BIOBANKCLOUD
Your PaaS for Biobanking

SEVENTH FRAMEWORK PROGRAMME

Project number: 317871
Project acronym: BIOBANKCLOUD

Project title: Scalable, Secure Storage of Biobank Data

Project website: http://www.biobankcloud.eu

Project coordinator: Jim Dowling (KTH)

Coordinator e-mail: jdowling@kth.se

WORK PACKAGE 2:
Scalable Storage

WP leader: Jim Dowling
WP leader organization: KTH
WP leader e-mail: jdowling@kth.se

PROJECT DELIVERABLE

D2.3
Scalable and Highly Available HDFS

Due date: 30th November, 2014 (M11)

Editor
Jim Dowling (KTH)

Contributors
Salman Niazi, Mahmoud Ismail, Steffen Grohsschmiedt, Jim Dowling (KTH)

Executive Summary

As of 2014, much of the biological material being collected for Biobanks will be sequenced using Next-Generation sequencing (NGS) machines. Whole-genome sequencing is, in particular, increasingly gaining in popularity due to the large amount of useful data produced at rapidly decreasing costs. High-coverage whole-genome sequencing (60x) produces up to 250 GB of raw data as well as roughly an equivalent amount of aligned genomic data after processing. All this data needs to be both stored and available for processing by bioinformaticians. This document describes how we built a scalable, highly-available storage system to support the storage of hundreds petabytes of genomic data. We based our storage system on Apache's Hadoop Distributed File System (HDFS) that can scale to store tens of petabytes. However, HDFS has scalability limitations that prevent it storing larger amounts of data. Its main scalability limitation is that its metadata must fit on the heap of the single Java virtual machine (JVM) that runs the NameNode service (a metadata service that maps filenames to blocks and hosts in the system). Another limitation of HDFS for genomics is that its data is replicated three times to ensure its high availability. In this document, we present Hops-FS, a new architecture for HDFS, that introduces stateless NameNodes where the metadata is now stored in an external in-memory, highly available, distributed database. By using an external database, Hops-FS can support significantly larger amounts of metadata compared to Apache HDFS. As Hops-FS supports multiple stateless NameNodes, the number of the NameNodes can be elastically increased or decreased according to the cluster size and the file system throughput requirements. Similarly, the distributed database we support, MySQL Cluster, can be scaled out to 48 data nodes if needed. Hops-FS uses a combination of fine-grained locking and transactions to maintain consistency of namespace metadata while massively increasing the level of parallelism for file system operations. Finally, Hops-FS reduces the storage cost of replication by supporting Reed-Solomon erasure coding to reduce the disk space consumption by 44% compared to three-way replication of files used in Apache HDFS.

Contents

Chapter 1

A Distributed file system for BiobankCloud

Next-Generation sequencing (NGS) machines are already capable of producing over a petabyte of data per year, but the rate of growth in amount of data they can sequence is growing faster than Moore's law. In 2014, Illumina introduced the HiSeq X Ten that can produce up to 1.4 petabytes (PB) of raw genomic data per year. The typical size of the raw data for a whole human genome (fastq files) with 30x coverage is 100 GB. However, even higher coverage (60x) is required if indel mutations are to be reliably identified using current technology. In addition to this, the data stored must be stored in redundant form to ensure that it can be reliablely accessed when it is needed. For example, the 1.4 PB per year from an HiSeq X 10 would be stored as 5.2 PB in a cluster running the Apache Hadoop Distributed File system (HDFS). Many large-scale sequencing projects are starting or going to start in the next few years in Europe. In the UK, the Hundred Thousand Genomes project intends to sequence the whole genomes of 100,000 cancer patients, while in Sweden the LifeGene project intends to sequence the whole genomes of several hundred thousand individuals. The scale of the storage requirements for these projects is on the order of hundreds of petabytes. Currently, Apache's HDFS does not scale to store that amount of data, due to a bottleneck in the metadata storage component (the NameNode). However, the option of partitioning the NGS data across multiple clusters is not an attractive option, as the Hadoop platform does not support the execution of analysis jobs over multiple clusters. With larger clusters, we can support analysis over larger sample sizes, and, thus, enable greater statistical power in making inferences on the NGS data available.

This document describes the design, implementation, and evaluation of our own version of the Hadoop Distributed File, Hops-FS, where we externalized the metadata of HDFS to an open-source, in-memory, highly-available, distributed database called MySQL Cluster. Hops-FS scales to clusters that can store hundreds of petabytes. This means that data processing jobs (MapReduce, Spark, Cuneiform) can be run over datasets that are hundreds of petabytes in size. We outline many of the technical challenges we faced in ensuring the consistency and integrity of our metadata, when it is partitioned over many nodes in the database, and how we handled scalability problems that occur when a single file system operation is performed on millions of files, such as delete a subtree.

Chapter 2

Distributed file systems

In the last decade, a number of applications with an ever-growing need for data storage and processing have emerged. Some examples include web indexing, genomic data storage, and social media applications. It is predicted that, by the end of this decade, data centers storing multiple exabytes of data will not be uncommon [12, 29].

The problem of storing large amounts of data is not new. The Grid Computing and High-Performance Computing (HPC) communities haveproduced numerous distributed file systems that can store many petabytes of data. Systems like GlusterFS [8], Vesta [11], Ceph [40], Lustre [4], iRods [20], and Lazy Hybrid (LH) [9] are some of the distributed file systems that have received attention from the research and development community. However, none of these HPC file systems gained any popularity in the big data community for the following reasons:

- they are not designed for commodity hardware: they typically rely on RAID functionality that requires expensive hardware.
- they lack built-in support for data locality, which is imperative for efficient big data analytics [42]
- data processing frameworks are not designed for them - writing big data applications and setting up working environment to use these file systems is a daunting challenge.

Big data processing and analytics platforms have recently emerged as a new class of system that provide both data storage and data processing services. Apache Hadoop [14] (including Cloudera and Hortonworks distributions), Databricks cloud [2], Google Cloud Platform [17], MAPR Hadoop [5], and Microsoft Azure [24] are some of the main big data platforms. These systems can be considered as the operating systems of the data centers. Distributed file systems, like Hadoop Distributed file system (HDFS) [35], Ceph [41], and Collosus [23], are at the heart of these big data platforms.

We have chosen HDFS to store the genomic data because Hadoop has become the de facto the standard open-source platform for storing and processing enormous datasets. In the following section we will briefly introduce HDFS.

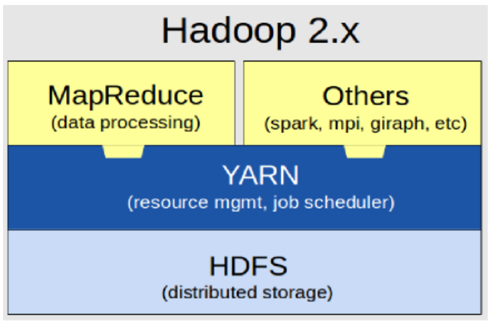

Figure 2.1: Apache Hadoop 2.X

2.1 The Hadoop Distributed File System (HDFS)

HDFS is a distributed, fault-tolerant file system designed to run on low-cost commodity hardware and to scale to store petabytes of data. It was created as an open source implementation of the Google File System (GFS) [16] and it is the file storage service in the Hadoop platform. Hadoop data processing applications use HDFS' classical storage APIs (similar to Posix APIs) as well as data locality APIs (on which node is this file block stored?). Data processing systems for Big Data began with Google's MapReduce [13], but on the Hadoop platform, they include MapReduce, Hive (database-like query support), giraph (graph processing), Spark (in-memory data processing), and a number of other YARN applications, see figure 2.1.

HDFS' architecture consists of a NameNode that stores metadata, DataNodes that store the blocks of data that make up files, there are also HDFS clients. Contrary to traditional file systems designed to run on a single node, HDFS does not store the file metadata alongside the files and directories contents (blocks). All the metadata is stored on the NameNode. The content of the files is split into small blocks which are evenly distributed among the DataNodes. The blocks are replicated (usually three times) to achieve high data availability and failure recovery. On top of storing the metadata, the NameNode manages the file system: DataNodes periodically send heartbeats to the NameNode indicating that they are alive and the NameNode takes management actions to ensure that the file system is in a safe state. For example, when the NameNode detects that a DataNode has failed, due to a number of consequetively missed heartbeats, for each block on the failed DataNode, it orders one of the surviving DataNodes that stores the block to replicate it to a different alive DataNode in the system. Clients communicate with both the NameNode and the DataNodes: in order to read a file the client first communicates with the NameNode and retrieves the metadata for the file; the metadata contains information about the DataNodes containing the blocks of the file; the client then contacts the corresponding DataNodes to retrieve the data blocks.

In HDFS v1, figure , all metadata is stored on the heap of a single JVM. Storing the metadata in memory is necessary because random access to metadata on either a magnetic disk or solid-state disk is an order of magnitude slower than reading from main memory. To avoid losing metadata in the eventuality of a NameNode restart or failure, all mutations to the metadata are persisted on disk in a log (EditLog) along with periodic snapshots (FSImage) of the NameNode state. The snapshots are required to reduce the time to recover the NameNode in the event of it failing. On NameNode recovery, it reads the latest FSImage, and then only needs to apply the mutations from the EditLog that occured since the latest FSImage was taken. Without FSImage, there could be an unbounded number of EditLog entries to apply. As all the clients must first contact the NameNode to read files, the NameNode is both a bottleneck and single point of failure (SPOF) for HDFS.

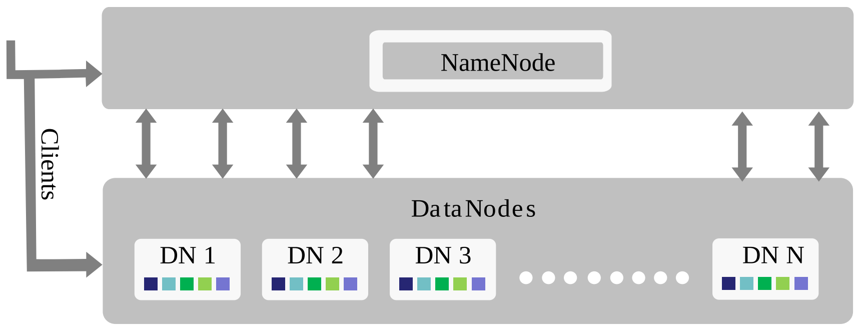

Figure 2.2: Apache Hadoop Distributed File System V1. The NameNode is a single point of failure.

In 2013, HDFS v2 introduced a highly available metadata architecture [39, 14]. The entire file system's metadata is still stored in memory on a single node [37], but changes to the metadata *edit log entries* are now replicated and persisted to a set of (at least three) Journal Nodes using a quorum-based replication algorithm. The log entries in the Journal nodes are applied to a standby NameNode that will take over as primary NameNode when the active NameNode fails and all outstanding log entries in the Journal Nodes have been applied. Figure 2.3 shows the highly available HDFS architecture, with the eventually consistent replication of the NameNode state from the Active NameNode to the Standby NameNode. As the replication protocol is eventually consistent, when the Active NameNode fails, it may take tens of seconds for the Standby NameNode to take over as Active NameNode. Also, as the standby NameNode is not used to satisfy file system operations unless the Active NameNode fails, the NameNode is still a bottleneck. Moreover, the NameNode is slower in this HA (high availability) configuration for write operations, as it must now persist metadata updates to a quorum of Journal Nodes before returning to the client. As a result, this new version improves fault tolerance, but decreases the throughput of HDFS for write operations.

Moreover, the new HA NameNode architecture requires additional services and nodes in the cluster. It needs at least three Zookeeper instances on different machines to allow all nodes in the cluster to reliably reach an agreement on which NameNode is currently active and which is passive. In large clusters, there is limited free memory on the heap of the NameNode(s), so it is not possible to efficiently create a snapshot of the NameNode's state. As such, an additional Checkpoint server is needed to periodically store a checkpoint of the NameNode's state to disk. Note that this server should have at least as much RAM as the NameNode.

2.2 Limitations of HDFS

The Hadoop platform has received tremendous attention from the research community in recent years. This is due to the fact that it is the most advanced and complete open-source ecosystem for big data analytics. However, the Hadoop Distributed File System (HDFS) [35] has received

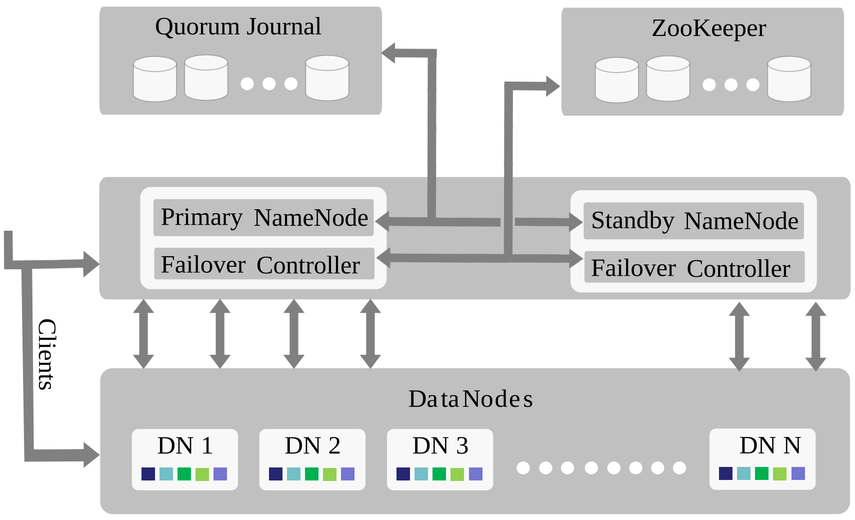

Figure 2.3: Apache Highly Available Hadoop Distributed File System V2

relatively less attention from the research community and suffers from two main problems. First, the core design of HDFS NameNode has not changed since the project was first started: a single NameNode is responsible for the maintenance of the file system metadata. Secondly, the reliability of the storage is only ensured by replication which is very storage inefficient and, in fact, has inferior high availability properties compared to erasure-coding replication in the case of multiple concurrent DataNode failures.

2.2.1 NameNode limitations

In order to reduce the software complexity of the NameNode and ensure its correctness and performance, the entire file system metadata is stored in the main memory of the NameNode. This makes it very simple to implement the file system operations, but this is not scalable and become a limitation for the platform. In fact, Hadoop clusters can grow very large and a single NameNode will have to serve thousands of DataNodes and hundreds of thousands of clients simultaneously while storing tens of gigabytes of metadata in RAM.

The problem with storing the namespace metadata in memory is two-fold. First, the namespace metadata cannot grow more than the main memory of the NameNode. Secondly, when the in-memory metadata grows to tens of gigabytes the application performance degrades due to JVM stop-the-world garbage collection events. Java garbage collectors are infamous for their poor performance for large heap sizes: the garbage collection thread consumes a significant amount of CPU resources and can pause applications for seconds at a time [37].

Moreover the namespace metadata is kept strongly consistent using *readers-writer* concurrency semantics, implemented using a global namespace lock. All the write operations are serialized, even if these operations intend to mutate different files in the different sub-trees of the namespace hierarchy. A write operation blocks all other operations; and a large write operation can potentially block the namespace for a very long time. Some operators have reported that due to sudden influx of *write*-operations and JVM stop-the-world garbage collection, the NameNode can become unresponsive for tens of minutes [22]. Due to the coarse grained NameSystem lock the NameNode doesn't scale well on multi-core hardware and the CPU is often underutilized for *write* intensive workloads [1, 3].

Storing all the namespace data on a single NameNode also makes the file system highly vulnerable. The file system becomes inaccessible when the NameNode fails. Hadoop V2 solved this problem with a hot swappable Standby NameNode that takes over as soon as it detects that the Active NameNode has failed. Yet, the time needed to propagate the logs from the main NameNode to the standby one is not negligible. From the time of detection of failure of the Active NameNode, the standby node can take tens of seconds to apply all the outstanding logs entries before serving read and write operations. Moreover, having a standby NameNode does not completely solve the single point of failure problem, the system can only tolerate a single NameNode failure. In case of failure of the Active NameNode, the standby NameNode becomes a single point of failure until the cluster administrators revive the failed Active NameNode. Although this problem can be resolved with even more redundant hardware (a standby for the standby), the costs involved and the fact the hardware does not contribute to the file system during normal operation, mean that operators typically go with just Active and Standby NameNodes.

Another problem with the Apache implementation of the NameNode is that it sometimes returns inconsistent results. Whenever the NameNode receives a write request it performs following steps:

1. acquires exclusive locks on the NameSystem,
2. performs permission checks and updates the namespace,
3. releases the Namespace lock,
4. saves the changes in the EditLog, and
5. finally, it notifies the client of the result of the operation.

As soon as the NameNode releases the namespace lock, the changes become visible to other clients and operations in the system. This can lead to inconsistent read operations. For example, consider a scenario where a client A creates a file F. The NameNode makes the changes in the namespace and releases the namespace lock. The new file is now visible to the other clients in the system and a client B can read the file. If the NameNode fails before it persisted the creation of F in the Journal Nodes, the standby NameNode will take over without any information about the creation of the new file F. If the client B tries to read the file F again will receive an error indicating that the file does not exist. Such inconsistencies arise because of the premature release of the namespace lock before mutations to the namespace have been persisted. To strengthen the NameNode's consistency semantics, the namespace locks should only be released once the changes have been persisted in the EditLog. But persisting to the EditLog is slow (tens of milliseconds) and holding the namespace lock while the file system changes are persisted would massively decrease the throughput of the NameNode. The designers surmised that weaker-than-Posix semantics are tolerable for users, given the increased performance it brings.

2.2.2 Storage limitations

HDFS is designed to be deployed on commodity hardware and achieves high data availability by replicating the data block, typically three times producing a storage overhead of 200%. However, even a replication factor of three is not high enough for huge clusters and such clusters will lose data with a high probability in case of correlated failures [10]. It has recently been shown that higher data availability can be achieved using Reed Solomon (RS) Erasure coding [34, 27]. Moreover, RS erasure coding imply a smaller overhead on the system (40%) which saves valuable disk space.

2.2.3 HDFS improvements

The two limitations presented above need to be solved. There is great need for a file system metadata management that is highly available, scales horizontally, and supports multiple concurrent write mutations. And HDFS storage reliability should not rely only on block replication. In this document we present our distribution of HDFS called Hops-FS. Hops-FS achieves horizontal scalability of metadata management by making the NameNode stateless. All the metadata is stored in a highly available, highly performant distributed database. This allows Hops-FS to have multiple stateless NameNodes which can process operations in parallel. The cluster administrator can increase or decrease the number of NameNodes depending on the workload and SLA requirements. All the clients in the system uniformly distribute their file system operations among all the NameNodes.

Hops-FS supports fine-grained *readers-writers* concurrency locking while strengthening the consistency semantics of HDFS. Unlike Apache HDFS, that locks the entire namespace to keep the namespace consistent, Hops-FS takes locks on the individual files or directories affected by each file system operation. This allows us to perform multiple file system read and write operations in parallel. Fine-grained locking and ACID transactions ensure that multiple namespace mutations do not violate the consistency of the file system. Additionally, Hops-FS supports RS Erasure coding to reduce the storage footprint and increase availability properties. See chapters (3) and (4) for architectural details of Hops-FS.

Chapter 3

Hops-FS architecture

Hops-FS replaces the Active-Standby NameNode architecture with a set of stateless NameNodes that access a distributed, in-memory, replicated database: MySQL Cluster [7]. MySQL Cluster is a real-time, ACID-compliant transactional, relational database, with no single point of failure. In scalability tests with 30 nodes, MySQL Cluster has performed more than a billion transactional updates of 100 bytes in size every minute [6]. Externalizing the metadata to a database makes the NameNodes stateless and disconnects the metadata scalability from the size of the memory heap of a single JVM. Moreover, it allows Hops-FS to run multiple NameNodes in parallel, which increases the number of concurrent metadata operations and delivers very high throughput. This chapter gives a bird's eye view of the Hops-FS architecture. Chapters 4 and 5 go into details of the design of the internal architecture and erasure-coding, respectively.

3.1 Architecture

All the changes in Hops-FS preserve the semantics of Apache-HDFS so that existing applications can run on Hops-HDFS without any change. Although the semantics of Hops-FS at least as strong as HDFS, its metadata management is fundamentally different. Fig. 3.1 shows the architecture of Hops-FS. Unlike HDFS (see figure 2.3), Hops-FS has multiple NameNodes that manage the namespace metadata. All Hops-FS' clients and DataNodes are aware of all NameNodes in the system. Whenever a NameNode fails the failed operations are automatically retried by clients and the DataNodes by forwarding the failed requests to a different live NameNode. In the remainder of this section we will present different architectural components of Hops-FS, starting from the metadata storage layer, followed by the NameNodes, the DataNodes and the clients.

3.2 Distributed metadata

All the NameNodes in Hops-FS are stateless and the namespace metadata is stored in MySQL Cluster® database. We have chosen MySQL Cluster for its high performance, high availability,

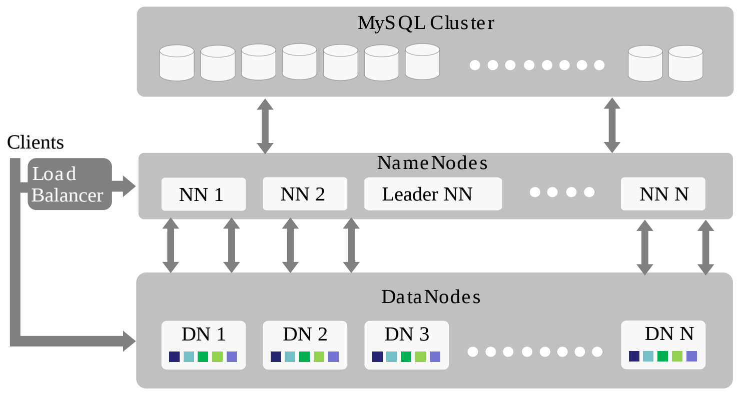

Figure 3.1: Hops-FS Architecture

real-time performance, online scale-out and high throughput [7]. MySQL Cluster provides both SQL and native NoSQL APIs for data access and manipulation. Hops-FS mostly use the NoSQL APIs because it allows us to reach a high throughput by bypassing the MySQL Server and directly communicate with the storage layer. It also allows Hops-FS to control how the data is distributed among the MySQL Cluster DataNodes. However, Hops-FS uses the SQL APIs for operations on aggregate data (for example, calculating the total number of files or corrupt blocks in the system) as the MySQL Server collects statistics from DataNodes that enable it to perform these operations efficiently.

Making the NameNode stateless turned out to be a big challenge as the NameNode contains diverse and highly optimized data structures. In order to have multiple completely independent stateless NameNodes, we needed to persist many data-structures, including inodes, blocks, blocks-locations and files-leases. Along with these data structures a lot of the secondary data structures also have to be persisted. Some examples are different replication queues; invalidated, corrupted, and excess replicas list; the lease manager state, and global variables. Figure 3.2 shows the key entities in the Hops-FS metadata model. Files and directories are represented by the *INode* entity. Each Inode contains a reference to its parent directory; which is used to construct a file system hierarchy. Each file contains multiple blocks whose information is stored in the *Block* entity. Each block is usually replicated multiple times. The location of each replica of the block is stored in a *Replica* entity. During its life-cycle a block goes through many phases. For example, when a DataNode fails some blocks will become under-replicated, such blocks are stored in the under-replicated blocks list (*URB*). The replication manager, periodically, selects some under-replicated blocks and tries to create more replicas for them. Blocks with ongoing replication are removed from the *URB* list and stored in the pending replication blocks list (*PRB*). Similarly, a replica of a block might have many states during its life-cycle. For example, if it gets corrupted will be moved to the corrupted replicas (*CR*) list. Whenever a client writes to a new block's replica, this replica is moved to the replica under construction (*RUC*) list. Replicas that are in excess are stored in the excess replicas (*ER*) list and replicas that are scheduled for deletion are stored in the invalidation (*Inv*) list.

3.3 Data partitioning

MySQL Cluster supports distributed transaction processing, where a Transaction Coordinator (TC) is located at evert DataNode in the cluster. Database transaction can be processed by any of the TCs in the cluster. When processing a transaction, a TC may communicate with TCs located on other DataNodes in order to retrieve data required for the transaction. As contacting other nodes implies costly network communications, MySQL Cluster is more efficient if all the data required by a transaction is located on the same host as the TC. We call transactions that can retrieve all their data locally *network-aware transactions*. Network-aware transactions require database tables to be partitioned by a user-defined *partition key*. If many different tables are all partitioned by the same partition key, then rows of all the tables with the same partition key value will all reside on the same DataNode. This means that if we have a transaction that will only access tables partitioned using the same partition key, all of the data will reside on the same DataNodes. If we are able to make sure that such network-aware transactions can on one of the DataNodes containing the transaction's data, then the transaction will be able to read all of its data locally, and any updates will only affect other DataNodes in the same replica group. In MySQL Cluster, it is possible to control where a transaction is started by specifying a partition key value when starting a transaction. The primary key column is the default partition key, but it is also possible to specify secondary indexes as partition keys. In the case of Hops-FS, an efficient partitioning of the files system metadata is to have all the data related to a same file (inode) placed on the same DataNode. For this purpose, we partition all the tables containing file information by the Inode_id column and start all transactions on the DataNode storing the file data. If the DataNode crashes, another DataNode in the same replica group will take over as responsible for the partition key, and future transactions will be started on the new DataNode.

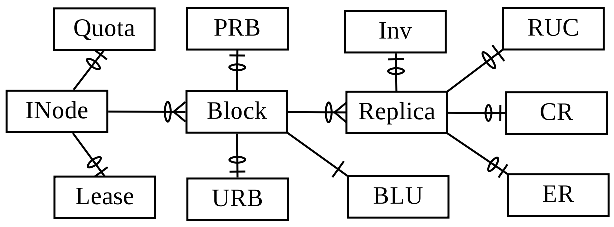

Figure 3.2: Hops-FS Architecture

3.4 Multiple NameNodes

Hops-FS supports multiple NameNodes. All NameNodes in Hops-FS are stateless and primarily provide an interface for reading and writing the namespace stored in the database (they also provide management, failover, and access control services). In Hops-FS, NameNodes can be easily added and removed depending on the load on the system. All client-facing file system operations can be performed by any of the NameNodes in the system. However, the execution of some management (housekeeping) operations must be coordinated among NameNodes. If these management operations are executed simultaneously on multiple NameNodes, they can potentially compromise the integrity of the namespace. For example, the replication manager service makes sure that all the blocks in the system have the required number of replicas. If any block becomes over-replicated, then the replication manager tries to remove replicas of the block until it reaches the required level of replication. If, however, we had several replication managers running in parallel without coordination, they might take conflicting/duplicate decisions and a block might end up being completely removed from the system. Another operation that can affect the consistency of metadata is the lease recovery operation. Only one NameNode should try to recover a file that was not properly closed by the client. We manage the coordination of management operations between NameNodes by electing a Leader NameNode that is solely responsible for management operations.

3.5 Hops-FS leader election and group membership

Apache HDFS uses ZooKeeper [21] as a coordination service. In contrast, Hops-FS uses the database for membership management and leader election, thus reducing the number of services that need to be managed and configured by our system. We implemented both leader election and group membership services using the database as strongly consistent distributed shared memory. All NameNodes periodically write to a shared table containing information about the NameNode (the NameNode's descriptor). NameNode need to successfully update their descriptor to show that they are alive in the system. While updating its descriptor, a NameNode also reads all the descriptors for other NameNodes from the database to maintain a local history of the NameNode descriptors. Using the local history, the NameNodes can easily discover dead NameNodes: the nodes that fail to update their descriptor within a configurable timeout period. A NameNode is declared leader when it has the smallest Id among all the remaining alive NameNodes. The NameNode Id is a monotonically increasing number that is persisted in the database. Whenever a new NameNode joins the system, it gets its a new id by incrementing a global id counter stored in the database. Nodes that are too slow to update their descriptor (and are hence ejected from the system) rejoin when they find out that their history is too old. The leader NameNode is also responsible for removing dead descriptors from the database.

3.6 DataNodes

The DataNodes are connected to all the NameNodes in the system. Whenever a block is modified, the DataNode notifies a random NameNode. Every hour each DataNode sends a block report to the NameNodes. These block reports can be very large: they contain information about all the blocks stored on that DataNode. A DataNode with 12TB of storage capacity may contain up to hundred thousand 128MB blocks. This means that in a cluster of 10,000 DataNodes, an average of 3 block reports will be generated every second. Processing such large number of block reports is not possible with the single NameNode of Apache-HDFS. In Hops-FS the DataNodes uniformly distributes the block reports among all the NameNodes in the system.

Each DataNode regularly sends a heartbeat message to all the NameNodes to notify them that it is still alive. The heartbeat also contain information such as the DataNode capacity, its available space, and its number of active connections and updates the list of DataNode descriptors at the NameNode. The DataNode descriptor list is used by NameNodes for future block allocations, and due to the frequency of updates to it, we decided not to persist it in the database. It is re-built on system restart using heartbeats from DataNodes. In the future, we intend to reduce the amount of heartbeat traffic by having DataNodes only send a single heartbeat message to a single NameNode.

3.7 Clients

The clients can send file system operations to any NameNode in the system. We support the following policies: *fixed*, *round robin*, and *random*. We can use these policies to load-balance client requests among all the NameNodes. If a file system operation fails because of a NameNode crash, the client removes the NameNode descriptor from its local list of NameNodes and retries the operation on a different NameNode. Clients periodically refresh their local list of alive NameNodes by querying a random NameNode. The refresh rate is configurable.

3.8 Metadata security

For performance reasons, MySQL Cluster does not support encrypted network channels between nodes in the cluster (clients, DataNodes, and Management nodes). Moreover, the data stored on the MySQL Cluster DataNodes is not encrypted - again for performance reasons. In a typical secure deployment, MySQL cluster is run on an isolated network, where all access is via applications that are also MySQL Cluster clients. The applications should support strong authentication and access control. Therefore, we recommend restricting physical access to MySQL Cluster using firewalls. Internally, MySQL Cluster nodes can use software firewalls to filter incoming network traffic that does not originate at known IP addresses/ports (the addresses of the other nodes in the cluster).

Hops-FS uses MySQL Server to perform aggregate queries on the metadata. Each NameNode has a local instance of MySQL server running on the machine. Securing MySQL server is straight forward. The MySQL server instances are configured to only accept connections from the localhost and from other NameNodes in the system.

Chapter 4

Hops-FS metadata management

This chapter focuses on different problems that arise when we externalize the metadata to a distributed database. We introduce the challenges with introducing fine-grained locking to support an increased level of parallelism for metadata mutation operations. We also describe how we maintain the consistency of the metadata when it has been partitioned over nodes in the distributed database. One technique we use to ensure consistency is to implement metadata operations using transactions. We base our approach is based on pessimistic concurrency control, where we ensure that all file metadata that will be read or written is locked with either a shared or write lock before they are operated on. We ensure freedom from deadlocks by enforcing concurrent clients to acquire locks in the same order. However, not all metadata operations can be encapsulated in a single transaction. For example, HDFS allows users to delete a directory containing millions of files. With current online transaction processing systems, it is not possible to efficiently execute such operations in a single transaction. So, how do we maintain file system consistency for operations that are not executed as single transactions? For these operations, which are all large subtree operations, we introduce our own protocol that provides the same consistency, isolation, and durability guarantees as transactions, but where operations are not atomic. Instead, our subtree operations guaranteee progress and eventual completion, even after recovery in the case of failures.

4.1 Namespace lock granularity

HDFS is a POSIX-like file system providing operations such as delete, move, rename, set permissions and set quota. File system operations are executed atomically: when file system operations return, the changes are immediately visible to all the other clients in the system. In order to keep the metadata consistent, the HDFS NameNode uses a single, global namespace lock to serialize all file system operations. The global namespace lock supports two locking modes: *shared* and *exclusive* locks. It is not possible to upgrade or downgrade the lock after its acquisition. All operations that modify the namespace, *write operations*, acquire an exclusive lock on the namespace. File system write operations include creating files and directories, delete, rename, and move. These operations are available through the file system client. However, there are many internal operations, such as metadata housekeeping operations, that also require

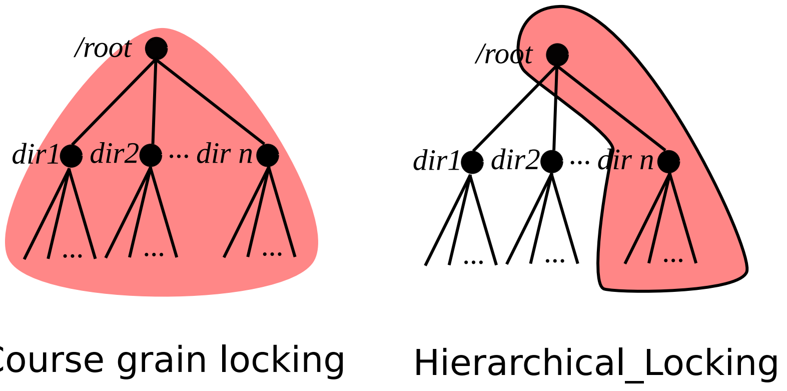

Figure 4.1: HDFS course grain locking vs hierarchical locking. In coarse grain locking, mutation operations lock the entire namespace, while hierarchical locking only locks a subpart of the tree.

exclusive access to the namespace. Similarly, operations that only read the metadata, *read operations*, such as listing a directory and reading files, acquire a shared lock on the namespace. An exclusive lock ensures that there is no other file system operation accessing the namespace in parallel. However, multiple read operations can acquire the shared lock simultaneously and read the namespace in parallel. Whenever a write operation arrives at the NameNode, it waits until all the read operations that precede it have relinquished the namespace lock. Large read operations, such as listing large directories can take tens of milliseconds and delay the write operations. HDFS delivers high throughput for workloads that have a substantially higher proportion of read operations to write operations. The performance of HDFS performance degrades when write operations become a significant share of the workload because HDFS serializes the write operations, that is, it cannot parallelize them. Figure 4.1 shows, on the left, an example where the entire namespace is locked by a write operation and other operations are put on hold until it completes. There are no fine-grained namespace locks in Apache HDFS.

4.1.1 Hierarchical locking

Coarse grain locks make it is easier to manage the consistency of the namespace, and the implementation of file system operations is also simplified. All operations in HDFS are path-based, and in an given file system operation only a subset of the path components are mutated. Most commonly, only the last component of the path is mutated. For example, the file operations *"rm /dir1/dir2/dir3/dir4/file.txt"*, *touchz /dir1/di2/file.txt"* and *"mv /dir1/dir2 /.recycle/"* only mutate the last component in the path. In our earlier work [19], we showed how we can increase the granularity of the namespace lock to allow multiple parallel operations on the namespace. Our work was inspired by Jim Gray's work on hierarchical locking [18]. We proposed to take shared lock on the path components that did not change, and exclusive locks on the file path components that needed mutations. For example, the command *hadoop fs -rm /dir1/dir2/file.txt* will remove the file *"/dir1/dir2/file.txt"* and update the timestamps on the directory *"/dir1/dir2"*. However, the directories *"/"* and *"/dir1"* do not change. This command will acquire shared locks on *"/"* , *"/dir1"* and exclusive locks on *"/dir1/dir2"* and *"/dir1/dir2/file.txt"*. Such locking mechanism frees up the root of the file system and other operations can also read/write the file system tree in parallel.

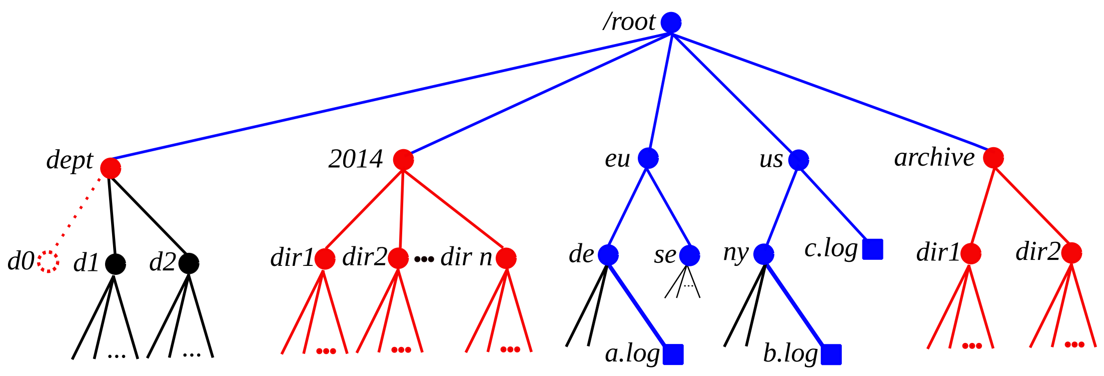

Figure 4.2: A snapshot of the namespace undergoing multiple mutations and read operations. The blue color represents shared locks and the red color represents exclusive locks. Files and directories are represented by squares and circles, respectively. File system operations that are simultaneously executing on the namespace are *"mkdir /dept/d0"*, emph"mv /2014 /archive/", *"ls /eu"*, *"cat /eu/de/a.log"*, *"/us/ny/b.log"*, and *"/us/c.log"*.

Figure 4.2 shows a snapshot of the file system namespace undergoing multiple concurrent read and write operations. In the figure shared and exclusive locks are presented by blue and red colors respectively. The namespace is running *"mkdir /dept/d0"*, *"mv -r /2014 /archive/"*, *"ls /eu"*, *"cat /eu/de/a.log"*, *"cat /us/ny/b.log"*, and *"cat /us/c.log"* operations concurrently. In large deployment, the NameNode will perform millions of operations every minute, and this scheme will lead to following problems. Firstly the distributed lock manager of the database will have to manage large number of locks, and it can easily get overloaded. In real deployments of Hadoop the file system workload primarily consist of read operations [32]; and the write operations would have to wait for long time to acquire exclusive locks on the file path components as some path component may be shared locked by large number of read operations. For example in the example about the file operations *"mkdir /tmp"* would have to wait for all operations to finish before it can acquire exclusive lock on the root to create the directory. Here exclusive lock on the root directory is required to update the time-stamp and quota values of the root directory. Similarly in the above example taking exclusive lock on *"/dept"* prevents read operations from reading files stored many level under the *"/dept"* folder.

4.1.2 Fine-grained locking

One of the advantages of hierarchical locking is that it is straight forward to implement and it automatically serializes conflicting operations. It makes it easier to maintain the consistency of metadata (and is a conceptually simple consistency model). Consider a situation where we want to run two operations *"mkdir /dept/d0"* and *"mkdir /dept/d2/dn"* in parallel on the file system tree shown in figure 4.2. Although the operations are non-conflicting, it is not possible to run both operations in parallel using hierarchical locking mechanism. If the first mkdir operation executes first then it will take exclusive lock on the *"dept"* directory which will block the second operation. The second operation will have to wait for the first operation to finish before it could execute. Similarly, if the second operation starts first then it would acquire shared lock on the *"dept"* directory and the first operation will have to wait for the previous operation to finish in order to take the exclusive lock on the *dept* directory. For many workloads where files being operation on share common parents, hierarchical locking fails to deliver very high throughput.

In our new concurrency model, we improve on hierarchical locking for typical workloads by removing the serialization bottleneck introduced by taking shared locks on ancestor directories. Our concurrency model increases the parallelism for common read/write file operations by eliminating the need of taking shared locks on the ancestor directories. The download of our solution is that for file system operations on subtrees of inodes, we now have to acquire a potentially huge number of locks compared to hierarchical locking. We will explain our solution with the help of some examples. Continuing the previous example, the operation *"mkdir /dept/d0"*, will acquire an exclusive lock on *"dept"* and no locks on *"/"* - it will read the last committed value for the *"/"* directory. The second operation *"mkdir /dept/d2/dn"* will take an exclusive lock on *"d2"* and no locks will be acquired on the *"dept"* and *"/"* directories (again, we read the last committed value for those directories). WThis way both the operations can execute in parallel and the consistency of the namespace is not violated. Figure 4.3 shows how our fine-grain locking solution locks the files and directories. Compared to hierarchical locking mechanism, our fine-grained mechanism allows more concurrent file operations for workloads with shared ancestor directories (such as in MapReduce jobs).

Our new concurrency control model introduces new potential inconsistencies for operations on subtrees of inodes that were not possible in the hierarchical locking model. For example, consider two operations *"mv /dept /dept_new"* and *"mkdir /dept/d2/dn"* simultaneously executing using the file system tree presented in figure 4.3. Assume the rename operation runs first. The rename operation will read the root directory without any lock and acquire exclusive lock on the *dept* directory. Assume after acquiring locks the operation is preempted. Now the second operation executes. The *mkdir* operation will read the */* and *dept* inodes without any locks (read committed) and it will acquire an exclusive lock on *d2*. Now the second operation is preempted and the first operation resumes. The first operation renames the directory and commits the new values in the database. Now if the *mkdir* operation continues it will create a new directory in a non-existant ancestor directory *"dept"*. Relaxing the locks on the ancestor directories causes the *mkdir* operation to corrupt the namespace and return incorrect results to the client.

However, removing shared locks on ancestor directories is imperative for achieving high operational parallelism and throughput. However, doing so lets conflicting operations run simultaneously on same subset of inodes. This problem can be solved by carefully analyzing all the operations supported by the HDFS client and identifying operations that operate explicitly or implicitly on subtrees of inodes, as these can potentially lead to inconsistent responses or corruption of the namespace. Such operations must be handled differently to operations that affect only a single inode. These operations include *move*, *rename*, *delete*, *file listing*, and *set quota* operations. These operations cannot execute in parallel with other operations on the same subset of inodes. We call these operations as *subtree operations* because these operations affect all the descendants of an inode manipulated by these operations. For example, move and rename operations will change the absolute paths of all the descendant inodes, while the delete operation will remove all the descendant inodes, and the set quota operation will affect how all the descendant inodes consume disk space. Similarly, the directory listing operation *ls* assumes that the contents of the directory do not change while user output is being generated.

One potential solution would be to take exclusive locks on all the descendants of the inode that is being manipulated by a subtree operation. This will ensure that no other operation can access the subtree while the subtree operation is ongoing. However, a subtree may have millions of descendants and this solution many not work online transaction processing systems, if we

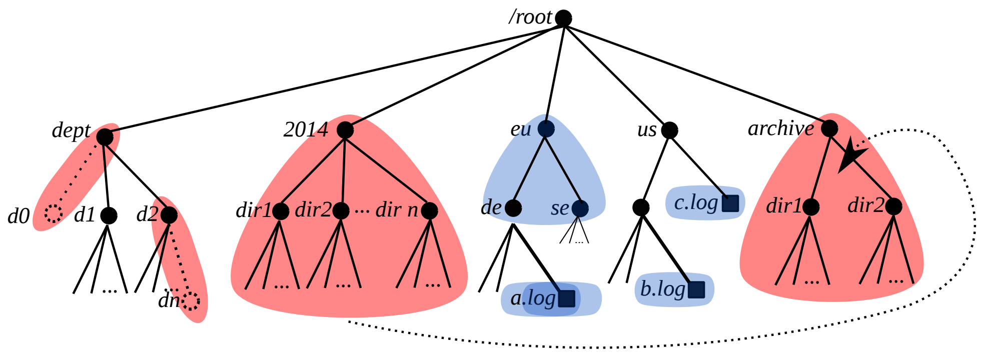

Figure 4.3: A snapshot of the namespace undergoing multiple mutations and read operations. The blue color represents shared locks and exclusive locks are represented by the red color. File and directories are represented by squares and circles respectively. File system operations that are simultaneously executing on the namespace are *"mkdir /dept/d0"*, *"mkdir /dept/d2/dn"*, *"mv /2014 /archive/"*, *"ls /eu"*, *"cat /eu/de/a.log"*, *"/us/ny/b.log"*, and *"/us/c.log"*. The file */eu/de/a.log* is read by two clients concurrently.

assume that each subtree operation is encapsulated in a single transaction. This isn't feasible in current in-memory online transaction processing (OLTP) systems, as the large number of locks cannot be acquired before practical TransactionTimeout limits being exceeded. Secondly, large transactions consume significant amounts of the memory, and transactions on tens of millions of inodes require all those inodes to be kept in memory, which again may not be feasible for in-memory OLTP systems. Lastly, acquiring large numbers of locks affects the performance of the distributed lock manager of the database management system which will result in poor performance for concurrent operations. In the following section, we introduce a new method for locking and isolating the entire subtree without relying on the distributed lock manger of the database management system.

4.1.3 Subtree operations

As we cannot implement large subtree operations as single transactions, we have developed our own protocol that provides the same consistency, isolation, and durability guarantees as transactions, but where operations are not atomic. In our protocol, subtree operations do not execute all-at-once, but rather as a series of batched operations, where each batch makes progress towards completing the subtree operation and when all batches have been executed, the subtree operation will have completed. Other concurrent clients are not affected by ongoing large subtree operations, and we maintain the same isolation semantics as Apache HDFS for the subtree operations.

Our large subtree operations are implemented as a sequence of transactions containing batches of operations. We maintain an invariant of *conservation-of-information*, so that progress will be made even in the event of failure of NameNode executing operation. As locks in OLTP systems have bounded timeouts, we cannot use them to isolate the subtree, so we introduce a flag based mechanism to mark and lock the subtree. Figure 4.4 shows the different steps in executing large subtree operations.

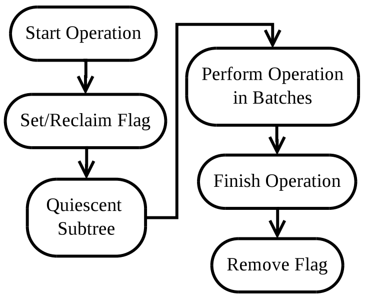

Figure 4.4: Sub tree operations

- In the first phase, an exclusive lock is acquired on the root of the subtree and a flag is set. The flag is an indication that all the descendants of the subtree are locked. The flag also contains the ID of the NameNode that owns the lock. In Hops-FS, there is a lock acquisition phase when resolving filename paths, and if an inode is encountered along the path whose flag is set, then the operation aborts and waits until the flag is unset by the operation that holds the flag. The client waiting for the lock periodically checks that the NameNode that holds the lock is still alive. If it detects that the NameNode that holds the lock is dead, then it reclaims the flag and continues with its operation. All subtree operations are designed in such a way that they leave the subtree in a consistent state if the NameNode fails during the operation.
- The NameNode reads the entire subtree from the database into memory using exclusive locks to make sure that all ongoing operations in the subtree have completed. The subtree is read in batches using multiple transactions and database locks are promptly released after executing each transaction. In order to reduce the memory footprint of the subtree in the NameNode, the in-memory representation of the subtree only contains inode ids and permission attributes. We reduce the time taken to read up the subtree by executing many transactions in parallel, up to a thread-pool size. Each transaction resolves the path components of a directory and can be executed in its own thread as a network-aware transaction, since all the data for the transaction will reside on the same DataNode in the database. That is, all inodes in the directory share the same partition key, which is the inode-id of their parent directory.
- Once the subtree operation has finished the flag is removed from the root of the subtree and client is notified about the result of the operation.

NameNode failure

NameNode failures can happen during the execution of a subtree operation either through failures or if the NameNode is declared dead by the membership management protocol. If the NameNode executing the subtree operation is declared dead, then the flag can be reclaimed by any of the remaining NameNodes. However, when the NameNode is very slow and declared dead, it may still believe it is alive and continue executing its subtree operation. In this case, there is the potential for multiple conflicting operations on the same subtree if another NameNode reclaims the flag. To solve this problem the slow NameNode terminates as soon as it detects that it has been evicted from the group membership management protocol. The terminated NameNode reboots and comes back with a different NameNode ID. In Hops-FS the NameNode IDs are never reused, see section 3.5. We also have to handle an edge case, where a slow NameNode continues its subtree operation before it checks the group membership management protocol. We handle this by maintaining a local lease time for the group membership management protocol, and if this lease times out, the NameNode needs to execute the group membership management protocol before it can execute another transaction on the database. When a NameNode wants to reclaim a flag it first waits for one a membership table update period to ensure that the NameNode that was declared dead has halted its operations.

In the following sections we present our solution for the different subtree operations in HDFS and justify why these operations do not leave the file system in an inconsistent state, even in the event of failures.

Delete operation

Once an exclusive lock is acquired on the root of the subtree, a flag is set to mark that the subtree is undergoing a mutation. The subtree is then read into memory with exclusive locks in batches to ensure that all ongoing activity in the subtree has completed. The NameNode then recursively deletes the leaves of the subtree until it reaches the root of the subtree, again in batches. The subtree is partially deleted if the NameNode fails before it completes the delete operation. This is acceptable outcome as even in any POSIX compliant file system, where the directory contents are partially deleted if the system fails during an ongoing recursive delete operation. In case of a failure, all the inodes that were not deleted remain in a consistent reachable state in the file system tree and the client can resubmit the delete request to a different NameNode to delete the remaining inodes.

Move and rename operations

Move and rename operations only mutate the root of the subtree, but they invalidate the path of all inodes in the subtree. Before the mutation is applied the whole tree has to be read to make sure that ongoing operations on inodes in the subtree have completed and that the client has access privileges for all the inodes in the subtree, and finally that quota restrictions will not be violated. The subtree is read after setting the flag on the subtree. If all conditions are met then the root of the subtree is mutated and the flag is removed in the same transaction. As move and rename operations only mutate filesystem state in the final (single) transaction, NameNode failure does not affect filesystem consistency. The flag can be reset by other NameNodes when they access the inode with the flag set.

Set quota and permissions operations

Set quota and permissions have similar semantics to the move operation. After setting the lock flag the entire subtree is read to make sure that ongoing operations on inodes in the subtree have completed and to check for access privileges and quota violations. The actual mutation is applied to the root of the subtree in the last transaction. This operation leaves the file system in a consistent state in case of NameNode failure, again as the only mutation to the filesystem state occurs in a single transaction.

File listing and content summary operations

These are read only operations. The subtree is locked to make sure that content of the subtree does not change while the user output is generated for operation.

4.2 Using transactions to maintain file system consistency in Hops-FS

All non-subtree file system operations in Hops-FS are encapsulated in and executed as a database transaction. In this section we will discuss the transaction isolation level required to ensure file system consistency.

4.2.1 Transaction isolation

Hops-FS requires serializable transactions to mutate the namespace and maintain its consistency. However, many distributed databases demonstrate very poor performance when they serialize all updates to shared state [38]. In addition to this, many distributed databases do not support serializable transactions because they result in low throughput.

Most OLTP systems support transactions with READ_COMMITTED isolation level, but stronger isolation levels are needed to maintain the consistency of the namespace. Hops-FS uses row-level locking to ensure stronger isolation guarantees on top of a READ_COMMITTED isoloation level. Our design goal with our concurrency model is to ensure that different transactions are fully isolated but we would like to support more concurrent operations on the namespace for non-interfering transactions. Building stronger isolation guarantees using row-level locks has another added advantage as most distributed databases support row-level locking and READ_COMMITTED isolation levels for transactions, including our default database MySQL Cluster. The limited assumptions we place on the database transaction processing system, support for at least READ_COMMITTED isolation level and row-level locking, should enable us to easily replace the underlying distributed database in future, if needed.

In Hops-FS, all NameNode operations that manipulate the namespace are executed as transactions. Each transaction takes row-level locks on the affected rows in the tables for inodes and other entities (see section 3.2 for Hops-FS' entities) to isolate transactions from other concurrent operations in the system. The problem with this approach is that the implementation of HDFS file system operations read and write from the entity tables in different orders, which will lead to deadlocks in the system if two or more transactions acquire locks to resources that result in a locking cycle. For example, consider two different clients that simultaneously submit the same rename operation, "mv /a/b/f1.log /a/b/f2.log". If the file *f2.log* exists then it will be deleted and the *f1.log* will be renamed to *f2.log*. In this case, the rename operation should lock both files to isolate them from the other operations in the system. Assume that the first rename operation first locks /a/b/f1.log and then tries to lock /a/b/f2.log. However, before the first operation could acquire the second lock, the second rename operation has already locked /a/b/f2.log and is now trying to acquire the lock for /a/b/f1.log. This locking cycle causes deadlock in the system, which is typically identified and fixed by the OLTP system using timeouts.

We solve the deadlock problem by exploiting the tree structure of the namespace, which is inherently a Directed Acyclic Graph (DAG). All operations in Hops-FS acquire locks on inodes in a total order, traversing the DAG in an agreed order based on depth-first search. The depth-first search traverses first towards the leftmost child and terminates at the rightmost child. Continuing the previous example with our depth-first search traversal, both rename operations would take locks in the same order. First, /a/b/f1.log is locked and then /a/b/f2.log is locked. Thus, deadlocks are avoided through all nodes acquiring locks in the same order, and no locking cycles on resources will happen because clients acquire locks on resources based on a total order defined using the file system's DAG.

4.2.2 Transaction cache

On performance issue, we addressed in Hops-FS was to cache partial transaction data at NameNodes. A file system operation might access the same metadata multiple times within a transaction before it is committed. For example, while moving a file from one folder to another the operations might traverse all the paths components multiple times to make sure that all the paths are valid, then it will check for user permissions, disk space availability, and the operation might modify multiple objects. Hops-FS stores all the objects in a distributed database and a naive implementation would access file path component multiple times from the database resulting in in multiple network round-trips. In order to avoid multiple network round-trips while reading same data, each transaction maintains a local per-transaction cache. The cache is used by the transaction first filling the cache and then the transaction reads and writes to the cache before the modified objects in the cache are committed along with the transaction. This approach led us to dividing each transaction into two distinct phases. In first phase, all the data that would be accessed by the operation is locked and read from the database in the total order based on the file system's DAG and depth-first search traversal. The retrieved data is stored as a cache in the main memory of the NameNode executing the transaction. In the second phase, the actual file system operation is executed, which reads and writes in an arbitrary order to the in-memory cache. Finally, when the transaction is committed all the modifications to the cache are sent to the database for persistent storage.

4.2.3 Transaction handling

The NameNode implements a thread-per-request policy for handling client requests. Requests are received as protocol buffer messages that are deserialized into Java objects, and then put into multiple queues, as shown in the figure 4.5. The NameNode has a pool of worker threads that, when they become available, pick up and process a request object from a message queue. In a typical configuration, the number of worker threads and the number of queues is the same, and the default number of workers is set to 10 (in Apache HDFS). After processing the a client's request, a response is returned to the client. All the RPC calls in HDFS are blocking (synchronous). If the NameNode becomes overloaded, more requests arrive at the NameNode than it can process, resulting in queue overflow and client requests timing out.

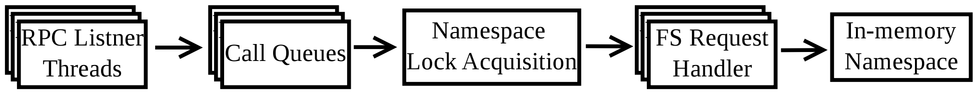

Figure 4.5: HDFS RPC request handling system.

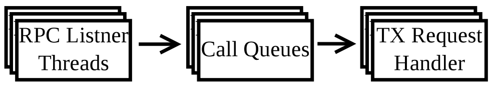

Figure 4.6: Hops-FS transaction processing system

Hops-FS has same code base as HDFS. We have designed a transactional processing system that encapsulates all the NameNode operations in a transaction. An advantage of this approach is that it does not require too many changes in the original NameNode's source code and later it will be much easier to migrate to new releases of HDFS. Figure 4.6 shows an overview of how Hops-FS handles the client requests. Note there is no global file system lock and each worker thread runs transactional operations to manipulate the metadata stored in the database. The code listing below shows how HDFS creates a new file. For clarity, the code for exception handling, journaling and other sanity checks has been removed.

```
private void startFile(String src, PermissionStatus permissions, String holder ... ) {
   FSPermissionChecker pc = getPermissionChecker();
   writeLock();
   try {
     checkOperation(OperationCategory.WRITE);
     startFileInternal(pc, src, permissions, holder ...);
   }
   ...
   finally {
     writeUnlock();
   }

   code to persist changes to EditLog
  }
```

First, the NameNode checks if the user has permissions to create the file, and the operation has acquired exclusive locks for the namesystem. After acquiring an exclusive lock the NameNode creates the file and the namesystem lock is released. The changes are then saved to a log file called the *EditLog*.

The code listing below show how Hops-FS performs the same operation in a transaction. All the code shown before is wrapped in an inner class *HDFSTransactionalRequestHandler*. The code for exception handling and sanity checks has been removed for readability. Three methods need to be overridden: setup, acquireLock and performTask function. HDFSTransactionalRequestHandler object controls the behavior of the transaction. Figure 4.7 shows some of key stages of the life-cycle of a transaction.

```
void startFile(final String src, final PermissionStatus permissions, final String holder ...
    ) ... {
     HDFSTransactionalRequestHandler startFileHandler = new
         HDFSTransactionalRequestHandler(HDFSOperationType.START_FILE, src) {
```

```
        @Override
        public TransactionLocks acquireLock() ... {
          HDFSTransactionLockAcquirer tla = new HDFSTransactionLockAcquirer();
          tla.getLocks().addINode(..., path).
        addBlock().
        addReplica().
        ...
          return tla.acquire();
        }

        @Override
        public Object performTask() throws PersistanceException, IOException {
            FSPermissionChecker pc = getPermissionChecker();
       try {
    startFileInternal(pc, src, permissions, holder ...);
       }
       ...
       return null;
        }

        @Override
        public void setUp() throws PersistanceException, IOException {
        }
    };
    startFileHanlder.handle(this);
 }
```

In Hops-FS, all client's operations are *path* based, that is, all the clients provide the full path of the files and directories that they are manipulate. However, there are many internal metadata operations that do not have access to the complete file paths, such as all the internal block operations that start with the block_id. For all of these operations, we use the setup phase to resolve the inode_id, given the block_id. After executing the setup phase, the transaction handler will automatically start the transaction (as it will now have the full path of all files and directories for the operation). After starting the transaction all the data that is needed for the operation is locked, read and stored in the in-memory cache at the NameNode. During the lock acquisition phase, the *acquireLock* method is called, which contains information about what entities are required by this operation and what the strength of locks for the entities should be. Hops-FS' total order locking system will only lock the entities identified in the *acquireLock* method. After populating the cache, the method *performTask*, which contains the operation-specific code, is called. The operation-specific code reads the data from the in-memory cache and all the modifications are also stored in the cache. When the *performTask* method returns the transaction manager extracts all the modifications from the cache and persists them in the database, committing the transaction. Note that the code in the *performTask* method is essentially the same as the code listing shown for HDFS' file create operation.

The class *EntityManager*, see figure 4.7, is used by all the transactions to read and write metadata. *EntityManager* provides static methods for metadata manipulation and it encapsulates all the cached objects for the transactions. Whenever a read/write request is received the *EntityManager* finds the corresponding object(s) in the transaction's cache. If the requested data is not available in the cache then the data is read from the database and a copy of the data is also stored in the cache. The *EntityManager* maintains an instance of the cache per transaction. When a transaction is committed or aborted, the *EntityManager* invalidates its cache.

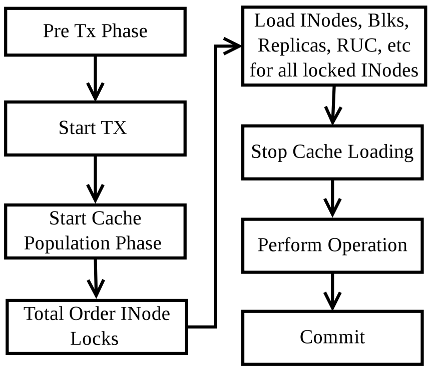
Pre Tx Phase
Start TX
Start Cache Population Phase
Total Order INode Locks
Load INodes, Blks, Replicas, RUC, etc for all locked INodes
Stop Cache Loading
Perform Operation
Commit

Figure 4.7: Hops-FS Transaction Request Hander

4.3 Distributed ID generation

A general requirement for Hops-FS is that all database records must be uniquely identified for their efficient data retrieval and manipulation (using primary key operations). However, distributed databases do not always provide an efficient means to generate unique keys at runtime, as this requires data nodes reaching agreement on which ids have been allocated and which ids are available. Agreement protocols (such as Paxos or 2-phase commit) could be used to allocate ids, but many application developers would prefer not to pay the cost of running an agreement protocol for something as simple as generating a unique id. In HDFS and Hops-FS, all inodes and blocks require unique ids. In Apache HDFS' NameNode this is a trivial problem, as there is a single namespace, and the NameNode can just generate a random number and perform a hash table lookup to check for a collision. In recent releases of the HDFS NameNode, a sequential id generator is now used to generate unique IDs. In Hops-FS, all NameNodes are stateless and it would be an overly expensive operation to check for the existence of the ID in the database. For performance reasons, it is infeasible to have a central sequence number generator, which would become a scalability bottleneck. Instead, in Hops-FS, each NameNode maintains a batch of preallocated IDs. An ID pool manager thread periodically replenishes the ID pool. Whenever the ID pool manager needs to replenish its pool of IDs it takes lock on a global counter for IDs, reads its current value (M) and increments it by N. Doing so, it pre-allocates a batch of IDs [M, M+N] for the requesting NameNode.

4.4 SafeMode

Apache HDFS provides a state for the file system called SafeMode that can be set whenever the system could potentially become corrupt if new reqeuests are allowed to execute on it. More specifically, SafeMode is a NameNode state in which the NameNode is read-only: it only allows read requests and throws an exception for any requests that try to modify the file system state. In the current version of HDFS, the NameNode enters SafeMode either manually upon an administrator's request or automatically during NameNode restart or due to low disk space that would prevent the NameNode from creating a new snapshot. To get out of SafeMode the NameNode loads its latest snapshot of the file system state (FSImage) from the disk and waits until it receives a block report from each of the DataNodes present in the system. These block reports account for the available blocks on each of the DataNodes and allow the NamNode to build a complete view of the cluster state. After exiting SafeMode (the SafeMode state), the NameNode uses the snapshot of the NameNode state from FSImage to recover missing and corrupted blocks.

In contrast, in Hops-FS, NameNodes are stateless and the file system metadata is persisted to the database. As a result, the NameNodes enter SafeMode only upon admin request and when the whole system restarts. As metadata is stored in the database, snapshots are no longer needed and when a unique NameNode restarts it can automatically access the metadata from the database. If the system enters SafeMode, each NameNode must read the last view of the cluster state from the database and wait until it receives block reports from each of the NameNodes. This is inefficient. In fact, (i) reading from the database is slower than reading a snapshot from a contiguous region on disk and (ii) sending block reports to each NameNode is redundant. In Hops-FS, these problems are solved by having the NameNodes collaborate when exiting SafeMode and recovering missing and corrupted blocks.

We now go describe our solution starting with the block reports phase and ending with the block recovery phase.

4.4.1 Processing block reports

In order to improve the block report processing phase, we use the fact that the cluster state is stored in the database to have each DataNode send only one block report and to have the NameNode process these reports in parallel. As shown in figure 4.8, each DataNode sends its block report to one and only one NameNode, and this NameNode then processes the report and writes the id of each correct block to a table in the database. This way it is possible to know which blocks have at least one correct replica by reading from the database. The only limitation with this solution is that it is not possible to enforce that each correct block should have more than one correct replica in order to allow the system to exit SafeMode. A trivial solution would be to add a column storing a counter to the table, but this would imply taking locks for each reported block and that would be inefficient. Finding a better solution is future work.

In order to optimize the block report processing phase, Hops-FS ensures that the reports are evenly distributed among all the NameNodes. When a DataNode sends a block report, it starts by asking the leader NameNode for a NameNode to which to send its block report. The leader then uses an internal policy to respond with a target NameNode for block report processing. The leader policy to select the target NameNode are currently either random or round robin. More advanced policies, taking in account the load of the NameNodes, is future work.

When a NameNode receives a block report it must compare the information contained in this report with the information stored in the database. The size of the block report depends on the number of blocks stored on the DataNode sending the report. On existing HDFS systems, this can be as much as a few million blocks. As a result it would be very inefficient to have a single database transaction per block present in the report. In fact, each transaction implies at least one round trip on the network and network latency would become a bottleneck. We solve this problem by processing the report in batches of blocks. The size of the batch depends on the database maximum batch size, but is typically in the thousands of blocks.

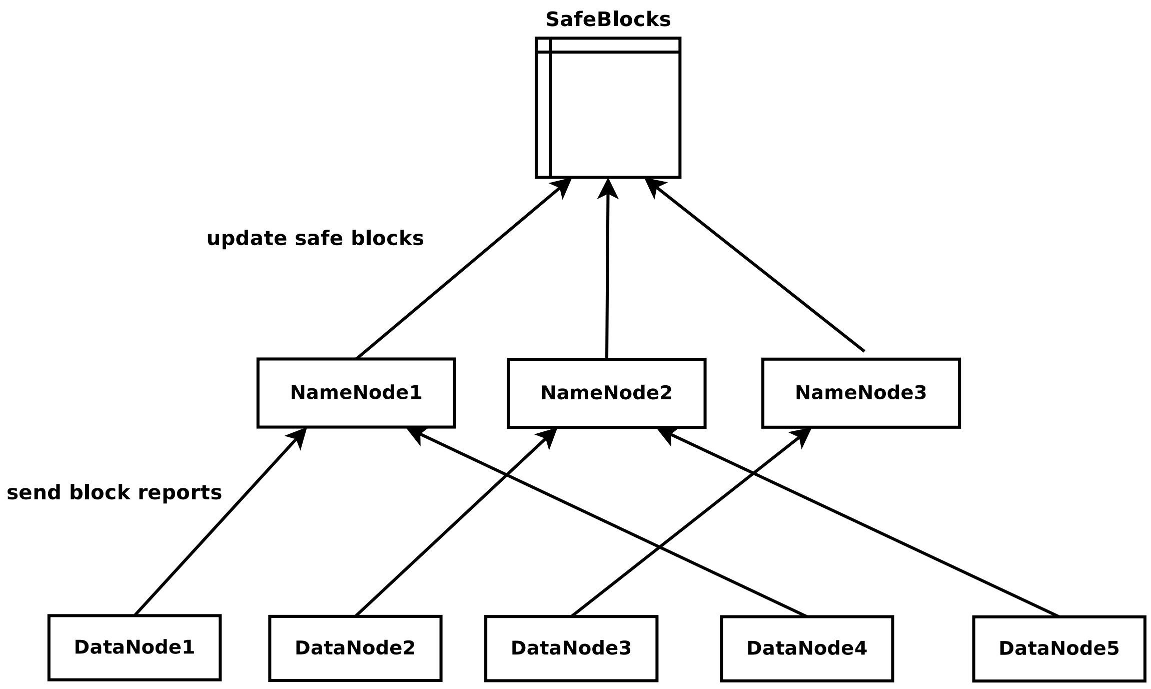

Figure 4.8: NameNodes update the safe blocks to get out of SafeMode in a cluster consisting of 3 NameNodes and 5 DataNodes.

4.4.2 Process missing and corrupted blocks

After exiting SafeMode, the NameNodes start to process all the files present in the database, in order to recover missing, corrupted, over-replicated and under-replicated blocks. In order to speed up the process, Hops-FS the NameNodes process files in parallel. Each file should be processed only once the NameNodes have to reach an agreement on which files each NameNode should process. To implement this scheme, Hops-FS use a distributed id generator 4.3: each time a NameNode wants to process a file it increases the value of the id generator and processes the file corresponding to the new id value. As it would be inefficient to repeat this process for each individual file, the NameNodes increase the id value S by X and process all the files with an id contained in the range $[S, S + X[$, with X a configurable parameter for the system.

As a NameNode can fail after acquiring a range of files to process and before finishing to process them, we need to ensure that these files will eventually be processed. For this purpose, once a NameNode has obtained a range of file ids to process, in the same transaction it writes this range in a table before starting the processing of associated files. The range is removed from the table once it has been processed. This way, when a NameNode is detected as dead, another NameNode can load the range of files to process from the table and process them. The full process of processing files is represented in figure 4.9.

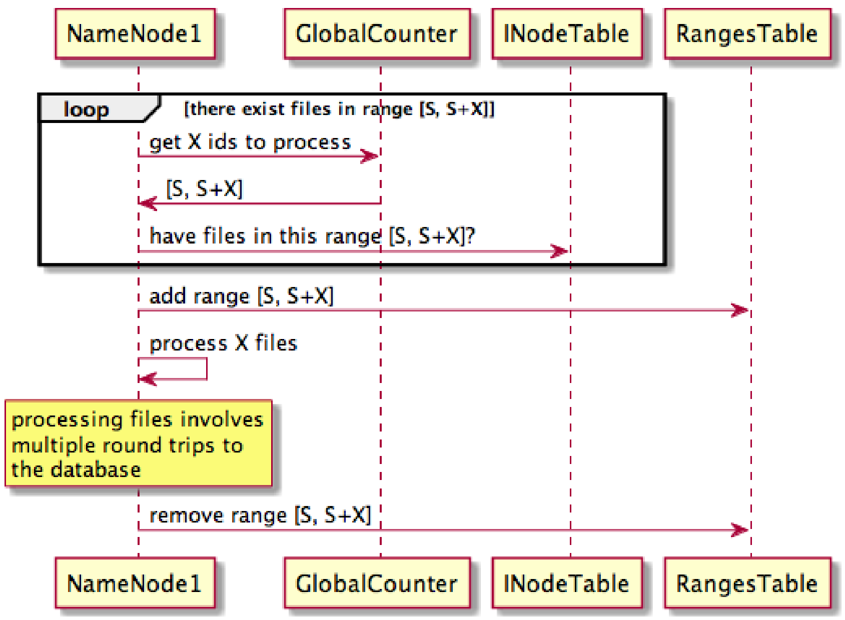

Figure 4.9: Simplified sequence diagram of NameNode1 contributing to the processing of all files.

Chapter 5

Hops-FS data storage

5.1 Erasure coding

HDFS is capable of handling petabytes of data spread over thousands of nodes, while maintaining availability if the face of node and even rack failures. The high level of fault tolerance is achieved by replicating data across nodes and racks. The default replication strategy is to replicate a chunk of data onto three different nodes located on two different racks. While this offers great resilience to failure, it also introduces large storage overhead. For instance, storing a file of 100GB requires 300GB of disk space. The total storage capacity, as available to users, is hence divided by three. A widely used approach solving this issue for non-distributed file systems, is the usage of so called Redundant Arrays of Inexpensive Disks (RAID). Advanced RAID systems divide files into stripes consisting of a predefined number of blocks. On each stripe, special mathematical functions are applied in order to generate parity blocks. The applied functions have the characteristic to allow the recreation of the original data without the presence of all blocks. In order to allow a specific number of disks to fail, while still being able to recompute the original data, the blocks of each stripe and the according parity blocks are spread over multiple disks. Using this concept, the redundancy storage overhead introduced by the parity blocks is smaller than a full replication of the data. The underlying concept of RAID is known as erasure coding.

For Hops-FS, we have transferred the concept of RAID onto HDFS in order to decrease the storage requirements for data stored in our file system. This part of our functionality is called Hops-EC and is based on a project called HDFS-RAID [15] which was released as open source and was included in previous releases of the official Apache distribution of Hadoop before being dropped for maintenance reasons. In comparison to HDFS-RAID, which adds an additional management node to the cluster, Hops-EC manages erasure-coded files directly in the NameNode, enabling Hops-EC to provide guaranteed block placement, a richer API, as early as possible detection of failures and prioritization of repair for critical files.

5.2 Goals

We identified the limitations of existing erasure-coding solutions for HDFS, and used them to define a set of goals for Hops-EC:

- add erasure coding to Hops to reduce storage requirements for genomic data;
- offer a flexible API allowing the encoding of individual files, at any time and with arbitrary codecs, as well as the revocation of the encoding;
- allow the implementation of custom strategies for identifying the files to encode and triggering the encoding;
- evaluate failure detection knowledge as fast as possible and prioritize the repair of critical files by consiering the number of erased blocks;
- enforce the block placement of encoded files to maximize file availability even in small clusters;
- provide a mechanism for transparent source file block repair on the client-side;
- automatically, atomically, and transparently remove the parity information during file deletion;
- enable the addition of new erasure codes;
- allow the combination of erasure codes and arbitrary replication factors (for example, two replicas plus erasure coding for a file).

5.3 Storage savings

When applying erasure coding, the replication factor of a file can be reduced without increasing the risk of data loss or sacrificing availability. A commonly applied codec is the combination of a $(14, 10)$-Reed-Solomon [31] code, which splits files into stripes of 10 blocks and creates four parity blocks for each of these stripes, with a replication factor of one. The storage overhead is hence decreased from the 200% required for triplication to 40%, see figure 5.1. 40% is an optimal value, meaning that it can be higher if partial stripes with less blocks than the given stripe length exist. For instance, using the given code for the encoding of a file with only one block would lead to a storage overhead of 400%. However, Hops-EC will prevent the encoding of files which lead to a higher storage overhead. Additionally, the code, including its stripe length and the number of parity blocks per stripe, can be freely configured on a per-file basis and an appropriate strategy can hence easily be chosen for smaller files. One of the advantages of Hops-EC for genomic data, is that BAM and fastq file sizes are typically in order of many gigabytes, and these files will have at least the 10 blocks needed for Reed-Solomon encoding.

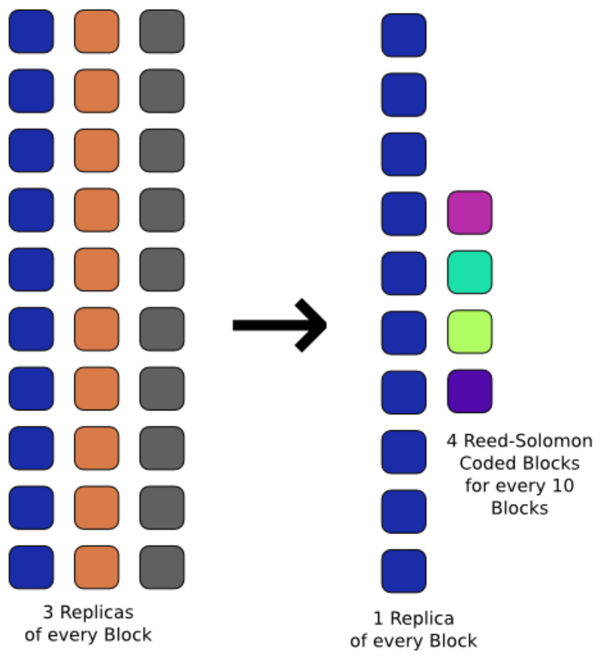
3 Replicas of every Block
1 Replica of every Block
4 Reed-Solomon Coded Blocks for every 10 Blocks

Figure 5.1: Storage savings through the application of erasure coding.

5.4 Architecture

The Hops-EC architecture, see figure 5.2, is divided into two main components, the ErasureCodingManager in the NameNode and an erasure coding library which is used by the ErasureCodingManager and implements the actual encoding logic. Additionally, a client-side ErasureCodingFileSystem interface is included which provides the transparent repair of erasure-coded files to client applications.

The ErasureCodingManager forms the core of Hops-EC. It is implemented as an active object (encapsulated in a single thread) on the leader NameNode. It is responsible for scheduling encodings and repairs, monitoring the revocation process of encoded files and executing the garbage collection of parity files. Its state is persisted in the database together with the NameNode state. Additionally, the ErasureCodingManager defines interfaces to be implemented by erasure codes or the library offering the actual encoding functionality.

In addition to the ErasureCodingManager, block management in the NameNode was extended to report missing and corrupted as well as recovered blocks to the ErasureCodingManger, which, therefore, is always notified of the latest state of the blocks.

The library includes an EncodingManager and a RepairManager which are used by the ErasureCodingManager to trigger and monitor encodings as well as repairs. It also includes the implementations of several erasure codes together with the encoding, decoding and repair logic. Computationally expensive tasks such as encodings and repairs are not executed on the NameNode but on the cluster in form of MapReduce jobs.

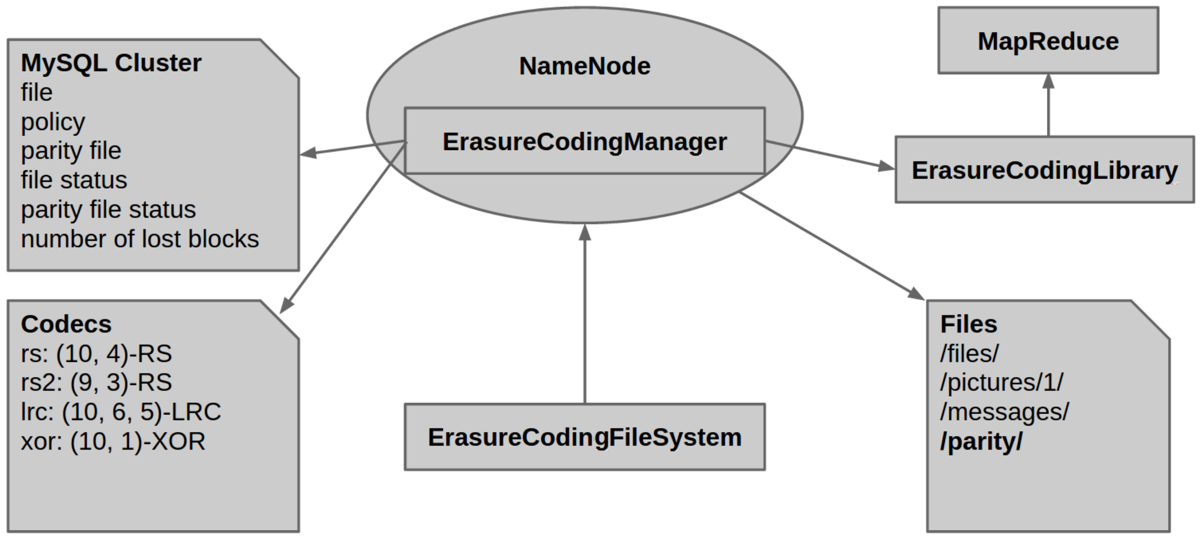

Figure 5.2: Hops-EC overview.

5.5 Supported erasure codes

The current implementation supports XOR (RAID 5) [28], Reed-Solomon [31] and Locally Repairable Codes [33]. The codes can be freely configured by specifying the source stripe length and the amount of parity blocks as well as the replication factor to be applied after encoding. New codes can be supported by implementing the provided interfaces and are automatically loaded during runtime if specified in a codec configuration.

The configuration of codes is done in a single configuration file. The code class and the requested code length, such as $(14, 10)$, in addition to the desired folder to store parity information, need to be given and named. The specific code configuration can then be used with the name given in this configuration, after restarting the NameNode. New codes simply need to be added to the class path in order to be used by this configuration.

5.6 Erasure coding API

Hops-EC provides an easy to use client API for erasure coding. For convenience to clients of the HDFS file system API, we embedded our erasure coding API in the same DistributedFileSystem class that implements the HDFS file system API. The client API was implemented in form of additional RPC calls to the NameNode. The method signatures of the API are shown below. Clients can request the encoding either during file creation or on an existing file. Either way, the encoding process will be executed asynchronously as soon as it is scheduled (normally shortly thereafter). The replication factor to be applied after the file was successfully encoded can be specified together with the erasure code to be used in a so-called policy. Additionally, a custom replication factor to be applied before the encoding can be set.

The encoding of files can be revoked at any time using the revoke method. When doing so, the replication factor to be applied before the parity information is deleted can be specified. The deletion of encoded files is done using the regular delete method of the HDFS file system API.

The simple but powerful API of Hops-EC enables client applications to apply their own encoding strategies. For instance, an archiving approach, which triggers the encoding of cold files, as in HDFS-RAID, can be implemented. Alternatively, an administration terminal could be built on top of it, allowing administrators to select files or folders to be encoded.

```
public HdfsDataOutputStream create(Path f, boolean overwrite,
    short replication, EncodingPolicy policy);
public void encodeFile(String filePath, EncodingPolicy policy);
public void revokeEncoding(String filePath, short replication);
public boolean delete(Path f, boolean recursive);
```

5.7 Prioritization of repairs

For each file, the number of lost source blocks, the number of lost parity blocks and the time when the first loss of a source respective parity block was detected, is stored. The repair of source files is now prioritized in the following order, leading to a prioritized repair of critical files.

1. Total number of lost blocks (descending)
2. Number of lost source file blocks (descending)
3. The detection time of the first block loss (ascending)

Parity file repairs are scheduled separately and are prioritized similarly.

5.8 Block placement

In order to guarantee reliability for encoded files, the placement of source and parity blocks needs to be considered. As discussed earlier, files to be encoded are divided into stripes and parity blocks are created for each of these stripes. The blocks of each stripe and the related parity blocks now need to be placed on different nodes, so that at most one of them is lost if a single node fails. Similarly, the block placement policy should enable a cluster to survive a rack failure by having files always store its blocks on at least 2 different racks.

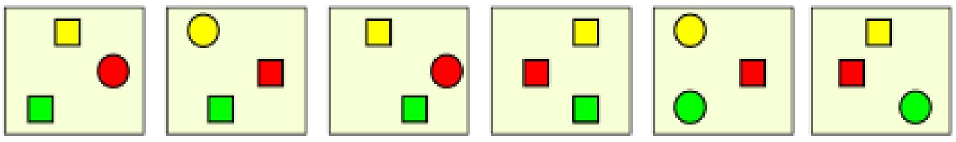

Figure 5.3: Block placement example for a $(6, 4)$ encoded file.

Whenever the encoding of a file is requested, Hops-EC enforces proper placement of stripe blocks by excluding those nodes that already store blocks for that particular stripe. Additionally the correct placement of repaired blocks is enforced. An example placement of an $(6, 4)$ encoded file with three stripes is shown in figure 5.3. Squares denote source blocks whereas circles denote parity blocks. Stripes are distinguishable by colour. A placement following this schema ensures that the maximal possible number of node failures can be tolerated. If the encoding is requested after the file has already been written, then the blocks for file may be migrated to enforce the Hops-EC block-placement constraints.

Chapter 6

Evaluation

We now evaluate the performance of Hops-FS. We first investigate the performances of the NameNode and then evaluate the Erasure Coding.

6.1 Throughput

In order to stress test the NameNodes and evaluate their read and write throughput, we designed a custom benchmark. Standard benchmarks, such as NNThroughputBenchmark, are designed to test the performances of a single NameNode and are not suited to evaluate the performance of a multi-NameNode system. Another limitation with NNThroughputBenchmark is that it does not include the cost of network communication between the clients and the NameNode, and it is also limited in that the whole benchmark runs on a single JVM. NNThroughputBenchmark also cannot balance load among different NameNodes.

We designed our benchmark to determine the throughput of read and write operations on a cluster containing several NameNodes. The benchmarks runs multiple clients distributed across multiples hosts. The clients are remotely controlled by a central operator that starts and stops all of them at roughly the same time. To measure the write throughput of the NameNodes, the clients send create file requests to the NameNodes, where the requests are load balanced across the NameNodes and the clients send requests in a loop in order to create as many files as possible. We then aggregate the number of files created by each client to obtain the number of files created per minutes. We proceed in the same way for read-operations.

Thanks to this benchmark we measured the performances of Hops-FS according to the number of NameNodes in the system. All the experiments were performed on 18 bare-metal machines running Ubuntu 12.04 containing 40GB of RAM, a six-core AMD Opteron(tm) Processor (2435), and the hosts are connected by a 1-gigabit ethernet switch with no network bonded interfaces. As the experiments were designed to stress the NameNode and not to test the DataNodes, we only needed three DataNodes for all experiments.

Setup for Apache HDFS: the number of NameNodes was set to two (one active and one standby NameNode). Three Quorum Journal Nodes were used for storing the EditLog, and

18,000 clients distributed on 18 hosts were used to send file operations requests to the NameNode.

Setup for Hops-FS: We varied the number of NameNodes from one to four. The metadata was stored in a deployment of MySQL Cluster containing 6 Database DataNodes, and 18,000 clients distributed on 18 hosts were used to send file operations requests to the NameNode.

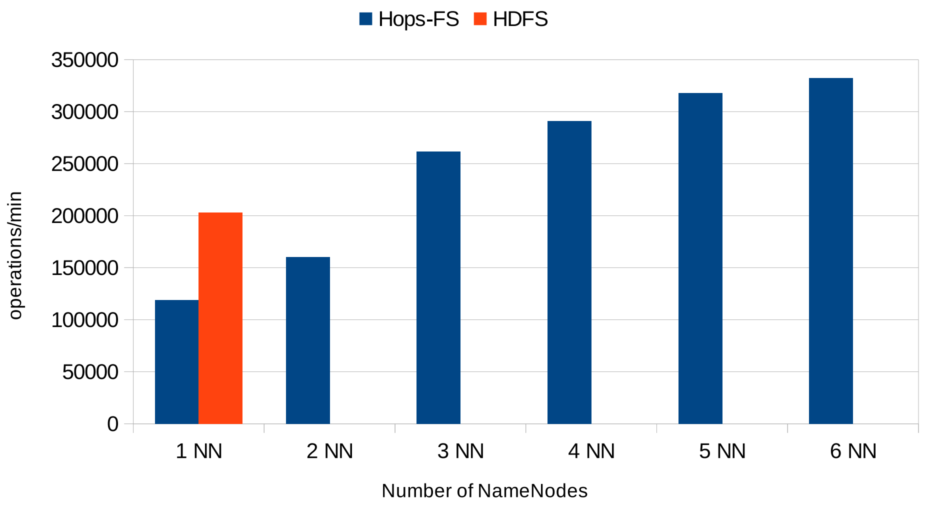

Figure 6.1: Hops-FS File Creation Performance.

Figure 6.1 shows the results of this benchmark when evaluating the write throughput of the system. We can see that with only three NameNodes Hops-FS can create 200,000 files/minute, which is more files per minutes than Apache HDFS. However, contrary to what was expected, the evaluation shows that our solution does not continue to scale linearly when increasing the number of NameNodes. We have identified the source of the problem and we are currently working on fixing it. The problem is located in the ClusterJ API of MySQL Cluster. In MySQL cluster each DataNode acts as a transaction coordinator so that the transaction can be uniformly distributed among all the DataNodes. However, the present ClusterJ API sends all the transactions to the same transaction coordinator, which results in the coordinator becoming a bottleneck for our system. We have submitted a bug report [25] to Oracle and are looking into the ClusterJ source code to fix the problem by ourselves. We are quite confident that once this problem will be solved, the throughput of our system will improve. The reading operations suffer from the same problem and shows similar limitations as the writing operations, see figure 6.2.

6.2 Namespace overhead and scalability

Storing the metadata in a database causes expansion in the amount of memory required to store the data. We call the amount of extra memory required the *memory expansion factor*. The extra memory is used to store primary and secondary indexes as welll as pad-out both rows and columns. In this section we evaluate the memory expansion factor and compare the scalability of the namespace of Hops-FS and HDFS in terms of number of files they can manage.

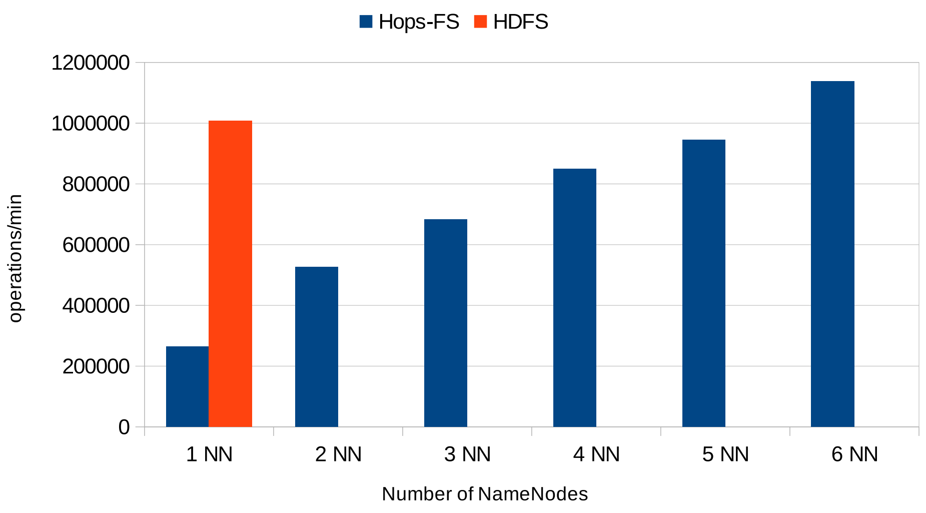

Figure 6.2: Hops-FS Read Performance.

6.2.1 HDFS namespace usage

As discussed in Chapter 2.2, HDFS keeps all the metadata state in the RAM of the NameNode. In order to allows the NameNode to store as many metadata as possible the developers of HDFS have put a huge effort in designing data structures that reduce the memory footprint of the metadata[37, 36]. We will now evaluate the size of each of these data structures.

The vast majority of the NameNode's memory is used by *INodes* and *BlockInfos* (cf:Figure 6.3). The memory footprint of each of these entities has been estimated by the past[36], but these estimations are outdated.

In order to conduct this evaluation we assume a 64-bit JVM. We then use the known size of Java primary types and the size of the main Java basic data structures overheads, listed in Table 6.1, to compute the size of each of the NameNode data structure classes. The results are presented in table 6.2, with L the length of the file name and R the replication factor of the blocks.

Object Overhead	16 bytes
Array Overhead	24 bytes
Object Reference	8 bytes
ArrayList Overhead	56 bytes

Table 6.1: Fixed memory overheads on 64 bit JVM.

We can now evaluate the amount of memory required to stored the metadata for a file. In production environments, Yahoo observed that each file contains on average 1.5 blocks[37] with a replication factor of 3. Armed with this information, we took a number of assumptions to enable us to build a simple model that estimates the memory consumption of each file: each file has a name of length 20; each file contains two blocks and the replication factor is set to 3. Given these assumption each file in HDFS will consume $((128 + 20) + 2 * (88 + 24 * 3)) = 468$bytes. Se we can store metadata for up to 2.1 million (2136752) files in 1GB of main memory.

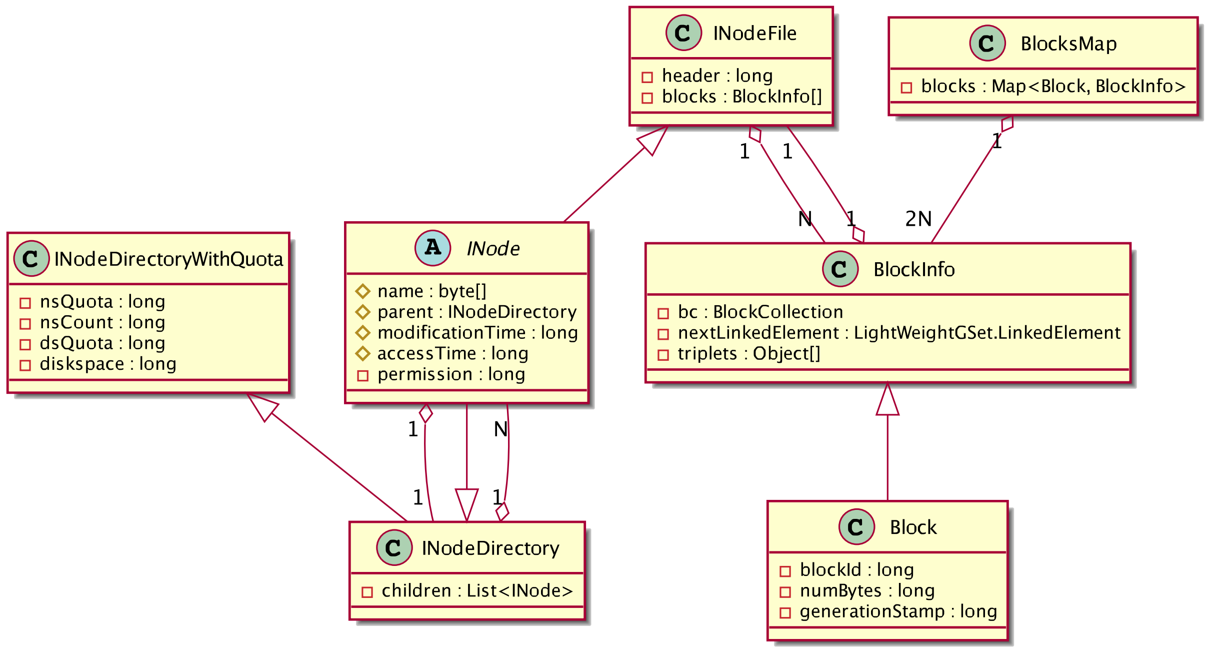

Figure 6.3: Class Diagram of HDFS main classes.

Object	Used memory in bytes
INode	$64 + L$
INodeFile	$128 + L$
INodeDirectory	$144 + L$
INodeDirectoryWithQuota	$176 + L$
Block	32
BlockInfo	$88 + 24 \times R$

Table 6.2: Memory requirements for the main data structures (objects) in HDFS.

6.2.2 Hops-FS namespace scalability

We will now estimate the memory consumption of the same metadata in a database. Storing metadata in a database incurs some additional costs. These additional costs come from normalizing the stored entities and storing more information for efficient data retrieval: primary keys, partition keys and indexes. Additionally in MySQL cluster all rows are 4 bytes aligned, that is, if a row is 21 bytes long then it will actually consume 24 bytes in the memory. In order to evaluate this over cost we need to know that in MySQL Cluster each index use 16 bytes per record and that the overhead of a variable sized columns (to store tables) varies from 4 to 12 bytes. In order to calculate the size of each entity we use a tool called *sizer*. Sizer accurately measures the memory usage of the entities stored in MySQL Cluster and is used to evaluate large scale cluster deployments [26].

Like HDFS, the entities that consume the most memory are INodes, Blocks, Replicas and Block_Lookup. INode entity contains five variable length columns. The columns *client_machine*, *client_node* and *client_name* stores information about the client that is writing data to the file. Once the file is closed these variables are emptied. Similarly, *name* and *symlink* store up to 3000 byte long file names. However, in typical production systems, filenames are short [32].

Column Name	Type	Size
under_construction	bit(1)	*
client_node	varchar(100)	104
parent_id	int(11)	4
dir	bit(1)	*
subtree_lock_owner	bigint(20)	8
symlink	varchar(3000)	3012
inode_id	int(11)	4
generation_stamp	int(11)	4
modification_time	bigint(20)	8
client_machine	varchar(100)	104
access_time	bigint(20)	8
name	varchar(3000)	3012
permission	Varbinary(128)	132
quota_enabled	bit(1)	*
subtree_locked	bit(1)	*
client_name	varchar(100)	104
header	bigint(20)	8
primary key index	hash index	16
inode_idx	btree index	16
Row Size		6512
Row Size with overhead		6586
* all bit fields are compacted into a 4 byte word		

Table 6.3: INode table memory consumption.

Table 6.3 shows that a row in an INode table, including all the overhead costs, can take up to 6586 bytes when the file is open. However, when the file is closed, the variable columns, *client_machine*, *client_node* and *client_name*, will be empty. As a result, assuming that the filename is 20 characters long, each INode row will require 306 bytes of memory, see equation 6.1.

$$\begin{aligned} 6586 - 3000(symlink) - 2980(filename) - 100(client\_name) \\ -100(client\_node) - 100(client\_machine) = 306bytes \end{aligned} \tag{6.1}$$

A row in the Block table takes 114 bytes (see table 6.4). Many internal operations of Hops-FS contain only information about the block_id. In order to efficiently determine to which file the block belongs we maintain an inverse block lookup table. This block lookup table has one row for each block in the system. The cost of storing one row in the block lookup table is 66 bytes (see table 6.5). Similarly each replica of a block uses 90 bytes (see table 6.6).

To conclude, in Hops-FS a file with two blocks and a replication factor of 3 will take 1206 bytes in memory, (see eq. 6.2). MySQL Cluster replicates the data to provide high availability of the stored data. By default MySQL Cluster replicates the data twice; which means a file in Hops-FS will actually take 2412 bytes. Hops-FS can store 0.4 million (414593) files in 1GB of metadata

Column Name	**Type**	**Size**
time_stamp	bigint(20)	8
primary_node_index	int(11)	4
num_bytes	bigint(20)	8
block_recovery_id	bigint(20)	8
block_index	int(11)	4
inode_id	int(11)	4
block_id	bigint(20)	8
block_under_construction_state	int(11)	4
generation_stamp	bigint(20)	8
primary key index	hash index	16
Row Size		72
Row Size with overhead		114

Table 6.4: Block table memory consumption.

Column Name	**Type**	**Size**
inode_id	int(11)	4
block_id	bigint(20)	8
primary key index	hash index	16
Row Size		28
Row Size with overhead		66

Table 6.5: Block lookup table memory consumption.

memory.

$$\begin{aligned} &306(INode) + 2 * 114(Blocks) \\ &+2 * 66(Lookup\_Table) + 6 * 90(Replicas) \\ &= 1206 bytes \end{aligned} \tag{6.2}$$

Hops-FS uses 514% of the memory used by the active HDFFS NameNode to store the metadata for each file. However, for a highly available (HA) deployment with an active, standby, and checkpoint NameNode,the expansion factor becomes only 171%. Nevertheless, HDFS' namespace scalability is limited by the size of JVM heap of the NameNode. Yahoo has one of the biggest Hadoop clusters in the world, containing 4000 DataNodes. In this large cluster, the NameNode's JVM uses 100 GB of heap space. Yahoo's Hadoop cluster cannot scale to larger sizes because increasing the size of the JVM heap beyond 100 GB is not practical, due to stop-the-world garbage collection events. At such scales the JVM pauses are very significant and the throughput of the NameNode can drop dramatically on JVM pauses. With current JVM technology, it is reasonable to conclude that HDFS's namespace cannot scale that much beyond a couple of hundred gigabytes. Meanwhile, Hops-FS has no such limitations. MySQL Cluster supports up to 48 DataNodes, which allows it to scale up to 12 TB of data in a cluster with 256 GB RAM on each DataNode. As shown in figure6.4), Apache-HDFS is limited to 213.6 million files on a 100 GB JVM, while Hops-FS can store up to 4.9 billion files using 12 TB of data (see figure 6.4), where the metadata has two copies (replicas) for high availability. Finally, the metadata overhead in Hops-FS also has utility for system operators, as the data tables contain

Column Name	Type	Size
storage_id	int(11)	4
replica_index	int(11)	4
inode_id	int(11)	4
block_id	bigint(20)	8
primary key index	hash index	16
storage_idx	btree index	16
Row Size		52
Row Size with overhead		90

Table 6.6: Replica table memory consumption.

a large number of indexes that support arbitrary queries on the metadata using SQL - a feature not supported by HDFS. If users don't wish to perform online queries on MySQL cluster, the cluster data can be replicated to a MySQL server, where offline analytics queries can be run.

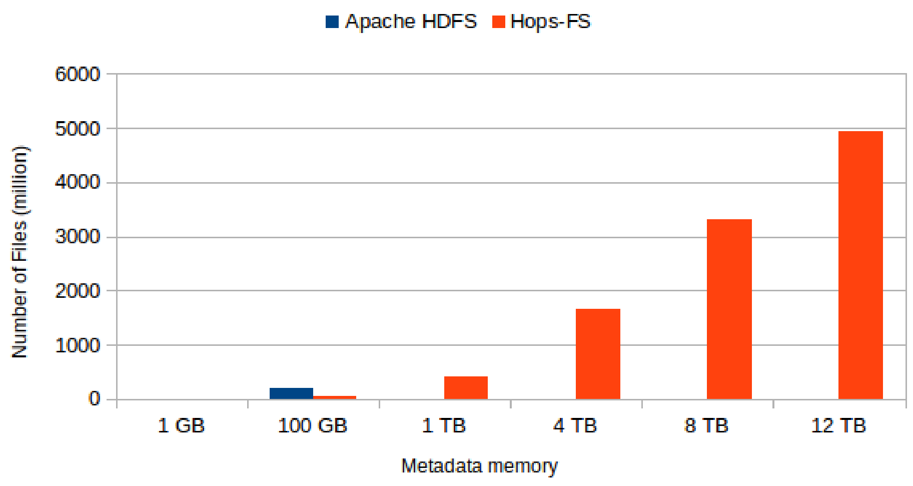

Figure 6.4: HDFS and Hops-FS namespace scalability. HDFS' scalability is limited by the fact that the JVM cannot handle much more than 100 GB, while Hops-FS can support up to 5 billion files.

6.3 Erasure coding

6.3.1 Resilience to failures

Two experiments were executed in order to evaluate the reliability of Hops-EC and to compare it to standard replication (or block triplication, as we call it). In each experiment, 100 random randomly sized files with 10 to 400 blocks with 4MB of random data were stored on 18 DataNodes, before failing n DataNodes. Failures were simulated by shutting down DataNodes.

The first experiment evaluated the effect of node failures when using block triplication and the default block placement policy. Blocks were spread evenly though nodes, by removing the writing node after file creation and waiting for the cluster to achieve triplication again. Figure 6.5 shows the results of the experiment. Three node failures are enough to corrupt all replicas of some blocks for 33% of the files, making the file unavailable. Having four or five node failures a majority of files becomes unavailable. This drastic result is most certainly caused by the small size of the cluster, in combination with a large number of blocks per file. Thus, many blocks of a file are stored on each node and hence not many failures can be tolerated.

The second experiment encoded the same files with a $(10, 6, 5)$-LRC code and applied a replication factor of one, before introducing the same node failures as in the first experiment. As the results in figure 6.6 show, up to four node failures can be tolerated without any file corruption. This is achieved by the block placement and the encoding of Hops-EC. With five nodes failing, the failure rate is similar to the triplication scenario. Thus, the reliability of Hops-EC is superior to triplication.

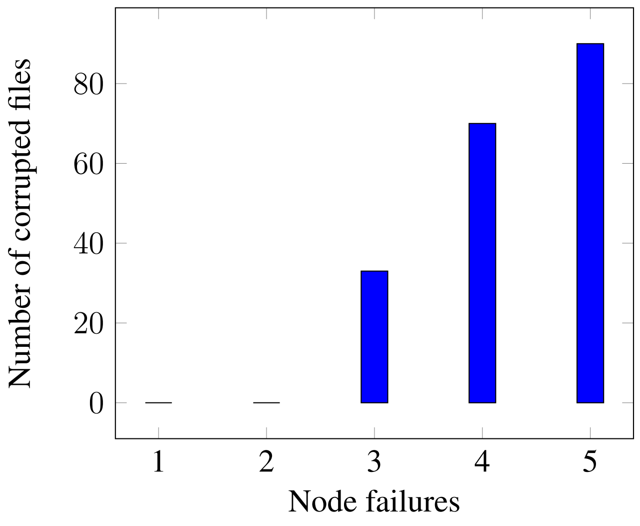

Figure 6.5: Triplication: Corrupted files per failed node.

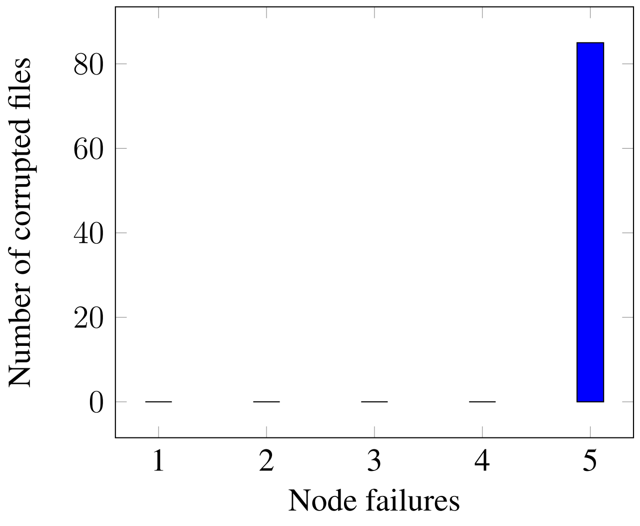

Figure 6.6: Hops-EC: Corrupted files per failed node.

6.3.2 Effect of block placement

In order to evaluate the benefits of placing the blocks of each stripe and its parity blocks onto different DataNodes, the experiments of creating 100 files with a random size of 10 to 400 blocks, with 4MB of data each, were repeated. Instead of failing nodes, blocks were randomly deleted in order to simulate failures without enforced block placement. The same cluster consisting of 18 DataNodes was used and the files were created with the same seed as in the experiment with failing nodes, resulting in the same files. Having 18 DataNodes, one node stores roughly 5% of all blocks. Hence, instead of failing one node, 5% of all blocks are randomly deleted.

The results from the experiment are illustrated in figure 6.7. While a block loss of 5% can still be tolerated in this case, a 10% loss already leads to a significant amount of corrupted files. Block loss factors of 15% and 20%, respectively, is disastrous and causes corruption of a majority of files. Comparing this result to the one benefiting from the block placement strategy, shown in figure 6.6, it is obvious how much a robust block placement strategy improves reliability. With the placement strategy of Hops-EC, not a single block would have been lost, even with 20% of lost blocks. The block placement strategy is, therefore, a very efficient way of increasing resilience to failures.

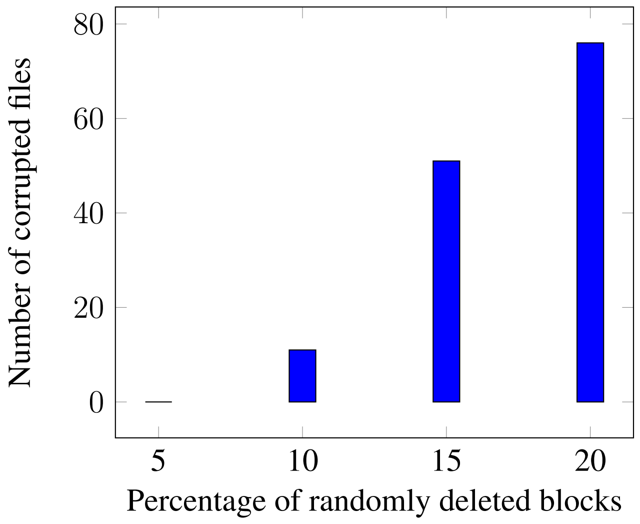

Figure 6.7: Hops-EC: Corrupted files during random block loss

6.3.3 Availability and degraded reads

As described earlier, Hops-EC ensures erasure-coded files are repaired transparently during reads, repairing lost blocks while blocking the client until the repair has completed. To evaluate the performance of degraded reads an experiment was conducted comparing the read performance of a healthy file to a scenario with one missing block and a scenario with one missing node. A file size of 10GB and a block size of 64MB was used, while reading the file with a single client. A $(10, 6, 5)$-LRC code was used for file encoding. The experiment was executed on the cluster of 18 nodes. Figure 6.8 shows the results of the experiment.

For one lost block, the read duration increases by 30 seconds from 1.5 to 2 minutes. Considering that more than 98% of failures in large distributed file systems are single block failures [30], this should not have a big influence on the expected read performance. Looking at the scenario in which a full node failed, the read time increases significantly. This is due to the fact that the cluster consisted of only 18 nodes and the failed node stored a sizeable percentage of the total set of blocks in the system. Also, when the recovery file system detects a block failure, it will retry to read the block multiple times, after waiting for a few seconds, before starting the reconstruction process. Hence for an increasing number of lost blocks, the wait time also increases linearly. Considering a realistic cluster size of several hundred nodes, a single node failure should not cause many losses in a single file and the degraded read performance should be comparable to the single block failure experiment. Consequently, having a transparent repair mechanism is an effective way of ensuring file availability after failures.

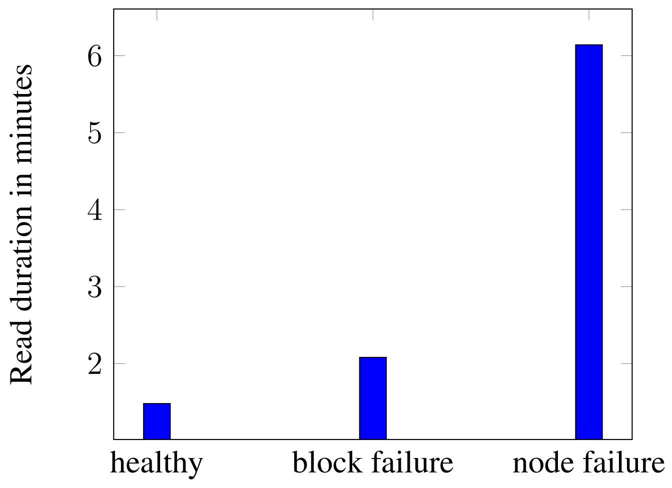

Figure 6.8: Read duration of a 10GB file

6.3.4 Comparison to HDFS-RAID

Table 6.7 shows a summarized feature comparison of Hops-EC and HDFS-RAID. It can be seen that Hops-EC continues to support most of the features of HDFS-RAID while adding support for important features such as a flexible API, enforced block placement and prioritized repairs. The most important feature that is currently not supported by Hops-EC is appending to encoded files. However, it will most certainly be supported in the near future.

Feature	**Hops-EC**	**HDFS-RAID**
Flexible API / Support for custom strategies	✓	✗
Detection of failures as early as possible	✓	✗
Low overhead to maintain the state	✓	✗
Prioritized repairs	✓	partially
Enforced block placement	✓	✗
Support for Hadoop 2	✓	✗
Transparent repairs	✓	✓
Configurable and extensible codecs	✓	✓
Grouped encoding	✗	✓
Support for append	✗	✓
HAR support for parity files	n/a	✓

Table 6.7: Comparison of Hops-EC and HDFS-RAID

Chapter 7

How to configure Hops-FS

Hops-FS is based on Apache Hadoop distribution. We have tried to make sure that Hops-FS does not break any semantics of HDSF, so that all the existing application and systems using HDFS can easily migrate to Hops-FS. Hops-FS supports most of the configuration parameters defined for HDFS [http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml]. In this chapter we will only point out the changes in the configuration parameters related to Hops-FS.

7.1 Unsupported configuration parameters

We have replaced HDFS 2.x's Primary-Secondary Replication model with shared atomic transactional memory. This means that we no longer use the parameters in HDFS that are based on the (eventually consistent) replication of *edit log entries* from the Primary NameNode to the Secondary NameNode using a set of quorum-based replication servers. Here are the parameters that are not used in the Hops-FS.

- **dfs.namenode.secondary.*:** None of the secondary NameNode attributes are used.
- **dfs.namenode.checkpoint.*:** None of the checkpoint attributes are used.
- **dfs.image.*:** None of the FSImage attributes are used.
- **dfs.journalnode.*:** None of the hadoop's journaling attributes are used.
- **dfs.ha.*:** None of the hadoop high availability attributes are used.
- **dfs.namenode.num.extra.edits.*:** None of the edit logs attributes are used.
- **dfs.namenode.name.dir.*:** FSImage is not supported anymore.
- **dfs.namenode.edits.*:** None of the edit log attributes are used.

7.2 Hops-FS configuration parameters

Hops-FS supports multiple stateless NameNodes. All of the NameNodes can mutate the namespace. Hops-FS' multi NameNode environment introduces a few new configuration parameters which are defined in the following subsection.

7.2.1 NameNode configuration parameters

- **dfs.quota.enabled**
  Quota can be en/disabled. By default quota is enabled.
- **dfs.leader.check.interval**
  The length of the period in seconds on which NameNodes run the leader election protocol. One of the active NameNodes is chosen as a leader to perform housekeeping operations. All NameNodes periodically update a counter in the database to mark that they are active. All NameNodes also periodically check for changes in the membership of the NameNodes. By default the period is to one second. Increasing the time interval would lead to slow failure detection.
- **dfs.leader.missed.hb**
  This property specifies when a NameNode is declared dead. By default a NameNode is declared dead if it misses a HeartBeat. Higher values of this property would lead to slower failure detection.
- **dfs.block.pool.id**
  Due to shared state among the NameNodes, Hops-FS only support one block pool. Set this property to set a custom value for block pool. Default block pood id is HOP_BLOCK_POOL_123.
- **dfs.name.space.id**
  Due to shared state among NameNodes, Hops-FS only support one name space. Set this property to set a custom value for name space. Default name space id is 911 :)
- **dfs.ndb.setpartitionkey.enabled**
  Partition hints can be used to start transactions on a specific MySQL datanodes. If this parameters is set to false then the transactions will start on random MySQL Cluster datanodes. For performance reasons it is better to start the transactions on the datanodes that hold the data for the transaction.

Memcached configuration

Each NameNode caches the metadata in memcached for later use. Note this is not same as transaction cache described in the section 4.2.3. Memcached entries have longer life than the individual transaction caches.

- **dfs.memcached.enabled**
  Enables/Disables the memcached for the NameNode.

- **dfs.memcached.server.address**
  Memcached server address.

- **dfs.memcached.connectionpool.size**
  Number of connections to the memcached server.

- **dfs.memcached.key.expiry**
  It determines when the memcached entries expire. The default value is 0, that is, the entries never expire. Whenever the NameNode encounters an entry that is no longer valid, it updates it.

Quota manager configuration parameters

- **dfs.namenode.quota.update.interval**
  In order to boost the performance and increase the parallelism of metadata operations the quota updates are applied asynchronously. The quota update manager applies the outstanding quota updates after every *dfs.namenode.quota.update.interval* milliseconds.

- **dfs.namenode.quota.update.limit**
  The maximum number of outstanding quota updates that are applied in each round.

Distributed unique ID generator configuration

ClusterJ APIs do not support any means to auto generate primary keys. Unique key generation is left to the application. Each NameNode has an ID generation daemon. ID generator keeps pools of pre-allocated id. For more detail see section 4.3. The ID generation daemon keeps track of IDs for inodes, blocks and quota entities.

- **Batch Sizes**
  When the ID generator is about to run out of the IDs it pre-fetches a batch of new IDs. The batch size is specified by these parameters.

  - **dfs.namenode.quota.update.id.batchsize** Prefetch batch size for Quota Updates. As there are lot of quota updates in the system the default value is set to 100,000.
  - **dfs.namenode.inodeid.batchsize** Prefetch batch size for inode IDs.
  - **dfs.namenode.blockid.batchsize** Prefetch batch size for block IDs.

  - **Update Threshold**
    These parameters define when the ID generator should pre-fetch new batch of IDs. Values for these parameter are defined as percentages i.e. 0.5 means prefetch new batch of IDs if 50% of the IDs have been consumed by the NameNode.
  - **dfs.namenode.quota.update.updateThreshold** Threshold value for quota IDs.
  - **dfs.namenode.inodeid.updateThreshold** Threshold value for inode IDs.
  - **dfs.namenode.blockid.updateThreshold** Threshold value for block IDs.

- **dfs.namenode.id.updateThreshold**
  It defines how often the IDs Monitor should check if the ID pools are running low on pre-allocated IDs.

7.2.2 Client configuration parameters

- **dfs.namenodes.rpc.addresses**
  HOP support multiple active NameNodes. A client can send a RPC request to any of the active NameNodes. This parameter specifies a list of active NameNodes in the system. The list has following format [hdfs://ip:port, hdfs://ip:port,]. It is not necessary that this list contain all the active NameNodes in the system. Single valid reference to an active NameNode is sufficient. At the time of startup the client will obtain the updated list of all the NameNodes in the system from the given NameNode. If this list is empty then the client will connect to 'fs.default.name'.
- **dfs.namenode.selector-policy**
  The clients uniformly distribute the RPC calls among the all the NameNodes in the system based on the following policies. See section 3.7 for more details.
  - ROUND_ROBIN
  - RANDOM

  By default NameNode selection policy is set of ROUND_ROBIN
- **dfs.clinet.max.retires.on.failure**
  The client will retry the RPC call if the RPC fails due to the failure of the NameNode. This property specifies how many times the client would retry the RPC before throwing an exception. This property is directly related to number of expected simultaneous failures of NameNodes. Set this value to 1 in case of low failure rates such as one dead NameNode at any given time. It is recommended that this property must be set to value ¿= 1.
- **dsf.client.max.random.wait.on.retry**
  A RPC can fail because of many factors such as NameNode failure, network congestion etc. Changes in the membership of NameNodes can lead to contention on the remaining NameNodes. In order to avoid contention on the remaining NameNodes in the system the client would randomly wait between [0,MAX_VALUE] ms before retrying the RPC. This property specifies MAX_VALUE; by default it is set to 1000 ms.
- **dsf.client.refresh.namenode.list**
  All clients periodically refresh their view of active NameNodes in the system. By default after every minute the client checks for changes in the membership of the NameNodes. Higher values can be chosen for scenarios where the membership does not change frequently.

7.2.3 Data access layer configuration parameters

- **com.mysql.clusterj.connectstring**
  Address of management server of MySQL NDB Cluster.

- **com.mysql.clusterj.database**
  Name of the database that contains the metadata tables.

- **com.mysql.clusterj.connection.pool.size**
  This is the number of connections that are created in the ClusterJ connection pool. If it is set to 1 then all the sessions share the same connection; all requests for a SessionFactory with the same connect string and database will share a single SessionFactory. A setting of 0 disables pooling; each request for a SessionFactory will receive its own unique SessionFactory. We set the default value of this parameter to 3.

- **com.mysql.clusterj.max.transactions**
  Maximum number transactions that can be simultaneously executed using the clusterj client. The maximum support transactions are 1024.

- **se.sics.hop.metadata.ndb.mysqlserver.host**
  Address of MySQL server. For higher performance we use MySQL Server to perform a aggregate queries on the file system metadata.

- **se.sics.hop.metadata.ndb.mysqlserver.port**
  If not specified then default value of 3306 will be used.

- **se.sics.hop.metadata.ndb.mysqlserver.username**
  A valid user name to access MySQL Server.

- **se.sics.hop.metadata.ndb.mysqlserver.password**
  MySQL Server user password

- **se.sics.hop.metadata.ndb.mysqlserver.connection_pool_size**
  Number of NDB connections used by the MySQL Server. The default is set to 10.

- **Session Pool** For performance reasons the data access layer maintains a pools of pre-allocated ClusterJ session objects. Following parameters are used to control the behavior the session pool.

  – **se.kth.hop.session.pool.size:** Defines the size of the session pool. The pool should be at least as big as the number of active transactions in the system. Number of active transactions in the system can be calculated as (num_rpc_handler_threads + sub_tree_ops_threds_pool_size). The default value is set to 1000.

  – **se.kth.hop.session.reuse.count:** Session is used N times and then it is garbage collected. The default value is set to 5000.

Chapter 8

Removed and Deprecated Features

Hops-FS fundamentally changes the design of metadata storage for HDFS. Due to these changes some of the functionality of HDFS is no longer needed. In this chapter we specify the architectural components of HDFS that have been removed from Hops-FS. The only functionality that we have removed from Apache HDFS is the support for federated namespaces. The need for federated namespaces arose from the scalability limitations of HDFS (more specifically, the NameNode). However, the federated support as it stands, is extremely limited - there is no consistent global view over all federated namespaces, and there is no support for executing computations (such as YARN applications) over all clusters in the federation. There are also no guarantees provided for cross namespace operations, such as moving files. As such, and because Hops-FS has mitigated the scalability, we decided to remove support for federation, At this moment in time, we see little benefit and only extra complexity in HDFS federation.

- **Secondary NameNode**
  The secondary NameNode is no longer supported. Hops-FS supports multiple NameNodes and all the NameNodes are active.

- **EditLog**
  The write ahead log (EditLog) is not needed as all the metadata mutations are stored in the highly available data store.

- **FSImage**
  We don't need to store checkpoints of the metadata (FSImage) as NameNodes in Hops-FS are stateless and metadata is stored in the external metadata store.

- **NFS and Quorum Base Journaling**
  Replaced by the external metadata store.

- **NameNode Federation**
  NameNode federations are no longer supported, due to reasons outlined above.

- **Viewfs**
  Viewfs is used by federated HDFS to view a namespace that contains multiple federated NameNodes.

- **ZooKeeper**
  HDFS uses ZooKeeer for coordination services. Hops-FS has replaced ZooKeeper with a coordination service built on distributed shared memory, see section 3.5 for more details.

Chapter 9

Future work

Hops-FS is a multi NameNode distribution of Apache HDFS. Making the NameNodes stateless poses many challenges, many of which has been addressed in this document. However, Hops-FS is an ongoing research work and some components of Hops-FS can be further optimized.

- **Block Report Handling:**
  Currently in Hops-FS, each NameNode maintains its own view of the datanodes' state. Each time a NameNode starts, it marks all the datanodes as stale. It then marks the DataNode as alive once it receives a block report from the datanode. As datanodes only send their block report to one NameNode each NameNode will have a different view of the state of the datanodes. This problem is illustrated in figure 9.1. As the block reports are sent periodically, the NameNodes views will eventually converge, but it would be more efficient to share this view using the database. Moreover, experimenting with different policies to balance the block reports among the NameNodes may yield even better results than our current random policy.

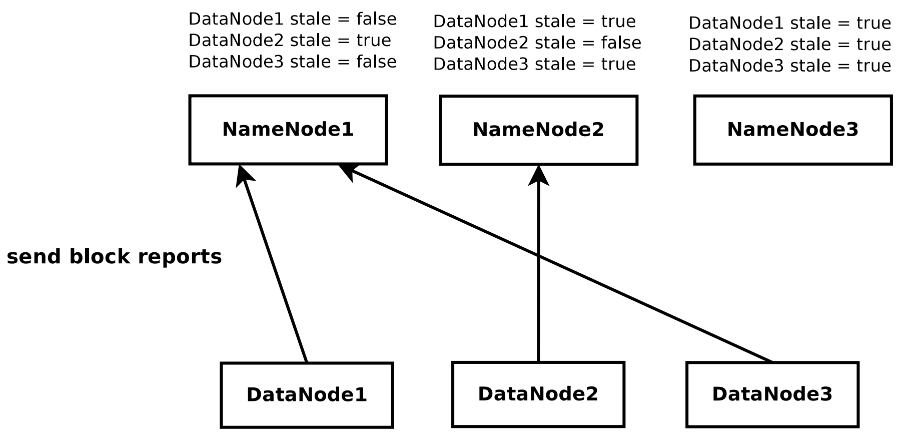

Figure 9.1: Example of DataNode view divergence: NameNode1 and NameNode2 receive reports from the 3 datanodes, while NameNode3 does not

- **Erasure Coded Files:**
  Currently the files are erasure coded by launching a map/reduce application for each file.

In future, Hops-FS will develop native support for erasure coding. The native support for erasure coding will reduce the time for encoding, decoding and repairing files.

- **Heartbeat Processing**
  Currently all the datanodes send heartbeats to all the NameNodes. This is done so because each NameNode maintains a local copy of the sate of the datanodes. We can reduce the number of Heartbeats by uniformly distributing the heartbeats among all the NameNode. This will require storing the state of the datanodes in the database so that all the NameNodes have same view of the state of all the datanodes.

- **Improving the throughput of the NameNodes**
  The performance of the NameNodes is greatly affected by the limitations of ClusterJ library. Due to a software bug in ClusterJ library all the transactions are sent to one transaction coordinator. The transaction coordinator gets overloaded and the performance of the system does not scale. We have an approach that we have tested to work around this bug, and improve the throughput of the NameNodes.

Chapter 10

Conclusions

In this document, we presented Hops-FS, a highly available, high performance, distributed file system that is compatible with Apache HDFS. Hops-FS supports multiple stateless NameNodes, with metadata stored in an in-memory, replicated, distributed called MySQL Cluster. We have presented solutions to the problems of coordinating NameNode management of the file system using leader election. We have also showed how we maintain the consistency of the file system operations, how we optimized block reporting, and and how we cache transaction state to reduce the number of database roundtrips per transaction. Finally, we have shown how we reduce the storage requirements for HDFS by implementing Reed-Solomon erasure-coding replication in NameNodes.

Bibliography

[1] The curse of the singletons! the vertical scalability of hadoop namenode. http://hadoopblog.blogspot.se/2010/04/curse-of-singletons-vertical.html. [Online; accessed 30-Nov-2014].

[2] Databricks. https://databricks.com/. [Online; accessed 30-Nov-2014].

[3] Improve namenode scalability by splitting the fsnamesystem synchronized section in a read/write lock. https://issues.apache.org/jira/browse/HDFS-1093. [Online; accessed 30-Nov-2014].

[4] The lustre storage architecture. http://wiki.lustre.org/manual/LustreManual20_HTML/UnderstandingLustre.html. [Online; accessed 30-Nov-2014].

[5] Mapr apache hadoop distribution. https://www.mapr.com/. [Online; accessed 30-Nov-2014].

[6] Mysql cluster benchmarks. http://www.mysql.com/why-mysql/benchmarks/mysql-cluster/. [Online; accessed 30-Nov-2014].

[7] Mysql cluster cge. http://www.mysql.com/products/cluster/. [Online; accessed 30-Nov-2014].

[8] An Introduction to Gluster Architecture Versions 3.1.x. Technical report, Gluster, Inc, 2011.

[9] S.A. Brandt, E.L. Miller, D.D.E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 290–298, April 2003.

[10] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 37–48, Berkeley, CA, USA, 2013. USENIX Association.

[11] P.F. Corbett, D.G. Feitelson, J.-P. Prost, and S.J. Baylor. Parallel access to files in the vesta filesystem. In *Supercomputing '93. Proceedings*, pages 472–481, Nov 1993.

[12] EMC Corporation. HADOOP IN THE LIFE SCIENCES:An Introduction. https://www.emc.com/collateral/software/white-papers/h10574-wp-isilon-hadoop-in-lifesci.pdf, 2012. [Online; accessed 30-Nov-2014].

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[14] Apache Foundation. Apache hadoop. https://hadoop.apache.org/. [Online; accessed 30-Nov-2014].

[15] Apache Software Foundation. HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID. [Online; accessed 30-Nov-2014].

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[17] Google. Google cloud platform. https://cloud.google.com/. [Online; accessed 30-Nov-2014].

[18] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared database. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394. IFIP, 1976.

[19] Kamal Hakimzadeh, Hooman Peiro Sajjad, and Jim Dowling. Scaling hdfs with a strongly consistent relational model for metadata. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, Lecture Notes in Computer Science, pages 38–51. Springer Berlin Heidelberg, 2014.

[20] Denis Hünich and Ralph Müller-Pfefferkorn. Managing large datasets with irods - a performance analyses. In *International Multiconference on Computer Science and Information Technology - IMCSIT 2010, Wisla, Poland, 18-20 October 2010, Proceedings*, pages 647–654, 2010.

[21] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIXATC'10*, pages 11–11. USENIX Association, 2010.

[22] Adam Kagawa. Hadoop summit 2014 amsterdam. hadoop operations powered by ... hadoop. https://www.youtube.com/watch?v=XZWwwc-qeJo. [Online; accessed 30-Nov-2014].

[23] Marshall Kirk McKusick and Sean Quinlan. GFS: Evolution on Fast-forward. *ACM Queue*, 7(7):10, 2009.

[24] Microsoft. Microsoft azure. http://azure.microsoft.com/en-us/. [Online; accessed 30-Nov-2014].

[25] Salman Niazi. Problems with setpartitionkey. http://bugs.mysql.com/bug.php?id=74431. [Online; accessed 30-Nov-2014].

[26] Several Nines. sizer - capacity planning tool. http://www.severalnines.com/resources/sizer-capacity-planning-tool. [Online; accessed 30-Nov-2014].

[27] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proc. VLDB Endow.*, 6(11):1092–1101, August 2013.

[28] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[29] Andreas J Peters and Lukasz Janyst. Exabyte scale storage at cern. *Journal of Physics: Conference Series*, 331(5):052015, 2011.

[30] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: a study on the facebook warehouse cluster. In *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems*, pages 8–8. USENIX Association, 2013.

[31] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[32] Kai Ren, Garth Gibson, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Poster: Hadoop's adolescence; a comparative workloads analysis from three research clusters. In *SC Companion*, page 1453. IEEE Computer Society, 2012.

[33] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xorbas. http://smahesh.com/HadoopUSC/. [Online; accessed 30-Nov-2014].

[34] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. *Proc. VLDB Endow.*, 6(5):325–336, March 2013.

[35] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[36] Konstantin Shvachko. Name-node memory size estimates and optimization proposal. https://issues.apache.org/jira/browse/HADOOP-1687, April. [Online; accessed 11-Nov-2014].

[37] Konstantin V Shvachko. HDFS Scalability: The limits to growth. *login*, 35(2):6–16, 2010.

[38] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[39] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas

Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[40] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[41] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of OSDI'06*, pages 307–320. USENIX Association, 2006.

[42] System Fabric Works. Using system fabric works lustre solutions for hadoop storage. http://www.systemfabricworks.com/products/system-fabric-works-storage-solutions/lustre-hadoop. [Online; accessed 30-Nov-2014].