DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

S3-HopsFS: A Cloud Native Distributed File System

Scalable, low-cost file system designed for the cloud

JOEL STENKVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

S3-HopsFS: A Scalable Cloud-native Distributed File System

JOEL STENKVIST

jstenk@kth.se

Master in Computer Science and ICT Innovation
Date: June 24, 2019
Supervisor: Gautier Berthou and Jim Dowling
Examiner: Vladimir Vlassov
School of Electrical Engineering and Computer Science
Host company: Logical Clocks
Swedish title: S3-HopsFS: En Skalbar Distribruerad Fil System med Moln Lagring

Abstract

Data has been regarded as the new oil in today's modern world. Data is generated everywhere - from how you do online shopping to where you travel. Companies rely on analyzing this data to make informed business decisions and improve their products and services. However, storing this massive amount of data can be very expensive. Current distributed file systems rely on commodity hardware to provide strongly consistent data storage for big data analytics applications, such as Hadoop and Spark. Running these storage clusters can be very costly; it is estimated that storing 100 TB in an HDFS cluster with AWS EC2 costs $47,000 per month. On the other hand, using cloud storage such as Amazon's S3 to store 100 TB only costs about $3,000 per month - however S3 is not sufficient due to eventual consistency and low performance. Therefore, combining these two solutions is optimal for a cheap, consistent, and fast file system.

This thesis outlines and builds a new class of distributed file system that utilizes cloud native block storage as the data-layer, such as Amazon's S3. AWS recently increased the bandwidth from S3 to EC2 from 5 Gbps to 25Gbps, sparking new interest in this area. The new system is built on top of HopsFS; a hierarchical, distributed file system with a scale-out metadata layer utilizing an in-memory, distributed database called NDB - which dramatically increases the scalability of the file system. In combination with native cloud storage, this new file system reduces the price of deployment by up to 15 times, but at a performance cost of 25% of the original HopsFS system (four times slower). However, tests in this research shows that S3-HopsFS can be improved towards 38% of the original performance by comparing it with only using S3 by itself.

In addition to the new HopsFS version, S3Guard was developed to use NDB instead of Amazon's DynamoDB to store the file tree hierarchy metadata. S3Guard is a tool that allows big data analytics applications such as Hive to utilize S3 as a direct input and output source for queries. The eventual consistency problems of S3 have been solved and tests show a 36% performance boost when listing and deleting files and directories. S3Guard is sufficient to support some big data analytic applications like Hive, but we lose all the benefits of HopsFS like the performance, scalability, and extended metadata - therefore we need a new file system combining both solutions.

Sammanfattning

Data har ansetts vara den nya oljan i dagens moderna värld. Data kommer från överallt - från hur du handlar online till var du reser. Företag är beroende på analysering av denna data för att kunna göra välgrundade affärsbeslut och förbättra sina produkter och tjänster. Det är väldigt dyrt att spara denna enorm mängd av data för analysering. Nuvarande distribuerade filsystem använder vanlig hårdvara för att kunna ge stark och konsekvent datalagring till stora dataanalysprogram, som Hadoop och Spark. Dessa lagrings kluster kan kosta väldigt mycket. Det beräknas att lagra 100 TB med ett HDFS-kluster i AWS EC2 kostar $47 000 per månad. På andra sidan kostar molnlagring med Amazons S3 bara cirka $ 3 000 per månad för 100 TB, men S3 är inte tillräckligt på grund av eventuell konsistens och låg prestanda. Därför är kombinationen av dessa två lösningar optimalt för ett billigt, konsekvent och snabbt filsystem.

Forskningen i denna thesis designar och bygger en ny klass av distribuerat filsystem som använder cloud blocklagring som datalagret, som Amazonas S3, istället för vanlig hårdvara. AWS ökade nyligen bandbredd från S3 till EC2 från 5 Gbps till 25Gbps, som gjorde ett nytt intresse i det här området. Det nya systemet är byggt på toppen av HopsFS; ett hierarkiskt, distribuerat filsystem med utökad metadata som utnyttjar av en in-memory-distribuerad databas som heter NDB som dramatiskt ökar filsystemets skalbarhet. I kombination med inbyggd molnlagring minskar detta nya filsystem priset för implementering upp till 15 gånger, men med en prestandakostnad på 25 % av det ursprungliga HopsFS-systemet (den är fyra gånger långsammare). Test i denna undersökning visar dock att S3-HopsFS kan förbättras till 38% av den ursprungliga prestandan genom att jämföra den med bara användning av S3.

Förutom den nya HopsFS-versionen, utvecklades S3Guard för att använda NDB istället för Amazons DynamoDB för att spara fil systemets metadata. S3Guard är ett verktyg som tillåter stora dataanalysprogram som Hive att använda S3 istället för HDFS. De eventuella konsekvensproblemen i S3 är nu lösta och tester visar en 36% förbättring av prestanda när man listar och tar bort filer och kataloger. S3Guard är tillräckligt för att stödja flera dataanalys program som Hive, men vi förlorar alla fördelar med HopsFS som prestanda, skalbarhet och utökad metadata. Därför behöver vi ett nytt filsystem som kombinerar båda lösningarna.

Acknowledgements

I would like to thank my supervisors prof. Jim Dowling and Dr. Gautier Berthou for their hard work in guiding me in this research, as well as all the employees of Logical Clocks. It was truly an exciting, fast-paced environment to work in with a high regard for team work. Specifically, Dr. Berthou spent many hours over the past few months discussing the design, problems, solutions, and related work. I would also like to thank my examiner Vladimir Vlassov for taking the time to help. Finally, I also thank you for reading this material!

June 24, 2019
Joel Stenkvist

Contents

1 Introduction **1**
- 1.1 Problem Definition . . . 2
- 1.2 Purpose . . . 3
  - 1.2.1 Cloud Computing . . . 3
- 1.3 Research Question . . . 4
- 1.4 Contributions . . . 4
- 1.5 Ethics and Sustainability . . . 4
- 1.6 Goals and Research Methodology . . . 6
- 1.7 Delimitations . . . 6

2 Background **7**
- 2.1 HopsFS . . . 7
  - 2.1.1 HDFS vs HopsFS . . . 8
  - 2.1.2 MySQL Cluster NDB . . . 9
  - 2.1.3 NameNode . . . 11
  - 2.1.4 DataNode . . . 12
- 2.2 Disk Architecture . . . 15
  - 2.2.1 Shared Disk . . . 15
  - 2.2.2 Shared Nothing . . . 15
- 2.3 Cloud Object Stores - Amazon S3 . . . 16
  - 2.3.1 Eventual Consistency . . . 16
  - 2.3.2 S3Guard . . . 17
- 2.4 Related Work . . . 18
  - 2.4.1 Mounting external storage in HopsFS . . . 18
  - 2.4.2 WekaIO - MatrixFS . . . 19

3 Implementation **20**
- 3.1 S3Guard . . . 20
- 3.2 New S3 DataNode in HopsFS . . . 22

3.2.1 Updated Read and Write Path . . . 22
3.2.2 Code Architecture (UML Diagram) . . . 24
3.2.3 S3DatasetImpl - Core Functionality of S3 DataNode . . . 25
3.3 Eventual Consistency Problems . . . 26
3.3.1 Enabling Consistent Reads and Writes . . . 26
3.3.2 Consistency Problem - S3 Block is Missing . . . 27
3.3.3 Consistency Problem - S3 Block Exists . . . 28
3.3.4 Block State Race Conditions . . . 28
3.4 Required Changes of Existing Packages . . . 30
3.4.1 S3A Connector . . . 30
3.4.2 S3A Circular Package Dependency . . . 31

4 Evaluation **32**
4.1 Performance of S3Guard . . . 32
4.2 S3 HopsFS Performance Analysis . . . 33
4.3 Performance of new S3 DataNode . . . 34
4.3.1 Create Workload . . . 34
4.3.2 Read Workload . . . 35
4.4 Network Performance . . . 37
4.4.1 Network Performance of S3 Alone . . . 37
4.4.2 Network Performance of S3 DataNode . . . 38
4.5 Cost of Cloud Storage . . . 39
4.6 Future Work . . . 41

5 Conclusion **43**
5.1 Results . . . 43
5.2 Motivation . . . 44

A Extra Results **45**

B How To Run **49**

Bibliography **52**

List of Figures

2.1 Architecture of HDFS compared to HopsFS [16] 8
2.2 Scaling beyond HDFS metadata capability [17] 10
2.3 Block States from the NameNode's perspective [19] 12
2.4 Replica States from the DataNode's perspective 14
2.5 How clients interact with S3 [7] 18

3.1 Design of S3 Guard in HopsFS 21
3.2 Flow of writing a new block with HDFS compared to standalone S3. 23
3.3 UML Class Diagram of new S3 DataNode 25
3.4 Potential read after write consistency problem 27
3.5 Steps taken when a block is missing in S3 28
3.6 Steps taken for each replica operation when a block in S3 already exists 28
3.7 Appending to a deleted block that still exists in S3 is not possible 30

4.1 S3Guard using DynamoDB vs NDB in AWS for listing files and directories 33
4.2 Creating 1 MB files on AWS EC2 instances with different number of clients 35
4.3 Reading 1 MB files on AWS EC2 instances with different number of clients 36
4.4 Reading and writing 1 MB files from S3 with 10 clients 38
4.5 Network performance of S3 HopsFS DN. The first half includes a create workload (purple line) and the second half is read (blue). The loopback interface (red) shows the same data being sent to the client. 39
4.6 Cost of storing 100 TB for 1 month 40
4.7 Cost calculation of 100 TB over 1 month 40

A.1 CPU utilization during workloads using 1 MB files with S3-HopsFS on AWS. Explanation for create workload bottleneck in section 4.3.1. The workload is running CREATE from the start to 13;03;19 then READ from 13;03;19 until the end. . . . 45
A.2 Disk utilization during workloads of 1 MB files with normal HopsFS on AWS. Explanation for read workload bottleneck in section 4.3.2. The workload is running CREATE from the start to 13;03;19 then READ from 13;03;19 until the end. . . . 46
A.3 Network performance of adding thread synchronization to S3-only test with 10 clients reading in parallel and locking on a get object metadata call. 46
A.4 Network performance during read of the static HopsFS cluster with 3 Datanodes and a replication factor of 3. 47
A.5 Time comparison of writing 30 10mb files in AWS EC2 47
A.6 Time breakdown of writing and reading 30 files with 10mb size in AWS. Note that most of the write phase (dark blue) is spent uploading for S3-HopsFS and S3. 48

List of Acronyms

ANN	Active NameNode
AWS	Amazon Web Services
DN	DataNode
DFS	Distributed File Systems
EMR	Elastic Map Reduce
EBS	Elastic Block Store
GDPR	General Data Protection Regulation
GS	Generation Stamp
GFS	Google File System
HDFS	Hadoop Distributed File System
HDD	Hard Disk Drive
HopsFS	Hadoop Open Platform-as-a-Service
NN	NameNode
NFS	Network File System
NDB	Network Database
POSIX	Portable Operating System Interface
RBW	Replica Being Written
RWR	Replica Waiting to be Recovered
RUR	Replica Under Recovery
RAID	Redundant Array of Independent Disks
RF	Replication Factor
S3	Simple Storage Service
SAN	Storage Area Network
SbNN	Standby NameNode
SMB	Server Message Block
SQL	Structured Query Language
SSD	Solid State Drive
VPC	Virtual Private Cloud

Chapter 1

Introduction

The world we live in today is increasingly becoming more dependent on technology that we take for granted, such as banking services, transportation, shopping, and communication. Companies today are spending fortunes on the digitalization of these processes through software and hardware. One common problem that has existed since the dawn of computing is storing data, or storing the information that these systems consume - such as customer information, photos, communication logs, video content, and much more. Traditional systems rely on hard drives located on the users computer or on a distant server, however, this approach that we take for granted today is actually extremely complicated. It is estimated today that the size of the internet is well beyond 1200 Petabytes - or 1.2 million Terabytes [1]! This estimate only includes the Big Four - Google, Amazon, Microsoft and Facebook (other sites such as Dropbox were not included in the estimate). To store and process this amount of data, complex solutions are required that companies and our world depend on today to function properly. These big data systems need to be fast, fault-tolerant, economically viable, and maintainable.

This massive store of data can be is very useful to understand how products and services are used and enjoyed by consumers. As such, information and data is the new oil in today's digital society. Companies are interested in learning more about what customers want, where they shop, and how they spend their money. This allows them to reach target markets better. Traditionally, marketing intelligence has relied on surveys to understand consumer behaviour, such as customer satisfaction surveys [2]. However, technological advances in the past decade has allowed this data to be mined and analyzed; a field commonly known as big data analytics.

Use cases for analyzing large amounts of data include customer segmenta-

tion and profiling, telecommunications, marketing analysis, recommender systems, and location-based advertising [2]. In telecommunications, call logs can be analyzed to cluster groups of consumers together in order to better understand their behaviour to improve operations, marketing, and sales of telecom providers [3]. Recommender systems can be used to offer consumers products and services that they enjoy personally, such as personal music recommendations at Spotify [4]. Location based advertising can be used to promote local businesses on mobile devices, such as pizza delivery services [5]. All of these use cases are powered by big data analytics, which needs massive compute and storage capabilities that offer high throughput and fault tolerance to analyze the data in real-time. Two common storage solutions include Distributed File Systems (DFS) and Cloud Block Storage such as Amazon's S3.

1.1 Problem Definition

Current Distributed File Systems rely on commodity hardware, such as any DELL server with spinning disk storage devices, to provide strongly consistent data storage for big data analytics applications, such as Hadoop and Spark. One of the most popular file systems for these big data applications is the Hadoop Distributed File System (HDFS), which runs on commodity hardware to provide high-throughput data access to files. Even though the usage of commodity hardware is much cheaper than using traditional network storage filers (such as NFS and SAN) with expensive dedicated hardware, running large HDFS clusters to accommodate for large datasets can still get expensive; Revx.io reports that running a 100 TB Hadoop cluster with EC2 on Amazon Web Services (AWS) is around $47,000 per month [6].

Native Cloud Storage solutions like Amazon's S3 can also be used as storage for big data apps because the API appears like a file system with create, read, and delete. Although these systems are much cheaper, they are generally much slower and do not provide consistency - meaning when you commit a file it may not appear until later. This is bad for big data analytics apps like Hive. When Hive scans a source path in S3 to list its contents, the query may fail because some data may still be present after being deleted and new data does not appear at all [7].

The problem of this thesis is to combine both of these worlds - cheap but consistent storage with high throughput. For example, since cloud storage like S3 is very low cost, AWS offers an Elastic Map Reduce (EMR) service that offers big data computation with S3 as the backend store for an estimated cost of $28,000 per month for a 100 TB cluster [6]. In AWS EMR, the user forms

a cluster using EC2 instances and whenever data is needed for a computation, it is copied down into HDFS from S3. Albeit this method is slower than the traditional Hadoop cluster, it reduces the cost dramatically by 40% since the entire dataset in S3 is not used for every query and EMR can scale EC2 compute nodes up and down as needed - hence the term "elastic". However, EMR is not a unique new file system since it is just HDFS copying data from S3.

1.2 Purpose

The purpose of this thesis is to design and build a new class of distributed file system that utilizes cloud native block storage as the data-layer and provides POSIX-style semantics, consistent distributed transactional metadata, a hierarchical file system, and high throughput all for a lower cost than traditional HDFS and HopsFS clusters. The beneficiaries of this project include any organization, company, or individual that needs to run large-scale big data analytics. The dataset for this type of analysis needs to be hosted on a system that provides high throughput of data, such as HopsFS, and typical end users include Data Scientists. Cloud storage is an essential part of Cloud Computing.

1.2.1 Cloud Computing

Cloud Computing compared to on-premise computing is essentially useful for many reasons. The biggest reason is the illusion of infinite scalability as well as flexibility. Instead of buying expensive hardware to run an application that requires floor space, cooling, electricity, and staff members for maintenance the application can be hosted in the cloud. This way, the app can easily scale up if usage increases or scale down if usage slows down [8]. Another reason applications and computation is moved to the cloud is cost. It is cheaper to run an application in the cloud than paying for your own servers. Although this cost is debatable because it depends on the size of the app, what it does, and how long you need it to run for (long-term costs could be higher with the cloud) [8]. However, moving to the cloud also increases internal resource pools and simplifies deployment of applications as employees can focus on product development instead of managing infrastructure. Finally, handing off infrastructure responsibility to knowledgeable vendors such as Google, Microsoft, and Amazon decreases downtime due to technical failures - these companies are very well known for their technical expertise and success.

1.3 Research Question

Is it possible to build a new class of Distributed File System that utilizes native cloud storage as the data layer, instead of using local hardware, in order to reduce costs and maintain a reasonable level of throughput? If so, what is the performance throughput and the associated costs?

1.4 Contributions

The results of this research includes a new version of HopsFS that uses S3 as the storage layer for blocks on the DataNode. S3 HopsFS cost can be reduced by up to 15 times compared to a normal HopsFS cluster when storing 100 TB in AWS EC2, mainly because of the ability to dynamically scale Datanodes up and down according to load because the blocks (replicas) are stored in S3. Performance tests indicate a reduction in read speed by four times (25% read speed of the baseline HopsFS cluster). This can be improved towards 38% of the original baseline performance by optimizing thread synchronization, reducing S3 round trips, keeping Datanode replica metadata in-memory, and writing blocks directly to S3 instead of first to local disk. The prototype is available on GitHub [9] and supports read, write, and append file operations.

In addition to S3 HopsFS a tool named S3Guard, which enables S3 to be used as direct input and output source for big data analytics applications like Hive, has been extended to use NDB from HopsFS instead of Amazon's DynamoDB as the database. Using S3 by itself introduces eventual consistency problems, but with S3Guard the metadata (hierarchy) of the file system is stored in a database, fixing the eventual consistency problem by double checking with a consistent store what the file system should look like. By replacing DynamoDB with NDB, S3Guard is now 36% faster mainly due to the short network distance between S3Guard and the database as well as the imperial speed of NDB. The code is available on the same GitHub repository [9]. These results are explained and outlined in terms of implementation and evaluation of cost and performance in this thesis.

1.5 Ethics and Sustainability

This project will reduce the overall cost of running these computer clusters, as well as improve the carbon footprint of IT businesses. It is expensive to

manage large compute clusters because of the need to replace broken hardware, deploy staff, and cool computers with air conditioning. This job can be done by cloud computing companies that solve these problems efficiently by sharing a pool of resources with multiple customers.

Accenture highlights the following features to support sustainability and make datacenters more green [10] -

- **Dynamic provisioning** - automatically moves applications from low capacity servers to higher capactiy servers based on load.
- **Multi-tenancy** - allows Software-as-a-Service (SAAS) providers to serve different companies on the same infrastructure to reduce energy, carbon emissions, and cost
- **Server utilization** - host multiple applications on the same server, isolated from each other. This reduces the number of total active servers.
- **Datacenter efficiency** - Utilize energy-efficient, state of the art datacenter technology to reduce carbon footprint

Moving the HopsFS storage layer away from local hardware to cloud native object storage, such as S3, will reduce the overall carbon footprint of IT businesses. For example, small businesses that moved to cloud services were able to reduce their carbon footprint up to 90 percent, while large companies reduced between 30 and 60 percent of their emissions [11]. This also moves more jobs away from on-premise datacenters to cloud companies such as AWS. However, talented engineers and managers are still required to maintain and use the technology.

Data confidentiality is also an issue when running big data analytics on large compute and storage clusters. Cloud companies such as AWS take security very seriously because it is vital to their business. If companies are managing big compute clusters in their on-premise datacenter, the security and confidentiality of data is their responsibility to take care of (in addition to the software vendors, but this is already the case with cloud computing) [8]. As such, this thesis project will improve data confidentiality, keep skilled workers employed, reduce the cost of data analytics, and reduce carbon emissions by moving the storage layer of HopsFS to the cloud. HopsFS also provides a feature called provenance, which adds metadata to the file system that shows details of how a file was created from other files, enabling policies like the European General Data Protection Regulation (GDPR) to be more effective.

1.6 Goals and Research Methodology

The goal of this thesis is to explore the space of using native cloud object storage systems as the backend storage for HopsFS. As a first step in this space, a tool named S3Guard has been developed by the Apache Foundation to enable S3 to be used as a standalone datastore for Hive, Spark, and MapReduce queries by resolving the eventual consistency problems associated with using S3. S3Guard uses Amazon's DynamoDB to keep a record of file tree changes, but we want to explore if it is possible to use NetworkDB from HopsFS to store S3 metadata. This will provide a first step towards changing the DataNode to use S3 instead of the local file system as it touches similar areas of code.

In short, this thesis project can be broken down into the following sub goals-

1. Extend S3Guard for HopsFS by replacing DynamoDB with NDB
2. Design and implement a solution for using S3 in the DataNode of HopsFS
3. Analyze the performance in terms of cost and speed of S3Guard and the proposed S3-HopsFS

The deliverables of the project include the S3-HopsFS prototype as well as a thorough analysis of the system, which includes the speed compared to using S3 alone, the cost of the new system compared to traditional HopsFS systems, as well as future work items for the new prototype that will enable further research in this subject area. Experiments will be carried out to measure the throughput of S3Guard, S3-HopsFS, and the original baseline HopsFS system.

1.7 Delimitations

This section includes what is not focused on in the research. Only Amazon's S3 will be utilized as the native cloud object storage system. In addition, not all features of HopsFS will be added. Most of the work is on the Datanode level. The Namenode has only been modified slightly to include new methods for S3Guard to query NDB. Only write, read, and append are supported for the S3-HopsFS prototype. Delete and truncate are not supported, since the block report has been disabled. In addition, the focus of the paper is to create a proof of concept of an S3-enabled Datanode, not make it optimized for performance (future work).

Chapter 2

Background

This section includes thorough background information that is necessary in order to understand the different components of the thesis, as well as related work in the area.

2.1 HopsFS

HopsFS is a hierarchical, distributed file system with a scale-out metadata layer utilizing an in-memory, distributed database. It was originally a derivative of the Apache Distributed File System (HDFS), which makes it compatible with any application utilizing HDFS already since the client is bascailly the same. HDFS is an open-source project based off one of the first distributed file systems, the Google File System (GFS). HDFS and GFS both exploit the usage of low-cost commodity hardware in a distributed fashion to gain high performance, horizontal scalability, and high fault-tolerance [12]. There are three main components to HDFS - the client, NameNode (NN), and DataNode (DN). The clients are relatively straight forward - they just read and write data via an IP address and port number of the HDFS cluster (a DataNode). The NameNodes manage the state of the file system by tracking this in a database, and the Datanodes actually store the data of the files in blocks.

HDFS provides file operations similar to POSIX, although with less-strict requirements. POSIX, or the Portable Operating System Interface, defines standards for how operating systems should behave, which includes standards for file and directory operations such as when a file is closed it should be available to readers immediately by writing the contents to disk [13]. Although some file systems like NetApp's WAFL postpone this until later by keeping the file contents in-memory, typically when using slow HDDs [14] [15]. Anyway,

the applications that run on HDFS are not general purpose apps that need to run on general purpose file systems that are POSIX compliant [12]. As such, HDFS is designed for batch processing and focuses on high throughput of data instead of low latency access to data, which might matter for the end-user of a web application. The hard requirements posed by POSIX are not needed and have been loosened in order to open the floodgates of increased data throughput [12]!

2.1.1 HDFS vs HopsFS

HDFS exposes a file system namespace that allows the clients (users) to store data in those files [12]. The main difference between HopsFS and HDFS is that a new metadata layer has been introduced - an in-memory, distributed database (NDB). Figure 2.1 shows the main components of each system and the differences in each layer - from the clients, metadata layer, and to the datanodes [16].

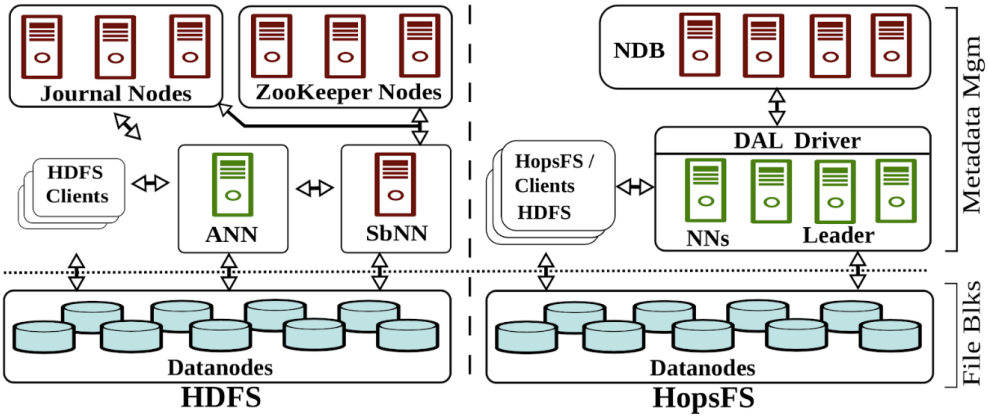

Figure 2.1: Architecture of HDFS compared to HopsFS [16]

The main difference is the new metadata layer. In HDFS, the NameNode keeps track of the file system metadata in a single server in memory. The metadata is kept in-memory because access to it needs to be extremely fast; otherwise it becomes the main bottleneck when reading and writing data due to the single Namenode design. For high availability, if this single Active NameNode (ANN) fails, the Standby NameNode (SbNN) takes over immediately [16]. This is possible because the SbNN mirrors the ANN's memory - or it keeps a copy of the same metadata information in memory. The mirroring is enabled by the ANN logging changes to the metadata to the journal servers using quorum-based replication, and the SbNN reapplying them asynchronously.

Zookeeper is used to coordinate failovers between the ANN and SbNN, as well as determine which one is active at any given time. One Namenode manages file system namespace, housekeeping functions, and metadata (stored on the JVM heap). In HopsFS, the scaling limitations of the single NN design in HDFS is solved by introducing the new MySQL Cluster Network Database [17], which now stores the metadata in a distributed, in-memory database.

HDFS faces a few scaling problems, namely the single Namenode design. In HDFS, the metadata is stored on the JVM Heap to provide ultra-fast access to it. However, the JVM heap space limits the namespace (file/directories) to about 460 million files (200GB heap space maximum). In addition, a single global lock on the namespace is used to implement atomic operations. The global lock ensures consistency of the file system by limiting concurrent access to the namespace to a one writer or multiple readers [17]. Mainly read heavy workloads can handle very high throughput scenarios, whereas write workloads are significantly slower. HopsFS solves the slow write throughput problem by enabling multiple writer semantics by implementing row-level locking in the database, where a row represents an inode (file or directory) [17]. Taking a lock on an inode actually locks all of its associated metadata. Now multiple clients can write to different files at the same time (only the same file can be modified one at a time).

To summarize, HopsFS offers the following benefits over traditional HDFS

1. Higher performance and scalability
2. Small files performance boost
3. Extended metadata
4. Provenance - Details of how a file was created

2.1.2 MySQL Cluster NDB

The state of the file system needs to be kept in a fail safe, permanent location that offers scalability and high performance. Recent advances in memory technology has lowered the price of hardware dramatically and reopened research in the field of in-memory distributed databases [16]. HopsFS has taken advantage of this by utilizing the shared-nothing, replicated, in-memory consistent relational database called MySQL Cluster Network Database. The NameNodes keep the state of the file system in this database at all times. This database enables HopsFS to overcome the single-node JVM heap size bottleneck that HDFS has by replacing it with a scale-out, distributed metadata layer.

NDB can currently scale up to 48 datanodes with 512 GB of data each; this sets the limit of metadata in the system to 24 Terabytes [17]. Figure 2.2 shows how HopsFS can scale beyond HDFS metadata capacity with the utilization of NDB, as outlined in [17].

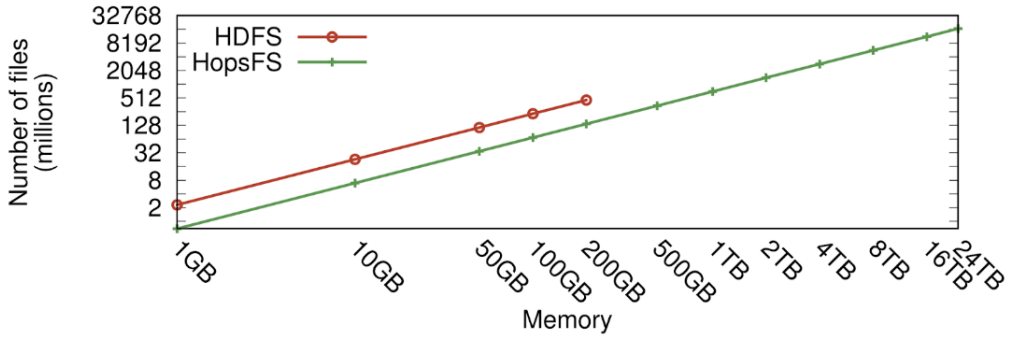

Figure 2.2: Scaling beyond HDFS metadata capability [17]

The architecture of NDB consists three types of nodes, the DataNode, cluster manager node, and the application node [18]. Similar to HDFS, the DataNode actually stores the data of the system. The cluster manager node is essentially the Namenode - it takes care of administration tasks, maintenance, monitoring, and more. Last but not least, the application node is the MySQL server that manages connections to clients and enables them to use the SQL query language.

2.1.3 NameNode

The NameNode (NN) is responsible for coordinating clients requests and maintaining file system metadata such as directories, permissions, last time modified, and users. Since the NN is stateless, it stores the state, or metadata, in NDB. NDB is the MySQL Cluster Database discussed in 2.1.2. The NN also manages file-level operations such as opening, closing, and renaming files. For write operations, the NN locks that file so other operations have to wait on the client side. This eliminates file corruption and race conditions, as well as solves many eventual consistency problems (outlined later in 2.3.1). Each file in HopsFS is split into one or more blocks, which usually have a configurable size of 128 MB. If a file only has one block, then the block can be smaller than this. The file blocks are stored in a set of DNs. The number of DNs the block is copied to is determined by the Replication Factor (RF), which is usually set to 3 for high availability.

Figure 2.3 summarizes all possible block state transitions from the NN's perspective. When a block is first created, it enters the *Under Construction* (UC) state either when a client issues the *addBlock* call or when the client issues *append* to the last block of a file. During *addBlock* or *close* calls, if the last block already has a Generation Stamp (GS) then it becomes *Complete* or *Committed* otherwise [19]. If the NN restarts, the last block of an unclosed file enters the UC state and the rest are *Complete*. One important thing to note is that once a block becomes *Committed* or *Complete*, every block replica has the same GS and should be finalized. Since the NN guarantees order of operations, or that one operation gets completed before the next one starts, we do not have to wait for updates to propagate through the distributed system in order to process a new operation, such as delete.

A short summary of block states

- **Complete** - a least one *Finalized Replica* and will not be modified.
- **Under Construction (UC)** - has recently been allocated for write or append.
- **Under Recovery** - the replica contents are being synchronized.
- **Committed** - client reported all bytes written, but no datanode has yet reported a *Finalized Replica*.

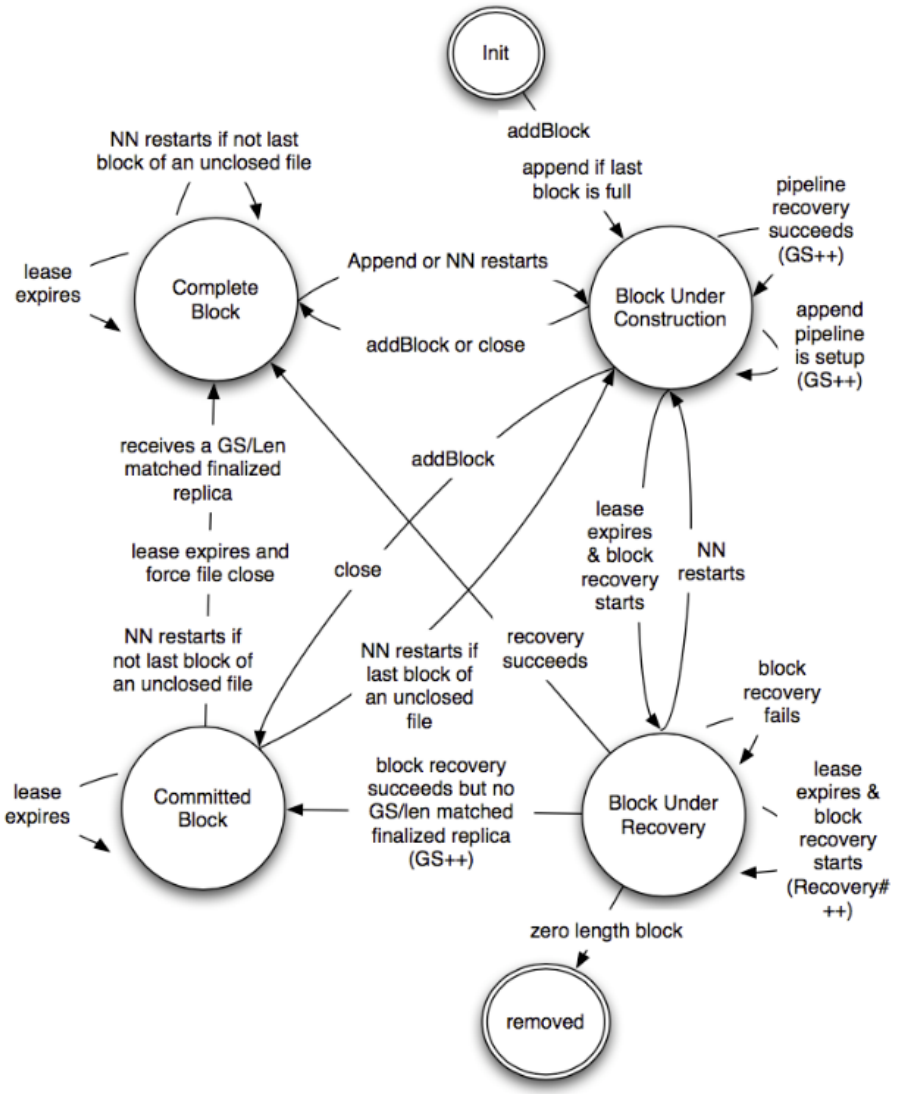

Figure 2.3: Block States from the NameNode's perspective [19]

2.1.4 DataNode

The DataNode (DN) is responsible for managing the storage attached to the nodes they run on. At this level, blocks become known as replicas, or "copies of the block". This is because on a new block write, the block is mirrored to a number of nodes (the Replication Factor) in order to provide high availability and reduce data loss due to hardware failure. The client needs to know which DN a replica is located on; as such, the mapping of replicas to DN is

maintained by the NN, which keeps this state in NDB.

The Replication Factor (RF) can have a positive effect on the read speed, since the NN can return several DNs that contain the block to clients. However, research in this area has shown that there is a "sweet spot" for the RF [20]. They were able to decrease job execution time by 18-20% by increasing the RF from 3 to 9. This was because all the DNs in the cluster were well balanced at 9 replicas. This increased availability was able to support concurrent mapper jobs [20]. However, increasing the RF further than 9 did not show any improvement and in some cases actually reduced performance specifically on write.

The states of replicas on DNs is outlined in Figure 2.4. A new replica is created either by a client or upon an instruction from the NN to create a new replica copy for the purpose of balancing [19]. The replica begins in the Replica Being Written (RBW) state. An RBW replica changes to Replica Waiting to be Recovered (RWR) when the DN restarts. When a lease expiration occurs and the replica recovery process beings, it changes to the Replica Under Recovery (RUR) state. Finally, when a client issues a close file operation, a replica recovery succeeds, or replication succeeds the replica moves to the Finalized state where it lives for the majority of its' life [19].

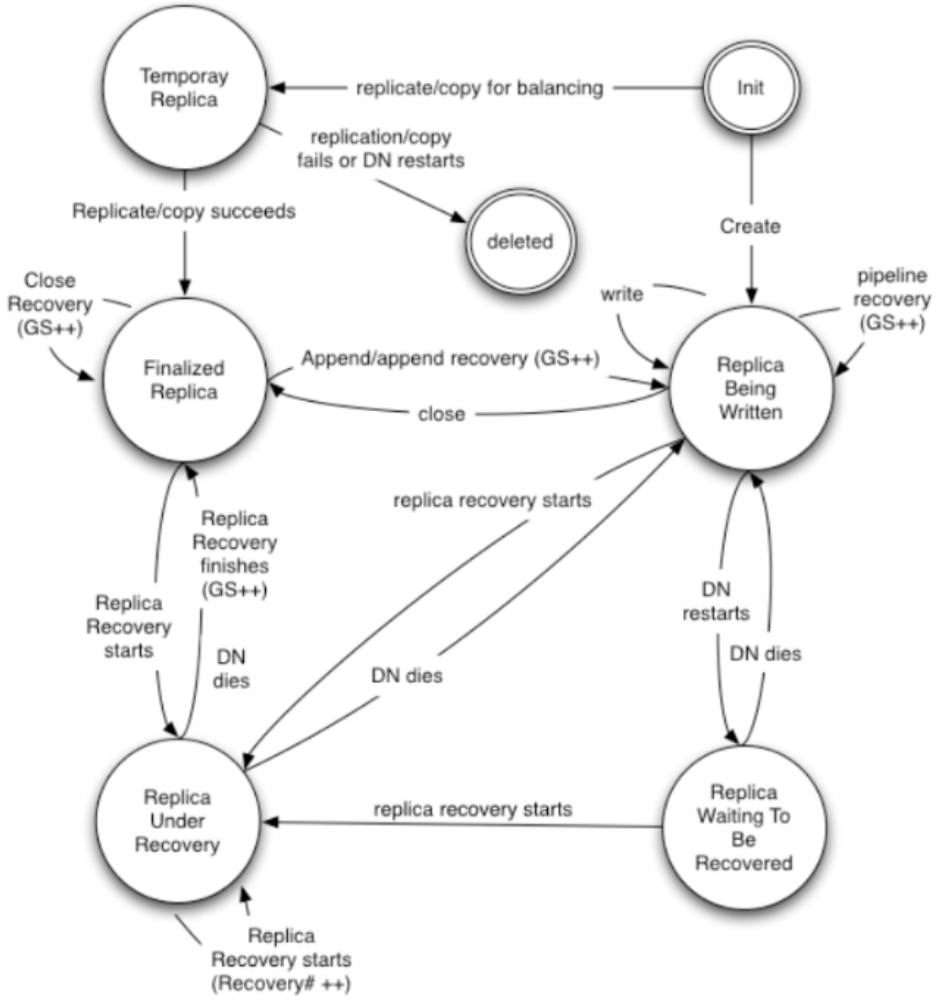

Figure 2.4: Replica States from the DataNode's perspective

All replica states include the following

- **Finalized** - the replica will not be modified
- **RBW** - the replica is being written to
- **RWR** - the replica is waiting to be recovered
- **RUR** - the replica is under recovery
- **Temporary** - temporary replica created for replication and relocation only.

2.2 Disk Architecture

Disk Architecture The main essence of storing data is the hard drive - which is typically composed of a spinning disk or a solid state drive. These hard drives are combined together to form systems that can store massive amounts of data. Two types of basic designs used for storage systems include Shared Disk and Shared Nothing architectures [21].

2.2.1 Shared Disk

Typically storage filers (NFS and SAN protocols) have Shared Disk architectures where two or more controllers share a pool of disks. The controllers are the "managers" that serve data to the users by keeping track of where data is stored and include common storage features such as who can access what (security policies), maintain the highest level of throughput for each user (commonly known as Quality of Service), and remote access to the data through the internet. The data stored on the disks is protected by RAID - or Redundant Array of Independent Disks [21]. The data is striped and replicated across the disks to provide high availability and reduce loss of data in the common occurrence of disk failure. Shared Disk Architectures are write-limited where multiple writer nodes must coordinate their locks around the cluster [21].

2.2.2 Shared Nothing

The Shared Nothing architecture includes the usage of commodity hardware to run highly available systems [21]. This approach includes more controllers, or nodes, that each have a section of the data being stored, such as a post office where data is stored in specific places and can only be accessed by specific entities. To protect data and provide high availability, the data is simply replicated to other nodes in the system, similar to our post office analogy the mail would be copied to multiple locations in case it gets lost. The Shared Nothing architecture is used primarily in the storage system focused on in this thesis - HopsFS. In contrast to Shared Disk, Shared Nothing architectures are write limited where a write that spans more than one partition needs to do a distributed two-phase commit [21]. An example of this two-phase commit includes first writing the replica locally in HopsFS, and then mirroring it to other DN's in the cluster during a write or append.

2.3 Cloud Object Stores - Amazon S3

There are many Native Cloud Object Stores in service today. One such popular option is Amazon's Simple Storage Service (S3). It has a simple interface to get, put, delete, and update any amount of data from anywhere on the web [22]. Specifically, S3 is an object store consisting of key value pairs. Every object in the datastore has a unique key that it can be identified with. Typically these keys represent file paths as seen on any Linux (POSIX) system. According to Ian Massingham at the AWS Summit in Stockholm of 2019, S3 consists of over 130 different micro services!

2.3.1 Eventual Consistency

In order for distributed computing to achieve high availability and scalability, the system cannot respond to state changes immediately. It takes time for updates to propagate throughout the system. This is known as Eventual Consistency, where an update to an object in the system takes time to commit. Only recently with the age of computing has information needed to be instantaneous. For example, the mailing service is eventually consistent; 100 years ago the results of a battle had to be sent by mail. The receiving general may make a bad decision because he doesn't know the battle has already been won (he has old information). This is also the case for a computer asking a distributed system for information, it may receive old data before the update has arrived.

Amazon's S3 offers read-after-write for new PUT object requests, meaning an object's contents can be read right after it has been put but only if it's a new key. However, read-after-write has eventual consistency if a HEAD or GET request is made to see if the object exists or not before it is created [22]. In short, S3 offers the following consistency model:

1. Read-after-write consistency after *PUTS* of new object keys.
2. Eventual consistency for *OVERWRITE PUTS* and *DELETES*.

The read-after-write caveat is critical to understand. Imagine the following scenario [23] -

```
GET /key-prefix/cool-file.jpg 404
PUT /key-prefix/cool-file.jpg 200
GET /key-prefix/cool-file.jpg 404
```

The first GET request for *cool-file* is made to check if the object already exists. After getting null back and putting the object in S3, the final GET request for this object *cool-file* with the same key becomes eventually consistent and may return null, or the object. This is due to S3's distributed nature with caching mechanisms, since it has internal replication of objects to provide high availability. In addition, if a key has to be updated with a new object (overwrite put), the new data has to propagate internally through S3's micro services and regions. So if a client requests the same object immediately, S3 may return old data from. This is why it is much better to always put new object keys in the datastore and then delete the old key - since it doesn't matter when the obsolete object disappears.

2.3.2 S3Guard

Often in the world of big data analytics, S3 can be used as the direct destination, and source, of analytical work. The API for S3 gives it an appearance of a file system because it supports most of the basic file operations - create, read, update, and delete - and therefore makes it possible for applications like Hive, Spark, and MapReduce to use S3 instead of HDFS for this [7]. This actually works (slowly) but sometimes returns incorrect results or fails altogether. This is because of the eventual consistency model that S3 implements, as explained in Section 2.3.1.

As outlined by [7], the following inconsistencies can occur when using S3 instead of HDFS

1. **List Inconsistency** - When a client lists a directory, new objects sometimes do not appear immediately, old objects may appear even though they are deleted, and updated objects may have their old metadata.

2. **Delete Inconsistency** - old files and folders may not disappear to clients immediately, causing operations such as checking if a file exists or not to fail.

3. **Update Inconsistency** - the updated contents of a file may not appear until later. For example, if a file is updated the client may still sometimes read the old data for some time.

However, S3Guard aims to overcome these inconsistency problems by simply keeping a record of the file tree and it's changes over time in a database. Then, when a client wants to interact with S3, the operation is double-checked

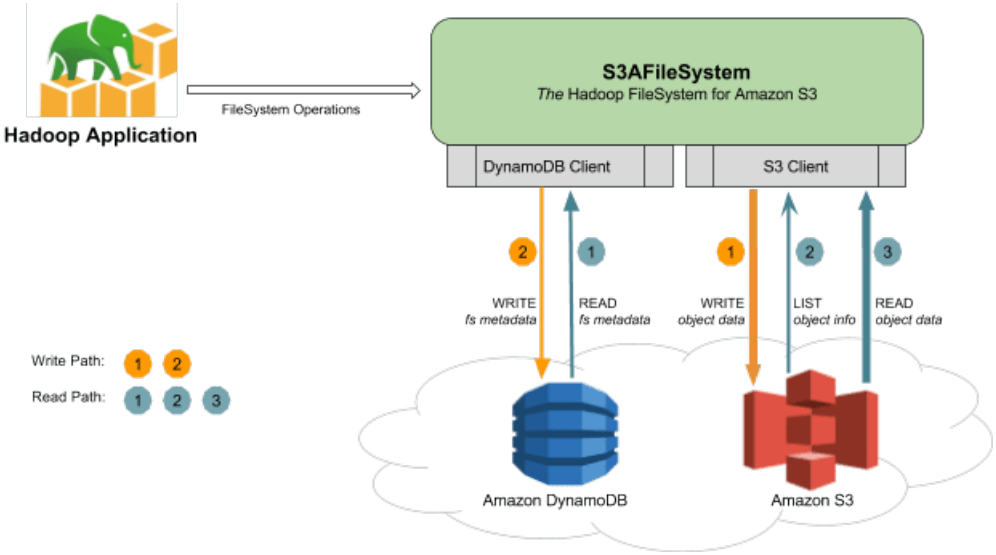

Figure 2.5: How clients interact with S3 [7]

with the database to make sure old data is not returned. Figure 2.5 illustrates how a Hadoop Application utilizes S3Guard to read and write data.

Queries through S3Guard on the data in S3 may become inconsistent with what is actually in S3. For example, new data may be omitted, objects can be overwritten, and clients may pick up deleted or old data. It is required that all clients or applications interacting with an S3 guarded bucket utilize this feature (s3guard) to avoid this inconsistency [24]. This makes sense since the S3 file tree metadata is kept in a separate database and every interaction with S3 needs to be updated in this database.

2.4 Related Work

This section outlines other systems that have utilized S3 with DFS. Besides AWS Elastic MapReduce (discussed in 1.1), this section discussed two specific solutions.

2.4.1 Mounting external storage in HopsFS

A previous attempt in this field has been made by Gabriel Vilen from TU Delft University in 2018, which involved mounting external storage in HopsFS [25]. This would enable workloads to read and write data that exists in external cloud repositories, such as S3. The work included mounting S3 repositories by tak-

ing a snapshot of the remote file system and copying it into HopsFS, making it a read-only approach. To support writes and synchronization, frameworks that layer stronger consistency models were used such as s3emper [25]. Although this solution did work for reading data, it proved to be overly complicated and did not reduce the cost of running HopsFS. The main difference between Gabriel's work is that the storage is mounted *into* an existing cluster, whereas the project outlined in this thesis is aiming to build a new file system ontop of the external storage (S3) to save on cost and over-complexity.

2.4.2 WekaIO - MatrixFS

WekaIO Matrix is a new software-only high-performance file based storage solution that is highly-scalable, elastic, and easy to configure written in the D programming language. It can run on standard Intel-based application servers. The main idea behind Matrix includes a radically simple storage solution that promises the performance of all-flash arrays but with the scalability and ease of use that comes with cloud computing [26]. As such, MatrixFS is a distributed, parallel file system that removes the traditional block volume layer that manages the underlying storage resources. HDFS has this volume layer. WekaIO can manage storage in POSIX, SMB, NFS, HDFS, and S3 [26]. One of the main features of MatrixFS includes the automated tiering of cold data with cloud object stores such as S3. In other words, data that is not used much on the MatrixFS storage cluster can be pushed to the cloud to save on costs [26]. Whenever this data is needed, it is downloaded back into the cluster. This type of configuration is deployed on AWS EC2 instances with SSDs attached. This is similar to the ideas presented in this thesis, but the main difference being that S3-HopsFS is storing all data in S3 and managing it directly, whereas WekaIO can also do this but also utilizes the mounted "local" file system for fast access and S3 for scaling out cheap storage. In addition, this thesis discusses S3Guard where applications can read and write directly from S3 without any extra layers of code (WekaIO does not have their own implementation of S3Guard currently).

Chapter 3

Implementation

This section includes the design and implementation of S3 into HopsFS in the two main components of the project - S3Guard and the new S3 DataNode. The engineering research method was chosen for this thesis - Conceive, Design, Implement, Test, and Operate (CDI(T)O).

3.1 S3Guard

As explained in Section 2.3.2, S3Guard is a tool that solves the consistency problem associated with S3, such as looking if files exist or not after updating or deleting them. For example, a Hive query can miss out on newly written data, pick up old data, or not find the data at all when it scans a source path in S3 [7]. For that reason, S3Guard is needed to keep this data consistent by keeping a record of the file tree and its' changes in a database. An analysis of the performance of NDB compared to the original DynamoDB implementation is included in Section 4.1.

The open source developers of Hadoop created an interface named *MetadataStore.java* that contains all the methods necessary to implement new databases for S3Guard in addition to a suite of black box tests (*MetadataStoreTestBase.java*) that test the integrity of the implemented metadata store. These tests check that s3guard - regardless of the underlying database - returns the expected data and updates properly according to changes in the file tree structure, such as deleting a subtree. In order to implement the *MetadataStore* interface which contains methods like *listChildren*, the HopsFS NameNode needs to implement these new methods to retrieve and insert data from NDB. These methods are optimized using a C++ engine named ClusterJ that enables these queries to be ultra-fast. S3Guard could use normal SQL queries to get and put data from

NDB using the MySQL Server, but the number of connections to NDB are also limited to keep the performance high.

Figure 3.1 shows the UML class diagram of how S3Guard was implemented in HopsFS. The greyed out elements represent implementation details; from a usage perspective S3Guard is just a black box that the client is reading and writing through to S3. *S3PathMeta* contains all the metadata automatically saved about a path in S3.

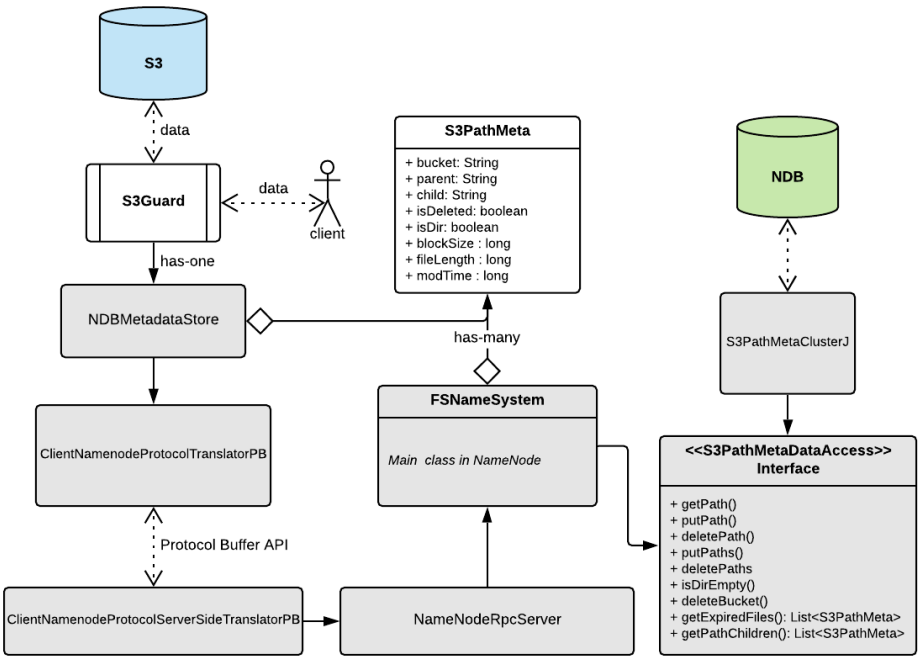

Figure 3.1: Design of S3 Guard in HopsFS

The important takeaway from Figure 3.1 is the *S3PathMetaDataAccess* interface that includes all the methods necessary to interact with NDB - *getPath, putPath, deletePath, putPaths, deletePaths, isDirEmpty, deleteBucket, getExpiredFiles, and getPathChildren*. These methods in turn are implemented in *S3PathMetaClusterJ*, which calls the C++ engine ClusterJ to query NDB for the data requested. Whenever the client wishes to interact with S3, and S3Guard is enabled on the bucket, the S3A code written by the Hadoop community actually handles the internal requests to S3 and compares the result with what is contained in S3Guard. For example, during a client list directory operation, if a file is marked as deleted in the database and is still returned by S3, the deleted file is simply omitted and the client sees the expected data. The logic that returns a directory's children is handled in *NDB-*

MetadataStore.listChildren - which essentially simply calls *getPathChildren* through the NameNode and then formats the data properly and returns it to the client (S3Guard).

The communication between the client (S3guard) and the server (NameNode) is handled via Google's Protocol Buffer API. Hadoop uses protocol buffer across it's protocols between the client, namenode, and datanode; as such, it was a fitting choice to use in this scenario to communicate with NDB [27]. The methods in the *S3PathMetaDataAccess* interface are included in this protocol between the client and server to pass information back and forth - the path information is bundled in *S3PathMeta* objects which can be either files or directories.

While S3Guard is very nice to fix the eventual consistency problem of S3 so that we can use it for Hive queries (example mentioned in 2.3.2), the big data analytic applications lose out on all the benefits of high-performance file systems like HopsFS (details in 2.1.1). Therefore, we need to combine these two solutions.

3.2 New S3 DataNode in HopsFS

A new DataNode implementation that utilizes S3 as the backing store is outlined in this section. The code for the new implementation can be found on GitHub [9]. How to setup and run the prototype is outlined in B.

3.2.1 Updated Read and Write Path

The main difference between the S3 DataNode and the normal HopsFS DN is the manner in which blocks are finalized. Figure 3.3 shows how the write path has been updated, and how it compares to just using a standalone S3 client. The green represents the new HopsFS DN and the blue represents just using an S3 client by itself to read and write data in a test environment.

The HopsFS client (HDFS compatible), first contacts the NameNode (NN) to get the address of which DN to use for writing a new file (blocks); the NN then checks NDB for metadata such as file state (if the requested operation is possible for example) and returns. The client then begins sending the file contents to the DN, which creates a new Replica Being Written (RBW) block on a volume (disk drive located on the DN) in a special "RBW" directory. As the client is writing the file, these local RBW block files are being filled with the data. When the client closes the file and finishes writing, the DN receives this close call and begins to finalize the RBW replica blocks by uploading

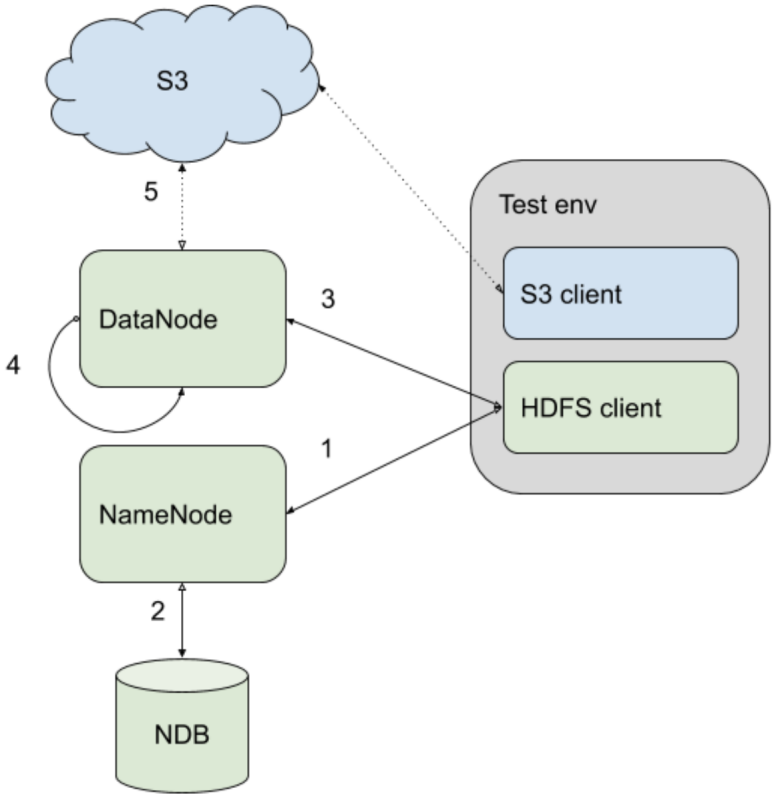

Figure 3.2: Flow of writing a new block with HDFS compared to standalone S3.

them to S3 and then removes the RBW block locally afterwards. Figure 3.3 visualizes this process.

This approach of only uploading finished blocks is elegant because it leaves most of the DN's features untouched, such as Block Recovery. For example, if the Datanode crashes while a block is being created (RBW state), it picks that process back up again when it comes back online since the block is still on the local disk. To initiate a recovery process on a finalized block, the only change necessary is to download the block to the RBW location. For append, the process is very similar - first the block is downloaded from S3, and then moved to the RBW directory just as the normal HopsFS DN does. However, the recovery process in general is not needed here because S3 is already over 99.99% reliable - but recovery was still included in this implementation because it is needed for delete and truncate operations.

The truncate operation does not need to be changed, since it relies on the recovery process to mark a block as being recovered. This automatically moves

the block from finalized to RBW location by downloading the block from S3 to the RBW directory on local disk like the original implementation. This behaviour to move the block is in *FsDatasetImpl.initReplicaRecovery()*, which is overridden in *S3DatasetImpl* to instead download the block from S3. Currently, deleting files is not supported because the NN block report needs to be changed (see Future Work section 4.6).

The read path is also relatively straight forward and remains very similar to the current approach. Instead of streaming the block from the local disk on the DN, the block is streamed from S3 instead by opening an HTTP connection. These methods are overridden from *FsDatasetImpl* in *S3DatasetImpl*, as explained in the section 3.2.2 and 3.2.3.

3.2.2 Code Architecture (UML Diagram)

The implementation is based on both new classes (or files) and existing classes and code. Figure 3.3 includes most of the critical changes needed to get S3 integrated as the backend store for the DataNode. In order to keep things simple, the UML diagram does not include an exhaustive list of every change necessary. Maven packages are highlighted in a different background color to visualize where code lives.

Most of the new code is added in extended Java classes that add new functionality and replace methods that need to be changed. The core interface is named *FsDatasetSpi* and includes methods such as *createRBW, getBlockInputStream, and finalizeBlock*. These methods include the essential block (or replica) lifecycle routines on the DataNode - creating a new block, writing data to it, finalizing and uploading it to S3, and reading the block from S3 via an *InputStream*. The existing implementation, *FSDatasetImpl*, was extended to produce *S3DatasetImpl* because many existing features remain the same because the local file system can still be used for most DataNode tasks, such as recovery. The system does not run out of space because finalized blocks (99% of stored data) is in S3 and is only downloaded when needed, and then removed. There is room for future optimizations here by caching popular blocks on the DN volume by not deleting certain downloaded blocks. In other words, keep finalized blocks around that have a high probability of being used or read again, such as an application log file.

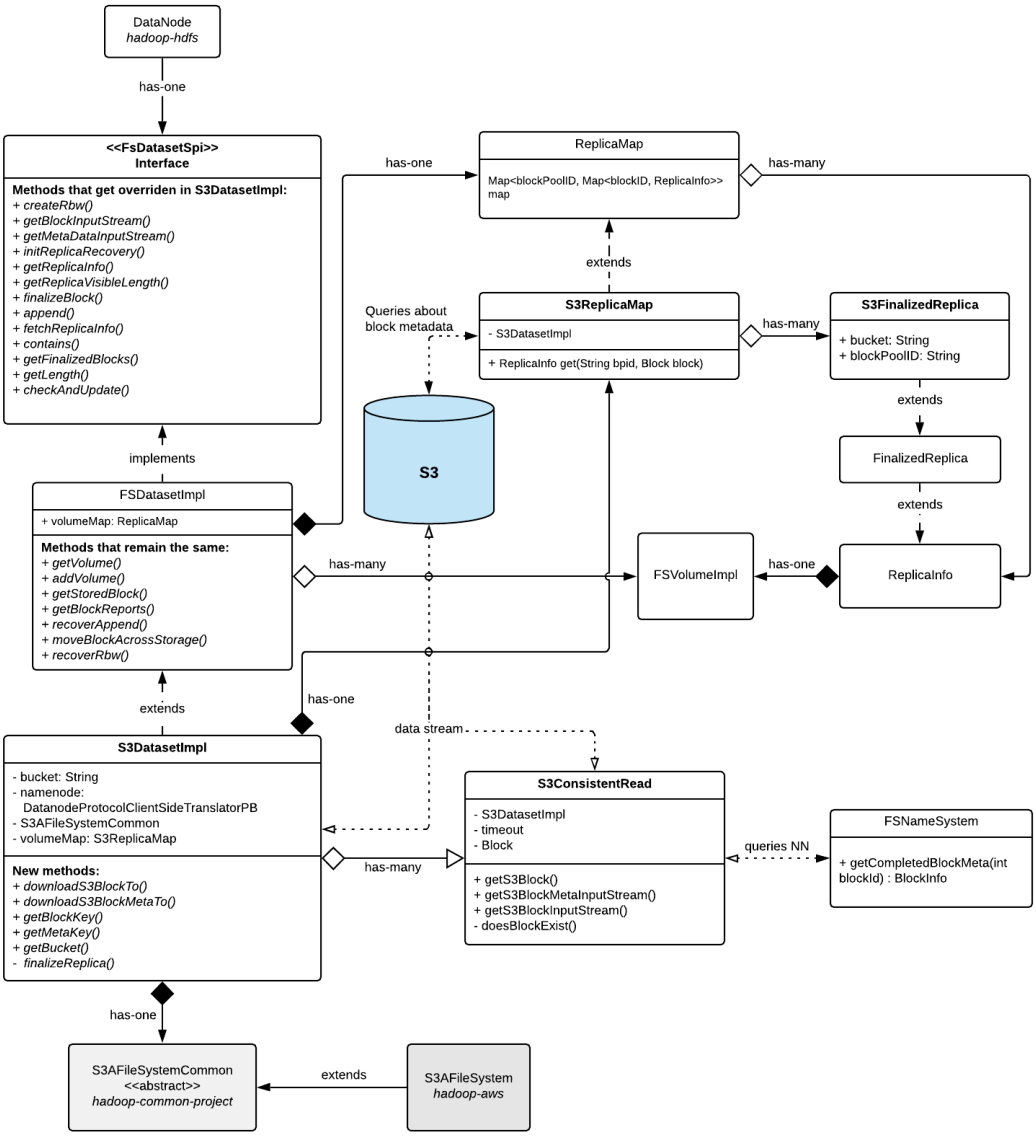

Figure 3.3: UML Class Diagram of new S3 DataNode

3.2.3 S3DatasetImpl - Core Functionality of S3 DataNode

An interface named *FsDatasetSpi.java* has all the core DataNode block operations such as *createRbw, getBlockInputStream, finalizeBlock, and append*. Many of these methods can be seen in state transitions in the replica state diagram in background section 2.1.4. The normal HopsFS code implements this interface in *FsDatasetImpl* for the local file system. The new S3 code simply

extends this *FsDatasetImpl* and only overrides the functions seen in Figure 3.3.

3.3 Eventual Consistency Problems

The main job of the Datanode is to manage blocks by reading and writing them from any storage that supports basic file functions - read, write, and delete. However, when storage systems that have eventual consistency are used, we cannot always trust the underlying system that what it returns is what we expect - meaning we need to use a second metadata layer to keep track of the file system state. HopsFS comes out of the box with a persistent metadata layer - the Network Database - that can be utilized to keep track of the exact file system state. Also, the NameNode guarantees order of operations and locking for writing files, so that a file cannot be read after it has been deleted for example. Most efficient and cheap cloud object stores offer eventual consistency due to the internal replicated state, such as AWS Simple Storage Service (S3). Background section 2.3.1 explains Amazon's S3 eventual consistency model, which mainly includes read-after-write for new keys and eventual consistency for overwrites and deletes. The next section outlines how the new S3 DataNode in HopsFS solves the remaining consistency problems with S3.

3.3.1 Enabling Consistent Reads and Writes

The DataNode needs full consistency in order to function properly, so we need to solve S3's eventual consistency problem. Luckily, we have a distributed metadata layer in the NameNode thanks to HopsFS. This system can be utilized to double-check if blocks actually exist, their current state, or if a block is not supposed to exist. A new system is introduced in the DN whenever it needs to read a block's metadata or content, which is implemented in a file called *S3ConsistentRead.java*. The basic logic is to try and get the unique block key from S3, and double check with the NN if null is returned. Each block key contains a Generation Stamp (GS) that shows it is the most up-to-date version of that block. For example, new files (or blocks) always start with GS 1000; if new data is appended to this block or it is truncated the GS becomes 1001. This enables the system to always return the correct block, or nothing at all since the generation stamp is included in the request from the client. This model exploits the usage of S3's read-after-write consistency model of new object keys (rule #1 from section 2.3.1). If a null is returned instead for some reason, the system then queries the NN to see if this version of the block is

supposed to exist. If the NN answers 'yes', *S3ConsistentRead.java* simply waits some time and tries again, since the block has not been fully written yet, or raises an exception if the NN answers 'no' such as when the client tried to read a block that got deleted by another client (the NN guarantees order of file-level operations). However, querying the NN from the DN showed some race condition problems in this research, outlined in section 3.3.4.

The main read-after-write caveat (issuing a GET before a new object is created - see background 2.3.1) needs to be optimized to prevent eventual consistency moments. In the current implementation, there are situations such as the following in Figure 3.4 where a READ needs to wait for the block to appear in S3. This can happen when a new block tries to be read immediately after it has been created, since there is a check if the block already exists during *createRBW* and *finalizeBlock* methods. The previous implementation of *FsDatasetImpl* has this check to make sure the DN does not corrupt blocks. The new *S3DatasetImpl* should not implement these checks by querying S3 for the block, since it will make the new block eventually consistent, thus impacting performance. The current prototype works because *S3ConsistentRead* gaurantees blocks that exist can be read.

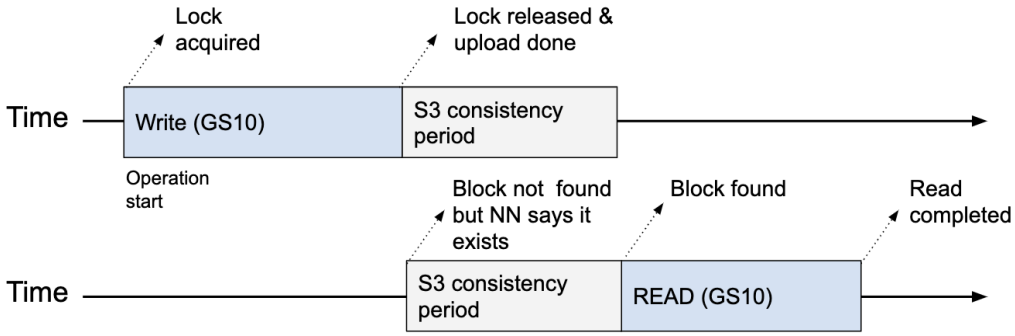

Figure 3.4: Potential read after write consistency problem

This problem can easily be solved by moving the Finalized Replica metadata back into the memory of the DN, and is part of the future work 4.6.

3.3.2 Consistency Problem - S3 Block is Missing

This section explores all possible operations that interact with a missing block in S3. The general idea is to check the NN for the block state, and retry again accordingly. The following table outlines every operation and the required steps to proceed if a block is missing in S3. For RBW and Finalize operations,

checking the NN for the block is useless because the new block version is guaranteed to be unique.

Operation	Solution
Read	Query NN. If exists, retry; otherwise exception
RBW	Continue
Finalize	Continue
Append	Query NN; if exists retry, otherwise exception
Delete	Continue anyway (S3 will honor latest request)
Recovery / Truncate	Query NN; if exists retry, otherwise exception.

Figure 3.5: Steps taken when a block is missing in S3

3.3.3 Consistency Problem - S3 Block Exists

This section explores all possible operations that interact with a block and it is actually found. The general idea is to simply continue with the operation, since all blocks (and versions of those blocks) have unique keys. This idea exploits S3's read-after-write consistency model of new object keys. The following table outlines every operation and the required steps to proceed if a block is found in S3.

Operation	Solution
Read	Continue (block key is unique)
RBW	Exception - block version already exists (impossible)
Finalize	Exception (older block version has different key)
Append	Continue downloading block
Delete	Continue deleting block
Recovery / Truncate	Continue with truncate

Figure 3.6: Steps taken for each replica operation when a block in S3 already exists

3.3.4 Block State Race Conditions

Since avoiding round-trips to S3 is critical for performance of the system, you may wonder why the Datanode just doesn't query the NN first if a block is supposed to exist or not. This approach was tested out and it was discovered that this did not always work as expected because of the distributed nature of

the system. The block the client wanted to read on the DN was indeed finalized and uploaded to S3, but not yet reported to the NN as *Finalized* - in this case the block was sometimes *Under Construction* and sometimes *Committed* from the NN's perspective. As such, the NN reported null and the read failed. To solve this distributed system race condition problem, the DN actually queries S3 first for the block, and returns it successfully if found. This is possible due to S3's read-after-write consistency for new object keys. This actually boosts performance, because now the DN avoided a query to the NN and went straight to S3 instead. If a block is not found in S3, then we query the NN and either try the read process again (due to S3 consistency problem) or raise an exception (non-existent block was read). By the time the DN has already tried S3, the NN knows the new metadata of the finalized block because a round trip has been completed already between the client, DN, S3, and the NN and the distributed system is in sync. The NN does not know about the new object metadata immediately after the block is finalized because the Datanode has not issued a block report to the NN yet that contains state information about blocks the DN has (the NN has it's own view of block state, as seen in background section 2.1.3). However, as clients are interacting with the NN to open, close, and read files, the NN builds it's own view of block state.

As such, it is not always possible to "query the namenode" to get a consistent view of block state for read-after-write. However, this problem was solved with unique block keys by including the generation stamp. This works for any modification to the block, since the GS is bumped every time.

In addition, since the NN guarantees order of operations it is not possible to append to a deleted block if that block is still found in S3, as figure 3.7 illustrates.

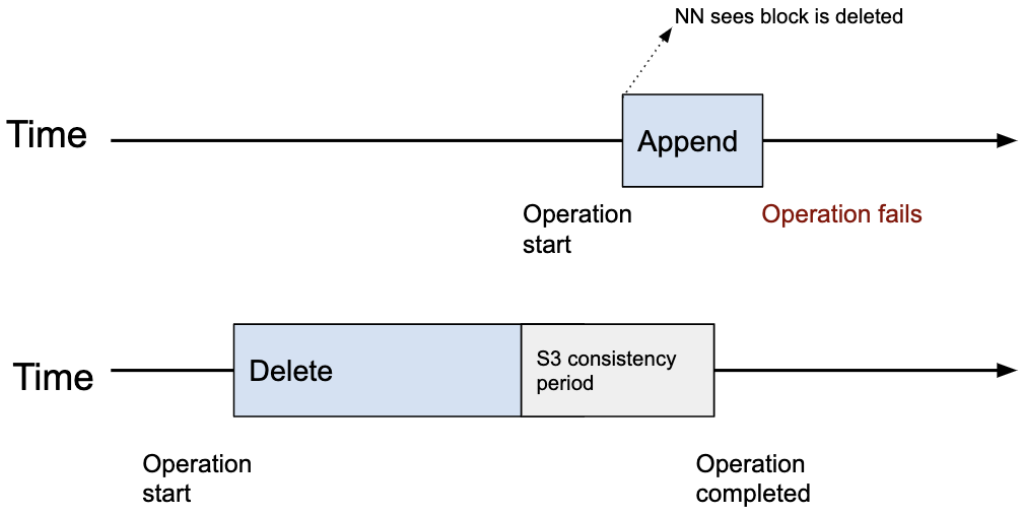

Figure 3.7: Appending to a deleted block that still exists in S3 is not possible

3.4 Required Changes of Existing Packages

This section includes bigger parts of the codebase that were changed outside of adding new files and config setting changes to get S3 to work.

3.4.1 S3A Connector

The S3A connector that already exists in the *hadoop-aws* package is required to interact with S3; this package makes it easy for existing Hadoop code to interact with S3 as if it was just another file system. For example, sometimes the datanode only needs to read a certain piece of a block, or seek into the file to just read to a certain offset and then close the file. When using the AWS SDK client directly instead, this creates problems when the S3 input stream is not drained and closed properly. The AWS SDK uses connection pools to speed up requests to S3 by reusing existing connections for new queries. This is not possible when the input stream is not read entirely - similar to a congested highway of large vehicles with only one passenger. The S3A connector fixes this problem by seeking to the end of the file after reading the required pieces and then aborting the request. The S3A connector also extends normal S3 for instance to include directories, faster uploads, file appends, and the capability of using S3Guard for a bucket. A few minor methods were modified in S3A, such as removing a get file status check before reading the file to increase performance (specifically *S3AFileSystem.open*).

3.4.2 S3A Circular Package Dependency

In order to use the S3A connector (from *hadoop-aws*) in the Datanode (*hadoop-hdfs* package), a few changes were required. Simply creating a new Java maven dependency on the *hadoop-aws* package from *hadoop-hdfs* does not work, because *hadoop-aws*'s dependencies have dependencies back on *hadoop-hdfs*, thus creating a circular dependency problem - much like how C does not let the parent library link back to the child again. This was solved by creating a simple interface - named *S3AFileSystemCommon.java* - in the hadoop-common package. It extends the FileSystem class and then has abstract methods for all the methods needed to call in the *S3AFileSystem* package. We can now import *hadoop-aws* into *hadoop-hdfs* during runtime using a named class from the configuration, because after the build is complete it is allowed to use any Java library. A few classes from S3A are pulled up into this new common package *hadoop-common.org.apache.hadoop.fs.s3a* to get the interface to compile.

Chapter 4

Evaluation

This section includes a thorough analysis of the performance of S3Guard and the S3 DataNode in terms of cost and throughput.

4.1 Performance of S3Guard

An evaluation of the performance of this new implementation of S3Guard utilizing NDB instead of DynamoDB is included in Figure 4.1. The analysis concludes that S3Guard with NDB is faster on average by 36% over 50 iterations of the experiment. The experiment was run on AWS m5a.xlarge EC2 instances and composed of listing all the files and directories in a file tree and then deleting them (how to run it is in B). The file tree was composed of 3000 files spread across sub directories with a depth of three. The time taken to contact S3 is irrelevant because both implementations do so and the same pattern emerges every time you run the experiment.

The client reading from NDB was located on one node and NDB was located on another node in the same Virtual Private Cloud (VPC) subnet. The performance of S3Guard utilizing NDB is better by 36% in this case mainly because the database is in the same subnet as the client; traceroute shows that the network distance between the two is only one network hop away. For DynamoDB, traceroute shows over 15 network hops between the EC2 instance and the DynamoDB endpoint dynamodb.eu-west-1.amazonaws.com, which are both in the same AWS region. This is an obvious benefit for using NDB over DynamoDB when utilizing S3Guard in applications that already have dependencies on HopsFS - namely because NDB is already deployed and paid for and there is no need to pay extra costs to deploy DynamoDB as well.

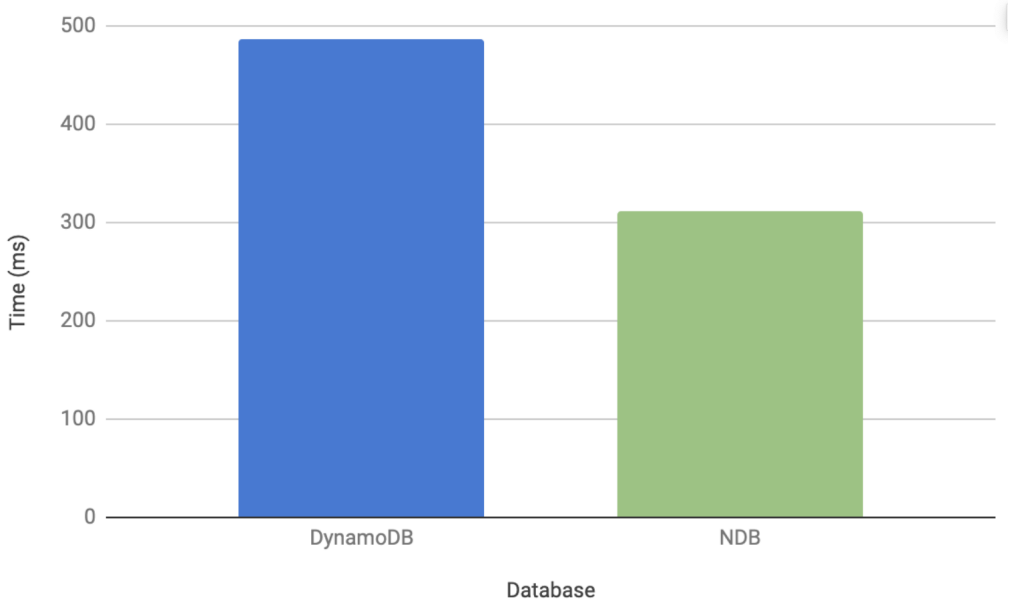

Figure 4.1: S3Guard using DynamoDB vs NDB in AWS for listing files and directories

4.2 S3 HopsFS Performance Analysis

This section includes a thorough analysis of the cost and performance of the new S3 DataNode as compared to the normal HopsFS DataNode that relies on the locally mounted file system.

The following areas will be explored

1. Why is the S3 DataNode slower than using S3 by itself?
2. What are the costs associated with S3 and normal HopsFS clusters?
3. Where does the S3 DataNode spend its time on reads and writes?
4. What are the bottlenecks?
5. How were the tests executed?

To get real world scenarios, the analysis was done in AWS to get the best performance from storage and compute. Recently in the beginning of 2018, AWS increased the bandwidth speed from EC2 to S3 from 5 to 25 Gbps [28]. This is a remarkable update because it dramatically increases the efficiency of

using S3 for storage since the data can now be utilized 5 times faster than before, putting it at serious competition with using HDDs and SSDs. To use this increased bandwidth capability, the ENS driver for Enhanced Networking on AWS Linux was enabled for all used instances [29]. However, achieving bandwidth speeds of 25 Gbit/s between EC2 and S3 require specialized workloads, using multiple buckets, and parallel clients that are reading and writing. Using S3 alone and parallel clients, only up to 1.5 Gbit/s speeds were seen, outlined in section 4.4.1. This implementation only included using 1 bucket per cluster so that all DataNodes can access any data, but extending the solution to use multiple buckets should be relatively simple.

4.3 Performance of new S3 DataNode

Each test was run on AWS EC2 instances with 4 CPUs, 16 GB of memory (m4a.xlarge), and EC2 bandwidth speeds of up to 10 Gbps. The Hammer Bench tool was utilized to test write, append, and read operations on the new prototype [30]; how to run hammer-bench and other tests is in Appendix B. Each workload reports successful operations at the end and gives an overall throughput per second. A brief overview of write and read performance is in the appendix figure A.6.

4.3.1 Create Workload

In Figure 4.2, the blue bars represent the new S3 Datanode and the green bars represents the standard HopsFS system. For each category of number of clients, each cluster type (S3 and non-S3) was tested with one and three nodes. For the three node cluster, S3-HopsFS set the Replication Factor (RF) to one, since S3 is already safe and replicated - whereas the normal HopsFS cluster had an RF of three for high availability. As the number of clients increases for the 1node S3-HopsFS cluster, the create throughput did not increase. However, for the 3 node S3-HopsFS cluster the throughput triples when the number of clients are 30 or greater. This is because there are now 3 DNs processing new files coming from the client and uploading them to S3. The normal HopsFS cluster did not triple in speed compared to 1 and 3 nodes when the number of clients are increased; this is because of the Replication Factor and the Datanodes have to mirror every block to the other nodes. Also, a CPU bottleneck was observed in every test scenario on the server hosting the NN, NDB, and one DN - it remained between 95% and 100% during the test (results in Appendix A.1). As such, the create throughput is very similar across the board

(the network bandwidth was not the bottleneck). In addition, the S3-HopsFS write speeds are lower since the DN is first writing the entire block to the local file system, and then uploading the finalized block to S3. This results in at least two round-trips to S3 to check if the block exists or not before upload, also contributing to a possible eventual consistency problem. This should be fixed in future implementations by checking the internal HashMap of finalized replicas (like the current *FsDatasetImpl* does).

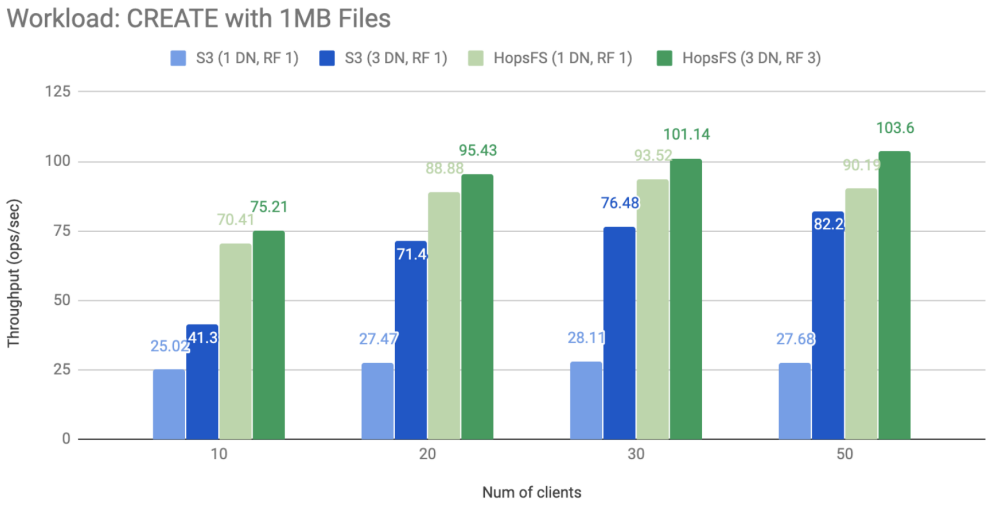

Figure 4.2: Creating 1 MB files on AWS EC2 instances with different number of clients

4.3.2 Read Workload

The read throughput for the same cluster definitions - one and three node S3-HopsFS and regular HopsFS - is significantly higher than the write throughput as seen in in Figure 4.3. This is because since the clients are running on the same hosts as the Datanodes, the read workload is extremely efficient for the normal HopsFS cluster where all the block data is going over the loopback interface - over 400 MB/sec was observed during read in these scenarios (see A.4). The Datanode is simply opening an InputStream and piping back the block data. The CPU is also no longer the bottleneck, as seen in Appendix A.1. In addition, the regular HopsFS cluster is running on SSD comparable throughput - Elastic Block Store (EBS).

EBS volumes are placed in Availability Zones where they are automatically replicated to protect from disk failures [31]. However, EBS volumes still

have an annual failure rate of between 0.1% - 0.2%, so if you run 1000 EBS volumes for one year you can expect 1-2 to have a data loss failure [31]. Because of this, we still need to triple replicate the HopsFS cluster using 3 datanodes to protect data if we were using EBS - thus wasting resources because the replication is in effect doubled without any protection gains. EBS was used in this experiment because due to the ease of use and has speed similar to SSD.

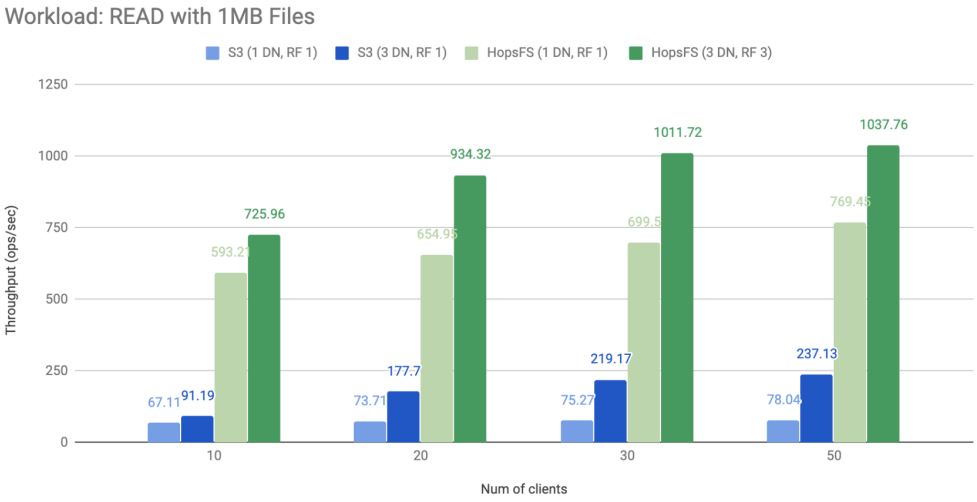

Figure 4.3: Reading 1 MB files on AWS EC2 instances with different number of clients

When using S3 to read and store data, several bottlenecks appear during read. The first is namely the distributed nature of the system; many thread synchronization issues appear since the latency from EC2 to S3 is much larger than the latency from the EC2 disk (EBS in this case). For example, during the read phase the Datanode has to check if the block file actually exists or not - accounting for at least one round-trip to S3. This round-trip is also synchronized in the code - meaning only one thread at a time can do this check. The new implementation was based on *FsDatasetImpl* (which uses the mounted file system) and stores the replica state in a memory-based HashMap; when the DN restarts this replica state is read from disk and put back into the HashMap. Changing and getting things out of the HashMap needs to be synchronized. The new S3 implementation checks the actual S3 repository for the finalized replica metadata, so that all Datanodes can access any block. Future work on this project involves fixing this bottleneck, with solutions such as storing finalized replica's in the HashMap like the original implementation, and then checking S3 as a last resort. When a simple thread synchronization to check

S3 is added to the S3-only tests in A.3, the throughput reaches similar levels to the S3-DN.

Another bottleneck for the S3 implementation is that the clients are technically reading from S3 and not the storage attached to the DN. In the tests run here, the normal DN saw throughputs of up to 3.3 Gbit/sec (420 MB/sec), but the maximum S3 throughput was 1.2 Gbit/sec per DN (calculated in 4.4.1). Since the S3 system has several bottlenecks, it is expected that the read throughput triples from 78 to 237 ops/sec when increasing the number of Datanodes from 1 to 3 with 50 clients. This behaviour is not seen in the normal HopsFS Datanodes since the bottleneck becomes the CPU again and disk for 1 and 3 nodes and not the network with just a few number of clients. Appendix A.2 shows the disk usage for normal HopsFS runs.

4.4 Network Performance

Network throughput is a key indicator of how Distributed File Systems perform due to their nature of communicating and transferring data over the network - clients are reading and writing data over the network to the Datanodes. As such, network throughput is key metric used to find the performance bottlenecks associated with the S3 DataNode - if it is lower than using S3 by itself to read and write data, we know S3 is not the main bottleneck in the system. Other bottlenecks include CPU, memory, thread synchronization mechanisms (deadlocks), latency between S3 and EC2, and where clients are located and how they are reading and writing data (such as network packet sizes).

4.4.1 Network Performance of S3 Alone

In order to determine the performance bottleneck in the new S3 DataNode, a Java Maven test was written to find the top speed of just using S3 alone to read and write the same block files as the Datanode in Java (how to run it is in B). Figure 4.4 shows the read and write speeds that were achieved using 1 bucket with 10 clients reading and writing 1 MB files from EC2 in the same region. A top speed of 1.2 Gbit/s (150 Megabytes per second) was achieved from a Java Maven test running in AWS.

To simulate the Datanode more closely, the write workload first checked if the file already exists in S3 before writing the file - contributing to 1 round-trip to S3 before uploading the 1 MB file. This contributes to a significant reduction in performance, as seen by the purple line in Figure 4.4 - this initial status check is not performed on reads in the test. The DataNode also performs this

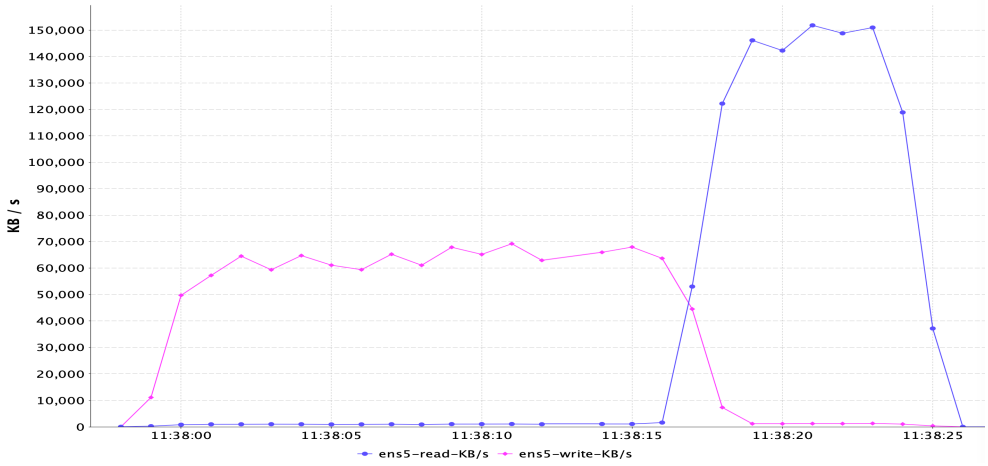

Figure 4.4: Reading and writing 1 MB files from S3 with 10 clients

status check when creating new RBW blocks, and throws an error if the block already exists, as well as during the finalize stage the DN checks if a previous block version exists and deletes it. These checks are necessary in order to improve the safety of the Datanode, reduce data corruption, and support append for example. Since every block has a unique key path because of the addition of the generation stamp, the same block with the same generation stamp should not co-exist during RBW stage - if it does happen it means there are other problems on the Datanode that are not related to eventual consistency.

4.4.2 Network Performance of S3 DataNode

The same test - 10 clients reading and writing 1 MB files (blocks) - was run on a S3-HopsFS cluster in AWS with 3 DataNodes and a Replication Factor of 1. The maximum read throughput achieved is 0.76 Gbit/s (95 Megabytes per second), which is lower than the expected outcome of 1.2 Gbit/s compared to just using the S3 SDK by itself as in 4.4.1.

The reason for this reduction in throughput for read and write workloads is quite complex. The following list provides a bottleneck summary

1. The S3-DN first writes the block to local disk, and then to S3. This results in 2 unnecessary round-trips to S3 to check if the block exists before upload

2. The clients are running on same machine as the Datanodes

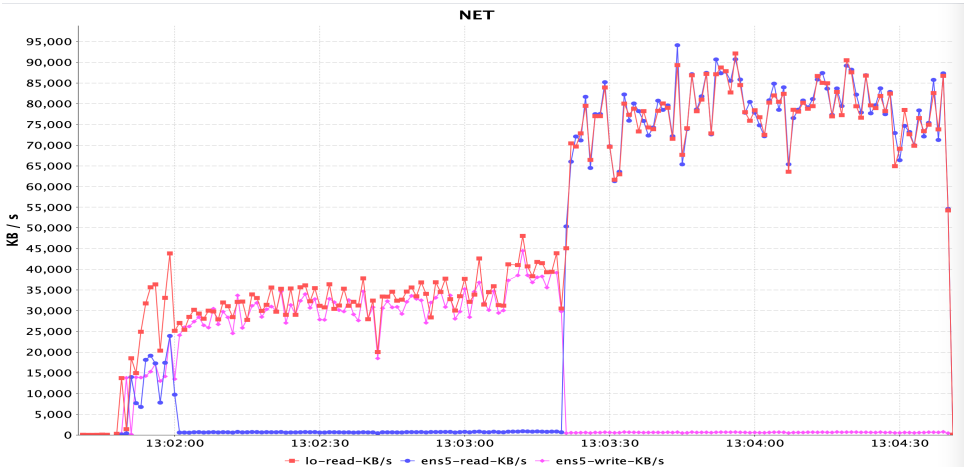

Figure 4.5: Network performance of S3 HopsFS DN. The first half includes a create workload (purple line) and the second half is read (blue). The loopback interface (red) shows the same data being sent to the client.

3. Thread synchronization issues for modifying shared structures such as the replica HashMap

4. On every replica-level read, the metadata is first read from S3 and not the in-memory HashMap (explained in 4.3.2). This also introduces possible eventual consistency problems even on new keys (discussed in 3.3)

5. The normal HopsFS cluster is reading from EBS, which is much faster than S3

When simple synchronization on getting file metadata from S3 is added to the S3-only test (A.3), the network throughput reaches similar levels as seen in 4.5. As such, the S3-DN experiences a throughput of about 25% of the normal HopsFS DN for read workloads, but can theoretically be improved to at most 38% of the original HopsFS speed (when using S3 by itself) - 150 MB/s of network throughput (see 4.4.1) compared to 400 MB/s (see A.4) when running workloads of 1 MB files.

4.5 Cost of Cloud Storage

The nature of business is to convert expenses into revenue for a company and its stakeholders. Therefore, it is critical to analyze the cost of these two different types of systems. An analysis of storing 100 TB over 1 month with HopsFS and S3 is included in Figure 4.6.

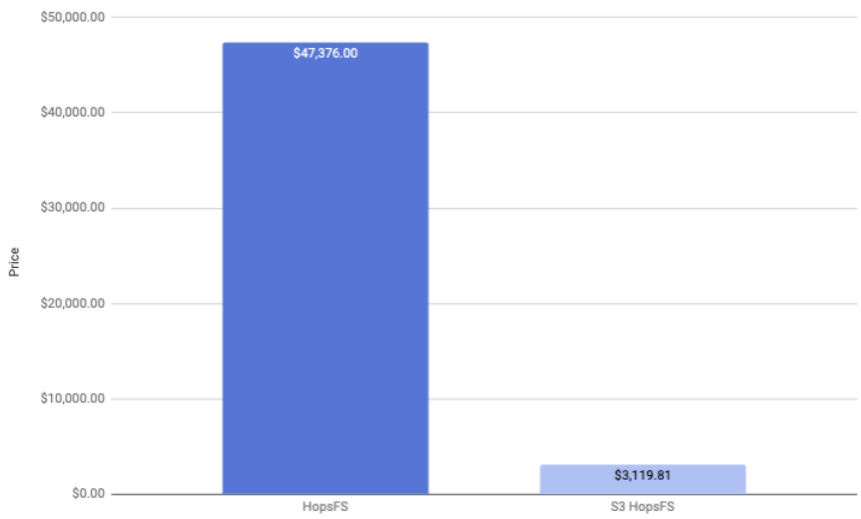

Figure 4.6: Cost of storing 100 TB for 1 month

The results show that using S3 HopsFS to store 100 TB can be at most 15 times cheaper than using a static HopsFS cluster on AWS EC2. To simply store 100 TB in HopsFS, a Replication Factor of 3 is needed to prevent data loss, making the total amount of data stored 300 TB. These machines need to be powered on all the time, because when an ephemeral EC2 instance (actual disks) is terminated all the data is lost. The HopsFS estimate uses the m1.xlarge instance, which has 4 CPUs, 15 GB RAM, four 420 GB HDDs each, and a bandwidth throughput of up to 25 Gbit/sec. The data was gathered using the Amazon AWS Cost Calculator tool [32]. Table 4.5 shows how the calculation was done.

System	GB per DN	RF	Storage (GB)	DNs	EC2 Price/hr	Total Price
HopsFS	1600	3	300 000	188	$0.35	$47 376
S3 HopsFS	–	1	100 000	1	$0.166	$3 120

Figure 4.7: Cost calculation of 100 TB over 1 month

The S3 cluster only included one Datanode because only the cost of storage is compared; it is not possible to compare the cost of throughput with storage in this case because the data is not available currently. The throughput levels seen in section 4.3.2 utilize a static HopsFS cluster with Elastic Block Store (EBS), which has read speeds similar to SSD, whereas this estimate uses HDDs so they cannot be compared cost-to-throughput wise. The S3-HopsFS

cost is also dynamic, meaning the S3 HopsFS cluster can theoretically automatically scale DNs up and down according to load. Throughput levels can reach similar levels as the static HopsFS cluster using HDDs - Databricks ran experiments where they saw S3 sustaining throughput at 600 MB/s [33]. When applications using the S3-HopsFS cluster do not need high throughput, the cost is extremely low (such as a Spark job only loading a small dataset once and then doing many calculations). In addition, S3 DNs can also be located on the actual compute cluster running Spark or MapReduce, further reducing cost and increasing throughput (since clients can read from the local DN). The compute power needed for the S3 DN is very low compared to that needed by Spark, since the S3 DN is essentially a gate to S3 and only forwarding data.

To calculate an upper bound for the cost of an S3-HopsFS cluster with equivalent throughput, a small scale experiment can be done with a static HDD HopsFS cluster. A brief estimate puts the cost at $25,523 for 100 TB - 188 t3.xlarge (4 cpu, 16 GB RAM) instances with 720 hours at $0.166/hour. The throughput can be comparable to the static HopsFS cluster using HDDs, since S3 also has similar read speeds (although no official report benchmark's S3 throughput due to the variability and black box of AWS).

The HopsFS cost estimate can also be verified by Revx.io [6]. They estimated that running a 100 TB Hadoop cluster with EC2 on AWS is around $78,000 per month; the only difference being they included a 40% free space requirement on the system; so the total reserved space includes 500 TB of data, after only storing 100 TB of data. This is because other jobs on the Hadoop cluster store transient query data on HDFS as well, thus requiring more additional space. If this requirement is removed, the estimate falls down to $47,000 per month.

4.6 Future Work

The scope of the S3 Datanode project is quite large. Here is a list of a few future work items that can improve the performance and stability of the software.

1. Cache downloaded blocks by not deleting them until later. Use Machine Learning to predict when or if a block will be used in the near future. Example application involves a log scanner tool that analyzes server logs where the same log file is analyzed over and over again as new data is appended.

2. Increase performance of the S3 DN by storing DN replica metadata in the memory HashMap, just as the current *FsDatasetImpl* does. If block

is not found in the HashMap, only then go to S3. When the DN restarts, this HashMap can be filled by first scanning the objects in S3. However, scalability bottlenecks could possibly occur if too many objects are in S3 for the DN to store in memory.

3. Modify the Namenode block report. It is not needed for the DN to report which blocks it has, since they are all stored in S3. But the block report is still needed for features such as delete.

4. Support load balancing of DNs. As system demand increases, start new DNs automatically, and remove DNs when demand is low to save cost.

5. Write the RBW blocks directly to S3 instead of to the local file system and then to S3.

Chapter 5

Conclusion

This section includes a summary of the work done in this thesis.

5.1 Results

This thesis outlines a new class of Distributed File System that utilizes native cloud storage as the data layer, instead of relying on the locally mounted file system found on commodity hardware. Named S3-HopsFS, this project reduces the cost dramatically of big data analytics, which is needed for modern businesses to make better decisions and reach their target market faster. The new system is built on top of HopsFS; a hierarchical, distributed file system with a scale-out metadata layer utilizing an in-memory, distributed database called NDB. NDB dramatically increases the scalability of the file system compared to HDFS. S3 by itself can also be used by big data analytic applications, but it is slow and eventually consistent - meaning new changes and data may not appear immediately. Therefore, we need to combine S3 with a fast, consistent file system like HopsFS.

When combining these two systems into S3-HopsFS, the price of deployment is reduced by up to 15 times compared to HDFS. Because of this, the new prototype is also four times slower than the original HopsFS system in the tests performed (25% of the original baseline speed). However, S3-HopsFS can be improved towards 38% of the original performance by optimizing thread synchronization, reducing S3 round trips, keeping Datanode replica metadata in-memory, and writing blocks directly to S3 instead of first to local disk. The prototype is available on GitHub [9] and supports read, write, and append file operations. Future work for this project includes performance optimizations and additional features, as outlined in the last evaluation section 4.6.

In addition to the new file system, a tool named S3Guard has been expanded and improved by utilizing NDB instead of Amazon's DynamoDB. S3Guard enables big data analytic applications such as Hive to utilize S3 as a direct input and output source for queries. The eventual consistency problems of S3 have been solved and improves reading files and directories by up to 36% in the tests performed. S3Guard is sufficient to support some big data analytic applications like Hive, but we lose all the benefits of HopsFS like the performance, scalability, and extended metadata - therefore we need a system like S3-HopsFS. The extended S3Guard code is available on the same GitHub repository [9].

5.2 Motivation

The motivation for utilizing cloud infrastructure compared to on-premise data centers consists of many beneficial reasons. The biggest reasons include the illusion of infinite scalability, economic savings, and flexibility. Instead of buying expensive hardware that requires floor space and technical staff members, the application can be hosted in the cloud. The app can scale according to demand; when nobody is using the app the scale can be brought down to save on costs. In addition, moving to the cloud increases internal resource pools and simplifies deployment of applications as employees can focus on product development instead of managing infrastructure.

Environmental impacts of IT businesses are also addressed by implementing more cloud computing resources. Data centers can become more green by efficiently sharing a pool of resources, automatically load balance applications, and use efficient energy-saving infrastructure hardware. In addition to these environmental impacts, data confidentiality and security are addressed by the cloud computing company, such as not allowing applications sharing infrastructure to read unauthorized content from another company, in addition to the application owners taking the right security protocols.

Appendix A

Extra Results

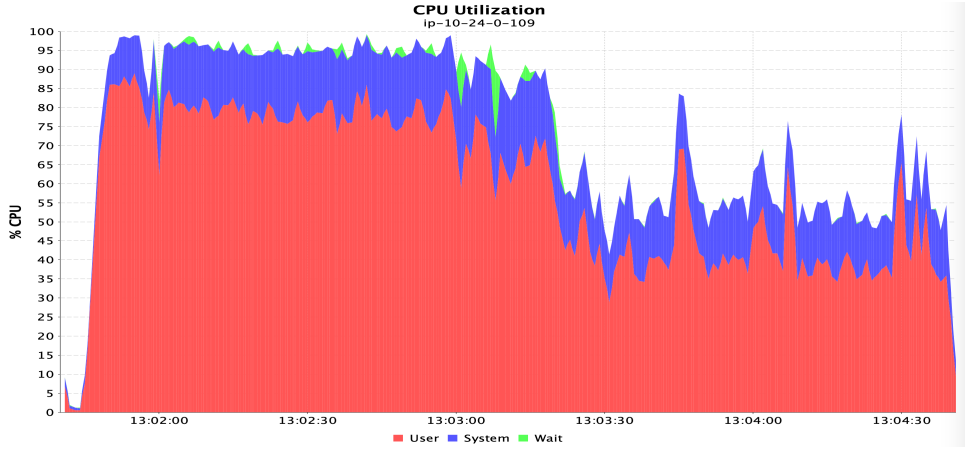

Figure A.1: CPU utilization during workloads using 1 MB files with S3-HopsFS on AWS. Explanation for create workload bottleneck in section 4.3.1. The workload is running CREATE from the start to 13;03;19 then READ from 13;03;19 until the end.

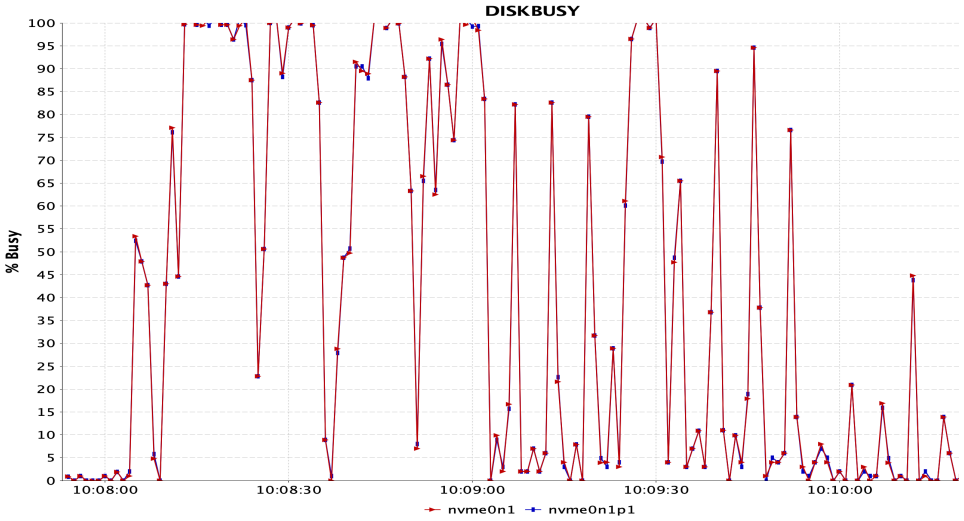

Figure A.2: Disk utilization during workloads of 1 MB files with normal HopsFS on AWS. Explanation for read workload bottleneck in section 4.3.2. The workload is running CREATE from the start to 13;03;19 then READ from 13;03;19 until the end.

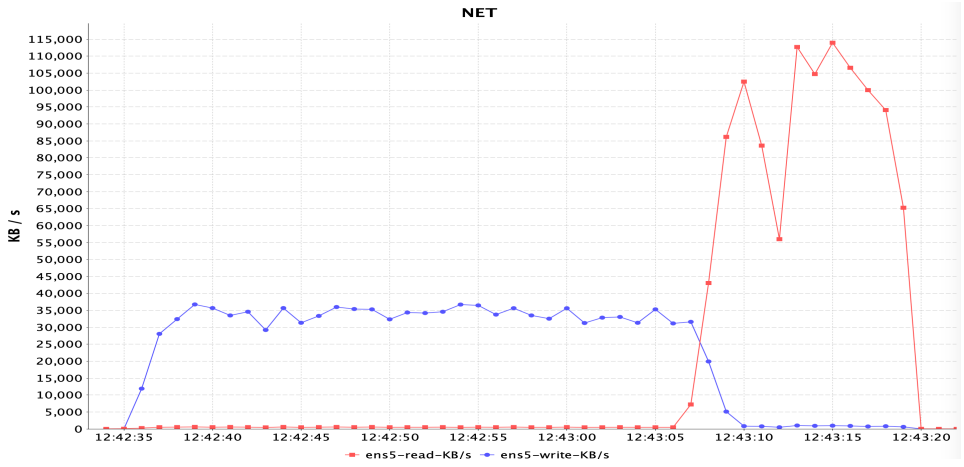

Figure A.3: Network performance of adding thread synchronization to S3-only test with 10 clients reading in parallel and locking on a get object metadata call.

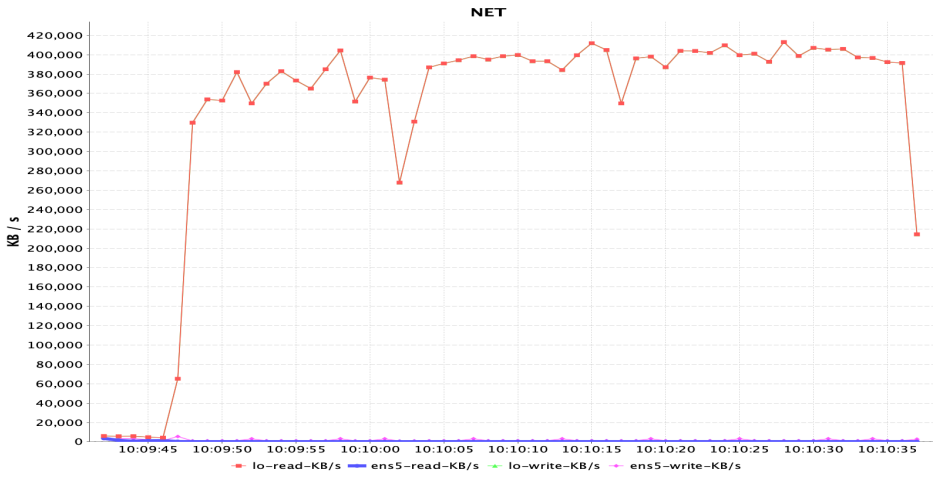

Figure A.4: Network performance during read of the static HopsFS cluster with 3 Datanodes and a replication factor of 3.

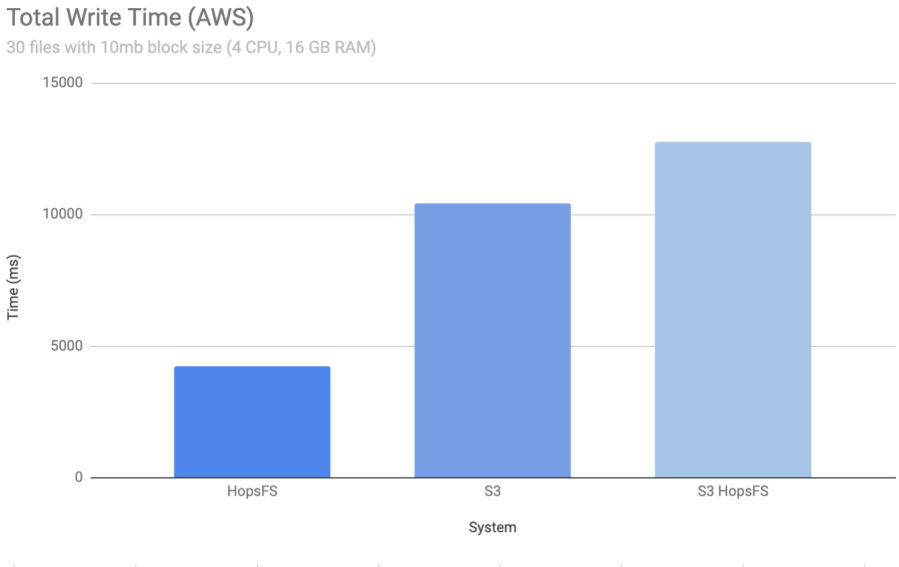

Figure A.5: Time comparison of writing 30 10mb files in AWS EC2

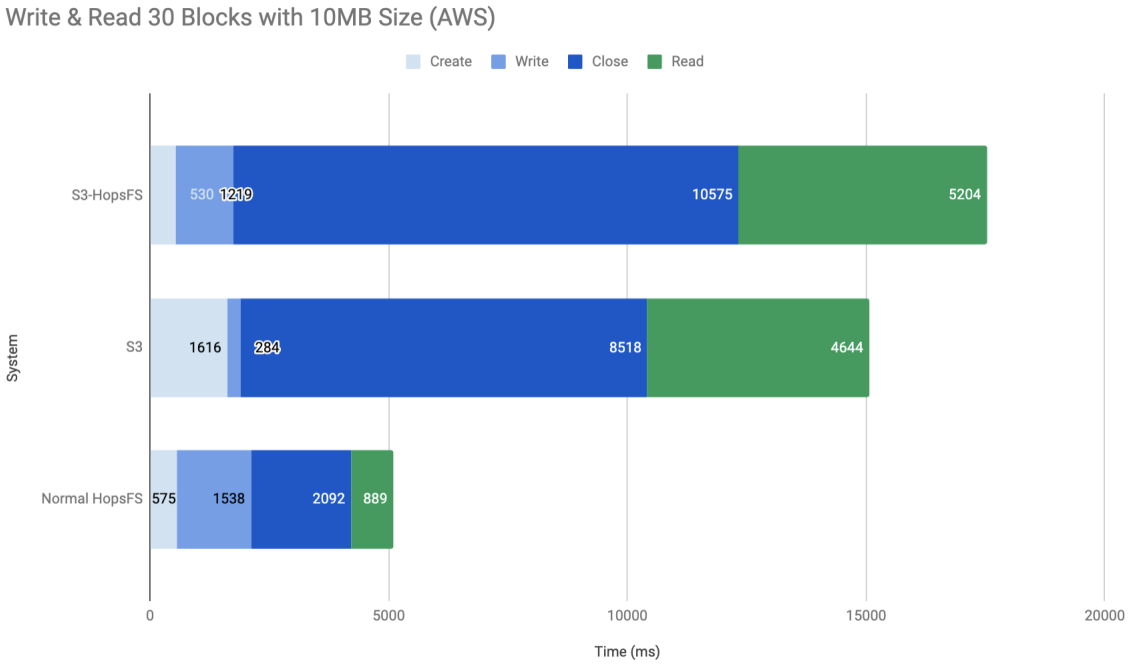

Figure A.6: Time breakdown of writing and reading 30 files with 10mb size in AWS. Note that most of the write phase (dark blue) is spent uploading for S3-HopsFS and S3.

Appendix B

How To Run

Git clone the repository Joel's S3 HopsHadoop Fork. Install and build HopsFS locally as described here HopsHadoop GitHub. IntelliJ is recommended for the coding environment.

1. Setup AWS S3, create bucket, etc
2. Setup NDB locally
3. Set in hdfs-default.xml
4. (a) fs.dataset.s3_bucket = your_bucket_name
   (b) Fs.dataset.use_s3 = true
   (c) Fs.s3a.impl = org.apache.hadoop.fs.s3a.S3AFileSystem
5. Add auth-keys.xml next to other .xml config files. For JUnit testing, add it to *hadoop-tools/hadoop-aws/src/test/resources/*
6. To benchmark the system, use this tool [30]

Example auth_keys.xml file

```
<configuration>
    <property>
        <name>fs.s3a.endpoint</name>
        <value>${ireland.endpoint}</value>
    </property>
    <property>
        <name>test.fs.s3a.name</name>
        <value>hopsfs-datanode-s3-test2</value>
    </property>
    <property>
        <name>fs.s3a.access.key</name>
        <value>*</value>
    </property>
    <property>
        <name>fs.s3a.secret.key</name>
        <value>*</value>
    </property>
</configuration>
```

How To Test S3Guard Performance

The test is located in the GitHub repository [9] and is called *S3GuardPerformanceTest*. HopsFS needs to be installed and running. Run the test on the command line in an AWS EC2 instance using this command

```
mvn -Dtest=S3GuardPerformanceTest test
```

How To Test S3-alone Performance

The test is located in the GitHub repository [9] and is called *S3PerformanceTest*. HopsFS needs to be installed and NDB running. Run the test on the command line in AWS using this command

```
mvn -Dtest=S3PerformanceTest#testS3RecordTime1MB \
_parallel_clients test
```

How To Test S3-HopsFS Performance (in JVM)

The test is located in the GitHub repository [9] and is called *S3PerformanceTest*. HopsFS needs to be installed and NDB running. Run the test on the command line in AWS using this command. Note that this uses virtualized components and only tests the code (not a deployed scenario)

```
mvn -Dtest=S3PerformanceTest#testHDFSRecordTime1MB test
```

How To Test S3-HopsFS Performance (deployed)

The hammer-bench tool [30] is used to test real, deployed clusters. Simply run `scripts/bench.sh` after setting the correct IP addresses in the following files - `datanodes`, `experiment-nodes`, `namenodes`, `master.properties`, `experiment-env.sh`

Bibliography

[1] Gareth Mitchell. “How much data is on the internet?” In: *Science Focus* (). https://www.sciencefocus.com/future-technology/how-much-data-is-on-the-internet/.

[2] J. Leon Zhaon Shaokun Fan Raymond Y.K. Lau. “Demystifying Big Data Analytics for Business Intelligence Through the Lens of Marketing Mix”. In: *Big Data Research* 2 (Mar. 2015). https://www.sciencedirect.com/science/article/pii/S2214579615000155.

[3] H. Ferhatosmanoğlu M.A. Abbasoğlu B. Gedik. “Aggregate profile clustering for telco analytics”. In: *Proceedings of the VLDB Endowment* (Aug. 2013). https://dl.acm.org/citation.cfm?id=2536284.

[4] M.B. Dias; D. Locher; M. Li; W. El-Deredy; P.J.G. Lisboa. “The value of personalised recommender systems to e-business: a case study”. In: *Proceedings of the 2008 ACM Conference on Recommender Systems, ACM, Lausanne, Switzerland* (2008). https://www.scopus.com/record/display.uri?eid=2-s2.0-63449114637&origin=inward&txGid=.

[5] Upkar Varshney Subhankar Dhar. “Challenges and business models for mobile location-based services and advertising”. In: *Communications of the ACM* (May 2011). https://dl.acm.org/citation.cfm?id=1941515.

[6] Shailesh Garg. “Cost Analysis of Building Hadoop Clusters Using Cloud Technologies”. In: *RevX.io* (). https://www.revx.io/resources/blog/cost-analysis-of-building-hadoop-clusters-using-cloud-technologies.

[7] Steve Loughran. “Using S3Guard for Amazon S3 Consistency”. In: *HortonWorks* (). https://hortonworks.com/blog/s3guard-amazon-s3-consistency/.

[8] Toby J. Velte PhD Anthony T. Velte and Robert Elsenpeter. “Cloud Computing - A Practical Approach”. In: *McGrawHill* (2010). http://www.academia.edu/download/1769375/0071626948_Cloud_Computin1.pdf.

[9] "S3 HopsFS Code Repository". In: *GitHub* (). https://github.com/MileanCo/hops.

[10] "Cloud Computing and Sustainability: The Environmental Benefits of Moving to the Cloud". In: *Accenture* (2010). https://download.microsoft.com/download/A/F/F/AFFEB671-FA27-45CF-9373-0655247751CF/Cloud%20Computing%20and%20Sustainability%20-%20Whitepaper%20-%20Nov%202010.pdf.

[11] B Golden. "Amazon Web Services for Dummies". In: *New York: John Wiley & Sons* (2010). https://fit.mta.edu.vn/files/DanhSach/Book_Amazon%20webservices%20for%20dummies.pdf.

[12] "HDFS Architecture Guide". In: *Apache Software Foundation* (). https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[13] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R))". In: *IEEE Standards Association* (2008). https://standards.ieee.org/standard/1003_1-2017.html.

[14] NetApp. "How memory works in ONTAP: NVRAM/NVMEM, MBUF & CP". In: (2018). https://whydoyoulikewafls.wiseinfotec.net/2018/12/27/how-memory-works-in-ontap-nvram-nvmem-part-1/.

[15] NetApp. "Write Anywhere File Layout". In: *Wikipedia* (). https://en.wikipedia.org/wiki/Write_Anywhere_File_Layout.

[16] Seif Haridi Salman Niazi Mahmoud Ismail and Jim Dowling. "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases". In: *USENIX Conference on File and Storage Technologies* (Mar. 2017). https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi.

[17] Mikael Ronstrom Seif Haridi Mahmoud Ismail Salman Niazi and Jim Dowling. "Scaling HDFS to more than 1 million operations per second with HopsFS". In: *KTH - Royal Institute of Technology* (2017). http://salmanniazi.com/files/2017-CCGRID-Paper.pdf.

[18] "MySQL Cluster Overview". In: *MySQL Documentation* (). https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster-overview.html.

[19] Hairong Kuang; Konstantin Shvachko; Nicholas Sze; Sanjay Radia; Robert Chansler. "Append Hflush Read Design". In: *Yahoo! HDFS Team* (June 2009). https://drive.google.com/open?id=1KOQVsHAb5O5D2KusNW-VRT4WMPbwCKTa.

[20] "Investigation of Replication Factor for Performance Enhancement in the Hadoop Distributed File System". In: *Companion of the 2018 ACM/SPEC International Conference, At Berlin* (Apr. 2018). https://www.researchgate.net/publication/324370408_Investigation_of_Replication_Factor_for_Performance_Enhancement_in_the_Hadoop_Distributed_File_System.

[21] benstopford. "Understanding the Shared Nothing Architecture". In: *benstopford* (2009). http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/.

[22] "Introduction to Amazon S3". In: *Amazon Web Services Documentation* (). https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html.

[23] Kyle Galbraith. "Quick Explanation Of The S3 Consistency Model". In: *Medium* (Feb. 2018). https://codeburst.io/quick-explanation-of-the-s3-consistency-model-6c9f325e3f82.

[24] "S3Guard: Consistency and Metadata Caching for S3A". In: *Apache Software Foundation* (). https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/s3guard.html.

[25] Gabriel Vilén. "Mounting External Storage in HopsFS". In: *TU Delft* (Oct. 2018). https://repository.tudelft.nl/islandora/object/uuid%3A1e1e9f5c-74ba-4cfd-aacb-2602977f148b.

[26] "WekaIO Matrix Architecture Technical White Paper". In: *Weka.io* (Dec. 2018). https://www.weka.io/pdf-content/wekaio-hpc-storage-architecture/.

[27] "ProtocolBuffers". In: *Hadoop Wiki* (). https://wiki.apache.org/hadoop/ProtocolBuffers.

[28] Jeff Barr. "The Floodgates Are Open – Increased Network Bandwidth for EC2 Instances". In: *Amazon Web Services News Blog* (Jan. 2018). https://aws.amazon.com/blogs/aws/the-floodgates-are-open-increased-network-bandwidth-for-ec2-instances/.

[29] "Enhanced Networking on Linux". In: *Amazon Web Services Documentation* (). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

[30] "HDFS-Distributed-BenchMark". In: *GitHub Repository* (). https://github.com/smkniazi/hammer-bench.

[31] "Amazon EBS features". In: *AWS Documentation* (). https://aws.amazon.com/ebs/features/.

[32] "AWS Cost Calculator". In: *Amazon Web Services* (). https://calculator.s3.amazonaws.com/index.html.

[33] Josh Rosen Reynold Xin and Kyle Pistor. "Top 5 Reasons for Choosing S3 over HDFS". In: *DataBricks* (May 2017). https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html.

TRITA-EECS-EX-2019:307

www.kth.se