Heterogeneous Storage in HopsFS

Bram Leenders

Supervisor: Mahmoud Ismail
Examiner: Jim Dowling
Stockholm 2nd September 2016

Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology

TRITA-ICT-EX-2016:123

Abstract

In the recent years, the Apache Hadoop distributed file system (HDFS) has become increasingly popular for the storage of large data sets. Both the volume of the data and the variety of applications is unprecedented. The variety of tasks, each with its own access pattern and demands, calls for a file system that supports specialized storages for different tasks.

This thesis describes the implementation of heterogeneous storage in HopsFS, a highly-available, highly-scalable version of HDFS. This makes the cluster aware of different storage types (e.g. hard disks and solid state drives) and allows users to specify preferred storage types for their data.

By introducing new storage types, we build in support for storage technologies like SSDs and RAID. The latter is especially of interest, since it increases both bandwidth and reliability of the storage on individual nodes while continuing commodity hardware. Since network bandwidth is increasing orders of magnitude faster than disk bandwidth, increasing the disk throughput is of vital importance to avoid local storage becoming a bottleneck.

The heterogeneous storage Application Programming Interface (API) described in this thesis offers HDFS administrators more control over their data while being compatible with the HDFS framework. Users can choose whether they want files stored on traditional disks, SSDs or more complex constructions using RAID and erasure coding.

Acknowledgements

This thesis would not have come to be without the support of many people.

Firstly I would like to thank my examiner, Jim Dowling, and my supervisor, Mahmoud Ismail, for their vision, insight and support that gave me direction while working with them. My appriciation also goes out to the rest of the Hops team at SICS.

I would also like to express my deepest gratitude to my family, who have always supported by every means possible. You helped me become who I am today.

I dedicate this thesis to my late grandfather, who supported me from my first steps almost until the end of my masters, who always encouraged me to learn and explore, who was always the most interested listener to any of my stories and the proudest reteller of them.

Contents

List of Figures

List of Tables

List of Code

Glossary

This document requires readers to be familiar with some technical terminology. To assist readers, this is a list of some of the terminology and acronyms used in this document.

Readers should take care to note the difference between the abbriviations for bytes per second (Bps) and bits per second (bps), also in combination with metric prefixes like MBps vs. Mbps.

API application programming interface: the interface of a piece of software exposed to developers.

GFS Google file system [19].

HDFS Hadoop distributed file system: an open source distributed file system maintained by Apache [40].

HopsFS a distributed file system based on HDFS.

IOPS input/output operations per second.

JBOD just a bunch of disks; a non-RAID configuration that pools disks in one volume without offering fault-tolerance.

MTTL mean time to (data) loss.

OLAP Online analytic processing; a class of systems that includes datamining and trend analysis, typically not in real-time.

OLTP Online transaction processing; a class of systems for transaction-oriented applications, providing atomicity, immediate feedback and high transaction throughput.

RAID redundant array of inexpensive disks.

RPC a remote procedure call uses network messages to perform a function call on a different system; it is a form of communication between processes in different address spaces.

SICS SICS Swedish ICT is a research institute for applied ICT in Sweden.

SSD solid-state drive, also referred to as flash storage.

URE unrecoverable read error.

Chapter 1

Introduction

Since the early 2000s, the demand for storage and processing of large volumes of data has rapidly grown. The key driver for this demand are various types of large scale data analytics, such as user behaviour analysis, search engines and recommendation systems. Usage of big data is not restricted to a select few technology companies but is an important aspect of many enterprises.

Two big innovators in the world of large distributed systems are Google and Yahoo. Both companies developed frameworks for dealing with big data. Google designed a programming model for distributed computing called MapReduce [16]. They also published information on the design of their proprietary software framework for distributed data storage, the Google File System [19] (GFS). In 2007, Yahoo! released an open-source framework called Hadoop [40] which offers functionality similar to Google's GFS.

Since the release of Hadoop, the entry barrier for big data analytics has drastically lowered. Many companies either use Hadoop products directly or build their own products on top of Hadoop, using for example the MapReduce API. A 2011 survey [36] indicated that 34% of the respondents was already using big data analytics, and 70% considered it a business opportunity. Research by International Data Corporation predicted the Hadoop-MapReduce market size to exceed 800 million dollar in 2016, which is a 60% compound annual growth rate since 2011 [46].

One of the core parts of Hadoop is the Hadoop distributed file system (HDFS). It is built to provide reliable storage for massive data sets on commodity hardware. HDFS clusters can consist several thousand nodes. Yahoo!, for example, runs multiple clusters with over 3000 nodes [40] and 10,000 CPU cores [51].

1.1 Motivation

During the initial design of Hadoop clusters were designed to be homogeneous. That is to say, all nodes and storages are treated as if they are the equal. Furthermore, the underlying storage type was typically assumed to be hard disks. At the time this made sense, because of some assumptions about the workloads running on Hadoop.

This section briefly describes some of the original assumptions and why they do not hold any more. As a result of these invalidated assumptions the original assumption that homogeneous storage suffices also becomes invalidated.

1.1.1 Original Design Rationale

HDFS is optimized for batch processing large files with sequential read/write access [40]. Most jobs running on HDFS are MapReduce jobs that stream a large dataset and have append-only output, so the MapReduce data access pattern matches HDFS' capabilities.

Furthermore, most MapReduce jobs using HDFS are built around the assumption that network access is the primary bottleneck in a cluster. When Hadoop was designed, a typical server would have a 100 Mbps network connection and disks could read at about 100 MBps, so single disk throughput was almost an order of magnitude faster than the network.

Lastly, Hadoop clusters were assumed to be homogeneous. That is to say, all nodes and all disks in the cluster were assumed to be equal.

These assumptions made it unnecessary for Hadoop to support multiple storage types. Both nodes and storages were physically homogeneous and there was no immediate need to support storage heterogeneity.

Changing Data Access Patterns

As previously stated, hard disks perform well for streaming data access. For most tasks, sequential reads will be fast enough not to form a bottleneck. However, hard disks perform poorly for random IO. While a single disk may have 150 MBps sequential read throughput, reading random 4 kilobyte blocks will yield a throughput of less than 1 MBps [1].

Although Hadoop was designed for MapReduce jobs with sequential data access over petabyte scale input data sets, it is now used for a much wider variety of applications. Apache HBase [2], a distributed database, is an example of a class of applications with a random IO pattern. It can scale up to terabytes but its definitely not the sequential data access application that HDFS was built for.

Another class of applications that would benefit from support for low latency storage are interactive query processing applications like Hive. When users wait for a query to finish, they want to see results in seconds or less. The traditional MapReduce batch processing approach on hard disks cannot provide such performance.

So, we want to have a file system that performs both well on sequential access on big data (as per the original design criteria) and offer decent random read performance for applications that need it.

Storing all data on solid-state drives (SSDs) is prohibitively expensive [31], but mixing hard disks and SSDs could be a viable option to improve performance while keeping the cost down. By having separate storage policies for HBase or Hive and regular MapReduce applications, the cluster could store the HBase data on SSD and the other data on hard disk. This allows both HBase

[1] A 7200 rpm hard disk can do up to 100-200 random reads per second, so reading random 4KB blocks has a throughput of 400 to 800 KBps.

[2] `https://hbase.apache.org`

and MapReduce jobs to run on the same nodes (thus sharing CPU and memory) but each have their specialized storage; SSD for HBase and hard disk for MapReduce.

Increased Network Bandwidth

Nodes in old data centres have a network connection of about 100 Mbps. Modern data centres have gigabit ethernet (GbE), which provides ten times more bandwidth. The next generation of data centres will have 10 GbE network connectivity. Thus, over the course of roughly a decade, the bandwidth in data centres will have increased by a factor of a hundred.

Disk throughput, however, has not increased significantly. Although the capacity of disks follows an exponential increase, disk speed is still under 200 MBps. Hadoop is built to run on cheap commodity hardware, which currently means that typical HDFS clusters use 7200 rpm SATA hard disks. These hard disks have a read speed ranging between 80 and 160 megabyte per second (640 Mbps to 1.2 Gbps).

This means that traditionally, network bandwidth was an order of magnitude lower than hard disk bandwidth. Currently, disk throughput is about on a par with network throughput (both at 1Gbps). For new clusters, network bandwidth can be a factor ten higher than hard disk bandwidth. To avoid hard disks becoming the bottleneck, we have to revisit the assumption that hard disks can keep up with the network.

Although SSDs provide better performance than hard disks, their cost is prohibiting widespread adoption for big data storage [31]. As such, simply replacing all hard disks with SSDs is not a viable solution.

Another strategy to keep up with the increased network bandwidth is striping reads/writes over multiple disks. By using a redudant array of inexpensive disks (RAID), the throughput of storage volumes can be increased without sacrificing data locality. However, there is no silver bullet since each RAID configuration has its advantages and disadvantages.

Heterogeneous Nodes and Tiered Storage

Although clusters typically start with all nodes being equal (an exception being the master node, which can have more reliable and faster components), clusters expand over time with nodes with different specifications. Differences in specifications can result from unavailability of the original hardware (or cheaper alternatives) or differences in needs for the cluster. In particular, the HDFS developers noted that demand for storage outstripes the demand for compute power [7], which implies clusters should be expanded for storage more often than expanded for compute power.

The amount of data stored in clusters grows rapidly and once existing clusters fill up, administrators must either delete old data or expand the cluster. By introducing a new type of node, the archival node, administrators could add a few nodes with very high storage capacity to existing clusters. These archival nodes could have tens or hundreds of terabytes storage per node with little computing power. Archival nodes add little compute power but provide very cheap storage compared to regular nodes in Hadoop clusters.

An interesting example is the company Backblaze, which builds storage pods with 240 TB storage per pod for about 10,000 dollar [9]. Compared to typical Hadoop nodes, assuming 2,500 dollar for a 10 TB node, the Backblaze pods are 83% cheaper per gigabyte. Typical Hadoop nodes provide far more compute power but for expansion of storage capacity storage pods are a good alternative.

1.1.2 Revisiting Design Rationale

As discussed above, assuming all nodes and disks to be equal limits the performance and versatility of clusters. A lot has changed since the first Hadoop release and the ecosystem has grown significantly. Supporting the new applications requires us to revisit the original assumption that nodes and storages are homogeneous. Clusters should support multiple storage types and allow users to leverage the strengths of each type.

By introducing support for heterogeneous storage, we make the file system aware of differences in performance characteristics of the storages in the cluster.

There is no silver bullet approach to storage, but by supporting multiple storage types administrators can choose storage strategies that are optimized for the applications running on their cluster. Databases can use SSDs, data analytics can use hard disks and archival of old data can be done on cheap storage pods.

1.2 Problem description

This thesis introduces support for heterogeneous storage in the HopsFS distributed file system. Instead of viewing the data storage nodes as black boxes, the cluster tracks which disks are present on each node and what type these disks are (e.g. hard disk, SSD or RAID).

Furthermore, this work adds new functionality that allows clients to specify storage policies for data in HopsFS. A storage policy describes the desired storage types used for storing replicas of data. For example, a policy could describe that every file in `/users` should have a single copy on SSD for fast reads and backup copies stored on hard disks for fault tolerance.

Applications using HopsFS can use this control for hierarchical storage: data that is accessed frequently or with strict time constraints can be placed on fast, expensive storage, while infrequently data can be placed on slow, cheap storage.

The solution described in this thesis is modelled after the implementation in HDFS. The API exposed to clients for querying the available storage policies and setting a policy is the same as in HDFS. Thus, applications written for HDFS can use the heterogeneous storage functionality in HopsFS without changes to the code.

1.3 Problem Context

This work is in the context of distributed storage and processing of massive data sets. Traditionally, this area consists largely of batch processing and online analytic processing (OLAP) systems.

Recently, the context has shifted to include online transaction processing (OLTP) applications as well. Not all applications running on HDFS or HopsFS process petabyte datasets as input and run for hours or days; many applications process several megabytes and finish running in seconds or minutes.

Some examples of traditional OLAP systems using HopsFS are customer behaviour analysis, stock market analysis and fraud detection. Concrete examples of OLTP applications are distributed databases like HBase and Impala and interactive query applications like Hive and Pig.

1.4 Thesis Outline

Chapter 1 introduces the problem and gives an overview of the context of this thesis. Chapter 2 further describes the context of this thesis and provides the information needed to understand the problem, context and solution. Related work and competing solutions are discussed in chapter 3. The goals and implementation of this work are discussed in chapter 4. Chapter 5 gives an analysis of this work and explains how it can be used and extended. Finally, chapter 6 evaluates the goals of this work and gives some final remarks.

Chapter 2

Background

This chapter provides information about the Hadoop ecosystem that the reader should have to fully comprehend the rest of this thesis. It will first give a high-level overview of distributed systems and explain what we mean when we talk about big data.

After defining these terms, we give a high-level overview of two of the earliest tools to handle big data in a distributed environment: MapReduce and the Google File System (GFS). Subsequently, we discuss Apache Hadoop, the open-source implementation of MapReduce and GFS. Special attention is paid to the Hadoop Distributed File System (HDFS), since this module is very similar to the topic of this thesis.

Finally, we compare HDFS with some its competitors: MapR, CephFS and HopsFS. The latter one is most important for this thesis, as the rest of this thesis discusses changes made to HopsFS.

2.1 Big data

Big data is a term used to describe datasets that are too big to process using traditional methods. A study by Laney [28] has identified three dimensions that force companies to adept new architectural solutions and trade-offs:

- **Volume**: the amount of data has massively increased over the last decade. E-commerce, for example, logs and processes all user transactions, which leads to 10 times as much data as traditional sales [28]. The scale at which data is stored and processed has moved from gigabytes to terabytes and terabytes to petabytes.

- **Velocity**: the speed at which data is processed has increased and the expected response time has lowered. Companies move from periodic batch processing to near real-time or real-time processing. For example: moving from product recommendations in e-commerce with (which has a relatively static product catalogue) to recommendations for news articles (which change by the minute) or stock options (which change every second) [15].

- **Variety**: the sources of data have become more more varied in terms of data structure, format and semantics.

These three aspects are commonly referred to as the three V's of big data. Other authors have added more V's such as value, which represents the increased financial gain companies have from their data; veracity, which represents the unreliability of big data sources; and volatility, which refers to how long data is valid and relevant.

Because of these new challenges, a demand has risen for systems that handle data in new ways. Examples of approaches are taking samples of the data to process only a fraction of all data, investing in more powerful infrastructure and using different algorithms for data processing. The former will not be discussed in this thesis, but the next sections will discuss the latter two approaches: scaling the system by using a cluster and using the MapReduce paradigm for solving problems in a distributed fashion.

2.2 Distributed systems

The term distributed systems refers to a networked set of hardware of software components that communicate only by message passing [14, p. 2]. Unlike many other forms of parallel computing, distributed systems do not have access to a shared memory. The lack of shared memory allows them to be more scalable, but introduces new problems with concurrency, timing, network failures and independent node failures.

The term can be used for a large variety of appliances such as sensor networks, the Bitcoin network and distributed databases. In this thesis we focus on a specific type of distributed systems, called (commodity) clusters. These are servers from off-the-shelf hardware connected with a fast network. The price per node is kept low by using commodity hardware, while scaling to many nodes allows for high performance.

Due to the large amounts of nodes in clusters, single node failures occur very frequently. Google reported a mean time to failure (MTTF) of 4.3 months for servers in their data centres [17]. For a cluster with 10,000 nodes, this translates to an average of one node failing per 20 minutes. Because of the high frequency of node failures, the cluster must be fault tolerant to be practically useful. Data stored in the cluster should not get lost and processing should not be interrupted when a node or group of nodes fail.

This fault tolerance is built in on a software level: programs running in clusters must be able to detect and recover from a fault. Writing fault-tolerant distributed programs is very complex, so many distributed programs use frameworks for error handling. One of these framework is the Apache MapReduce model, which is currently the main framework for big data processing.

2.3 MapReduce

The concept of the MapReduce model is to express a program in two functions: a mapper and a reducer [16]. The map function processes the input and emits key-value pairs. These pairs are then shuffled: the values are grouped by key, resulting in a key with a list of values. These grouped key-value pairs are then processed by the reducer, which reduces a key with a list of values to a single key-value pair.

To illustrate this consider the example code in Listing 2.1, which defines a mapper and reducer that count how often words occur in a given input:

```
# key = document name (ignored in this example)
# value = document content
def map(string key, string value):
  for word in value.split():
    emit(word, "1");

# key = word
# intermediate_values = list of counts
def reduce(string key, int[] intermediate_values):
  sum = 0;
  for value in intermediate_values:
    sum += value;
  emit(key, sum);
```

Listing 2.1: MapReduce wordcount mapper and reducer

The MapReduce model allows for simple horizontal scaling: each node that contains part of the input data runs the map function on the local data. The intermediate values are then distributed such that all values with the same key end up on the same node. Each node that receives a list of intermediate values will then run the reduce function to produce the result.

Figure 2.1 shows how the mappers and reducers only communicate during the shuffle phase. Each mapper runs independently from other mappers and the same goes for the reducers: only the intermediate values produced by the mappers are sent over the network.

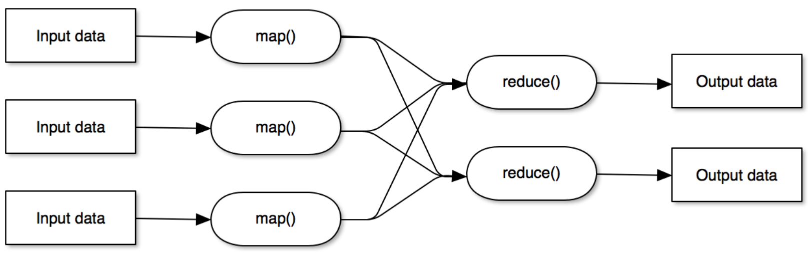

Figure 2.1: MapReduce diagram

To reduce the load on the network, a combiner can be used. This is a function that takes a key with a list of intermediate values and outputs the same key with a (smaller) list of intermediate values. It runs on the output of the mapper before it is shuffled over the network. The combiner can run an arbitrary amount of times (zero, once or multiple times), the exact number of times is not controlled by the programmer.

As an example, the wordcount program defined in 2.1 can be enhanced by providing the combiner defined in Listing 2.2. This reducer combines a list of integers to the sum of all integers in the list. For example, it could combine `("and", [1,1,1,1,1])` to `("and", [5])`. For a word that occurs frequently in a text, it could save a lot of bandwidth.

Note that the types emitted by the mapper (a string key and an integer value) match the types emitted by the combiner. Also, note that the combiner can take its own output as an input, so even if the mapper hasn't finished all input the combiner can already combine the intermediate results to save resources. So, in a subsequent run of the combiner, it could combine `("and", [5,1,1,1])` to `("and", [8])`.

```
# key = word
# intermediate_values = list of counts
def combine(string key, int[] intermediate_values):
  intermediate_sum = 0;
  for value in intermediate_values:
    intermediate_sum += value;
  emit(key, intermediate_sum);
```

Listing 2.2: MapReduce wordcount combiner function

To make this paradigm fault tolerant, the MapReduce framework uses a task tracker. Each instance of a mapper or reducer that should run during the processing is called a task. If a task does not complete in time, the task tracker will execute it on another node. The tracker also guarantees that only one instance of each task has an effect on the processing; it makes the tasks idempotent, so to say.

This ensures regardless of any node failures, the wordcount example will count each word exactly once. And all of this happens opaquely to the programmer, which makes development a lot easier.

2.4 Google file system

The GFS was developed in the early 2000's to store large amounts of information on commodity hardware [19]. Its original goal was to store the datasets generated by crawling and indexing the web. As such, it is optimized for storing many large files (in order of gigabytes per file) and allowing very high read/write throughput.

There are two types of nodes in a GFS cluster: a single master which stores metadata for the entire file system, and many chunkservers that store the data. File are split into fixed-size chunks of 64MB, which are distributed over the chunkservers. Each chunk is replicated to three different chunkservers for reliability. Clients connect to the master for metadata operations (such as querying which datanode stores a chunk) but connect directly to chunkservers for datatransfer. Figure 2.2 shows a high-level overview of the GFS architecture.

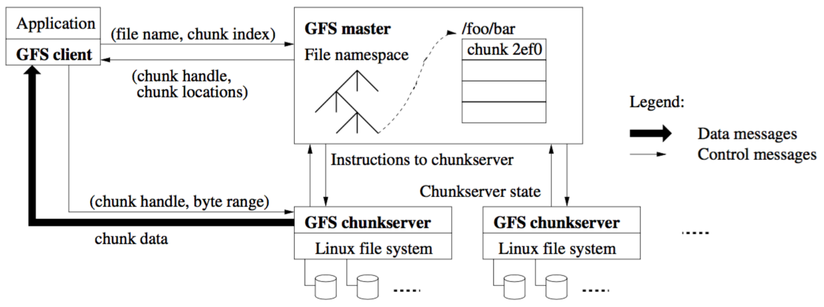

Figure 2.2: GFS architecture [19]

The GFS was designed to have a single master in order to simplify metadata storage and operations (e.g. garbage collection and replica control) [29]. The downside is that the master node is a single point of failure: chunkservers store blocks redundantly and can fail without affecting the cluster but the master node does not have a failover.

A second problem of the single-master architecture is that the master can become a bottleneck when scaling out the cluster to more chunkservers. The Google team solved this by splitting a file system up in multiple subvolumes, called cells, each with its own master. This distributes the load over multiple masters while maintaining the single-master architecture.

The downside of this solution is that it adds complexity, as clients must be aware of the different cells. The design also does not support atomic changes (such as moving data) between cells, which has to be added on top if this is a requirement.

2.5 Apache Hadoop

Apache Hadoop [51] is an open-source Java framework for storing and processing large volumes of data. Its core components are Hadoop MapReduce and HDFS, which are designed to be similar to MapReduce and GFS.

Hadoop MapReduce is an open-source Java framework for running MapReduce programs on a cluster. Users specify a mapper, a reducer and data source. Hadoop will then execute the mapper on nodes that contain the input data, shuffle the data such that it is grouped by key and run the reducer for every key. By running the mapper on a node that stores input data, Hadoop attempts to minimize the amount of network traffic. If a node fails while processing the data, Hadoop will automatically retry the execution on another node to provide fault-tolerance.

Besides its two core components, there are many other parts of Hadoop that form an ecosystem. Examples are YARN [45], a resource negotiator which is now so tightly integrated it could be considered a core component; Pig [33], a high-level platform to write MapReduce programs in Java, Python, JavaScript, Ruby or Groovy; and Hive [44], a data storage system based on SQL that allows querying very large datasets. Some of the parts in the Hadoop ecosystem have pluggable replacements. Section 2.6 discusses three alternatives for HDFS.

What is important for this thesis, is that most components in the Hadoop ecosystem use HDFS. MapReduce tasks use HDFS for data in- and output, distributed databases (e.g. Hive and Impala) use it to store the raw data, et cetera.

Hadoop Distributed File System

The design choices for HDFS are similar to those of GFS. More specifically, it has the following characteristics [11, 40][51, p. 41,42]:

- **Built for fault tolerance**: HDFS is designed to provide reliable storage on unreliable commodity hardware. By default, each block is replicated to three different nodes spread over two racks [51, p. 74]. This allows any two nodes or an entire rack to fail without causing interruptions or losing data.

- **Optimized for large files**: files stored in HDFS are typically in the gigabyte to terabyte range. They are split into smaller blocks (64 or 128 MB) and spread over multiple nodes. This allows storage of files larger than any single node can store and parallel reads.
- **Optimized for streaming access**: jobs on Hadoop typically read entire datasets in sequential order, so reads are optimized for high throughput instead of low latency.

Also, like GFS, HDFS has a master/slave architecture: an HDFS cluster has a single master, called the namenode, and multiple slaves, called datanodes.

The namenode stores the locations of each replica and all other metadata of the file system, such as the file system tree and file permissions. An HDFS cluster has only one active namenode and optionally a secondary namenode acting as failover (it does not allow load-distribution). This is an important limitation, as the namenode must be able to store all metadata in the Java heap: for very large clusters, this can become a bottleneck [41, 42].

The datanodes store the blocks and periodically report to the namenode which blocks they store. When clients want to access a file, they first query the namenode to ask which datanodes store replicas of a block, and then directly connect to one of these datanodes.

A single HDFS cluster can scale up to several thousand datanodes and tens of petabytes of storage. Already in 2010, Yahoo! ran clusters with 3500 nodes and tens of petabytes of storage [40].

2.6 Overview of Distributed File Systems

There are other distributed file systems that are comparable to GFS and HDFS. This section discusses three alternatives for HDFS, all of which are interchangeable without breaking compatibility with other parts of Hadoop.

The basics of each file system are very similar; files are split into chunks (sized 64, 128 or 256 MB) and these chunks are distributed and replicated over the nodes. The main differentiator between the file systems is how they store the file system metadata, such as the directory structure, file permissions and physical locations of chunks (which nodes store a chunk).

2.6.1 Ceph

Ceph [48][1] is an open-source[2] implementation of a fully distributed file system. Like HDFS, it has separate nodes for data storage and metadata storage, called respectively object storage deamons (OSDs) and metadata servers (MDSs). Ceph also uses a third group of nodes, called monitors, to keep track of which OSDs are online and which are down.

The key difference between Ceph and HDFS is that Ceph has multiple MDSs, whereas HDFS effectively has only one namenode. Load balancing over MDSs is achieved by dynamic subtree partitioning [50], where each MDS is responsible for a subtree of the file system. Each MDS tracks the popularity of the subtree

[1] `https://ceph.com`
[2] `https://github.com/ceph/ceph`

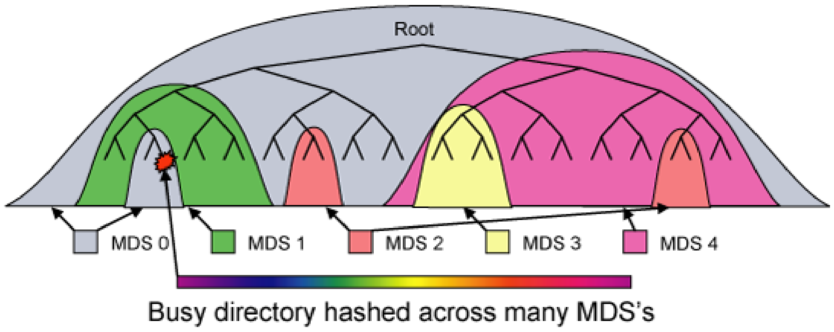

Figure 2.3: Ceph dynamically load-balances metadata storage by distributing subtrees over multiple metadata servers [48]

it is responsible for. The MDSs periodically compare load values and can move responsibility for a particular subtree to balance the load.

In addition to load balancing, Ceph reduces the amount of metadata operations by dividing the file system into placement groups (typically 100 per OSD) and using a hashing function called CRUSH [48, section 5.1][49] to pseudo-randomly distribute the placement groups over the datanodes. Because the CRUSH hash is a deterministic function, any node with up-to-date information about the cluster topology can perform the CRUSH hashing function to calculate which nodes should store a piece of information. This contrasts to HDFS where the namenode keeps a map to store the location of each block, and where clients must query the namenode for every read operation.

Replication is built in on a per-placement group basis: each placement group is mapped to n different OSDs to provide n-way replication. The OSDs are periodically pinged by the monitors to keep an up-to-date overview of the cluster. When an OSD is unresponsive, it is marked as *down* and excluded from updates; when coming back online, it will ask the primary replica of each of its placement groups for a log of all recent changes. If an OSD is unresponsive for a longer period, it is marked as *out* and re-replication is started. The network topology is stored by a group of monitors that use Paxos (a distributed consensus algorithm [27]) to maintain consensus of the current state of the cluster.

2.6.2 MapR

The MapR[3] distributed file system is a commercially available replacement for HDFS.

Unlike HDFS and Ceph, all nodes in the MapR file system are equal. That is to say, MapR does not distinguish between namenodes and datanodes. Data stored in a MapR file system is distributed over containers; a container is a data structure that contains both metadata (directory structures and pointers to chunks of data) and actual data, the latter consisting of files split into chunks of 256 MB. The physical locations of a container is stored in a so-called container location database.

The rest of the metadata is all stored inside the containers. Directory structures, access permissions and any indexes are all stored inside containers, just

[3] `https://www.mapr.com`

like regular data. Since containers are replicated for fault-tolerance, this means all metadata is automatically replicated as well.

By having very large containers (10 to 30 gigabyte), the overhead of storing the location of each container is relatively small compared to storing locations of blocks in HDFS, since there are far fewer containers in MapR than blocks in HDFS. A petabyte file system in HDFS with 128MB blocks has close to eight million blocks. A similarly sized file system in MapR has only 50,000 containers.

2.6.3 HopsFS

HopsFS[4] is part of Hops, a distribution of Apache Hadoop that focusses on simplifying Hadoop. It is developed in cooperation of KTH and Swedish Institute of Computer Science (SICS) and available as open-source project[5].

HopsFS is based on HDFS but made a big architectural change; unlike HDFS, all metadata is stored in a distributed relational database instead of in memory on a namenode [23, 32, 47]. Namenodes in HopsFS are stateless, since all metadata is stored in a database. Currently HopsFS is implemented to work with MySQL Cluster, but the modular design allows other databases to be used.

Figure 2.4 shows the architecture of HopsFS. Note that clients do not communicate with the database directly; they always communicate via a namenode. In fact, the communication between clients and namenodes is almost the same as in HDFS; HopsFS uses the same RPC messages as HDFS. The main difference is that clients are aware of multiple namenodes and are free choose which one they use.

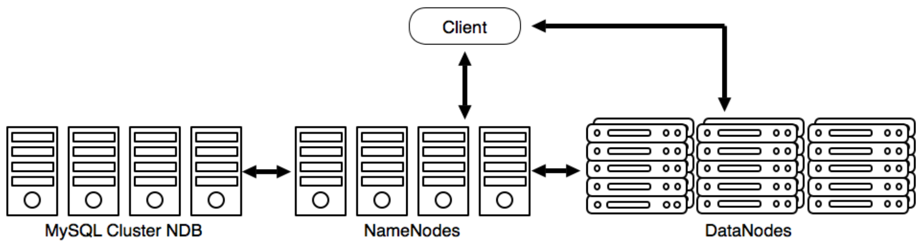

Figure 2.4: HopsFS architecture

Because namenodes are stateless, a highly available and scalable setup can be achieved by using MySQL Cluster with multiple namenodes. This is resistant against node failures; as long as at least one namenode is up and the MySQL cluster is working, the HopsFS cluster will continue working as well.

Another advantage of using a distributed database is that it allows HopsFS to store far more metadata than HDFS. Benchmarks indicate that HopsFS can store an order of magnitude more metadata than HDFS [32]. Since the size of HDFS file systems is limited by the amount of metadata (which must fit in memory on the namenode) [41], HopsFS can accommodate larger file systems than HDFS.

[4]http://www.hops.io
[5]https://github.com/hopshadoop/hops

HopsFS also uses the metadata storage to replace many of the services running in a Hadoop cluster. Running HDFS as a highly available cluster requires a Zookeeper cluster, as a distributed coordination service; standby namenodes, as a hot failover for namenodes; and journalling servers, to store edit logs to keep the namenodes (active and standby) synchronized. Altogether, it is a pretty complex setup. In HopsFS, all these services are integrated in the metadata storage and add no operational complexity.

Chapter 3

Related work

3.1 MapReduce on SSD

The price of SSDs has rapidly dropped in the last decade. In 2007 the price of SSDs was about 40 dollar per gigabyte [35], 120 times more expensive per gigabyte than hard disks. By 2014 the price difference had dropped to a factor 25. Although hard disks will have a better price per gigabyte for the foreseeable future, SSDs are quickly closing the price gap.

If we compare prices not based on capacity but on random read performance, SSDs easily outperform hard disks already. A commodity hard disk can do in the order of 100 input/output operations per second (IOPS), where most SSDs have between 50,000 and 100,000 IOPS [30]. Thus, SSDs outperform hard disks by orders of magnitude on random access workloads. Even considering a difference in price by a factor 25, as argued above, some workloads justify the higher price per gigabyte by the massive performance advantage of SSDs.

Comparing based on sequential read/write performance, the gap between SSDs and hard disks is considerably smaller. An SSD outperforms a hard disk by a factor of about 5 to 10 [26, 30], so a factor 25 in price difference may not be justifiable. By pooling hard disks with RAID or JBOD, one can achieve a throughput that is comparable to that of an SSD for a lower price.

Since SSDs have become a real option for storage systems, companies have to consider using SSDs for their storage needs. It is important to keep in mind that results vary heavily per type of workload, so care has to be taken in selecting which workloads will benefit. This section discusses research by various companies and the results they achieved.

3.1.1 Cloudera Benchmarks

Cloudera benchmarked various synthetic MapReduce test jobs to test the performance of SSD based versus hard disk based clusters [26]. As a test suite they used some of the most well-known Hadoop programs such as WordCount, TeraSort, TeraGen, TeraValidate and simply writing data to HDFS. This set of programs covers most possible workloads; read-only (TeraValidate), write-only (TeraGen), CPU-heavy (WordCount) and mixes of the previous.

Figure 3.1 shows the results of the various job types with different configurations. Tests were executed with 6 disks, 11 disks, 6 disks plus one SSD (hybrid) and 6 disks plus one SSD where the SSD is split in 10 directories. The disks are treated as a JBOD with round-robin distribution of load. By splitting the SSD into 10 directories, HDFS will put a factor 10 more load on it.

The most relevant comparison is between the first bar and the fourth bar; these represent the difference of simply using hard disks versus using hybrid storage.

The authors observed a performance increase of up to 70% for jobs with a large intermediate result set, e.g. TeraSort. The reason for this is that large intermediate results are very read/write intensive in the shuffle phase, so they benefit greatly from SSDs.

However, many workloads do not benefit significantly. As visible in figure 3.1, the difference between HDD-6 and Hybrid with split SSD is very small. The reason for this is that some jobs bottleneck predominantly on the CPU; improving the storage layer does not affect their runtime significantly.

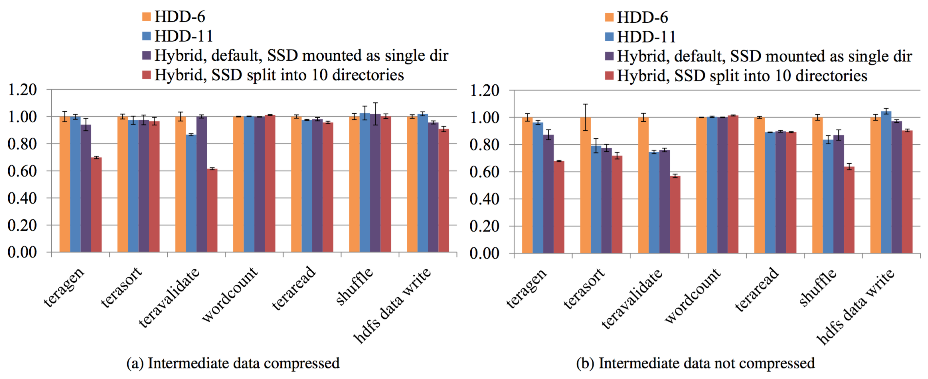

Figure 3.1: Normalized job durations for various MapReduce tasks in various setups (lower is better) [26]

The authors conclude that, although performance increased by up to 70% for some workloads, most workloads do not benefit significantly from SSDs. They also conclude that even for benefits that do benefit from SSDs, the cost of SSDs is disproportionally high. The observed price-per-performance (dollar per MBps of throughput) of SSDs is 2.5 times that of hard disks. For many workloads this will be an even bigger difference since many jobs are not (exclusively) I/O bound.

However, the authors chose a very expensive SSD to do their comparison. The model they chose is a PCIe SSD that costs 14,000$ and provides 1.3 GBps random read/write throughput. At the time the paper was published, SATA SSDs were available for a fraction of the price of their PCIe SSD. One could get an SSD that performs nearly half as good (600 MBps) for less than 5% of the price. If the authors had chosen such an SSD for their comparison, they would likely have reached a different conclusion for the price-per-performance comparison.

3.1.2 Facebook Messages

Harter et al. [24] performed a real world case study of Facebook Messages, which stores messages between Facebook users in HBase [18] (which uses HDFS as underlying storage layer). Messages is a good example of an ideal usecase for heterogeneous storage: it has random I/O, 90% of the files are small (less than 15 MB) and only a third of the data accessed is older than 8 days. Thus they hypothesized that putting a fraction of the data on SSD can boost performance.

The Facebook team noted that the hot data is too large to fit in memory and cold data too large to fit on flash storage. However, a hybrid storage where hot data is stored on flash and cold data on disk was a viable option. Their analysis leads them to the following observations:

- Most files are small and short-lived, resulting in many metadata operations per gigabyte of storage. This could cause the namenode to become a bottleneck. Note that HopsFS should deal better with this as it allows easier scaling of the namenode.
- Using flash storage caching in addition to caching in RAM greatly improves cache hit rates for reads (going from a 45% to a 63% hit ratio). Additionally, since flash is non-volatile, a hybrid cache can avoid over half the extra disk seeks after a node reboot.
- Using flash storage as write buffer did not yield significant improvements (a few percent at most). Many writes are sequential and do not benefit from SSDs. They recommend flash only to be used as read cache as reads are not sequential for their workload.

They also made a financial overview of various configurations of their servers, in which they observe that using SSDs can triple performance while adding only 5% to the cost of a server.

It is important to stress that the analysis of Messages is an analysis of an ideal scenario; most workloads will not triple their performance quite so easily. The authors also note that other optimizations, such as reducing the amount of writes by tuning the amount of compactions and logging in HBase, also greatly improves performance.

3.1.3 HDFS Workload Types

The two studies discussed in the last sections show that the benefit of using SSDs largely depends on the type of workload on the cluster. The workloads that benefit most are the ones with much shuffling and those with a random read/write access pattern. Workloads that are mostly CPU bound do not benefit significantly and workloads with a sequential data access pattern may not benefit enough to justify the high monetary cost of SSDs.

Thus, it is important to know what type of jobs are executed on a cluster in order to predict the benefit of SSDs.

A study by Chen et al. [13] describes over two million MapReduce jobs run by Facebook and a customers of Cloudera. Their study observes that over 90% of all analysed jobs are small; they run for minutes and accesses megabytes or several gigabytes of data. Thus, if we are able to predict which data this is, storing a small fraction of the data on SSD will speed up the majority of the jobs. Over 80% of all observed data re-accesses happen within minutes to hour range (as shown in figure 3.2). This indicates that there is a potential gain from

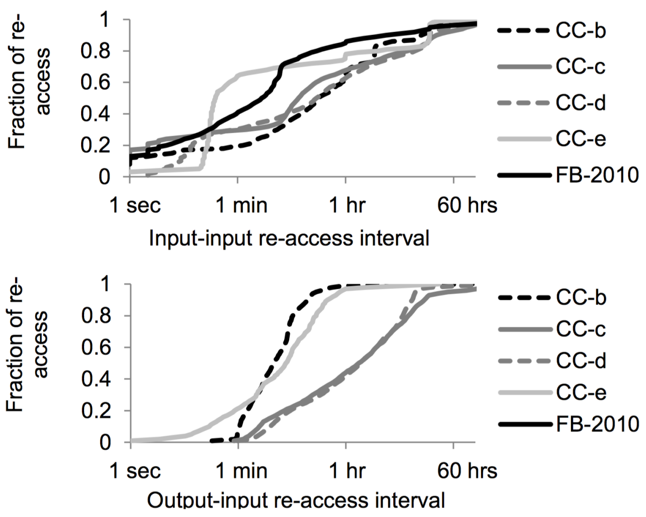

Figure 3.2: Data re-access intervals. Shows distribution for re-accessing input files (a) and using output of one job as input for another (b) [13]. Different lines indicate the different clusters: four Cloudera customers (CC-*) and one Facebook cluster.

caching of recently accessed data. The study did not specify the spatial locality of the data access.

Research by Ananthanarayanan et al. [4] on Facebook and Microsoft Bing workloads observed that 79% of a job duration is spent in an IO-intensive phase. Given that the median memory usage in the Facebook cluster is just 19%, there is a lot of potential for caching data in memory at no extra cost. They observe an improvement of over 50% (up to 77% for small interactive jobs) in completion time both on the Facebook and the Microsoft Bing cluster after implementing a distributed cache in the file system.

The main reason for this speedup is that many higher-level applications (in this paper Hive [44] and SCOPE [12] jobs) execute a chain of MapReduce jobs. For example, calculating the top 10 search queries is split into two jobs: one aggregation (count) and the second to order the aggregated data. Because the output of one job is the input of the next, the access pattern of the application exhibits strong temporal locality: data written to the file system as output of the aggregation is read shortly after as input for the next job.

Lastly, a study by Appuswamy et al. [8] argues that there is too much focus on scaling out to petabyte scale. They observed 174,000 MapReduce jobs running on a Microsoft cluster, and observed a median input data set size of 14 GB and a median running time of 3.83 hour. Considering the low price of memory, they argue that scaling up is a better approach for most jobs. Scaling out a cluster improves performance of big jobs but does not affect small jobs that suffer from latency. Scaling up nodes by adding memory and SSDs for caching also improves the performance of small jobs by decreasing latency.

3.2 HDFS heterogeneous storage

As of version 2.6, the Apache HDFS has support for heterogeneous storage. Instead of tracking blocks per datanode, the namenode now has a finer granularity and tracks blocks per storage. Thus, the namenode knows which disks in a cluster store a block and what type of disks they are (hard disk, SSD, etc.).

The HDFS implementation is split up over multiple releases and multiple JIRAs (tasks). The following three umbrella JIRAs cover most of the changes:

- HDFS-6584 [7]: support multiple types of nodes; compute nodes with little (several terabytes) storage but much computing power and archive nodes with low compute power but many terabytes of storage.

- HDFS-2832 [5]: change a datanode to a set of storages instead of a single node and blockreport each storage individually.

- HDFS-5682 [6]: expose an API to set the storage policy for a file and store blocks on storages of the desired type.

Initially (the first JIRA), HDFS only differentiated between two types of nodes. Later JIRAs added more types and moved the granularity from nodes to individual storages.

The HopsFS and HDFS implementation of heterogenenous storage are very similar, as one of the requirements for this work is to keep it compatible with HDFS. Since the implementation of HDFS is so similar to HopsFS, it will not be discussed in this section but in the next chapter. Important differences between HopsFS and HDFS will also be pointed out there. The next section will discuss an HDFS use case to illustrate the potential of heterogeneous storage.

3.2.1 HDFS Heterogeneous Storage Case Study

eBay reports using HDFS heterogeneous storage functionality to implement tiered storage [10, 43]. The rationale of such a setup is described in section 1.1.1.

The eBay cluster, which is briefly characterized in table 3.1, has two types of nodes: regular (compute) nodes and high-storage archival nodes. The cost per gigabyte of storage for the archival nodes is about one-fifth of the cost for regular nodes. The downside is that processing of data on the archival nodes is slower, because they have less computing power.

However, this need not be a problem if data stored on archival nodes is processed infrequently. As shown by the access pattern in figure 3.3, the access pattern is heavily skewed towards new data. Data older than a year is accessed at most a few times a month, new data is accessed tens of thousands times a month.

Blocks of frequently accessed files are stored on regular nodes. When files are accessed less frequently, only one copy of each block in the file is stored on a regular node and all other copies are stored on archival nodes. Eventually, when a file is accessed very infrequently, all copies of the blocks are stored on archival nodes.

The process of moving the blocks is automated; a policy can be derived from the age of the data but will likely depend on the type of data. For example; browser session logs (the access pattern of which is plotted in figure 3.3) become

	Regular node	Archival node
# drives	12	60
Storage capacity	40 TB	210 TB
# cores per node	32	4
Memory	128GB	64GB
Relative cost (per GB)	100%	20%
# nodes	2000	48
Combined Storage Capacity	80 PB	10 PB

Table 3.1: eBay cluster - regular vs. archival node [43]

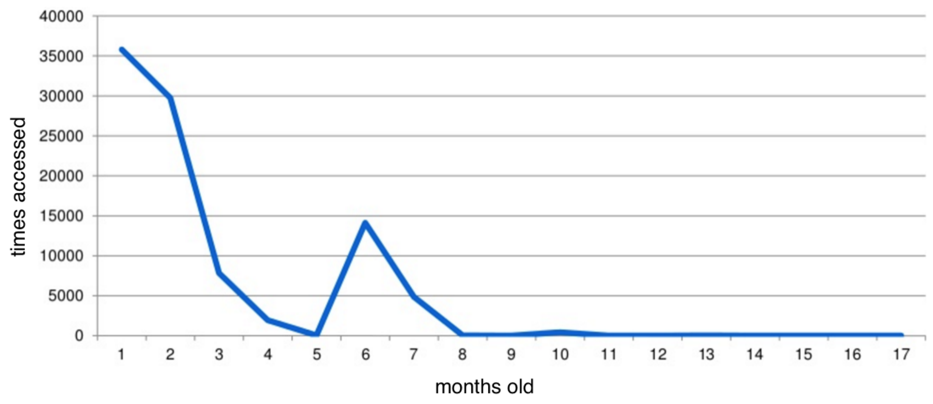

Figure 3.3: eBay access pattern for user browser session table [43]

almost irrelevant after more than a year, but the output of an analytics job may become irrelevant after a day. This requires manual tuning to reach an optimal strategy.

Moving the data from one node type to another can happen at high bandwidth. Since data is spread out over tens to thousands of nodes, data transfer can happen in parallel. eBay reports that it takes roughly twelve hours to move one petabyte from regular to archival storage, which means the throughput is almost 1.4 terabyte per minute. This means regular nodes have a average network traffic of 10Mbps and archival nodes close to 4Gbps per node. Thus, the performance degradation of the cluster is hardly noticable during the transfer of data to archive.

Interestingly, when eBay published this information the full heterogeneous storage functionality was already implemented. However, they still talk of types of nodes (instead of storages) and do not mention considering use of SSDs. This may be simply because their reason to start using heterogeneous storage was a lack of storage space, not a lack of performance.

Chapter 4

Method

This chapter discusses the heterogeneous storage implementation of HopsFS. Since it is very similar to the HDFS implementation, sometimes a comparison will be made to explain why we have or have not opted to follow the HDFS implementation.

We first define the goals for the implementation in section 4.1 and the delimitations in section 4.2.

Section 4.3 describes the implementation of heterogeneous storage support. It starts by describing internal changes that users do not notice directly, such as the internal application systematics (section 4.3.1) and the protocol (section 4.3.2).

After having described some of the internals section 4.3.3 and 4.3.4 present the new storage types and storage policies visible to end users. Section 4.3.5 explains how the storage policies work in combination with the exising block placement and volume-choosing policies.

Section 4.3.6 and 4.3.7 show how users can interact with the new features; either via the API or via the commandline tool. Section 4.3.8 describes the extensibility for adding new storage types and policies.

Lastly, section 4.4 discusses differences between the implementation of HopsFS and HDFS and the effects of these deviations.

This section focusses on changes made in this project, not on giving a description of the full HopsFS implementation. For more information on the design I refer to literature on HopsFS [23, 32, 47] and HDFS [11, 40].

4.1 Goals

The purpose of this work is to add support for heterogeneous storage types in HopsFS. This high-level purpose can be described as the following concrete goals:

1. Clients should be able to specify what storage types a file should be stored on.

2. If the desired storage type is not available, there should be a fallback or default storage type.

3. Administrators should be able to check whether desired storage types are used, and if not move blocks to the desired storage types.

The first goal will be implemented by specifying a list of storage policies, where each policy describes the desired state. That is, each policy describes what storage types a replica should be stored on.

In addition to specifying the desired state, a storage policy also specifies fallback storage types in case the desired type is unavailable. This implements the second goal.

The third goal is implemented in a Mover tool, described in section 4.3.7. This tool scans for policy violations and moves replicas to different storages to meet the policies.

To guarantee compatibility, the following two additional goals are defined:

4. The heterogeneous storage API on the client should be the same as for HDFS, to maintain compatibility between HDFS and HopsFS.

5. Legacy applications can continue to use HopsFS without being aware of heterogeneous storage features.

The fourth goal implies that HopsFS heterogeneous storage must offer the same API as HDFS. The HDFS API only specifies two new functions related to heterogeneous storage (for setting a policy and for requesting a list of available policies).

The fifth goal is satisfied by using sensible default values for all heterogeneous storage features. Specifying a storage type for storages on datanodes is optional (see section 4.3.3) and so is specifying the storage policy for files.

The goals specified so far lay the foundation of heterogeneous storage support, which can be further integrated in HopsFS to offer the following HopsFS-specific benefits (HDFS does not offer these benefits):

6. Integration with the Erasure-Coding library, to offer storage with multiple layers of fault-tolerance. Each node could store data on RAID volumes to tolerate disk failures and on top of that, erasure coding over multiple nodes to tolerate node or rack failures.

7. HopFS' metadata storage allows the system to track data accesses. This information can be used to automatically decide appropriate storage policies for files.

4.2 Delimitations

This project does not cover upgrading existing clusters; the amount of HopsFS clusters running in production is limited, so upgradeability did not have high priority. The database schema has changed significantly and cannot easily be upgraded. Specifically; the old schema maps replicas to datanodes, whereas the new schema maps replicas to storages. Unless each datanode contains only one storage (which is highly unlikely), this mapping cannot be deduced from the old database contents.

Existing configuration files are forward-compatible; default values will be assumed for any missing fields.

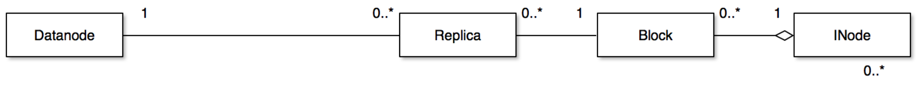

Figure 4.1: Old data model of datanodes, replicas, blocks and inodes.

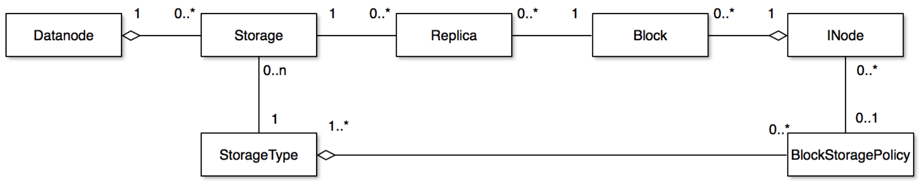

Figure 4.2: New data model, including storages, storage types and policies.

4.3 Implementation

This section provides information on the on the implementation of heterogeneous storage in HopsFS. It contains information both on what HopsFS end users can interact with -the commandline, API, storage types and storage policies- and on internal changes -the data model, protocol and interplay of block storage, block placement and volume-choosing policies.

4.3.1 HopsFS Systematics

In the old model, the namenode tracked which datanodes were present in the cluster. Each datanode mapped to a list of replicas stored on the datanode, and each replica was stored on exactly one datanode. Each replica belongs to a block and each block belongs to an inode. Each inode exists of zero or more blocks; empty files and directories are represented as inodes with no blocks. This schema is shown in figure 4.1.

In the new model, datanodes consist of a collection of storages. Each storage has a storage type (`DISK`, `SSD`, `ARCHIVAL` or `RAID5`) and a list of replicas stored on the storage. Each file has one storage policy; if it is not specified, it inherits the policy of the parent directory. Each storage policy is defined as three arrays of storage types; one array with the preferred types, one array with fallback types for creation and one with fallback types for replication. This schema is shown in figure 4.2.

From an end user perspective only the inodes, storage policies and storage types are visible. Users do not interact with individual blocks, replicas or storages; they only interact with files or directories. The system tracks metadata relevant to users (permissions, replication factor, storage policy) on the inode level.

4.3.2 Protocol

Communication between clients, namenodes and datanodes happens via remote procedure call (RPC) messages.

The most important steps of the protocol are discussed in this section. Messages that have not changed are not discussed unless there is a specific reason the reader should know about them. Acknowledgements will be ignored; typically all messages with meaningful content are acknowledged by the receiver. Names of the messages will be shortened, e.g. `GetStoragePoliciesRequestProto` is written as `getStoragePolicies` for readability.

Datanode Registration

A datanode registers by making a `registerDatanode` RPC to the namenode. The request contains some basic information about the datanode, such as its identifier and software version. The namenode stores a new entry for the datanode in the database, but will not use it for storage until it has reported a storage.

This RPC has not changed, it is only ment to clarify the process of discovering datanodes.

Block Reporting

Block reporting is a mechanism to synchronize the metadata stored on the namenode and the actual state of datanodes. A block report contains the datanode identifier, a list of storages on the datanode and a list of the block IDs of all blocks stored on each of these storages.

The first block report is sent immediately after the datanode registration is confirmed by the namenode. The block report of a newly formatted datanode contains only the datanode identifier and a list of storages, since there are no blocks to report.

When the namenode receives a block report for a storage that has not been reported before, it will create a new record for the storage. Once this has happened, blocks can be written to the storage.

After a block report, the namenode has up-to-date information on what blocks are stored on each of the reported storages.

A block report for a 24 terabyte datanode filled with 64 megabyte blocks could contain a list of up to 375,000 blocks. Processing such a large block report in a single transaction could put a significant peak load on the namenode. To avoid big these peaks, datanodes can choose to split the block report into multiple reports; one block report per storage. This results in a more uniform distribution of the load and is processed by the garbage collector more easily.

Heartbeating

Every datanode periodically sends a heartbeat to the namenode with current information about each of its storages and total usage. If a datanode fails to update the information on one or more of its storages, these storages will be marked as stale.

When a storage is reported as failed, the namenode will start the recovery process for all blocks on the failed storage.

If a datanode fails to heartbeat for a certain period, it is considered a failure and the namenode will initiate rereplication of the blocks on the namenode.

Creating and Writing to Files

This process requires multiple RPC messages, but since it is a fixed flow it is described in one go. Figure 4.3 shows the message flow as swimlane diagram.

Note that the failure recovery process is more complicated than the process described here. However, the recovery process has not changed significantly and as such will not be discussed here. I refer to documentation by Cloudera [52]

Clients make a `createRequestProto` RPC to the namenode that contains some metadata about the file that is to be created, including the path and the replication factor. If the request is valid, the namenode adds a new inode to the database. This message has not changed, although a difference between HopsFS and HDFS is that clients can specify whether the file should be erasure coded (see section 5.1.3).

To change the storage policy, clients can now make an `setStoragePolicy` RPC to the namenode, specifying the path of the file and the name of the desired policy. This only affects blocks for which `addBlock` has not been called yet; other blocks are only moved after calling the mover tool (section 4.3.7).

For every block a client writes, it first makes an `addBlock` RPC to the namenode. The namenode responds with a message that contains the datanodes responsible for storing the block, the identifiers of the storages to use on those datanodes and the types of those storage.

The clients make an `OpWriteBlock` RPC to the first datanode returned by the namenode. The message includes the types of storage that the block should be stored on.

The datanode pipes the block to the second node, which pipes it to the third, and so on. Clients wait until the current block is finalized (all acknowledgements from the datanodes are received) before calling `addBlock` again.

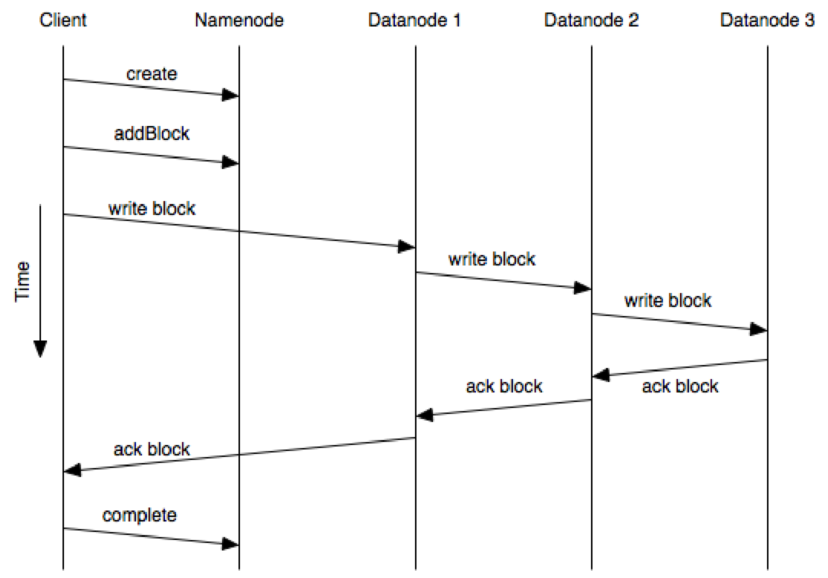

Figure 4.3: Hops/HDFS file creation/write protocol overview.

The code that executes the process of creating and writing to a file is given in listing 4.2 on page 31.

4.3.3 Supported Storage Types

The folders used on a datanode are specified by setting `dfs.datanode.data.dir`. Older versions of Hadoop already allowed adding multiple paths, which would be used in a JBOD setting; blocks would be randomly distributed over the folders. New HopsFS configurations allow the user to indicate a particular underlying storage type, e.g. `[DISK]file:///mnt/disk0`. There is no automatic detection of storage types. Storages that do not have a type specified are assumed to be `DISK`s, which is the default prior to heterogeneous storage.

The storage types supported are:

- `SSD`: flash storage with low latency/high throughput but small size.
- `DISK`: spinning disk with higher latency but larger size.
- `RAID5`: same as `DISK`, but higher fault tolerance and some storage overhead for the parity.
- `ARCHIVE`: bulk storage; cheap but lower performance.

It should be noted that HopsFS is unaware of the underlying implementation of storage types. There is no difference between a software RAID5 in ZFS/BtrFS and one implemented with a hardware raid controller.

Also, the storage type could reflect the type of node a storage is located on. If a cluster has some nodes for bulk storage, disks on these nodes could be marked as `ARCHIVE`, even though they are physically harddisks. This strategy, which is described in a Hadoop JIRA [7], allows administrators to add a few high-density storage nodes to increase the total storage capacity with minimal cost. Because the storages are marked `ARCHIVE`, clients can avoid reading from these nodes or executing tasks on them but instead use the normal nodes for these tasks. This allows the clusters storage capacity to grow independent of the clusters compute capacity.

4.3.4 Supported Storage Policies

Hops supports most of the storage policies that HDFS supports, and one that is not supported by HDFS. The list of storage policies in Hops is given in table 4.1.

The policy not supported by Hops is the `Lazy_Persist` policy, which stores one replica in the ram of a datanode. Since Hops does not support volatile storage, this policy does not apply to Hops. For more information, see section 4.4.2.

In addition to implementing the HDFS storage policies, Hops also supports a special `RAID5` storage policy. This policy implies a higher fault tolerance per storage than `DISK` (since a RAID5 storage can tolerate one single disk failure).

If no storage policy is defined for a file, the `Hot` strategy is assumed as a default value. This is most similar to how clusters used to work before heterogeneous storage; all replicas are stored on disk. This is in accordance with the fifth goal defined in section 4.1.

The HopsFS strategy for distributing replicas over storages is identical to the HDFS approach; it uses a block storage, block placement and volume-choosing policy. This is extensively described in section 4.3.5.

Policy name	Block placement	Fallback for creation	Fallback for replication
All_SSD	SSD: n	DISK	DISK
One_SSD	SSD: 1, DISK: $n-1$	SSD, DISK	SSD,DISK
Hot	DISK: n		ARCHIVE
Warm	DISK: 1, ARCHIVE: $n-1$	ARCHIVE, DISK	ARCHIVE, DISK
RAID5	RAID5: n	DISK	DISK
Cold	ARCHIVE: n		

Table 4.1: Supported storage policies

4.3.5 Storage, Placement and Volume-choosing Policies

There are three different policies that influence where a block is physically stored: the block storage, block placement and volume-choosing policy. The first two run on the namenode, the latter on the datanodes.

The concept of the block storage policy is newly introduced in this project, the old block placement policy in Hops is largely rewritten to incorporate the storage policy. The volume-choosing policy has not changed significantly.

Block storage policy

The block storage policy defines the storage types a block should be stored on.

A policy is defined as an ordered list of storage types of length n. For the i'th replica, the desired storage type is either the storagetype at position i (if $i \leq n$) or the storagetype at position n (if $i \geq n$). The storage policy does not specify any particular replication factor: this is independently handled by the placement policy.

To illustrate how this would work, consider the following examples:

- A policy {SSD, SSD, HDD} with replication factor of 1 stores one replica on an SSD.
- A policy {SSD, SSD, HDD} with replication factor $n \geq 2$ stores two replicas on an SSD and $n-2$ on a HDD.
- A policy {HDD} with replication factor n stores n replicas on HDDs.

If the cluster is used intensively, it could be that there is no available storage space on the desired storage type. To handle this scenario, each policy has two fallbacks: one for creation and one for replication. These fallbacks specify how a block should be stored if the desired storage type is not available.

Hadoop 2.6 introduced a new tool, Mover, to deal with mismatches between desired and actual state. This tool is discussed in section 4.3.7.

Block placement policy

The block storage policy specified a desired state, the block placement policy maps this to physical storages. It takes the following factors into account:

- the physicial location of storages, to avoid multiple replicas on one datanode.

- the physicial location of datanodes, to spread multiple replicas over multiple racks or data centres.
- the load on datanodes, to avoid peaks in load.
- whether a datanode/storage is a match according to the block storage policy.

Given a list of storages, a network topology, a storage policy and a replication factor, the placement policy will return a list of storage IDs where the block should be stored.

An important note is that datanodes do not have to abide the choice of storage; datanodes are allowed to store a block on a different storage of the same type (see the next section).

Volume choosing policy

When the datanode receives a `writeBlock` command from either the client or another datanode in the pipeline, the command includes the desired storagetype to be used to store the block on the receiving datanode. The namenode does not decide what storage the datanode should use; only which storage type.

A rationale for letting the datanode decide is that this allows for more flexibility with regards to, for example, load balancing. The namenode tracks the combined load on a datanode, but does not load-balance over multiple storages on a single datanode. By allowing the datanode to decide which storage to use, it could avoid writing to heavily used storages.

Currently, the only policy implemented in Hops is round-robin scheduling, which is implemented by `RoundRobinVolumeChoosingPolicy`. This effectively makes all storages of the same type a single JBOD: blocks are randomly distributed over all storages of the same type.

4.3.6 Heterogeneous Storage API

As specified by goal 4 in section 4.1, the API is intentionally kept the same as the HDFS heterogeneous storage API to maintain compatibility. Concretely, this means that the `DistributedFileSystem` must implement the first two methods shown in listing 4.1. The third method is offered in Hops as a convenient way to query for the currently used storage policy for a file.

following two methods:

```java
/** Set the storage policy for the specified path. */
void setStoragePolicy(Path src, String policyName) throws IOException;

/** Get all the existing storage policies. */
BlockStoragePolicy[] getStoragePolicies() throws IOException;

/** Get the effective storage policy for a file. */
public BlockStoragePolicy getStoragePolicy(Path src) throws IOException
    ;
```

Listing 4.1: Heterogeneous storage API in DistributedFileSystem

A code example of how the `setStoragePolicy` command could be used is given in snippet 4.2.

Furthermore, clients can query the storage policy of a file or directory by calling `getFileInfo(String src)`. This function already existed, but the returned `HdfsFileStatus` object is extended with a `getStoragePolicy()` function that returns the id of the storage policy. It returns the effective policy. This means that if there is no policy defined for the file, it will return the policy of the parent. If none of the parents have a policy defined, the default (Hot) is returned.

```java
// (supposing we have a DistributedFileSystem object called fs)

// Create the file with replication factor 3 (default)
FSDataOutputStream stream = fs.create("/file1.dat");

// Set the policy to All_SSD
fs.setStoragePolicy("/file1.dat",
    HdfsConstants.ALLSSD_STORAGE_POLICY_NAME);

// Write to file using ALL_SSD storage policy
stream.write(RandomUtils.nextBytes(fileSize), 0, fileSize);

// We could now change the policy again and write more blocks.
// The new blocks would be distributed according to the new policy.

stream.close();

// Get the policy id of the file (HDFS way)
HdfsFileStatus fileStat = fs.getFileInfo("/file1.dat");
byte policyId = fileStat.getStoragePolicy();

// Get the policy of the file (HopsFS way)
BlockStoragePolicy policy = fs.getStoragePolicy("/file1.dat")
```

Listing 4.2: Create file and set storage policy

4.3.7 Storage Policy Commands

Besides programmatic access through the API, HopsFS users can also manage storage policies via the commandline tool. The following options are added to the hdfs executable:

- `hdfs storagepolicies -listPolicies`
  Lists all policies available.

- `hdfs storagepolicies -setStoragePolicy -path <path> -policy <policy>`
  Sets the storage policy of a file/directory.

- `hdfs storagepolicies -getStoragePolicy -path <path>`
  Queries the storage policy of a file/directory. It returns the effective policy, which may be defined in a parent directory.

These operations only affect the metadata; they do not have an immediate effect on the stored data. Changing the storage policy does not initiate transfer of blocks to match the new policy. For that, there is the special mover tool, discussed in the next section.

Mover tool

Hadoop 2.6 introduced a new tool, similar to the Balancer, that scans for violations of the storage policy. For every block stored in the file system, it checks whether the actual storage types match the desired storage types. Reasons for a mismatch could be that the desired storage type was not available during creation of the block (in which case the fallback storage type would be used) or because the storage policy of a file has changed.

If there is a mismatch and the desired storage type is available, the Mover will create a new replica on the desired storage type and remove the mismatch.

The Mover program is called as follows: `hdfs mover [-p <files/directories>]`. This will scan for policy violations for all blocks belonging to the specified files.

If no files or folders are specified, the root folder is assumed. The Mover tool is not automatically triggered on a policy change, so a common operation would be to first change the policy for a directory and immediately after execute the Mover on the same directory.

Note that executing this command can be very time consuming, as a lot of data may be transfered over the network.

4.3.8 Extensibility

New policies and storage types can easily be added, although both require a recompilation of the code and as such cannot be done on a live cluster without rebooting.

All storage policies are defined in `BlockStoragePolicySuite`, the policy names and identifiers are defined in `HdfsConstants`.

New storage types must be defined in the `StorageType` enum and in `hdfs.proto`. Additionally, the conversion between the Java enum and the protobuf enum must be specified in `PBHelper#convertStorageType` (note that there are two functions; one for either direction).

4.4 Deviation from HDFS

The design of this work is very similar to the HDFS implementation of heterogeneous systems, but it has a few key differences which are discussed in the next sections.

4.4.1 Quotas

The new Hadoop versions extend the original implementation of quotas. In the old model (which is supported by HopsFS), administrators specify one storage quota per file or directory. In new versions of HDFS, administrators can set a quota per storage type per file or directory. For example, the `/tmp` directory could be limited to 100 gigabyte of SSD storage and 500 gigabyte of hard disk storage.

However, the storagetype-specific quotas introduce many new edgecases and raises new questions such as:

- Does a quota violation cause an error or do we use fallback storage types?

- Do we allow policy changes if the change causes an exceedance of a quota? Policy changes do not directly move blocks (the Mover tool does) so an alternative would be to let the Mover throw an error.
- If a storage medium supports data compression, do we count compressed or uncompressed data?
- Can an administrator lower the quota for a storage type if the lower quota would be exceeded? If so, what happens during a rereplication; is the new quota honored or does the system rereplicate to the original type and exceed the quota?

The HDFS team dedicated a full page to various usecases in their original design document [2]. Not all scenarios are discussed (e.g. changes in replication factor are not addressed) and complicated scenarios may arise. Additionally, not all possible scenarios are fully tested in the HDFS test suite (e.g. rereplication).

To avoid this added complexity, HopsFS does not yet include storagetype-specific quota.

4.4.2 Transient Storage

In addition to the `SSD`, `DISK` and `ARCHIVE` storage types, HDFS also has a `RAM_DISK` type. A ramdisk stores data in memory, which is very expensive but also far more performant than any other storage type. This storage type also has a property that other types do not; volatility. When a node is powered off, all data on the storage is lost.

The solution to the volatility is to also write the data to a non-volatile storage (such as DISK). There are two ways to do this: write-through or write-back. The first waits until data is written to disk and memory before acknowledging the write, the latter acknowledges as soon as data is stored in memory and lazily writes to disk.

Using the write-back strategy is more performant but may cause loss of data if the node is shut down before the data is written to disk. Read performance is equal for both strategies, since reads always happen from memory.

The HDFS team opted for the write-back strategy, trading durability for performance. If the `RAM_DISK` type is only used for intermediate results (i.e. if data can be recreated) and non-critical data, this can be a good trade-off.

HopsFS does not support the `RAM_DISK` storage type. However, the behaviour of the `RAM_DISK` can be mimicked by using ZFS features like the ARC and L2ARC cache [21]. This allows administrators to configure either memory or SSDs as caching layer in the local filesystem in a way that is opaque to HDFS or HopsFS. Given that ZFS is very thoroughly tested and widely used in production systems, choosing ZFS (with caching) as underlying layer for HopsFS is a good alternative to implementing transient storage ourselves.

The only storage policy in HDFS that uses the `RAM_DISK` is `Lazy_Persist`, which is also not supported in HopsFS.

4.4.3 RAID Storage Types

The RAID storage types (for now only RAID 5 is supported) allow cluster administrators to combine multiple physical disks into one logical unit to increase performance or reliability. Some RAID systems also allow combinations of multiple disk types to benefit from the performance characteristics of each type.

Administrators can use the `RAID5` storage type to provide nodes with storage type focussed on fault tolerance. One disk can fail opaquely to the system, so HopsFS does not have to start rereplication or recovery over the network; the node will handle it internally. Section 5.3.1 discusses how this can be used in combination with other fault tolerance mechanisms.

One classic argument against using large RAID 5 volumes is the chance of unrecoverable read errors (UREs) during recovery, since a URE could break the data recovery process. Disk vendors certify most consumer disks as having URE rate of one in 10^{14}, so 1 bit every 12.5 TB. Considering that RAID 5 volumes can be around 20 to 30 terabyte, one would expect an average of two UREs during recovery. In other words; almost no recovery would succeed, thus RAID 5 provides a false feeling of fault tolerance.

This argument is false for several reasons. Firstly, many systems succesfully use RAID 5; they do not experience problems during recovery. Secondly, the URE of most disks is much better than most disk vendors certify [20]. Lastly, modern filesystems like BtrFS and ZFS can continue recovering data when a URE occurs. Only the block where the URE occurred will be skipped.

That last point is important; if a RAID 5 storage of several terabyte fails and ZFS skips a few blocks while rebuilding the volume, HopsFS can still recover any blocks that were skipped due to UREs -supposing some other fault tolerance like erasure coding is used as well. Thus, even if UREs occur during recovery, only a few megabytes of network traffic are needed to restore the entire volume.

The real cost of data recovery in RAID 5 is a decreased performance on the node containing the data. ZFS can recover RAID-Z (which is comparable to RAID 5) volumes at a speed of roughly 100 MBps [3], so a 4 TB disk failing will take about 11 hours to restore. During that time performance on that node is degraded and blocks may be temporarily inaccessable. Thus, the `RAID5` storage type should only be used for data where the risk of a long latency is acceptable.

Chapter 5

Analysis

To analyse the result of this project, we give an evaluation of the goals defined for this work in section 5.1. To give a more formal evaluation, we also discuss the validation of the code in section 5.2. Section 5.3 discusses some considerations in choosing between storage space, fault-tolerance and recovery performance.

There is no experimental analysis of the newly added features. Benchmarking the performance of heterogeneous storage is difficult, as the performance varies wildly based on the type of workload running on the cluster. IO intensive jobs can be expected to benefit a lot from SSDs or hybrid storage, whereas CPU intensive jobs will not perform noticably faster.

Section 3.1 discusses benchmarks of MapReduce on SSDs, and shows how difficult a comprehensive benchmark is simply by the number of studies and variety in outcomes. For a ballpark estimate of how heterogeneous storage is expected to perform in HopsFS I refer to that section.

A usecase of heterogeneous storage for cost-saving through archival nodes is discussed in section 3.2.1. Everything discussed in that section applies to HopsFS as well as HDFS.

5.1 Evaluation of Goals

To evaluate the success of the implementation, we consider the goals specified in section 4.1. As a reminder for the reader, these goals are:

1. Clients can specify a desired storage type for a replica to be stored on.
2. Clients can specify a fallback storage type, in case the desired storage type is unavailable.
3. Replicas can be migrated from non-desired storage types to storage types.
4. The system must be with the HDFS API.
5. The system must support legacy applications.
6. Clients can combine heterogeneous storage with the Erasure-Coding library.
7. The system can automatically choose an appropriate policy for files based on usage statistics.

5.1.1 Heterogeneous Storage Support

The first three goals are completely satisfied; clients can specify policies that define the desired storage types (first goal) and fallback storage types (second goal). The mover tool moves any replicas that do not match the storage policy (third goal).

5.1.2 Compatibility

HopsFS uses the same API and commandline tools as HDFS. Thus, applications written for Hadoop are compatible with HopsFS (fourth goal).

Legacy clients use the default strategies, which store all replicas on hard disk. This is how both HopsFS and HDFS worked before the heterogeneous storage functionality, so legacy clients continue to work as they did before. Note that users can manually change the storage policy without the (legacy) applications noticing, so legacy applications can be set to use SSDs.

For example, an old version of HBase that is configured to store its data in `/hbase` can use SSDs if the user sets the `All_SSD` policy for `/hbase`. Since files inherit the policy from their parent, this will cause all HBase tables to be stored on SSD without changing any application code. Thus, even applications that are not designed to use heterogeneous storage can use it.

5.1.3 Erasure Coding and Automated Policies

The sixth goal, support for erasure coding in combination with heterogeneous storage is partially implemented. Users can set a storage policy for a file and then erasure code it, both the data blocks and the parity blocks will then have the storage policy. Since data blocks are not moved during the process of erasure coding, the blocks will not be redistributed. Since parity blocks are newly generated, the parity blocks will be distributed according to the storage policy. Calling the mover tool will also move the data blocks.

One limitation is that changing the storage policy of an erasure coded file will not update the policy of the parity blocks. Since the erasure coding library splits files into two files during the erasure coding process (a data file and a parity file), the storage policy has to be set for both files separately.

Clients wishing to set a storage policy for erasure coded files can look in `erasure-coding-default.xml` (on the local file system) to see the locations for parity files. These storage policies for both data and parity files can then be manually set through the commandline or API.

The seventh goal, automated policies, has not been implemented due to time constraints. Users can manually set the storage policies, which suffices for most usecases.

5.2 Validation

To validate the behaviour of the heterogeneous storage, the tests in table 5.1 were added. This is only a list of classes where new tests are added. In addition to this list, many of the old tests were adapted or extended to verify the behaviour of the newly added features.

Test Description	**Test Class**
Verify namenode behavior when a given DN reports multiple replicas of a given block.	`TestBlockHasMultipleReplicasOnSameDN`
Simulates situations when blocks are being corrupted, modified, etc. before a block report happens. For each test case it runs two variations: one block report for all storages and one block report per storage.	`TestBlockReport`
Verify parsing of datadir config files, e.g. whether `[DISK]/mnt/disk1` is parsed correctly.	`TestDataDirs`
Verify that block verification occurs on the datanode.	`TestDatanodeBlockScanner`
Verify that the datanode Uuid is correctly initialized before `FsDataSet` initialization.	`TestDataNodeInitStorage`
Verify that after a volume failure: • Blocks are marked underreplicated • Other volumes on DN are still usable • Number of blocks and files has not changed	`TestDataNodeVolumeFailure`
Verify that datanodes can correctly handle errors during block read/write.	`TestDiskError`
Test behaviour of appending to file.	`TestFileAppend4`
Verify that policies are correctly set, and that policies of parents are correctly inherited.	`TestFileCreateWithPolicies`
Verify that incremental block reports from a single DataNode are correctly handled by NN. Tests the following variations: 1. Incremental BRs from all storages combined in a single call. 2. Incremental BRs from separate storages sent in separate calls. 3. Incremental BR from an unknown storage should be rejected.	`TestIncrementalBrVariations`
Test conversion of `DatanodeStorage` and `LocatedBlock`.	`TestPBHelper`
Test replication with sufficient amount of nodes but only one rack available.	`TestReplicationPolicy`
Verify that datanodes correctly report all storages and storage types.	`TestStorageReport`

Table 5.1: Tests added

Due to changes in behaviour, many tests were updated to reflect the new behaviour. Some tests became obsolete because they test functionality that is now covered by other tests. For that reason, the tests in table 5.2 have been removed.

Removed test	Reason for removal
`TestBlockReport.blockReport_01`	Replaced to cover more scenario's
`TestBlockReport.blockReport_02`	Replaced to cover more scenario's
`TestBlockReport.blockReport_03`	Replaced to cover more scenario's
`TestBlockReport.blockReport_06`	Replaced to cover more scenario's
`TestBlockReport.blockReport_07`	Replaced to cover more scenario's
`TestBlockReport.blockReport_08`	Replaced to cover more scenario's
`TestBlockReport.blockReport_09`	Replaced to cover more scenario's
`TestDataDirs.testGetDataDirsFromURIs`	Replaced by more extensive tests

Table 5.2: Tests removed

5.3 Heterogeneous Storage and Erasure Coding

One of the goals posed in section 4.1 is the integration of the erasure coding library in HopsFS.

This library implements fault tolerance over nodes with more sophisticated strategies than simply storing multiple copies of data. There are three erasure coding strategies currently supported: RAID5, Reed-Solomon (RS) and Locally Repairable Codes (LRC) [22]. What these codes have in common is that they produce parity blocks that can be used to restore the original data blocks. The ratio between number of data blocks and parity blocks is configurable.

For example, Reed-Solomon (14,10) takes 10 data blocks and produces an additional 4 parity blocks. If up to four of the blocks (data or parity) are lost, the remaining 10 can be used to restore all missing blocks.

The advantage of using these erasure codes is that they allow many failures (up to 4 in case of Reed-Solomon (14,10)) while having minimal storage overhead. The HDFS default fault tolerance strategy of triple replication has a 200% overhead and withstands any two blocks to be corrupted. Reed-Solomon (14,10) has a 40% overhead and withstands any four blocks to be corrupted.

The disadvantage is that to recover blocks, many of the remaining blocks have to be processed. Thus, restoring blocks generates a lot of network traffic. If a 4 terabyte disk in a Reed-Solomon (14,10) configuration fails, restoring all blocks on that node generates 40 terabyte of network traffic.

To reduce the amount of recoveries caused by disk failures, one could use a RAID 5 on individual nodes to tolerate disk failures.

5.3.1 Balancing Fault Tolerance, Storage Overhead and Efficient Data Recovery

HopsFS supports many techniques for fault tolerance; RS, LRC, RAID5 over multiple nodes, RAID5 as storage type and replication. This begs the question of which technique(s) to use in a cluster. To shed some light on this question,

Technique	Storage overhead	Fault tolerance		Repair Traffic	
		#storages	#nodes	Failed storage	Failed node
No replication	0%	0	0	-	-
Triple replication	200%	0	2	1x	1x
RAID5 (storage type)	20%	1	0	0x	-
RAID5 (EC library)	20%	0	1	5x	5x
RS(10,14)	40%	0	4	10x	10x
LRC(10,6,5) [37]	60%	0	4	5x	5x
RS(10,4)+RAID5	68%	1	4	0x	10x
LRC(10,6,5+RAID5	92%	1	4	0x	5x

Table 5.3: Comparison of fault tolerance techniques. RAID5 is assumed to have 6 disks resp. nodes for local resp. distributed. RAID5 combinations are using the RAID5 storage type (not EC library). Fault tolerance is defined as number of storages per node that fail without causing node failure and as number of nodes that can fail without data loss. Repair traffic is expressed as a multiplier of the amount of data on the failed storage/node. I.e. a 10 TB node failure with 5x multiplier causes 50TB of network traffic for recovery.

this section discusses how each technique balances three things: low storage overhead, high fault tolerance and efficient data recovery.

Table 5.3 describes the characteristics of most fault tolerance techniques supported by HopsFS. Figure 5.1 gives a visual representation of how these techniques compare.

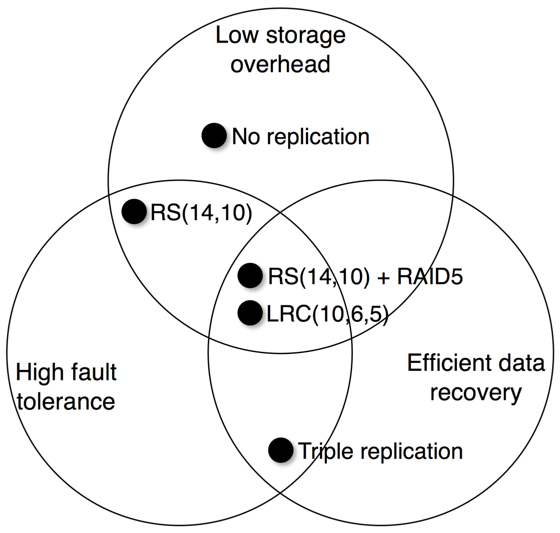

Figure 5.1: Visual representation of fault tolerance, storage overhead and efficient data recovery. See caption of table 5.3 for explanation.

It should be noted that this figure cannot accurately represent the details. For example, RS+RAID5 and LRC are both in the intersection of all three circles, indicating that they satisfy every description to some degree. However, they are both less efficient in recovering data than triple replication is.

RS+RAID5 is very efficient at restoring data on a single disk failure, as the RAID5 storage has parity data stored on the same disk as the actual data and requires no network traffic for recovery. However, node failures require it to use RS to recover data, which is very inefficient.

Likewise, triple replication is fault tolerant, up to two node failures, but RS is even more fault tolerant, up to four node failures. The most fault tolerant technique is RS+RAID5 (up to four node failures and remaining nodes can have one disk failure each).

One more aspect that is not reflected in these characteristics is the mean time to (data) loss (mttl). This depends on the level of fault tolerance, but also on the recovery time and the failure model (i.e. whether failures are correlated). This becomes very difficult to model, especially when considering storage vs. node vs. rack failures. Interested readers are referred to studies by Schroeder et al. [38, 39] for more information on modelling the MTTL.

Using a RAID5-like storage type like ZFS RAID-Z in combination with a low-storage overhead fault tolerance technique like RS(14,10) can be an attractive new option. Several studies [34, 38] show that clusters still have many disk failures. Reed-Solomon data recovery causes these disk failures to result in very expensive data recoveries. A study by Sathiamoorthy et al. states that data recovery accounts for 10 to 20% of the traffic in Facebook clusters that use Reed-Solomon erasure coding. RAID5 makes nodes resistant against single disk failures and can reduce the amount of network traffic significantly.

Chapter 6

Conclusions

This chapter states the final thoughts on the work presented in this thesis. Section 6.1 summarizes and concludes the project described in this thesis. Section 6.2 presents some directions for future work on heterogeneous storage.

6.1 Conclusion

Heterogeneous storage support allows cluster users to better leverage the different performance characteristics disks in the cluster. By supporting multiple storage types, HopsFS is more flexible and adaptable to the requirements of the various workloads running on Hops clusters.

By mixing SSDs and hard disks, administrators can update their cluster to follow the trend of running interactive query processing on Hadoop. Tests on HDFS indicate a great potential for performance increase; Cloudera observed a 70% increase in performance for IO bound syntetic tests [26] and Harter et al. trippled the performance for specific workloads on Facebook clusters [24].

Not all workloads benefit this much from using flash storage, but there is a great potential. Rapidly dropping prices of SSDs [35] make using hybrid storage an option everyone should look into when setting up a new cluster or expanding an existing cluster.

For clusters that fill up with data that is accessed very infrequently, the newly added archival storage provides a way of cheaply adding bulk storage to clusters. eBay reports an 80% reduction in price per gigabyte for data stored on specialized archival nodes [10, 43]. Although these nodes perform not too well for data processing, the cheap hardware allows users to store more data on clusters.

Especially in combination with HopsFS, which allows far larger file systems than HDFS [23], adding cheap storage to clusters is essential. Where HDFS file systems cannot grow beyond 100 terabyte to 1 petabyte, HopsFS clusters can scale to an order of magnitude more. When dealing with file systems of this size, the price per gigabyte of storage can be a key differentiator.

Combining archival storage with the erasure coding [22] offers even greater cost savings. Erasure coding offers reliable storage with minimal storage overhead at the cost of increase computional cost and network traffic for recovering data. Using the `RAID5` storage policy, there is a higher level of fault tolerance on a node-level. This reduces the amount of expensive recovery operations, since only node failures have to resort to erasure coding recovery; disk failures are resolved with the RAID 5 recovery.

6.2 Future Work

This thesis provides an implementation of heterogeneous storage support, but is not completely finished. It must be further tested and integrated before it reaches its full potential. This section proposes some extensions that are omitted in the implementation due to time constraints.

6.2.1 Further Testing and Benchmarking

Although the API is tested with unit tests and simple integration tests, the commandline interface is not tested. Also, the system is not tested with real workloads nor on a large scale. Before deploying the system as it is today in production environments, it should be tested more extensively to reveal undiscovered bugs.

The heterogeneous storage implementation has not been benchmarked yet. To really put the system to the test, it would be good to benchmark the system with several standardized benchmarks and real workloads. A promising suite for synthetic benchmarking is HiBench [25].

Some questions that are of particular interest are:

- How does the performance of HopsFS with hybrid storage (i.e. SSD and hard disk) compare to HopsFS on hard disk only?
- How does the performance of HopsFS compare to HDFS?
- How does caching on a local file system level, e.g. ZFS' L2ARC caching on SSDs [1], affect performance?
- How does data recovery in the local file system (e.g. RAID-Z resilverin in ZFS) affect the performance of the cluster?
- How do various RAID configurations affect performance?

The first question is already partially discussed and answered in several papers, albeit for HDFS rather than HopsFS. Running similar benchmarks for HopsFS should prove whether the implementation behaves similar to HDFS - as we would expect it to.

Testing heterogeneous storage for various RAID configurations seems like a novel research area. In particular, it would be interesting to see how the number of clients affect the performance of the different RAID levels, for example a RAID-0 versus RAID-5 versus JBOD configuration for 10, 100 and 1000 clients.

Since an increase in number of clients also increase the amount of disk seeks, it might show that some configurations have higher throughput for few clients but degrade more quickly when the number of clients increases. Also, it might show that some configurations strongly favor reads over writes.

6.2.2 Storagetype-specific Quotas

As discussed in section 4.4.1, HopsFS does not support quotas for specific types. It is possible to restrict the size of a folder but not to restrict the size per storage type, e.g. maximum of 20 GB on `SSD` and 40 GB on `DISK` (which would be a 20 GB folder with triple replication and `One_SSD` policy). Thus, there is no difference between using a terabyte of hard disk space and using a terabyte of SSD space, even though the latter is considerable more expensive.

Extending the existing system for quotas to account for the differences in storage types would be a valuable addition to the current implementation.

6.2.3 Automated Adaptive Storage Policies

The current implementation requires users to manually set policies through the commandline, or programmers to add support for applications through the heterogeneous storage API. However, it would be beneficial to the system if it could tune itself to automatically select storage policies for files.

HopsFS has an advantage over HDFS on this point, as HopsFS allows for more scaleable metadata storage. It is relatively easy to track when and how frequently files are accessed, and derive the appropriate storage policy from this information.

The main challenge identified for this project is to add adaptive storage policies without complicating the system for the end users. Any haziness in the workings of the system can result in unexpected behaviour. The logics behind the automatically adapting policies must be clearly understandable and predictable for administrators.

Bibliography

[1] B. aDam LeVenthaL. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[2] A. Agarwal, S. Radia, S. Srinivas, and T.-W. Sze. Heterogeneous Storage for HDFS. *https://issues.apache.org/jira/secure/attachment/12615761/20131125-HeterogeneousStorage.pdf.*

[3] M. Ahrens. Openzfs: a community of open source zfs developers. *AsiaBSDCon 2014*, page 27, 2014.

[4] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.

[5] Apache Software Foundation. HDFS JIRA 2832 - Enable support for heterogeneous storages in HDFS. *https://issues.apache.org/jira/browse/HDFS-2832.* Online: accessed 09-06-2016.

[6] Apache Software Foundation. HDFS JIRA 5682 - Heterogeneous Storage phase 2. *https://issues.apache.org/jira/browse/HDFS-5682.* Online: accessed 09-06-2016.

[7] Apache Software Foundation. HDFS JIRA 6584 - Support Archival Storage. *https://issues.apache.org/jira/browse/HDFS-6584.* Online: accessed 09-06-2016.

[8] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 20. ACM, 2013.

[9] Backblaze. Storage Pod 6.0: Building a 60 Drive 480TB Storage Server. *https://www.backblaze.com/blog/open-source-data-storage-server/.* Online: accessed 09-06-2016.

[10] A. Benoy. HDFS Storage Efficiency Using Tiered Storage. *http://ebaytechblog.com/2015/01/12/hdfs-storage-efficiency-using-tiered-storage/.* Online: accessed 08-07-2016.

[11] D. Borthakur. The hadoop distributed file system: Architecture and design. *The Apache Software Foundation*, 2007.

[12] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[13] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[14] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[15] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280. ACM, 2007.

[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *OSDI*, pages 61–74, 2010.

[18] L. George. *HBase: the definitive guide.* " O'Reilly Media, Inc.", 2011.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[20] J. Gray and C. Van Ingen. Empirical measurements of disk failure rates and error rates. *arXiv preprint cs/0701166*, 2007.

[21] B. Gregg. Zfs l2arc. *Fishworks Engineering, dated Jul*, 22, 2008.

[22] S. Grohsschmiedt. Making big data smaller: Reducing the storage requirements for big data with erasure coding for hadoop. 2014.

[23] K. Hakimzadeh, H. P. Sajjad, and J. Dowling. Scaling hdfs with a strongly consistent relational model for metadata. In *Distributed Applications and Interoperable Systems*, pages 38–51. Springer, 2014.

[24] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, Santa Clara, CA, 2014. USENIX.

[25] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.

[26] K. Kambatla and Y. Chen. The truth about mapreduce performance on ssds. In *28th Large Installation System Administration Conference (LISA14)*, pages 118–126, 2014.

[27] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[28] D. Laney. 3-d data management: Controlling data volume, variety and velocity. *META Group File*, 949, 2001.

[29] K. McKusick and S. Quinlan. Gfs: evolution on fast-forward. *Communications of the ACM*, 53(3):42–49, 2010.

[30] M. Moshayedi and P. Wilkison. Enterprise ssds. *Queue*, 6(4):32–39, 2008.

[31] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating enterprise storage to ssds: analysis of tradeffs. *In Proceedings of EuroSys 09*, 2009.

[32] S. Niazi, M. Ismail, S. Grohsschmiedt, M. Ronström, S. Haridi, and J. Dowling. Hopsfs: Scaling hierarchical file system metadata using newsql databases. *arXiv preprint arXiv:1606.01588*, 2016.

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[34] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, volume 7, pages 17–23, 2007.

[35] H. K. Ramapriyan. Evolution of archival storage (from tape to memory). In *Manual of Remote Sensing (MRS), 4th Edition.* The American Society of Photogrammetry and Remote Sensing, 2015.

[36] P. Russom et al. Big data analytics. *TDWI Best Practices Report, Fourth Quarter*, pages 1–35, 2011.

[37] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[38] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *FAST*, volume 7, pages 1–16, 2007.

[39] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):337–350, 2010.

[40] K. Shvachko, H. 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.

[41] K. V. Shvachko. Hdfs scalability: The limits to growth. *;login: The Magazine of USENIX*, 35(2):6–16, 2010.

[42] K. V. Shvachko. Apache hadoop: the scalability update. *;login: The Magazine of USENIX*, 36:7–13, 2011.

[43] T.-W. Sze and A. Benoy. Reduce Storage Costs by 5x Using The New HDFS Tiered Storage Feature.
Slides: *http://www.slideshare.net/Hadoop_Summit/reduce-storage-costs-by-5x-using-the-new-hdfs-tiered-storage-feature*
Video: *https://www.youtube.com/watch?v=cUTvklVTWfw*. Online: accessed 08-07-2016.

[44] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[45] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[46] D. Vesset, B. McDonough, M. Wardley, and D. Schubmehl. Worldwide business analytics software 2012-2016 forecast and 2011 vendor shares. *IDC Market Analysis*, 2012.

[47] M. Wasif. A distributed namespace for a distributed file system. 2012.

[48] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[49] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006.

[50] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 4. IEEE Computer Society, 2004.

[51] T. White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[52] Y. Zhang. Understanding HDFS Recovery Processes. *http://blog.cloudera.com/blog/2015/02/understanding-hdfs-recovery-processes-part-1/*. Online: accessed 11-07-2016.