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

Qi Qi

Instituto Superior Tcnico - IST (Portugal)
Royal Institute of Technology - KTH (Sweden)

**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.

**Keywords:** HDFS, MySQL Cluster, Concurrency Control, Snapshot Isolation, Throughput

## 1 Introduction

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 [1]. Inspired by the *Google File System* (GFS) [2], 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.

Built upon the single namespace server (*the NameNode*) architecture, one well-known shortcoming of HDFS is the limitation to growth [3]. 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.

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.

However, in HDFS, the correctness and consistency of the namespace is ensured by atomic metadata mutation [4]. 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.

**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.

## 2 Background and Related Work

### 2.1 The Hadoop Distributed File System

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, Pig, Mahout, 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 [5].

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*. 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.

### 2.2 Hadoop Open Platform-as-a-service and Hop-HDFS

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 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.

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: (1) the scalability of the namespace - the memory size restricts the storage capacity in the system; (2) the throughput problem - the throughput of the metadata operations is bounded by the ability of the single machine (NameNode); (3) 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.

The architecture of Hop-HDFS consists of multiple *NameNodes*, multiple *DataNodes*, a *MySQL cluster* and is accessed by multiple *clients* as shown in Figure 1. The design purpose for Hop-HDFS is to migrate the metadata from NameNode to an external distributed, in-memory, replicated database *MySQL Cluster*. Therefore, the size of the metadata is not limited by a single NameNode's heap size 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.

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

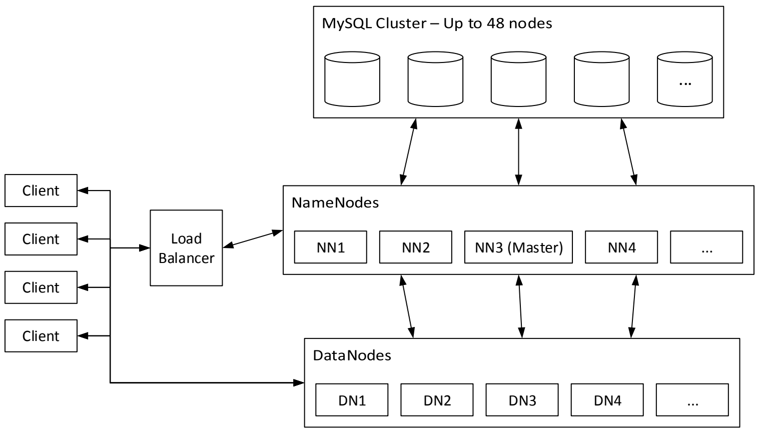

Fig. 1: The Architecture of Hop-HDFS

maximum storage capacity for HDFS in practice. For MySQL Cluster, it supports up to 48 datanodes, 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 [3] for HDFS from 2010.

### 2.3 Namespace Concurrency Control and Performance Assessment in Hop-HDFS

**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.*

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 2. The table representation in the database is like Table 1.

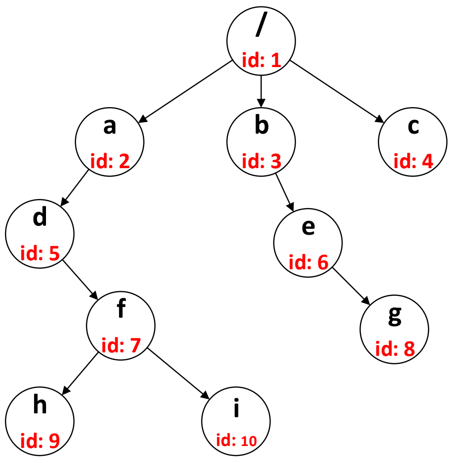

Fig. 2: Filesystem Hierarchy with ID for INodes in Hop-HDFS

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* 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.

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

Table 1: INode Table for Hop-HDFS

**Namespace Concurrency Control** In the first version of Hop-HDFS [6] (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 [7] (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. Besides, *implicit locking* [8] is used to lock on the root of a subtree in a transaction, which implicitly acquires locks on all the descendants, so that the strong consistent semantics from original HDFS can be maintained.

**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 2, 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 2.

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 2, 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.

**Namespace Operation Performance Assessment** Here we will give the namespace operation performance assessment between the second version (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 Section 4.

We aims to give a performance comparison between HDFS and PCC. Since the workload is generated from a single machine by the *NNThroughtBenchmark* [3], 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

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

| **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!)** |

Table 2: Implicit Lock Table in Hop-HDFS

directories created is 100 and they are also placed under 4 different directories equally. See Table 3 for the operation performance comparison between HDFS and PCC.

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 *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% |

Table 3: Operation Performance Comparison between HDFS and PCC

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. 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 Table 4, 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 **??**, 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.

| **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% |

Table 4: 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.

## 3 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.

### 3.1 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* [9] 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*.

Each *Path* object is related to a string representation of the "/" based absolute path name. For example, in Figure 2, the path 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.

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

Table 5: Table Representation for the Semantic Related Group

### 3.2 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) [9] on database server side. However, we need to implement snapshot isolation on the application side since MySQL Cluster supports only the READ COMMITTED isolation level.

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

**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: *2. Fuzzy Read* is precluded by *snapshot isolation*: Transactions read from snapshot instead of database, not affected by the value committed by others. *2. Phantom Read* is also precluded by *snapshot isolation on Semantic Related Group*: Transactions snapshot the semantic related group of after the operation. So if same operation peformed, it is not affected by the value committed by others since it operate from the snapshot.

## 3.3 ClusterJ and Lock Mode in MySQL Cluster

*ClusterJ* is a Java connector based on object-relational mapping persistence frameworks to access data in MySQL Cluster. It doesn't depend on the MySQL Server to access data in MySQL Cluster as it communicates with data nodes directly, which means that ClusterJ can perform some operations much more quickly.

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. 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.

## 3.4 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.

**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.

## 3.5 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 rows needed to be during update phase, they will be sorted first by the *id* value. 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.

### 3.6 Pseudocode of the Complete Algorithm

See Algorithm 1.

**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 versionCompare(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
```

## 4 Evaluation

The goal of this chapter is to proof 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.

Testbed Setup: 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.

### 4.1 Parent Directory Contention Assessment

This experiment is the same as described in Section 2.3 but we expand it to include the results with OCC. Therefore, we have a full performance comparison here among HDFS, PCC and OCC. From Table **??**, 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%** |

Table 6: OCC Performance Improvement on Parent Directory Contention

### 4.2 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 2.3, 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 *mkdirs()*.

From Table **??**, 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%** |

Table 7: OCC Performance Improvement on Read-Write Mixed Workload

### 4.3 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 due to conflicts. These operations will try to create same sub-directories in different numbers from 1 (all conflicts), to 10000 (no conflicts). Therefore, we have different size of conflicts.

From Table 8, we can calculate that the maximum performance degradation is 19.1% when all operations conflict: $(14.53 - 11.75) \div 14.53 = 19.1\%$

## 5 Conclusion

This thesis addresses the problem by investigating the fault-tolerance mechanism of Hadoop, a popular implementation of MapReduce. It presents some results to illustrate Hadoop's behaviors under different

| Total Num. of Sub-DirectoriesCreated for 10000 Operations | Elapsed Time (Second) |
|---|---|
| 1 (all conflicts) | 14.53 |
| 10 (1000 conflicts) | 14.11 |
| 100 (100 conflicts) | 13.51 |
| 1000 (10 conflicts) | 12.72 |
| 10000 (no conflicts) | 11.75 |

Table 8: OCC Performance with Different Size of Conflicts

situations. The intent was to confirm the mechanism and analyze the drawbacks of how Hadoop handles failures.

We proposed the designed of a new feature for Hadoop: the waste-free preemption function. By allowing a task to preserve its state and release the slot in a timely manner, Hadoop can have more control over resources. This in turn will help increase the performance of Hadoop under the occurrence of failure. The preemption feature was implemented not only with the intention of improving performance under failure, but also to open new possibilities for further improvement under other circumstances.

Finally, we evaluated the effectiveness of the new feature considering some basics metrics such as execution time and data locality. In this work, we compare the original Fifo scheduler with a preemptive version. Experiments show that our new feature improves the execution time of Hadoop job when failure occurs. Although the preemption mechanism imposes some overhead, if wisely used, it can greatly improve Hadoop's performance.

### Future work

Our work suffers from some concerns regarding the usability, as well as efficiency. Firstly, the Reduce Pause and Resume mechanism should be further extended to allow preemption during Reduce phase. This allows a more fine-grain control over Reduce tasks. Secondly, it would be interesting to evaluate the cost of preempting a Reduce task, in comparison with Killing. Killing instantly releases the slot, but wastes the effort, while Preemption may take sometime before the slot is released. Also, upon resuming, Reduce tasks also require some extra time to load up the data from the local storage. These two delays add up and in some cases, it is more beneficial to decide to Kill rather than Preempt a Reduce task.

## References

1. Borthakur, Dhruba. "HDFS architecture guide." HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf (2008).
2. Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS Operating Systems Review 37, no. 5 (2003): 29-43.
3. Shvachko, Konstantin V. "HDFS Scalability: The limits to growth." login 35, no. 2 (2010): 6-16.
4. Shvachko, Konstantin, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system." In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, pp. 1-10. IEEE, 2010.
5. White, Tom. "Hadoop: The definitive guide." (2012).
6. Wasif, Malik. "A Distributed Namespace for a Distributed File System." (2012).
7. Peiro Sajjad, Hooman, and Mahmoud Hakimzadeh Harirbaf. "Maintaining Strong Consistency Semantics in a Horizontally Scalable and Highly Available Implementation of HDFS." (2013).
8. Lorie, Raymond A., Gianfranco R. Putzolu, and Irving L. Traiger. "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. 1976.
9. Berenson, Hal, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. "A critique of ANSI SQL isolation levels." ACM SIGMOD Record 24, no. 2 (1995): 1-10.
10. Wasif, Malik. "A Distributed Namespace for a Distributed File System." (2012).
11. Wasif, Malik. "A Distributed Namespace for a Distributed File System." (2012).
12. Wasif, Malik. "A Distributed Namespace for a Distributed File System." (2012).
13. Wasif, Malik. "A Distributed Namespace for a Distributed File System." (2012).
14. Wasif, Malik. "A Distributed Namespace for a Distributed File System." (2012).