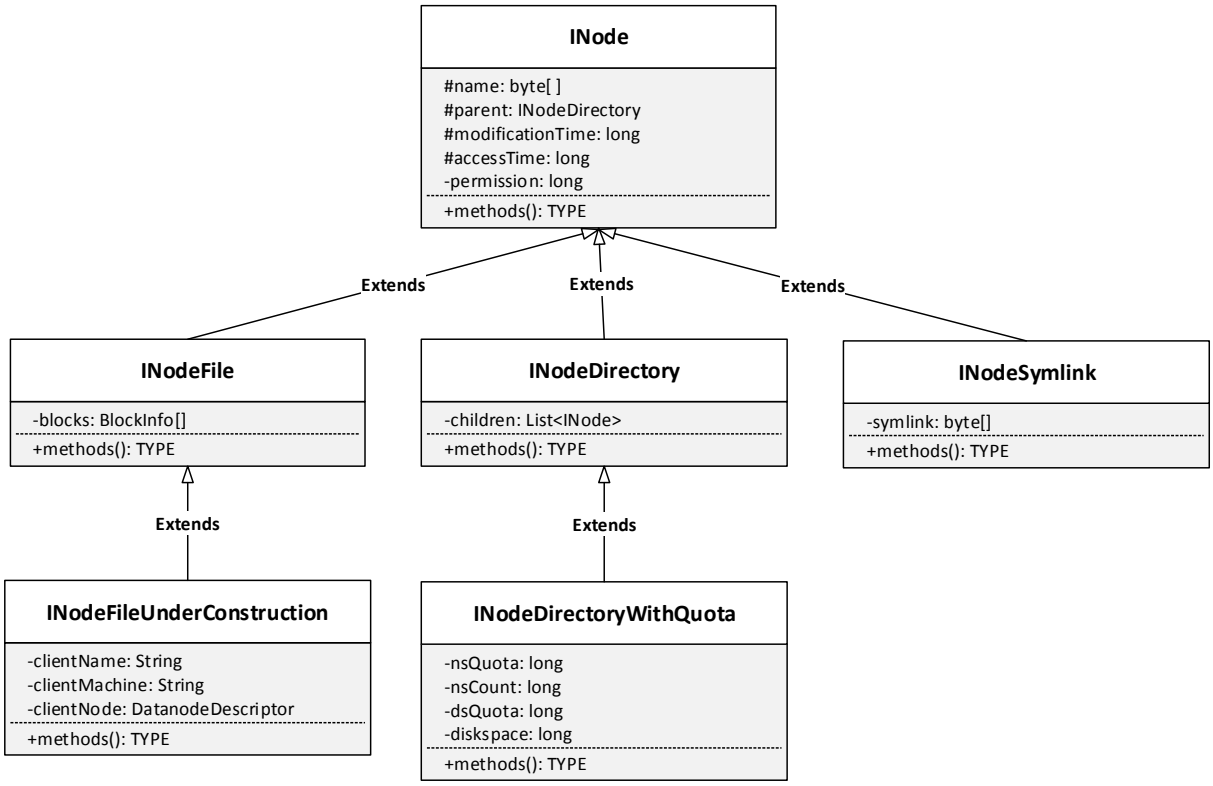


Figura 3.2: The Namespace INode Structure in HDFS

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. We call it *system-level lock*.

HDFS categorizes the metadata operations into *read operations* and *write operations*. The NameNode allows concurrent threads to access shared object for read operations, 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. (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 Limitations

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 for the process:

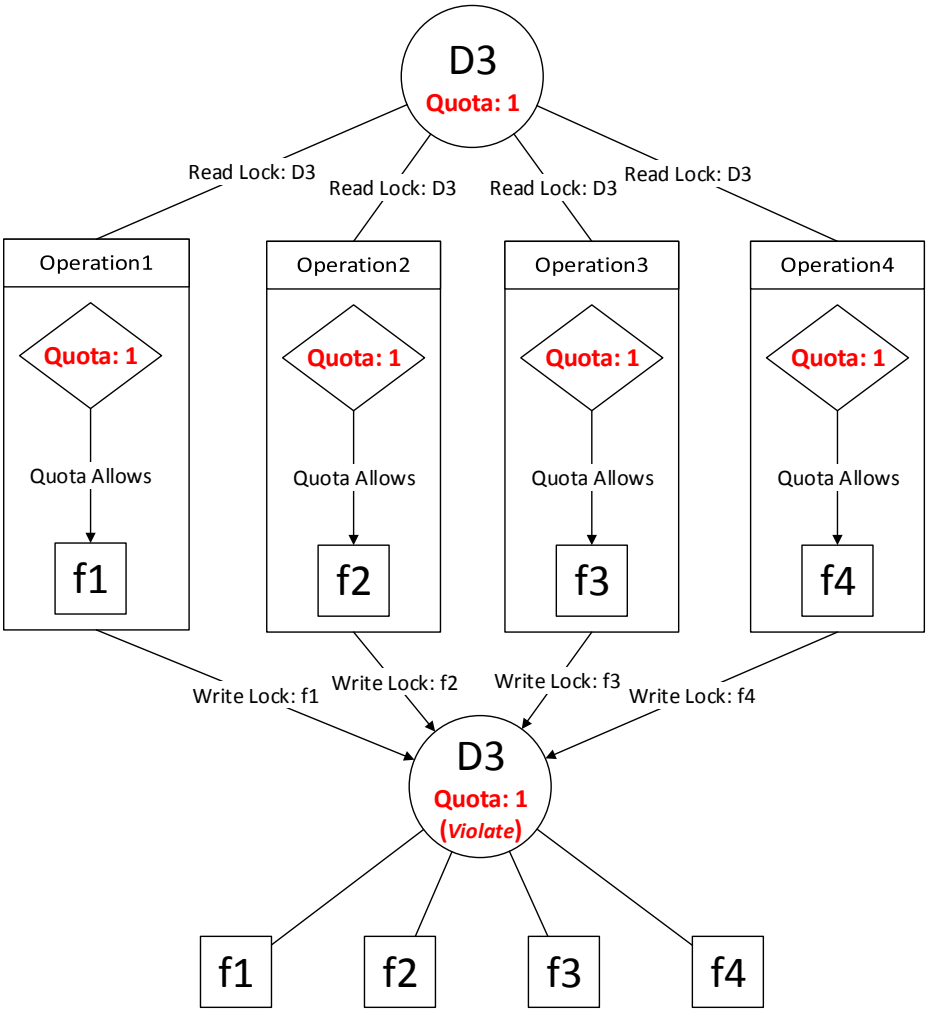


Figura 3.3: Violation in Quota Semantic

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 communication 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.

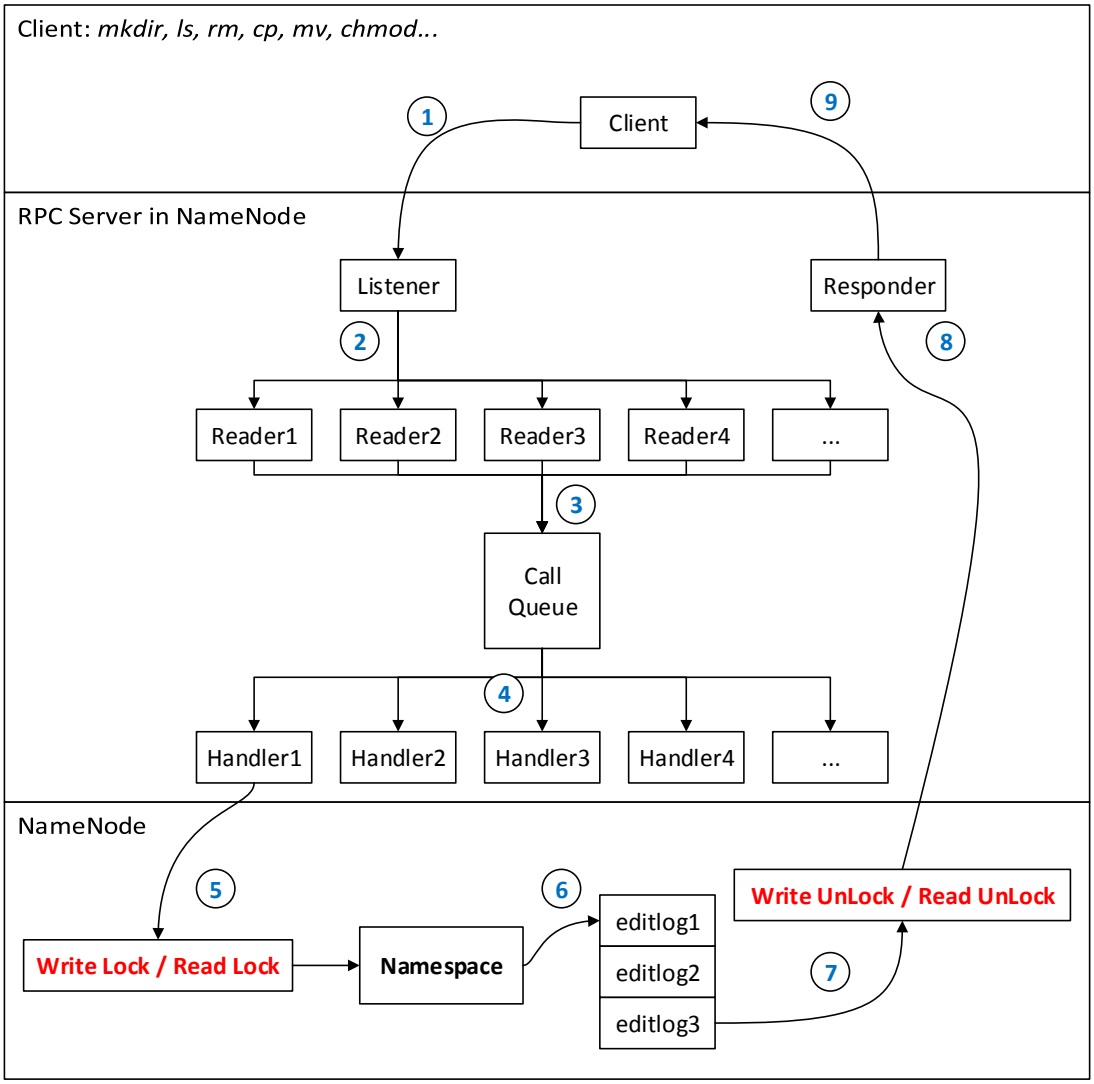


Figura 3.4: RPC between Clients and NameNode for Namespace Operations

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.

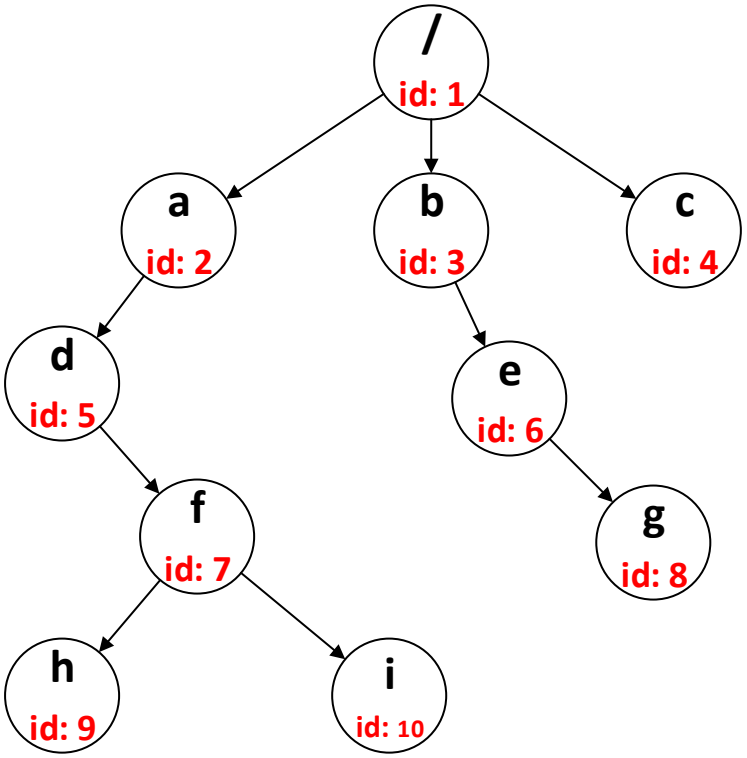


Figura 3.5: Filesystem Hierarchy with ID for INodes in Hop-HDFS

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. Because the INode *id* is not known beforehand on the application side, but the <name, parent_id> pair is known since it can be resolved from the path string. Therefore, data rows can be looked up by the <name, parent_id> pair *Primary Key*

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	...

Tabela 3.3: INode Table for Hop-HDFS

directly from database on the application side.

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

3.3.2 Namespace Concurrency Control

In the first version of Hop-HDFS (Wasif 2012) (also named as KTHFS), the main task is to migrate the metadata from memory to MySQL Cluster. Therefore, it still depends on the system-level lock in HDFS NameNode (fsLock in *FSNamesystem - ReentrantReadWriteLock* to serialize the operations and maintain the semantics. This becomes a big problem since the network latency between NameNode and database is far more larger than it was when operated in-memory in original HDFS. Hence, each operation will take a long time lock on the name system. The throughput heavily decreases. A fine grained locking scheme is needed to improve the concurrency.

In the second version of Hop-HDFS (Peiro Sajjad & Hakimzadeh Harirbaf 2013) (also named as KTHFS), it adopts a fine-grained row-level locking mechanism to improve the throughput while maintaining the strong consistency semantics. It uses transactions with *Pessimistic Concurrency Control* (PCC) to ensure the safety and progress of metadata operations.

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. See Figure 3.6.

Besides, *implicit locking* (Gray et al. 1976) is used to lock on the root of a subtree in a transaction,

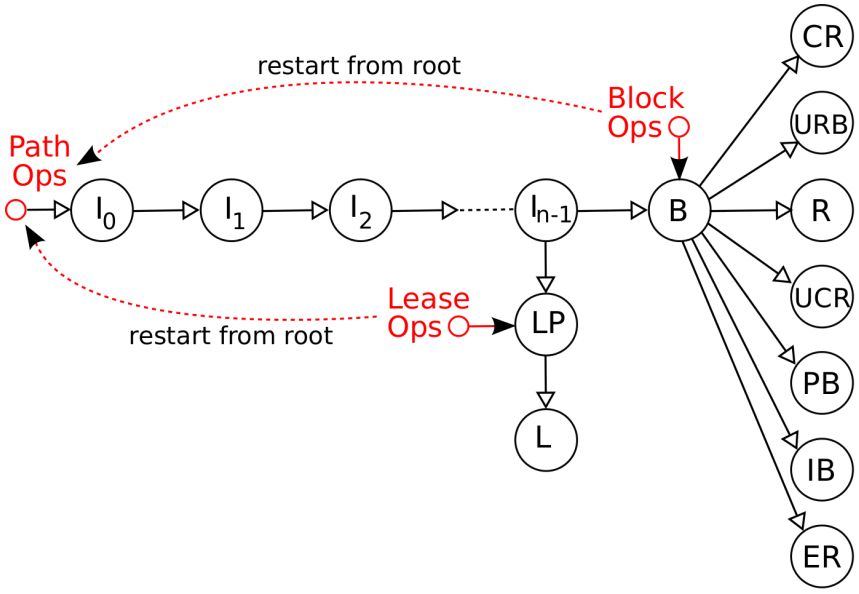


Figura 3.6: Acyclic DAG. Operations start from root, locks taken in order from leftmost child (Hakimzadeh et al. 2014)
I: INode, B: BlockInfo, L: Lease, LP: LeasePath, CR: CorruptedReplica, URB: UnderReplicatedBlock, R: Replica, UCR: UnderConstructionReplica, PB: PendingBlock, IB: InvalidBlock, ER: ExcessReplica

which implicitly acquires locks on all the descendants, so that the strong consistent semantics from original HDFS can be maintained.

3.3.3 Limitations

There are two major limitations in this locking scheme:

1. It lowers the concurrency when multiple transactions try to mutate different descendants within the same subtree. Only one writer is allowed to work on INodes under one directory due to the implicit lock (Write Lock) for the parent directory.
   For example, if transaction Tx1 wants to mutate INode *h*, and another transaction Tx2 wants to mutate INode *i* concurrently in Figure 3.5, Tx1 will take a parent lock on INode *f* first and then perform operations. No more transactions can work under INode *f* at the moment. Tx2 will be blocked by the implicit lock until Tx1 commits. See Table 3.4.
2. There is un-avoided duplicated database round trips overhead. It takes two transactions to finish the implicit locking. The first transaction is used to resolve the path in the database so that we know which rows existed in the database so that the INode's parent

directory can be taken the implicit write lock in the second transaction, or last existing INode directory can be taken the implicit write lock if the path is not full resolved(HDFS will build up the missing intermediate INodeDirectories).

For example, if transaction Tx1 wants to mutate INode *h* in Figure 3.5, in the first database round trip, it needs to resolve the path to see if the related rows of INode */, a, d, f, h* are all in the database. If yes, in the second database round trip, INode */, a, d* will be taken Read Locks [1] and the INode *f* will be taken a Write Lock; if no, the last existing INode will be taken a Write Lock while others will be taken Read Locks.

id	parent_id	name	Locks by Tx1	Locks by Tx2
1	0	/	Read Lock	Read Lock
2	1	a	Read Lock	Read Lock
3	1	b		
4	1	c		
5	2	d	Read Lock	Read Lock
6	3	e		
7	5	f	Write Lock	Write Lock **(Blocked!)**
8	6	g		
9	7	h (Mutated by Tx1)	Write Lock (Implicit)	Write Lock (Implicit) **(Blocked!)**
10	7	i (Mutated by Tx2)	Write Lock (Implicit)	Write Lock (Implicit) **(Blocked!)**

Tabela 3.4: Implicit Lock Table in Hop-HDFS

Summary

In this chapter, we discussed the namespace architecture in GFS, HDFS and Hop-HDFS. We analyzed their namespace concurrency control scheme and concluded with related limitations separately.

[1] The third version of Hop-HDFS is trying to Replace Read Lock to Read_Committed for this PCC scheme

4 Namespace Operation Performance Assessment

In this chapter, we will give the namespace operation performance assessment between the second version (we called it PCC version) of Hop-HDFS and original HDFS under single NameNode. All the tests in this chapter are performed under same the experimental testbed described in Chapter 6 Section 6.1.

4.1 NameNode Throughput Benchmark

A *NNThroughtBenchmark* (Shvachko 2010) tool has been developed to measure the NameNode performance. It is included in the Apache HDFS test package. The tool used in this assessment is based on the code from Apache Hadoop HDFS 2.0.4 Aplha (We integrate the part which tests *mkdirs* operation from HDFS 2.3.0 NNThroughtBenchmark into this one).

NNThroughtBenchmark starts a single NameNode and runs multiple client threads on the same node. The same NameNode operation is performed by each client repetitively via directly calling the implemented method. The number of operations performed by the NameNode per second is measured.

In this section, we aims to give a performance comparison between HDFS and PCC. Since the workload is generated from a single machine by the *NNThroughtBenchmark*, we set the number of operations to be 100 and the number of threads to be 3. For operations *create, delete and rename*, the total number of files involved is 100. They are placed under 4 different directories equally. For operation *mkdirs*, the total number of directories created is 100 and they are also placed under 4 different directories equally. See Figure 4.1 for the operation performance comparison between HDFS and PCC.

From Table 4.1, we find that the throughput of *mkdirs* in PCC is 64.9 % of HDFS, while others are all less than 30%. The reason why the performance of *create, delete and rename* is worse is because they involve multiple NameNode primitive operations. For example, to finish the

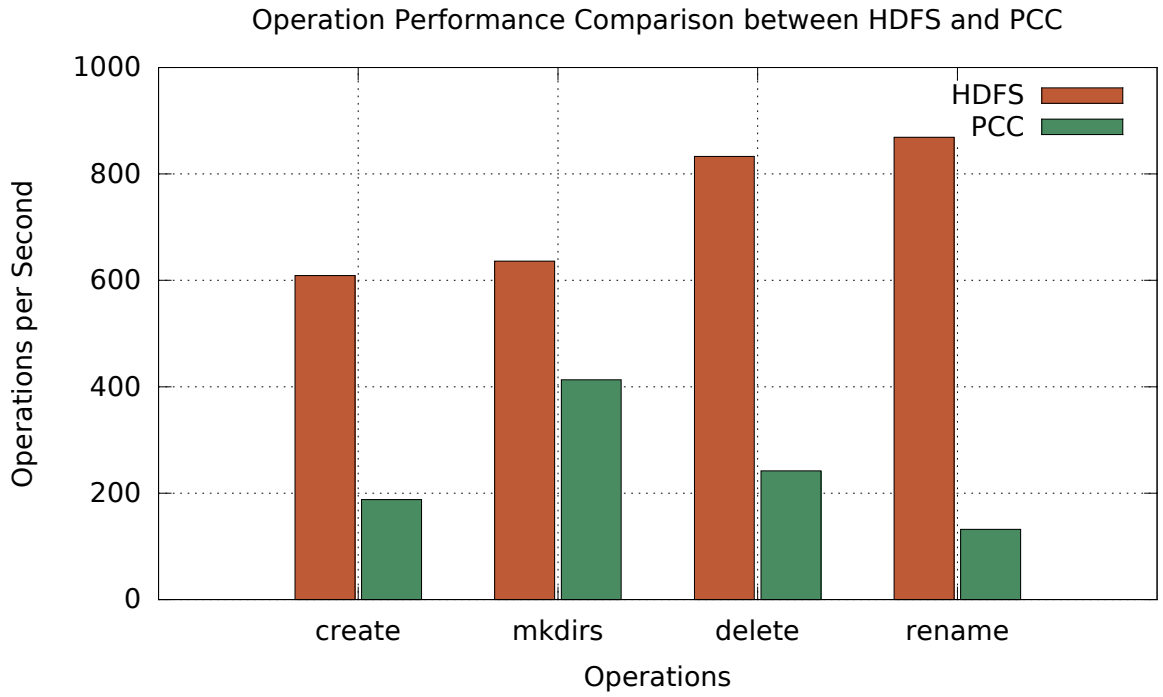


Figura 4.1: Operation Performance Comparison between HDFS and PCC

create operations, it takes two NameNode primitive operations (two transactions): *startFile* and *completeFile*. Since each NameNode primitive operation is implemented as a single transaction, the more primitive operations involved, the more parent write locks will be, which means that more transactions will be blocked.

Operations per Second	**create**	**mkdirs**	**delete**	**rename**
HDFS	609	636	833	869
PCC	188	413	242	132
PCC / HDFS	30.9%	64.9%	29.1%	15.2%

Tabela 4.1: Operation Performance Comparison between HDFS and PCC

4.2 Parent Directory Contention Assessment

The worst case in PCC happens when all concurrent operations try to work under the same directory. Even though they mutate different INodes, all handling transactions will put a parent directory write lock to block each other. Therefore, the parent directory becomes a contention point.

Here we design a test for the parent directory contention assessment. We build a thread pool

with size 1024 for clients. We have three tests with 1000, 10000 and 100000 concurrent clients separately. Each client creates (*mkdirs()*) one sub-directory. All these sub-directories are different, but they are all created under the same parent directory. The parent directory is the contention point in each task. We measure the elapsed time to finish all the concurrent creation tasks in each test.

As we can see from Figure 4.2 and Table 4.2, it takes 4 - 5 more times in PCC to finish all these tasks compared to HDFS. However, when the size of concurrent tasks increases, this ratio decreases. Because as mentioned in Section 3.2.3, under heavy workload, the edit logs in HDFS degrade the NameNode performance. Since there is no edit logging and check pointing part in Hop-HDFS, it works more efficiently than HDFS.

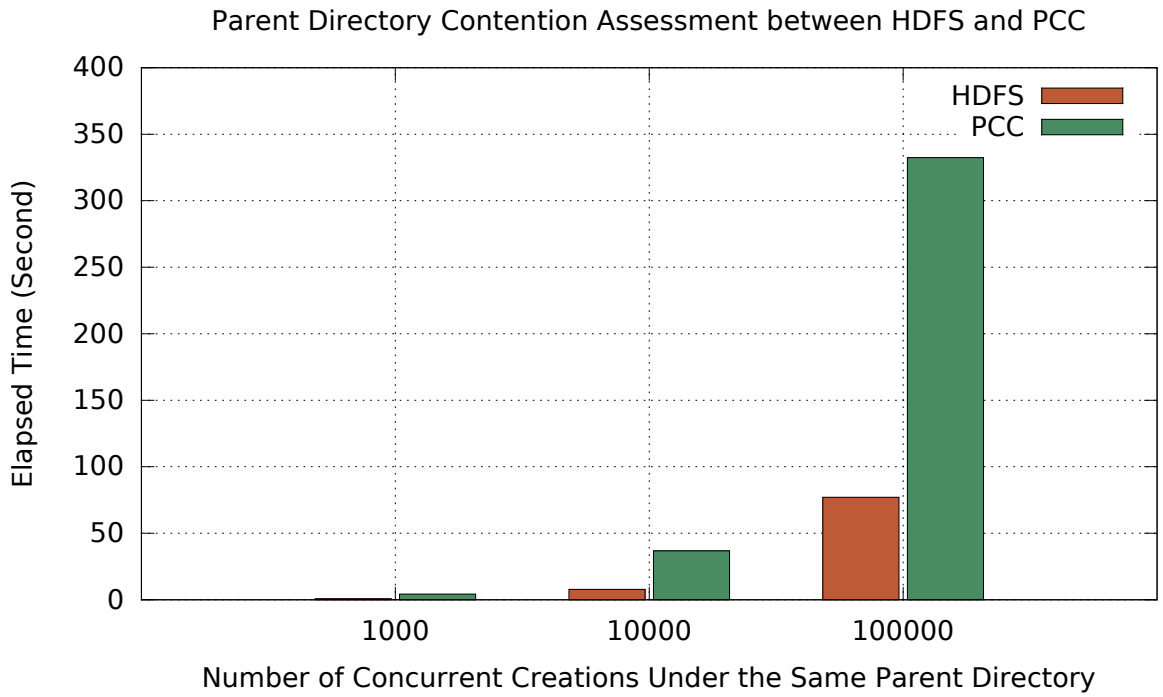


Figura 4.2: Parent Directory Contention Assessment between HDFS and PCC

Num. of Concurrent Creation	**1000**	**10000**	**100000**
HDFS	0.82s	7.83s	77.13s
PCC	4.35s	36.74s	332.36s
PCC / HDFS	530.5%	469.2%	430.9%

Tabela 4.2: Parent Directory Contention Assessment between HDFS and PCC

Note that the tests performed in this chapter is based on single NameNode. The multi-NameNode architecture in Hop-HDFS will help to improve the overall throughput.

Summary

In this chapter we provided a systematic namespace operation performance assessment between the second version of Hop-HDFS (PCC version) and original HDFS. We found that with single NameNode, the performance on PCC was worse than HDFS, especially on write-write intense workload with parent directory contention point. However, when the size of concurrent tasks increases, this gap became smaller because check pointing and journaling were not needed in Hop-HDFS. But those part degraded the NameNode performance in HDFS.

III Algorithmic Solution

5 Solution

The solution we propose to improve the throughput can be summarized as *Optimistic Concurrency Control with Snapshot Isolation on Semantic Related Group*. The solution algorithm consists of 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 their versions all match with the snapshot copy, go to update phase; else, abort and retry current transaction.
4. **Update Phase**: update related data in the database table. Abort and retry transactions if the instance already exists in the database for "new"data. For successful updates, the versions of the modified rows will be increased by 1.

The phases mentioned above will be illustrated in the following sections with more details. 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 a single 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 operation 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 resolve a semantic related group for each INode along the edge of ancestors into 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 in original HDFS.

For each row in *inodes table*, the <name, parent_id> pair is the *Primary Key*. With the full path string, we can iteratively 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	...

Tabela 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. If a transaction reads the same data twice, it will return the same data, that was gather from a snapshot earlier. So all transaction operations will be performed on its own snapshot. Therefore, we called it Per-Transaction Snapshot Isolation.

5.2.1 Fuzzy Read and Phantom Read are Precluded

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

- *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 on 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 since it counts from the snapshot.

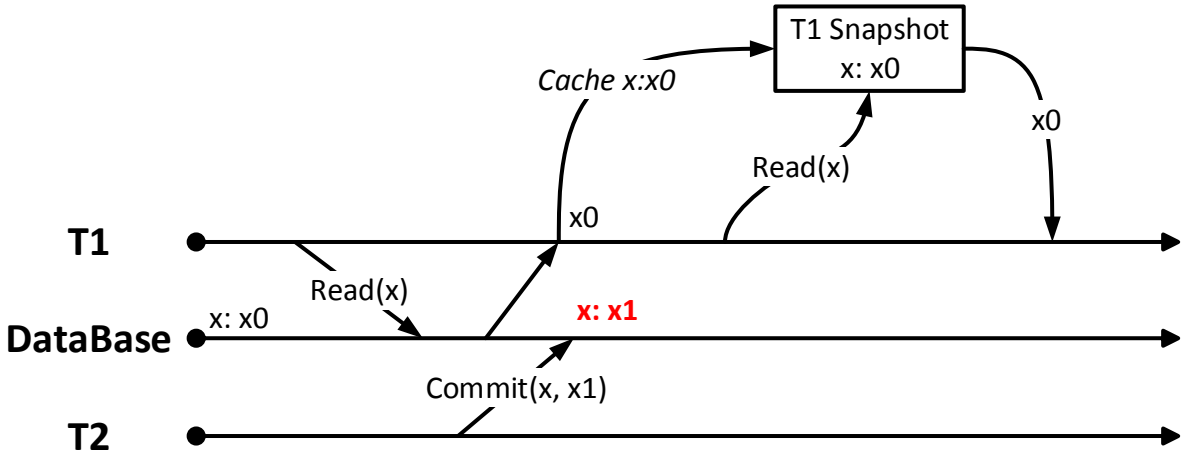


Figura 5.1: Snapshot Isolation Precludes Fuzzy Read

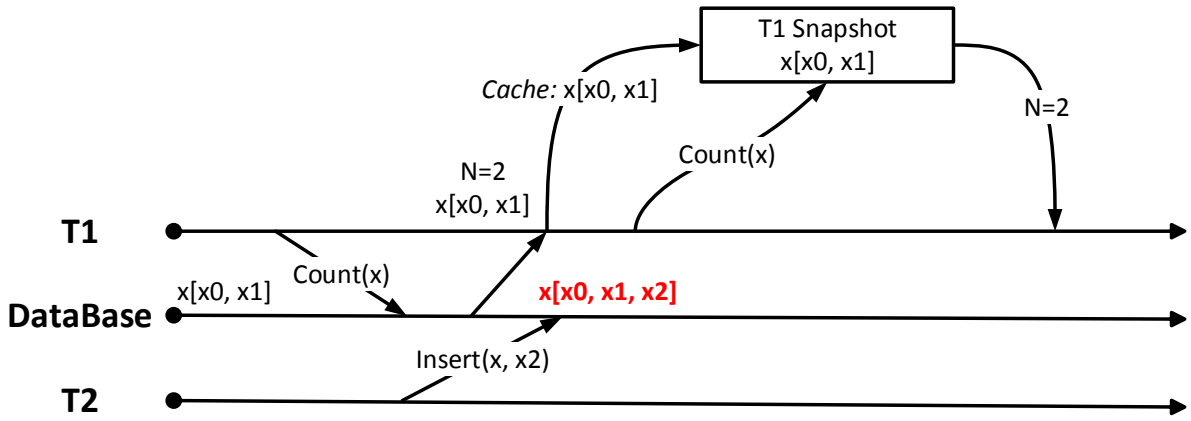


Figura 5.2: Snapshot Isolation with Semantic Related Group Precludes Phantom Read

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), 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	✓	✓	✓

Tabela 5.2: Locks Blocking Table in MySQL Cluster

5.4 Optimistic Concurrency Control

Our algorithm is based on *Optimistic Concurrency Control* (OCC) method 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* 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 so that the parent directory lock is not needed in Hop-HDFS. Hence, transactions can operate under the same directory concurrently.

In *read phase*, transactions use *Read_Committed Lock Mode* to fetch semantic related group as snapshots and cache them in-memory for their own use without being blocked.

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 snapshot copy in the cache for their *versions*. If versions 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 commits while allowing other Shared Locks taken on the same rows for their validation phase.

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

5.4.1 Write Skew is Precluded

The *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 for how 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 by lock cycle. If multiple INodes needed to be updated during update phase, they will be sorted first by the

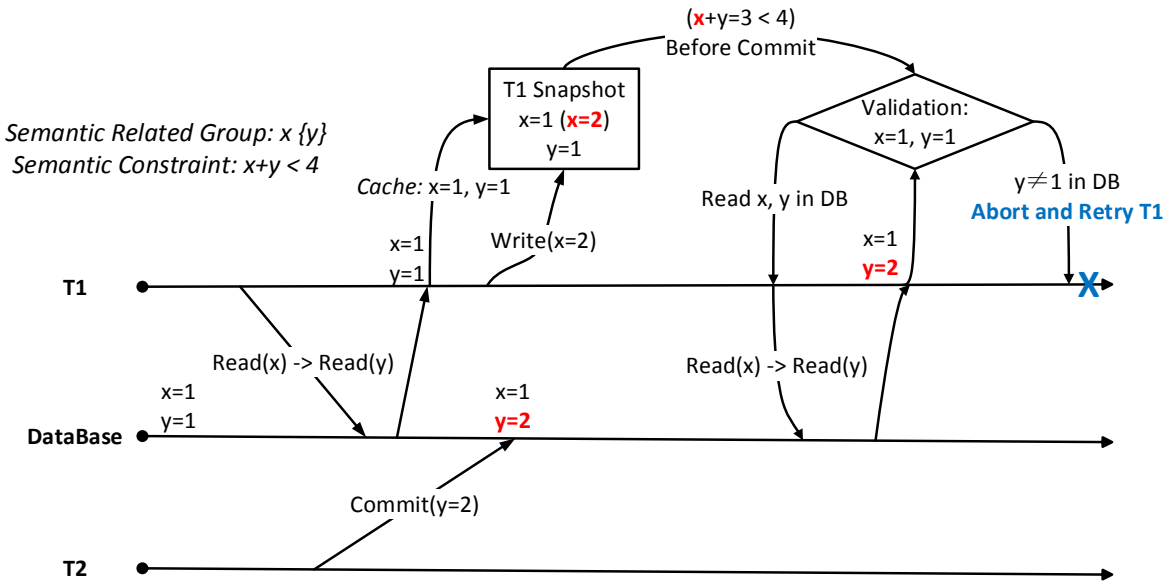


Figura 5.3: Optimistic Concurrency Control with Snapshot Isolation on Semantic Related Group Precludes Write Skew

id values. Then they will be updated in ascending order according by *ids*.

Since we can not take an Exclusive lock on the ”new”row which not yet exists in the database, multiple transactions may try to persist ”new”rows with the same *Primary Key*, and one might be overwritten by the other. Using *makePersistent()* function in ClusterJ can throw exception if the instance already exists in the database.

Finally, for successful updates, the versions of the modified rows will be increased by 1.

5.6 Pseudocode of the Complete Algorithm

The complete algorithm pseudocode can be found in Algorithm 1.

Summary

In this chapter, we proposed a solution to improve the throughput in Hop-HDFS based optimistic concurrency control with snapshot isolation on semantic related Group. We gave all the details on related phases of the algorithm and discuss how anomalies are precluded in our solution. Finally, we provided the complete algorithm pseudocode for our solution.

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

```
1: init: restart = true, try = 0, path = operation.src, TotalRetry = 10
2: while restart and try < TotalRetry do
3:    restart = false
4:    try += 1
5:    tx.snapshot.clear()
6:    tx.begin()
7:    /* 1. Read Phase */
8:    tx.lockMode(Read_Committed)
9:    tx.snapshot = resolve_semantic_related_group(path)
10:   /* 2. Execution Phase */
11:   operation_performTask(tx.snapshot) // HDFS operation performs on its snapshot
12:   /* 3. Validation Phase */
13:   tx.lockMode(Shared)
14:   relatedRows_DataBase = batchRead_Database(tx.snapshot)
15:   tx.lockMode(Exclusive)
16:   modifiedRows_DataBase = batchRead_Database(tx.snapshot)
17:   if versionCompare(relatedRows_DataBase, tx.snapshot) == true and versionCom-
      pare(modifiedRows_DataBase, tx.snapshot) == true then
18:      /* 4. Update Phase */
19:      operation.modifiedRows.version+=1
20:      total_order_sort(operation.modifiedRows)
21:      if batchPersist_Database(operation.modifiedRows) success then
22:         tx.commit()
23:         return SUCCESS // Return HDFS Operation Success
24:      else
25:         tx.abort()
26:         waitForRandomMilliseconds()
27:         retry = true
28:      end if
29:   else
30:      tx.abort()
31:      waitForRandomMilliseconds()
32:      retry = true
33:   end if
34: end while
35: return FAIL // Return HDFS Operation
```

IV Evaluation and Conclusion

6 Evaluation

The solution *Optimistic Concurrency Control with Snapshot Isolation on Semantic Related Group* (OCC) is built on top of the transactional framework (Peiro Sajjad & Hakimzadeh Harirbaf 2013) in the second version of Hop-HDFS (PCC). The goal of this chapter is to prove that our OCC model performs better than PCC. As a proof of concept, we implemented the OCC version for the operation *mkdirs* and also give a detailed evaluation on it compared with the PCC version. For this purpose, we concern about the execution time (elapsed time) needed to finish all the concurrent tasks.

6.1 Experimental Testbed

The MySQL Cluster consists of six data nodes connected by 1 Gigabit LAN. Each data node has an Intel Xeon X5660 CPU at 2.80GHz, and contributes 6 GB RAM (5 GB Data Memory + 1 GB Index Memory) separately. Therefore, the total available memory for the cluster is 36 GB. The number of data replicas is 2. The maximum concurrent transactions is 10000 for each data node, and the inactive timeout for each transaction is 5 seconds.

To avoid any communication overhead caused by RPC connections and serialization, we run the NameNode and Clients on the same machine with Intel i7-4770T CPU at 2.50GHz and 16 GB RAM. This machine is connected with the MySQL Cluster data nodes by 100 Megabits LAN.

6.2 Parent Directory Contention Assessment

This experiment is the same as described in Section 4.2, but we expand it to include the results with OCC. See Figure 6.1 for the workload visual diagram. Here we have a full performance comparison here among HDFS, PCC and OCC.

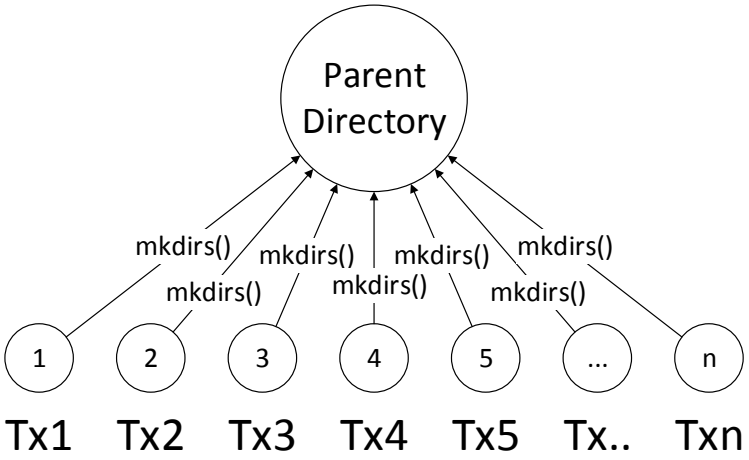


Figura 6.1: Workload of Parent Directory Contention Assessment

From Figure 6.2 and Table 6.1, we can see that OCC significantly outperforms PCC by almost 70 % on this concurrent write-write parent directory contention workload. Under heavy workload, the execution time is just 1.3 times of HDFS. Remember that this is just a single NameNode performance test. We believe that OCC can greatly outperform HDFS in our multiple NameNodes architecture.

Num. of Concurrent Creation	**1000**	**10000**	**100000**
HDFS	0.82s	7.83s	77.13s
PCC	4.35s	36.74s	332.36s
OCC	1.36s	12.01s	103.23s
PCC / HDFS	530.5%	469.2%	430.9%
OCC / HDFS	165.9%	153.4%	133.8%
OCC Improvement: (PCC-OCC) / PCC	**68.7%**	**67.3%**	**68.9%**

Tabela 6.1: OCC Performance Improvement on Parent Directory Contention

6.3 Read-Write Mixed Workload Assessment

In this experiment, we did a test for a read-write mixed workload assessment while the parent directory is still the contention point for PCC. So we assume that OCC will still outperform PCC in this kind of workload.

Similar to the experiment in Section 6.2, we have 1000, 10000 and 100000 concurrent clients' operations running under the same parent directory. But in each task, half of them will do the metadata read operation *getFileStatus()*, while the other half will do the write operation

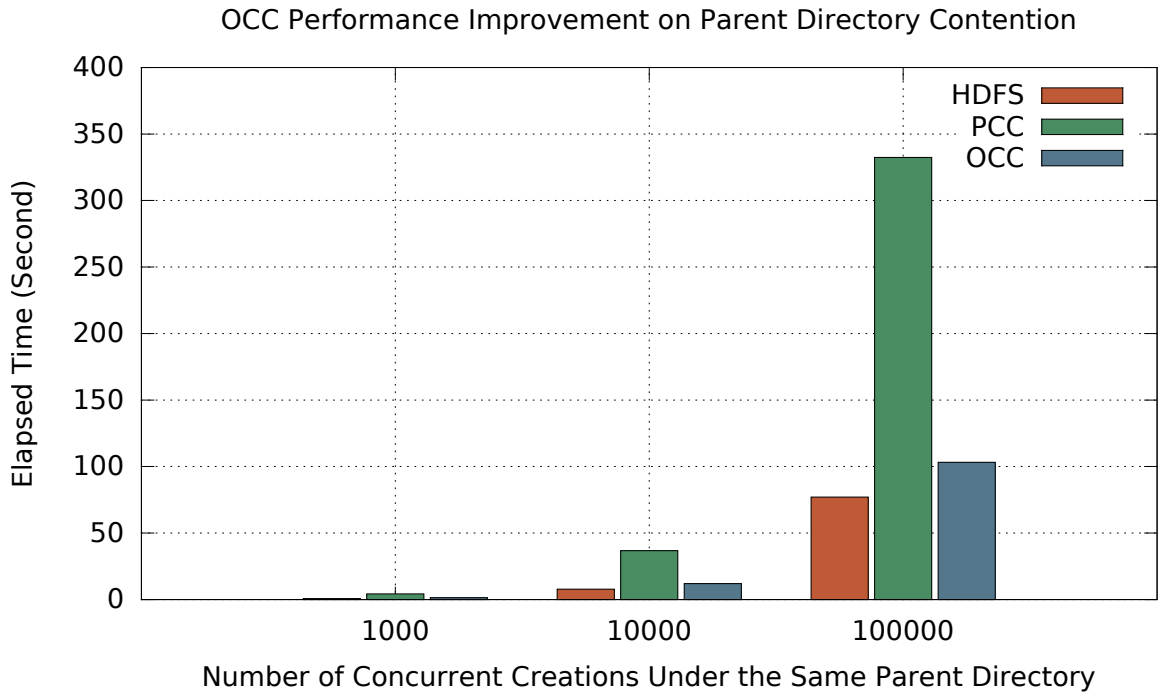


Figura 6.2: OCC Performance Improvement on Parent Directory Contention

mkdirs(). See Figure 6.3 for a visual reference.

From Figure 6.4 and Table 6.2, we can see that OCC still significantly outperforms PCC by 65 % on this concurrent read-write mixed workload.

Num. of Concurrent Creation	**1000**	**10000**	**100000**
PCC	4.92s	50.69s	352.25s
OCC	1.78s	15.31s	120.64s
OCC Improvement: (PCC-OCC) / PCC	**63.8%**	**69.8%**	**65.8%**

Tabela 6.2: OCC Performance Improvement on Read-Write Mixed Workload

6.4 The Size of Semantic Related Group

In the read phase and validation phase, we need to fetch the semantic related group. The more levels of directories involved, the more related data rows needs to be fetch. The depth of the path equals to the size of the semantic related group. But since the namespace is a tree structure, the depth of the namespace won't be too much due to the logarithmic order. Also, HDFS limits the maximum number of levels to be 1000, and maximum number of characters

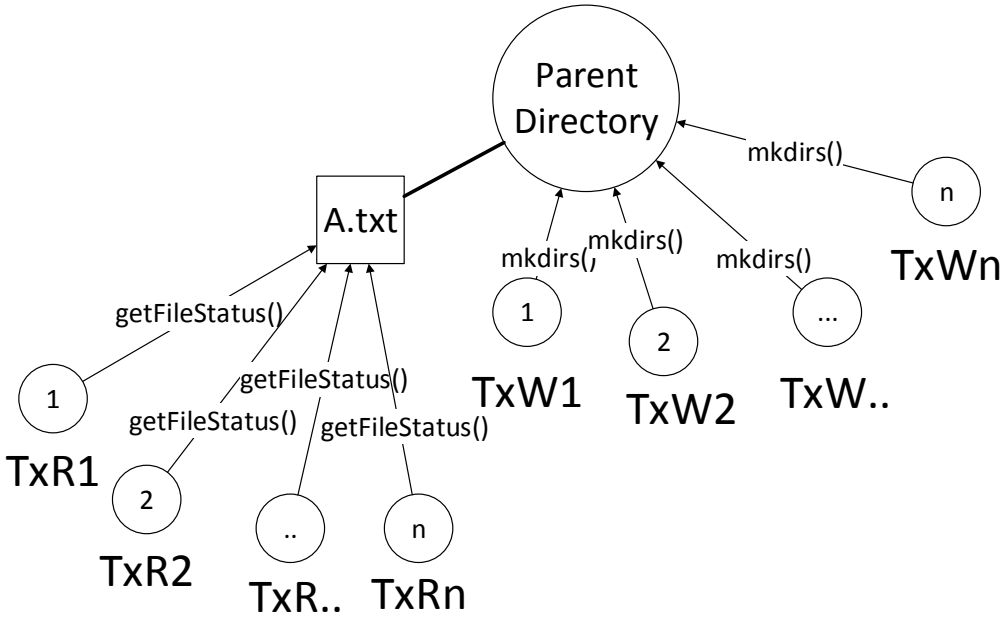


Figura 6.3: Read-Write Mixed Workload

for the full path name to be 3000 [1].

In addition, batch reading is also used to minimize the network round-trips so that multiple data can fetched in one round-trip. Therefore, the size of semantic related group will not be a limitation in practice.

Here we did a test to see how performance affected by different size of semantic related group. Similar to the experiment in Section 6.2, we have 100 concurrent operations (*mkdirs()*) running under the same parent directory. See Figure 6.5 for the linear relationship between the size and the elapsed time.

6.5 OCC Performance with Different Size of Conflicts

When OCC conflicts happen, transactions will abort, wait for random milliseconds and retry. Eventually one transaction will success, and others will get updated values after retry and return RPC callbacks.

Here we have 10000 concurrent operations running under the same parent directory. Each operation creates only one sub-directory. Some of them will success and some others will fail

[1] If we assume that 10 characters for one directory name, the maximum level will be 300.

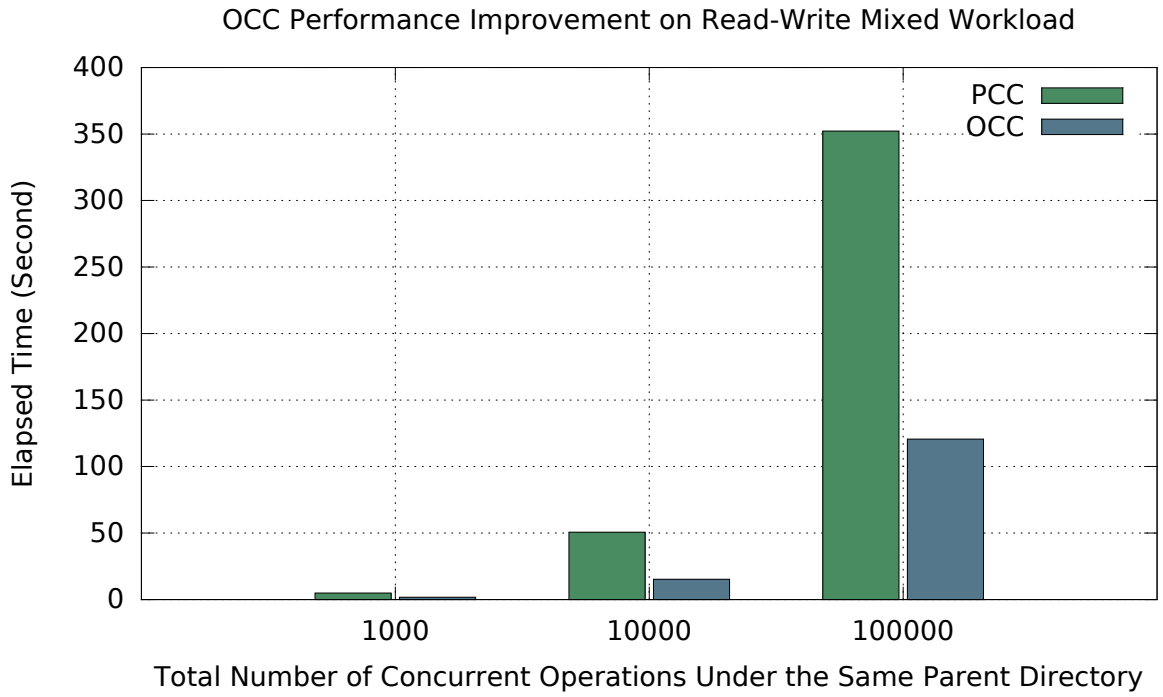


Figura 6.4: OCC Performance Improvement on Read-Write Mixed Workload

due to conflicts. These operations will try to create same sub-directories in different numbers from 1 (100 % conflicts), to 10000 (0 % conflicts). Therefore, we have different size of conflicts.

Total Num. of Sub-Directories Created for 10000 Operations	**Conflict Size**	**Elapsed Time (Second)**	**Performance Decrease Compared to Zero Conflict**
1	100%	14.53	23.7%
10	10%	14.11	20.1%
100	1%	13.51	15.0%
1000	0.1%	12.72	8.23%
10000	0%	11.75	0%

Tabela 6.3: OCC Performance with Different Size of Conflicts

From Figure 6.6 and Table 6.3, we can find that the maximum OCC performance decrease is only 23.7% when 100 % of the operations conflict:

$$(14.53 - 11.75) \div 11.75 = 23.7\%$$

Besides, with Figure 6.7, we find that the OCC performance decrease rate grows very slowly after conflict size 10%. From conflict size 10% to conflict size 100 %, the performance decrease rate only grows from 20.1 % to 23.7 %.

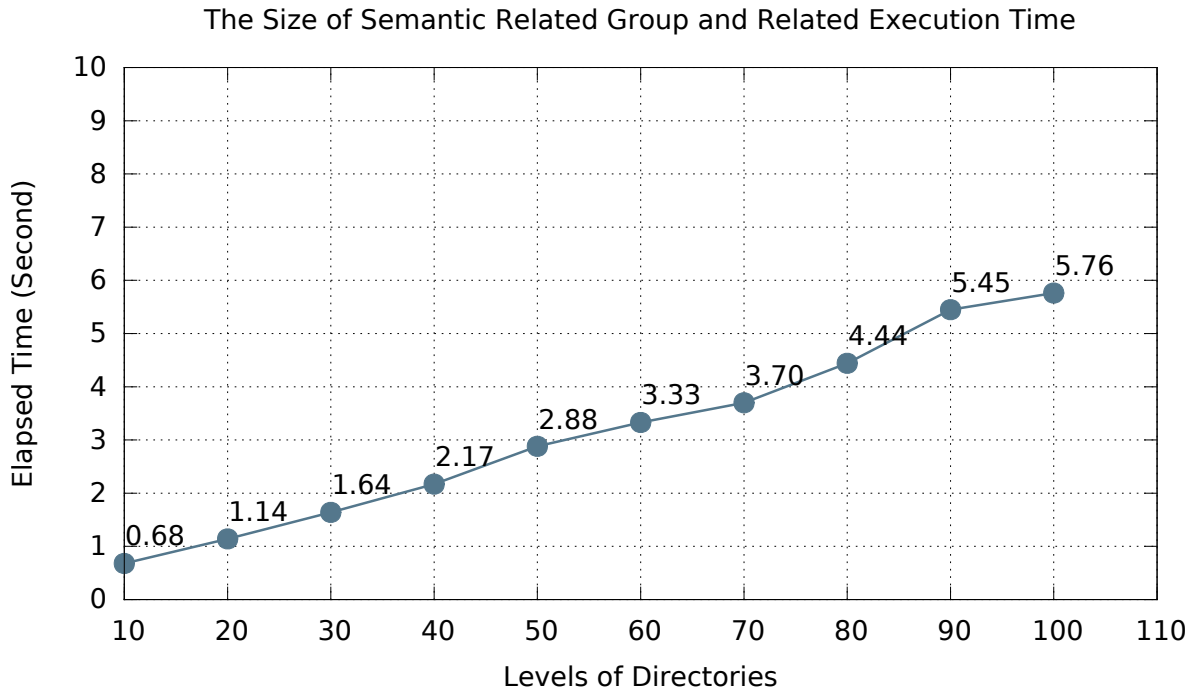


Figura 6.5: The Size of Semantic Related Group and Related Execution Time

6.6 Correctness Assessment

The correctness of our OCC implementation for *mkdirs()* [2] has been validated by 300+ Apache HDFS 2.0.4 Alpha unit tests passing. The full passing tests list can be found in Appendix A.

Summary

In this chapter, we gave a detailed evaluation on our OCC solution compared to the previous work on Hop-HDFS (PCC version). We proved that our OCC model performs better than PCC up to **70 %** on write-write intense concurrent workload and **65 %** on read-write mixed intense concurrent workload. We also evaluated OCC performance decrease on different size of conflict. We found that the maximum OCC performance decrease was only 23.7% when 100 % of the operations conflict, and the decrease rate grew very slowly from conflict size 10% to conflict size 100 %.

[2] other operations are PCC

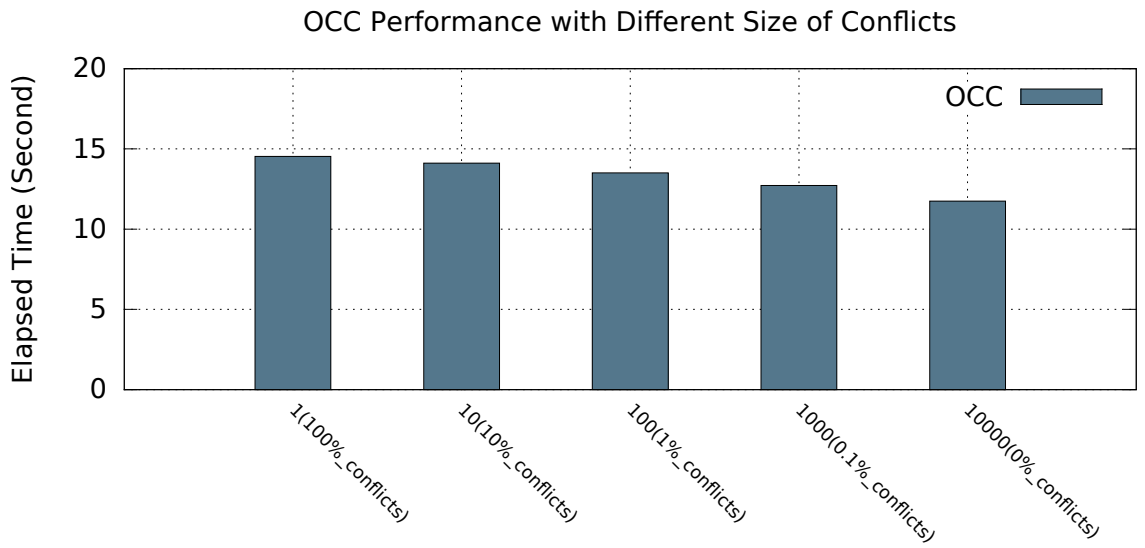


Figura 6.6: OCC Performance with Different Size of Conflicts

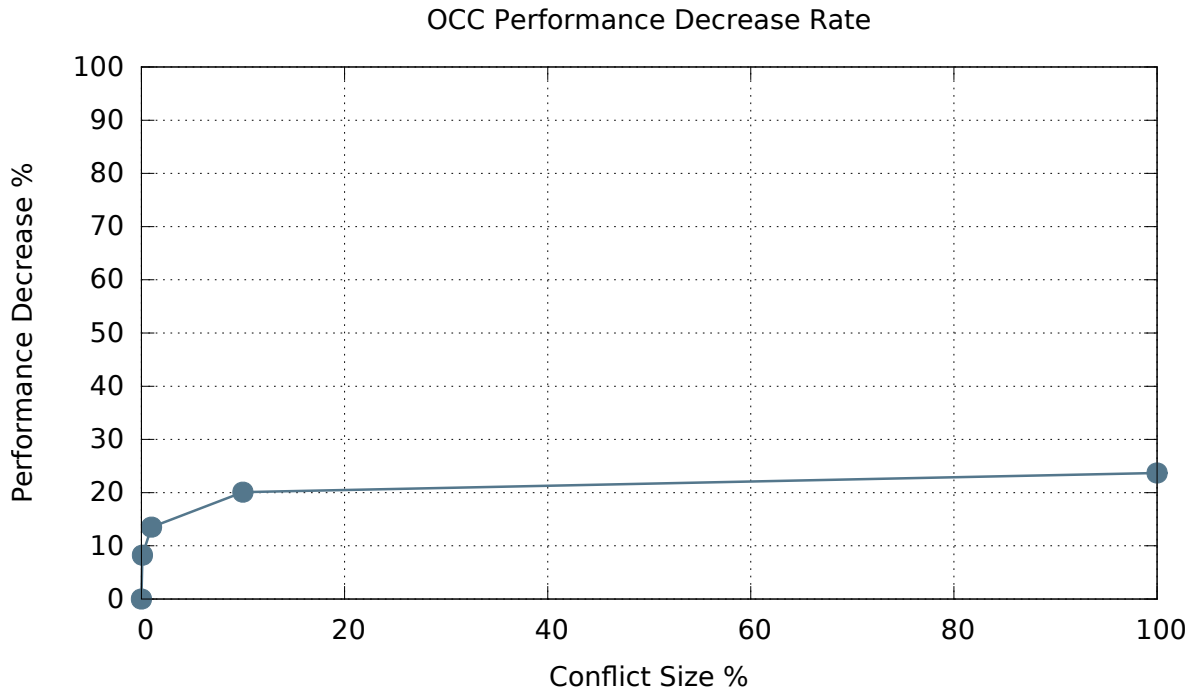


Figura 6.7: OCC Performance Decrease Rate

7 Conclusion and Future Work

7.1 Conclusion

In this thesis, we provide a solution for Hop-HDFS based on optimistic concurrency control with snapshot isolation on semantic related group to improve the operation throughput while maintaining the strong consistency semantics in HDFS.

First, we discuss the architectures of related distributed file systems, including Google File System, HDFS and Hop-HDFS. With focus on their namespace concurrency control schemes, we analyzes the limitation of HDFS's NameNode implementation and provide an overview of Hop-HDFS illustrating how we overcome those problems in the distributed NameNode architecture.

MySQL Cluster is selected to be the distributed in-memory storage layer for the metadata in Hop-HDFS due to its the high operation throughput and high reliability. However, the trade off is that the NDB cluster storage engine of MySQL cluster supports only the *READ COMMITTED* transaction isolation level. *Anomalies* like fuzzy read, phantom, write skew will appear because the write results in transactions will be exposed to reads in different concurrent transactions without proper implementation.

Then, based on optimistic concurrency control with snapshot isolation on semantic related group, we demonstrate how concurrency is improved and anomalies - *fuzzy read*, *phantom*, *write skew* are precluded, so that the strong consistency semantics in HDFS is maintained.

Finally, as a proof of concept, we implemented the OCC version for the operation *mkdirs* and also give a detailed evaluation on it compared with the PCC version. Our solution outperforms previous work of Hop-HDFS up to 70 %. Under heavy workload, the single NameNode performance of HDFS is just a slightly better than OCC. We believe that OCC can greatly outperform HDFS in Hop-HDFS multiple NameNodes architecture. The correctness of our implementation has been validated by 300+ Apache HDFS unit tests passing.

7.2 Future Work

The result of our OCC solution is promising. Other operations in Hop-HDFS can also adopt the same algorithm to achieve better performance.

Future evaluation on Hop-HDFS in multiple NameNodes architecture with OCC solution is needed to prove that it can achieve better performance than HDFS in single NameNode architecture.

Bibliography

Ansi, A. (1992). x3. 135-1992, american national standard for information systems-database language-sql.

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.

Bernstein, P. A. & N. Goodman (1981). Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR) 13*(2), 185–221.

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. 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

```
TestAbandonBlock
TestAllowFormat
TestAppendDifferentChecksum
TestAtomicFileOutputStream
TestAuditLogger
TestAuditLogs
TestAuthFilter
TestBalancerBandwidth
TestBalancer
TestBalancerWithEncryptedTr
ansfer
TestBalancerWithHANameNodes
TestBaseService
TestBestEffortLongFile
TestBlockInfo
TestBlockMissingException
TestBlockPoolManager
TestBlockReaderLocal
TestBlockReplacement
TestBlockReport
TestBlocksScheduledCounter
TestBlocksWithNotEnoughRack
s
TestBlockToken
TestBlockTokenWithDFS
TestBlockUnderConstruction
TestBookKeeperAsHASharedDir
TestBookKeeperConfiguration
TestBookKeeperEditLogStream
s
TestBookKeeperHACheckpoints
TestBookKeeperJournalManage
r
TestByteRangeInputStream
TestCase
TestCase
TestCheck
TestCheckpoint
TestCheckUploadContentTypeF
ilter
TestClientBlockVerification
TestClientProtocolForPipeli
neRecovery
TestClientProtocolWithDeleg
ationToken
TestClientReportBadBlock
TestClusterId
TestConfigurationUtils
TestConnCache
TestCorruptFilesJsp
TestCorruptReplicaInfo
TestCrcCorruption
TestCurrentInprogress
TestCyclicIteration
TestDataDirs
TestDatanodeBlockScanner
TestDatanodeConfig
TestDatanodeDeath
TestDatanodeDescriptor
TestDataNodeExit
TestDatanodeJsp
TestDataNodeMetrics
TestDataNodeMultipleRegistr
ations
TestDataNodeMXBean
TestDatanodeRegister
TestDatanodeRegistration
TestDatanodeReport
TestDatanodeRestart
TestDataNodeVolumeFailure
TestDataNodeVolumeFailureRe
porting
TestDataNodeVolumeFailureTo
leration
TestDataTransferKeepalive
TestDataTransferProtocol
TestDeadDatanode
TestDecommissioningStatus
TestDefaultNameNodePort
TestDelegationTokenFetcher
TestDelegationTokenForProxy
User
TestDelegationToken
TestDelegationTokenManagerS
ervice
TestDelegationTokensWithHA
TestDelimitedImageVisitor
TestDeprecatedKeys
TestDFSAddressConfig
TestDFSClientExcludedNodes
TestDFSClientFailover
TestDFSClientRetries
TestDFSHAAdmin
TestDFSMkdirs
TestDFSPermission
TestDFSRemove
TestDFSRename
TestDFSRollback
TestDFSShellGenericOptions
TestDFSStorageStateRecovery
TestDFSUpgradeFromImage
TestDFSUtil
TestDirectBufferPool
TestDirectoryScanner
TestDirHelper
TestDir
TestDiskError
TestDistributedFileSystem
TestEditLogFileInputStream
TestEditLogFileOutputStream
TestEditsDoubleBuffer
TestEncryptedTransfer
TestEpochsAreUnique
TestExactSizeInputStream
TestExceptionHelper
TestException
TestExtendedBlock
TestFcHdfsCreateMkdir
TestFcHdfsPermission
TestFcHdfsSetUMask
TestFiDataTransferProtocol2
TestFiDataTransferProtocol
TestFiHFlush
TestFiHftp
TestFileAppend2
TestFileAppend3
TestFileAppend4
TestFileAppend
TestFileAppendRestart
TestFileConcurrentReader
TestFileCorruption
TestFileCreationClient
TestFileCreationDelete
TestFileCreationEmpty
TestFileCreation
TestFileJournalManager
TestFileLengthOnClusterRest
art
TestFileLimit
TestFileStatus
TestFileSystemAccessService
TestFiListPath
TestFiPipelineClose
TestFiPipelines
TestFiRename
TestFsck
TestFSImageStorageInspector
TestFSInputChecker
TestFsLimits
TestFSMainOperationsWebHdfs
TestFSNamesystem
TestFSOutputSummer
TestFuseDFS
TestGetBlocks
TestGetConf
TestGetGroups
TestGetImageServlet
TestGetUriFromString
TestGlobPaths
TestGroupsService
TestGSet
TestHABasicFailover
TestHABasicFileCreation
TestHAConfiguration
TestHAFailoverUnderLoad
TestHARead
TestHDFSCLI
TestHDFSConcat
TestHDFSFileSystemContract
TestHdfsHelper
TestHdfs
TestHdfsNativeCodeLoader
TestHDFSServerPorts
TestHDFSTrash
TestHeartbeatHandling
TestHFlush
TestHftpDelegationToken
TestHftpFileSystem
TestHftpURLTimeouts
TestHost2NodesMap
TestHostnameFilter
TestHostsFiles
TestHSync
TestHttpFSFileSystemLocalFi
leSystem
TestHttpFSKerberosAuthentic
ationHandler
TestHttpFSServer
```

Figura A.1: Apache HDFS 2.0.4 Alpha Unit Tests Passing List 1

```
TestInjectionForSimulatedSt
orage
TestINodeFile
TestInputStreamEntity
TestInstrumentationService
TestInterDatanodeProtocol
TestIPCLoggerChannel
TestIsMethodSupported
TestJettyHelper
TestJetty
TestJMXGet
TestJournal
TestJournalNode
TestJSONMapProvider
TestJSONProvider
TestJsonUtil
TestJspHelper
TestLargeBlock
TestLargeDirectoryDelete
TestLayoutVersion
TestLease
TestLeaseRecovery2
TestLeaseRecovery
TestLeaseRenewer
TestLightWeightHashSet
TestLightWeightLinkedSet
TestListCorruptFileBlocks
TestListFilesInDFS
TestListFilesInFileContext
TestListPathServlet
TestLoadGenerator
TestLocalDFS
TestMD5FileUtils
TestMDCFilter
TestMetaSave
TestMiniJournalCluster
TestMissingBlocksAlert
TestModTime
TestMultiThreadedHflush
TestNameCache
TestNamenodeCapacityReport
TestNameNodeJspHelper
TestNameNodeMetrics
TestNameNodeMXBean
TestNameNodeResourcePolicy
TestNetworkTopology
TestNNMetricFilesInGetListi
ngOps
TestNNStorageRetentionManag
er
TestNNThroughputBenchmark
TestNNWithQJM
TestNodeCount
TestOfflineEditsViewer
TestOfflineImageViewer
TestOffsetUrlInputStream
TestOIVCanReadOldVersions
TestOverReplicatedBlocks
TestPacketReceiver
TestParallelLocalRead
TestParallelRead
TestParallelReadUtil
TestParam
TestParam
TestPathComponents
TestPBHelper
TestPendingDataNodeMessages
TestPendingReplication
TestPermission
TestPersistBlocks
TestPipelines
TestPread
TestProcessCorruptBlocks
TestProxyUserService
TestQJMWithFaults
TestQuorumCall
TestQuorumJournalManager
TestQuorumJournalManagerUni
t
TestRBWBlockInvalidation
TestReadWhileWriting
TestRefreshNamenodes
TestRefreshUserMappings
TestRenameWhileOpen
TestReplaceDatanodeOnFailur
e
TestReplicaMap
TestReplication
TestReplicationPolicy
TestResolveHdfsSymlink
TestRestartDFS
TestRoundRobinVolumeChoosin
gPolicy
TestRunnableCallable
TestSafeMode
TestSchedulerService
TestSecureNameNodeWithExter
nalKdc
TestSecurityTokenEditLog
TestSeekBug
TestSegmentRecoveryComparat
or
TestServerConstructor
TestServer
TestServerWebApp
TestSetrepDecreasing
TestSetrepIncreasing
TestSetTimes
TestShortCircuitLocalRead
TestSimulatedFSDataset
TestSmallBlock
TestSocketCache
TestStartSecureDataNode
TestStartupOptionUpgrade
TestStickyBit
TestStreamFile
TestTransferRbw
TestUnderReplicatedBlockQue
ues
TestUnderReplicatedBlocks
TestUrlStreamHandler
TestUserProvider
TestValidateConfigurationSe
ttings
TestViewFileSystemAtHdfsRoo
t
TestViewFileSystemHdfs
TestViewFsAtHdfsRoot
TestViewFsFileStatusHdfs
TestViewFsHdfs
TestVolumeId
TestWebHdfsDataLocality
TestWebHdfsFileSystemContra
ct
TestWebHDFS
TestWebHdfsUrl
TestWebHdfsWithMultipleName
Nodes
TestWriteConfigurationToDFS
TestWriteRead
TestWriteToReplica
TestXException
TestXMLUtils
```

Figura A.2: Apache HDFS 2.0.4 Alpha Unit Tests Passing List 2