KTH ROYAL INSTITUTE
OF TECHNOLOGY

Doctoral Thesis in Information and Communication Technology

Distributed File System Metadata and its Applications

MAHMOUD ISMAIL

Stockholm, Sweden 2020

Distributed File System Metadata and its Applications

MAHMOUD ISMAIL

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2020

School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2020:65
ISBN 978-91-7873-702-4

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i informations och kommunikationsteknik fredagen den 18 december 2020 klockan 9.00 i Sal C, Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista .

© Mahmoud Ismail, December 2020

Tryck: Universitetsservice US AB

Abstract

Distributed hierarchical file systems typically decouple the storage and serving of the file metadata from the file contents (file system blocks) to enable the file system to scale to store more data and support higher throughput. We designed HopsFS to take the scalability of the file system one step further by also decoupling the storage and serving of the file system metadata. HopsFS is an open-source, next-generation distribution of the Apache Hadoop Distributed File System (HDFS) that replaces the main scalability bottleneck in HDFS, the single-node in-memory metadata service, with a distributed metadata service built on a NewSQL database (NDB). HopsFS stores the file system's metadata fully normalized in NDB, then it uses locking primitives and application-defined locks to ensure strongly consistent metadata.

In this thesis, we leverage the consistent distributed hierarchical file system metadata provided by HopsFS to efficiently build new classes of applications that are tightly coupled with the file system as well as to improve the internal file system operations. First, we introduce hbr, a new block reporting protocol for HopsFS that removes a scalability bottleneck that prevented HopsFS from scaling to tens of thousands of servers. Second, we introduce HopsFS-CL, a highly available cloud-native distribution of HopsFS that deploys the file system across Availability Zones in the cloud while maintaining the same file system semantics. Third, we introduce HopsFS-S3, a highly available cloud-native distribution of HopsFS that uses object stores as a backend for the block storage layer in the cloud while again maintaining the same file system semantics. Fourth, we introduce ePipe, a databus that both creates a consistent change stream for HopsFS and eventually delivers the correctly ordered stream with low latency to downstream clients. That is, ePipe extends HopsFS with a change-data-capture (CDC) API that provides not only efficient file system notifications but also enables polyglot storage for file system metadata. Polyglot storage enables us to offload metadata queries to a more appropriate engine - we use Elasticsearch to provide a free-text search of the file system namespace to demonstrate this capability. Finally, we introduce Hopsworks, a scalable, project-based multi-tenant big data platform that provides support for collaborative development and operations for teams through extended metadata.

Sammanfattning

Distribuerade hierarkiska filsystem kopplar vanligtvis bort lagring och hanteringen av filens metadata från filens innehåll (filsystemets block) för att göra det möjligt för filsystemet att skala bättre för att lagra mer data och stödja högre genomströmning. Vi utformade HopsFS för att ta skalbarheten i filsystemet ett steg längre genom att även koppla bort lagring och hantering av filsystemets metadata. HopsFS är en öppen källkod, nästa generations distribution av Apache Hadoop Distribuerade Filsystem (HDFS) som ersätter den huvudsakliga skalbarhetsflaskhalsen i HDFS, en nod som lagrar all metadata i minnet, med en distribuerad metadatatjänst byggd på en NewSQL-databas (NDB). HopsFS lagrar filsystemets metadata fullt normaliserat i NDB, och använder sedan låsande primitiver och applikationsdefinierade lås för att säkerställa starkt konsistent metadata.

I denna avhandling använder vi den konsistenta distribuerade hierarkiska filsystemmetadata som tillhandahålls av HopsFS för att effektivt bygga nya klasser av applikationer som är tätt kopplade till filsystemet samt för att förbättra filsystemets interna funktioner. Först introducerar vi hbr, ett nytt blockrapporteringsprotokoll för HopsFS som tar bort en skalbarhetsflaskhals som hindrade HopsFS från att skalas till tiotusentals servrar. För det andra introducerar vi HopsFS-CL, en mycket tillgänglig molnbaserad distribution av HopsFS som distribuerar filsystemet över tillgänglighetszoner i molnet samtidigt som samma filsystemsemantik bibehålls. För det tredje introducerar vi HopsFS-S3, en mycket tillgänglig molnbaserad distribution av HopsFS som använder objektlagring som en backend för blocklagringslagret i molnet samtidigt som samma filsystemsemantik bibehålls. För det fjärde introducerar vi ePipe, en databus som båda skapar en konsistent förändringsström för HopsFS och så levererar korrekt beställd ström med låg latens till nedströmsklienter. Det vill säga ePipe utökar HopsFS med ett CDC-API (Change-data-capture) som inte bara ger effektiva filsystemmeddelanden utan också möjliggör polyglot-lagring för filsystemets metadata. Med polyglot-lagring kan vi avlasta metadatafrågor till en mer lämplig sökmotor - vi använder Elasticsearch för att tillhandahålla en fritext-sökning i filsystemets namnområde för att visa denna förmåga. Slutligen introducerar vi Hopsworks, en skalbar, projektbaserad big data-plattform om stödjer flera användare och ger stöd för samarbetsutveckling och drift för team med hjälp av utökad metadata.

To the loving memory of my parents.

Acknowledgements

I owe my deepest gratitude to my primary advisor, *Jim Dowling*, for his guidance, patience, and support during all these years. Without whom, this thesis would not have been possible. I would also like to express my gratitude to my second advisor, *Seif Haridi*, for his valuable discussions, suggestions, and contributions that have helped me not only within this thesis context but also to broaden my understanding of distributed systems. I want to extend my appreciation to my advance reviewer, *Sarunas Girdzijauskas*, for his comments and insightful suggestions to this thesis.

I want to thank all my co-authors for their contributions to the papers included in this thesis. Primarily, my colleague and friend, *Salman Niazi*, for all the time we spent developing and optimizing HopsFS. It was a long process with a lot of challenges to overcome. I would also like to thank *Mikael Ronström* for his insightful suggestions and support, making it easier to use NDB optimally.

I want to thank all my friends and colleagues at KTH, RISE SICS, and Logical Clocks AB for their support during my studies. I want to thank *Alex*, *Lars*, and *Paris*, for the interesting discussions we had over many lunches, coffee breaks, and concerts. I would also like to extend my appreciation to *Ahmed*, *Efi*, *Nicolae*, *Ian*, and *Marco*. I would also like to thank *Robin* for helping me with the Swedish abstract.

I'm very grateful to my friends and housemates, *Bakr*, *Karim*, and *Mina* for making Sweden home to me over the past years. I want to extend my gratitude to all my friends across Egypt, Europe, and the USA. Specially, *Shady*, *Loay*, *Obda*, *Walid*, and *Hatem* for all the travels, discussions, and all day long availability that kept me going through all that time. Furthermore, I would like to thank all the many friends I have made over the years in Sweden. Not to forget anyone, thank you all for making my stay in Sweden so enjoyable.

I'm very thankful to *Gintare* for always supporting and motivating me along this process. I'm also very thankful to my *sisters* and their families for always being there. Finally, no words can describe how much I am grateful to my *parents* for their love, motivation, support, and sacrifices over the years. They have always put my sisters and me ahead of themselves. May their souls rest in peace.

Included Papers

Paper *I*

Scaling HDFS to more than 1 million operations per second with HopsFS.
Mahmoud Ismail, Salman Niazi, Mikael Ronström, Seif Haridi, Jim Dowling. *In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain.* **Winner of IEEE Scale Challenge Award.**

Paper *II*

HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases.
Salman Niazi, ***Mahmoud Ismail***, Mikael Ronström, Steffen Grohsschmiedt, Seif Haridi, Jim Dowling. *In 15th USENIX Conference on File and Storage Technologies, Santa Clara, California, USA, 2017.*

Paper *III*

Scalable Block Reporting for HopsFS.
Mahmoud Ismail, August Bonds, Salman Niazi, Seif Haridi, Jim Dowling. *In IEEE International Congress on Big Data (BigData Congress), Milan, Italy, 2019.* **Winner of Best Student Paper Award.**

Paper *IV*

Distributed Hierarchical File Systems strike back in the Cloud .
Mahmoud Ismail, Salman Niazi, Mauritz Sundell, Mikael Ronström, Seif Haridi, Jim Dowling. *In 40th IEEE International Conference on Distributed Computing Systems (ICDCS), Singapore, 2020.*

Paper *V*

HopsFS-S3: Extending Object Stores with POSIX-like Semantics and more.
Mahmoud Ismail, Salman Niazi, Gautier Berthou, Mikael Ronström, Seif Haridi, Jim Dowling. *In Proceedings of the 21st International Middleware Conference Industrial Track, The Netherlands, 2020.*

Paper *VI*

ePipe: Near Real-Time Polyglot Persistence of HopsFS Metadata.
Mahmoud Ismail, Mikael Ronström, Seif Haridi, Jim Dowling. *In 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Larnaca, Cyprus, 2019.*

Paper *VII*

Hopsworks: Improving User Experience and Development on Hadoop with Scalable, Strongly Consistent Metadata.
Mahmoud Ismail, Ermias Gebremeskel, Theofilos Kakantousis, Gautier Berthou, Jim Dowling. *In 37th IEEE International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 2017.*

Contents

I Thesis Overview 1

1 Introduction 3
1.1 Thesis Statement 6
1.2 Thesis Contributions 7
1.3 List of Publications 7
1.4 Thesis Outline 9

2 Thesis Context 11
2.1 Hadoop Distributed File System (HDFS) 11
2.1.1 File system operations 12
2.1.2 File system notifications 14
2.1.3 Authorization systems 15
2.1.4 S3A connector 15
2.2 HopsFS 16
2.2.1 The metadata storage layer 16
2.2.2 The metadata serving layer 25
2.2.3 The block storage layer 29
2.3 Current Limitations of HopsFS and HDFS 29

3 Thesis Contributions 31
3.1 Improving HopsFS high availability and scalability 31
3.1.1 hbr: Scalable block reporting for HopsFS 32
3.1.2 HopsFS-CL: Highly Available HopsFS in the cloud . 33
3.1.3 HopsFS-S3: Extending Object Stores with POSIX-like semantics 35

3.2 New Classes of applications 38
3.2.1 ePipe: Polyglot Persistence for HopsFS metadata . . 38
3.2.2 Hopsworks: Improving user experience and data management 39

4 Conclusions and Future Work 41

Bibliography 45

II Included Papers 51

Paper I: Scaling HDFS to more than 1 million ops/sec with HopsFS 53
1 Introduction 55
2 Background 57
2.1 Hadoop Distributed File System (HDFS) 57
2.2 HopsFS 58
2.3 Network Database (NDB) 58
2.4 Different Types of NDB Read Operations 59
3 HopsFS Key Design Decisions 60
3.1 Partitioning Scheme 62
3.2 Fine Grained Locking 62
3.3 Optimizing File System operations 63
3.4 Caching 63
4 Configuring HopsFS and NDB 64
4.1 Optimizing NDB cluster setup 64
4.2 Thread locking and Interrupt Handling 65
5 Scalability Evaluation 66
5.1 Throughput Scalability 66
5.2 Metadata Scalability 67
5.3 Effect of the Database Optimization Techniques . . . 68
6 Load on the Database 68
7 Conclusions 70
8 Acknowledgements 70

Paper II: HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases 73
1 Introduction 75
2 Background 77
2.1 Hadoop Distributed File System 78

2.2 Network Database (NDB) 79
3 HopsFS Overview 81
4 HopsFS Distributed Metadata 82
4.1 Entity Relation Model 83
4.2 Metadata Partitioning 84
5 HopsFS Transactional Operations 85
5.1 Inode Hint Cache 86
5.2 Inode Operations 87
6 Handling Large Operations 89
6.1 Subtree Operations Protocol 90
6.2 Handling Failed Subtree Operations 91
6.3 Inode and Subtree Lock Compatibility 92
7 HopsFS Evaluation 92
7.1 Experimental Setup 92
7.2 Industrial Workload Experiments 93
7.3 Metadata (Namespace) Scalability 95
7.4 FS Operations' Raw Throughput 96
7.5 Operational Latency 98
7.6 Failure Handling 99
7.7 Block Report Performance 101
8 Related Work 102
9 External Metadata Implications 103
10 Summary 104
11 Acknowledgements 104

Paper III: Scalable Block Reporting for HopsFS 113
1 Introduction 115
2 Background 118
2.1 Block and Replica States 119
2.2 Block Reporting Protocol 120
3 Problem Definition 120
4 *hbr* 121
4.1 System Model 121
4.2 Implementation 122
5 Evaluation 127
5.1 Throughput of *hbr* 127
5.2 Latency of *hbr* 128
5.3 Effect of invalid buckets 129
5.4 Block Report Size 130

5.5 Load on NDB and Namenodes 131
6 Related Work 132
7 Conclusion 133

Paper IV: Distributed Hierarchical File Systems strike back in the Cloud **137**
1 Introduction 140
2 Background 141
2.1 HopsFS 141
2.2 NDB 143
3 Challenges to deploy HopsFS in the cloud 146
4 AZ-Awareness throughout the Stack 147
4.1 Metadata Storage Layer 147
4.2 Metadata Serving Layer 152
4.3 Block Storage Layer 153
5 Evaluation 154
5.1 Experiment Setup 154
5.2 Throughput 156
5.3 End-to-End latency 159
5.4 Resource utilization 160
5.5 AZ-Local Reads 164
5.6 Failures 165
6 Related Work 166
7 Conclusions 167

Paper V: HopsFS-S3: Extending Object Stores with POSIX-like Semantics and more **173**
1 Introduction 176
2 Background and Related work 177
3 HopsFS-S3 180
3.1 HopsFS operations 180
3.2 HopsFS-S3 operations 181
4 Evaluation 184
4.1 Terasort 185
4.2 TestDFSIOEnh 188
4.3 Metadata operations 190
5 Conclusions 191

Paper VI: ePipe: Near Real-Time Polyglot Persistence of HopsFS Metadata **199**

1 Introduction 201
2 Background 203
2.1 Hadoop Distributed File System (HDFS) 203
2.2 HopsFS 204
3 Notifications for HopsFS 205
3.1 Selective Logging of Files/Dirs in HopsFS 205
3.2 Consuming the ***frol*** entries 206
3.3 NDB Event API 206
4 ePipe 207
4.1 System Model 207
4.2 Architecture 209
4.3 Failure Recovery 211
5 ePipe Use Cases 212
5.1 Metadata search 212
5.2 Applications for ePipe 213
5.3 Extended Metadata 213
5.4 Hopsworks 214
5.5 Apache Hive 214
6 Evaluation 215
6.1 Overhead of *frol* extension to HopsFS 216
6.2 ePipe vs. HDFS INotify vs. Trumpet 217
6.3 ePipe vs. HDFS find 219
6.4 ePipe performance 220
6.5 ePipe Recovery 222
6.6 Statistics from a Production Cluster 223
6.7 AutoIncrement Columns Overhead 223
7 Related Work 225
8 Conclusions 226

Paper VII: Hopsworks: Improving User Experience and Development on Hadoop with Scalable, Strongly Consistent Metadata 231
1 Introduction 234
2 System Overview 236
2.1 Concepts 237
2.2 Under the hood 239
3 Services 241
3.1 Extended Metadata 241
3.2 Search 242
3.3 Job Launcher 242

3.4 Kafka 242
3.5 Zeppelin 243
4 Demo 243

Part I

Thesis Overview

1 Introduction

Not Dead Yet!
Hierarchical File Systems Won't Die.
— Tony Mason, Margo Seltzer

Over the past decades, file systems have been the defacto standard to organize and store users' data. Files and directories are organized in a tree where inner nodes represent directories, and the leaves represent files. The increased demand for bigger file systems to support the increasingly growing data has lead to the development of distributed hierarchical file systems. Typically, these file systems decouple the storage and serving of the file system's metadata from the actual file system's data to enable higher throughput, better availability, and scalability. The Hadoop Distributed File System (HDFS) [1] is one of the most prominent file systems developed in that domain to support big data platforms. HDFS mainly consists of a metadata server, called Namenode, and a set of data servers called Datanodes. The file system's metadata is stored in the metadata server's main memory, while the actual file's data is split into blocks that are replicated across the datanodes.

HDFS suffered from bottlenecks that limited its scalability, preventing it from supporting larger clusters. The metadata server (Namenode) is the major scalability bottleneck in the HDFS architecture, where the memory

is limited to the amount of memory that be handled on the heap of the Java virtual machine. This memory limitation, typically around 300-500 GB, limits the number of files and directories that a cluster can store and limits the metadata server's throughput and performance. As a solution, we took the scalability one step further by decoupling the storage of the metadata from the serving of the metadata. We developed HopsFS, a next-generation distributed hierarchical file system, to mitigate the HDFS scalability bottlenecks. HopsFS leverages the recent improvements in the NewSQL databases, a new class of distributed databases with a shared-nothing architecture, and supports cross partition transactions.

HopsFS stores the file system's metadata fully normalized in a shared-nothing, distributed database. More specifically, HopsFS stores the metadata for files and directories in an inodes table where each file or directory is mapped to a single row. HopsFS ensures consistent access to the file system's metadata through the use of transactions and primitive locking supplied by the database layer, as well as application-defined locking provided by HopsFS. HopsFS consists of three main layers; the metadata storage layer, the metadata serving layer, and the block storage layer. The default and recommended metadata storage layer is NDB. The metadata serving layer consists of multiple stateless metadata servers. The block storage layer consists of multiple datanodes that store the data blocks for the file system's files. The decoupling of the metadata storage from the metadata serving opened up the file system potential to support new classes of application. It also enabled the development of algorithms and protocols to ensure the high availability and scalability of the file system in different and new environments.

Decoupling the storage and serving of the file system's metadata enables the scalability of the file system. However, it can result in inconsistencies between the metadata storage layer and the block storage layer. Thus, it requires introducing a periodic synchronization protocol between the metadata layer and the block storage layer to ensure the consistency of the file system's metadata and its blocks. In HopsFS and HDFS, this synchronization protocol is called block reporting, where each datanode in the cluster periodically sends information about its local blocks. The network and processing overhead of the existing block reporting protocol increases with cluster size, which would ultimately limit the cluster scalability. We developed hbr, a new block reporting protocol for HopsFS to tackle the existing protocol bottlenecks enabling HopsFS to scale to tens

of thousands of block storage servers.

HopsFS was developed mainly for on-premise installations, where an administrator manages the cluster of machines. To deploy HopsFS in the cloud, cloud service providers recommend deploying across availability zones to ensure availability in case of failures of an availability zone. Availability zones are, in fact, data centers that are connected with low-latency links to form a region. Each availability zone has independent power and networking infrastructure, reducing the probability of correlated failures of all availability zones in a given region. A service is considered highly available (HA) in a region if it can survive the failure of an availability zone. Internally, HopsFS uses synchronous replication protocols at the metadata storage layer, which requires low-latency (sub-millisecond) between servers. However, recent cloud networking advancements between availability zones have made it more viable to deploy HopsFS in the cloud. We developed HopsFS-CL, a redesign of HopsFS, that natively provides high availability across availability zones while maintaining the same file system semantics.

Cloud service providers offer object stores a cheap and scalable alternative to distributed hierarchical file systems despite their lower performance and relaxed metadata semantics. Object stores lack critical atomic operations such as atomic directory rename, an essential operation for implementing transactional operations in big data lake frameworks [2–4], and SQL-on-Hadoop frameworks [5]. On the other hand, HopsFS provides strong metadata semantics, ensuring atomic operations such as directory rename. We developed HopsFS-S3 to leverage the scalability, high availability, and low cost of object stores as well as the strong metadata semantics provided by HopsFS. HopsFS-S3 is a redesign of HopsFS that transparently uses object stores as a backend for the block storage layer in cloud deployments.

The externalizing of the file system's metadata in a distributed database enables the administrators to leverage the SQL API provided by the database to draw insights from the file system's metadata, for example, "Which are the biggest files in the file system?", "Which files have been modified this week?" etc.. However, many such queries have the potential to overload the metadata storage layer, which in turn will overload the whole file system. Also, NDB doesn't support full-text search functionality. As the conventional database wisdom says, no size fits all; that is, we need the most suitable engine to query the metadata with different query

patterns supplied by the user without adding non-negligible overhead to the file system. Therefore, we designed ePipe to provide replicated metadata as a service with negligible overhead on HopsFS. ePipe is a databus that streams the file system's changelog and eventually delivers the correctly ordered stream of changes to downstream subscribers. The ePipe architecture is pluggable to allow different types of downstream applications and different use cases.

HopsFS is extensible by design thanks to the externalization of the file system's metadata to the metadata storage layer (NDB). That allows us to implement new abstractions and features on top of the file system, without interrupting its internal operations, using extended metadata. Moreover, we can ensure the strong consistency of these abstractions and features using both transactions and foreign key constraints, assuring the file system metadata's integrity. Recent growth in machine learning and analytics has led to specialized development and infrastructure roles such as data scientist, data analyst, data engineer, and machine learning infrastructure engineer. Teams with different roles often collaborate to curate and analyze large datasets, necessitating developing a secure and collaborative platform for storing and analyzing large datasets. We introduce Hopsworks, a secure, scalable, multi-tenant, and collaborative big data platform built on top of HopsFS. Hopsworks provides system support for collaborative development and operations between the diverse roles, where sensitive data can be securely stored on a shared cluster. Hopsworks leverages the extended metadata of HopsFS to implement three new abstractions projects, datasets, and users. A project is a set of capabilities/privileges defined over a subset of users and a subset of datasets. A dataset is a subdirectory in the file system that holds the user's data and can be securely shared with other users.

1.1 Thesis Statement

> Consistent, customizable metadata in distributed hierarchical file systems enables improvements in the scalability and high availability of the file system and allows for new capabilities such as fast free-text search, new security models, and cloud-native integration.

1.2 Thesis Contributions

This thesis contribution is threefold. First, we show how HopsFS uses modern NewSQL databases to store the file system's metadata consistently, providing higher throughput and better scalability than HDFS. Second, we identify different bottlenecks with HopsFS that limit its scalability and adoption in cloud environments. To tackle these bottlenecks, we developed a new block reporting protocol (hbr) and two new extensions to HopsFS (HopsFS-CL and HopsFS-S3) to enable efficient, scalable and highly available deployments of HopsFS in the cloud. Third, we leverage the extensibility of HopsFS to build different systems on top that both extend the file system's metadata and have strongly consistent lifecycles for the extended files/directories. More specifically, we developed ePipe to provide consistent replication of the file system's metadata into different stores, allowing efficient query and storage of the file system metadata. We then developed Hopsworks to ease the development and operations of different teams, enabling secure and multi-tenant access while using the same HopsFS cluster.

1.3 List of Publications

The list of publications that are part of this thesis:

1. Scaling HDFS to more than 1 million operations per second with HopsFS. ***Mahmoud Ismail***, Salman Niazi, Mikael Ronström, Seif Haridi, Jim Dowling. *In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain.*

   Author's Contributions: The thesis author contributed to choosing efficient design decisions and optimization techniques to allow the proposed system (HopsFS) to scale to 1 million operations per second. He also contributed to the implementation and experimentation of HopsFS. The thesis author wrote the majority of the text and designed the figures in this paper. This paper won the 10th IEEE International Scalable Computing Challenge (SCALE 2017).

2. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. Salman Niazi, ***Mahmoud Ismail***, Mikael Ronström, Steffen Grohsschmiedt, Seif Haridi, Jim Dowling. *In 15th USENIX*

Conference on File and Storage Technologies, Santa Clara, California, USA, 2017.

Contributions: The thesis author contributed to the design, implementation, performance optimization, and evaluation of the proposed system (HopsFS). He also contributed to the writing and figures design of this paper. The thesis author is one of the two main contributors to HopsFS.

3. Scalable Block Reporting for HopsFS. ***Mahmoud Ismail***, August Bonds, Salman Niazi, Seif Haridi, Jim Dowling. *In IEEE International Congress on Big Data (BigData Congress), Milan, Italy, 2019.*

   Contributions: The thesis author designed and implemented the proposed protocol (hbr) presented in this paper. He also contributed to developing and performing experiments and evaluations. He wrote the majority of the text and designed the figures in this paper. This paper won the Best Student Paper award in IEEE BigData-Congress 2019.

4. Distributed Hierarchical File Systems strike back in the Cloud . ***Mahmoud Ismail***, Salman Niazi, Mauritz Sundell, Mikael Ronström, Seif Haridi, Jim Dowling. *In 40th IEEE International Conference on Distributed Computing Systems (ICDCS), Singapore, 2020.*

   Contributions: The thesis author designed and implemented the proposed system (HopsFS-CL) presented in this paper. He also performed the experiments and evaluations, wrote the majority of the text, and designed the figures in this paper.

5. HopsFS-S3: Extending Object Stores with POSIX-like Semantics and more. ***Mahmoud Ismail***, Salman Niazi, Gautier Berthou, Mikael Ronström, Seif Haridi, Jim Dowling. *In Proceedings of the 21st International Middleware Conference Industrial Track, The Netherlands, 2020.*

   Contributions: The thesis author contributed to the design and implementation of the proposed system (HopsFS-S3) presented in this paper. He also performed the experiments and evaluations, wrote the majority of the text, and designed the figures in this paper.

6. ePipe: Near Real-Time Polyglot Persistence of HopsFS Metadata.***Mahmoud Ismail***, Mikael Ronström, Seif Haridi, Jim Dowling. *In 19th*

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Larnaca, Cyprus, 2019.

Contributions: The thesis author designed and implemented the proposed system (ePipe) presented in this paper. He also performed the experiments and evaluations, wrote the majority of the text, and designed the figures in this paper.

7. Hopsworks: Improving User Experience and Development on Hadoop with Scalable, Strongly Consistent Metadata. ***Mahmoud Ismail***, Ermias Gebremeskel, Theofilos Kakantousis, Gautier Berthou, Jim Dowling. *In 37th IEEE International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 2017.*

   Contributions: The thesis author contributed to the design and implementation of the proposed system (Hopsworks) presented in this paper. He also wrote the majority of the text and designed the figures in this paper.

Other publications by the author of the thesis that are not part of this thesis:

1. Leader Election using NewSQL Database Systems. Salman Niazi, ***Mahmoud Ismail***, Gautier Berthou, Jim Dowling. *In Distributed Applications and Interoperable Systems (DIAS): 15th IFIP WG 6.1 International Conference, DAIS 2015. Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015.*

2. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. Salman Niazi, ***Mahmoud Ismail***, Seif Haridi, Jim Dowling. *In: Sakr S., Zomaya A.Y. (eds) Encyclopedia of Big Data Technologies. Springer, Cham.*`https://doi.org/10.1007/978-3-319-77525-8_146`

1.4 Thesis Outline

This thesis is organized into two parts. The first part provides an overview of the thesis. Chapter 2 gives an overview of the thesis context. Chapter 3 provides a summary of the thesis contributions, and Chapter 4 provides the thesis conclusions and future work. The second part presents the

thesis contributions in further detail as a collection of published research papers.

2
Thesis Context

In this chapter, we provide the essential background needed to understand the context of the thesis contributions. First, we provide an overview of the architecture of the Hadoop Distributed File System (HDFS) and its pitfalls. Then, we give an overview of the architecture of HopsFS, its improvements to HDFS, and its shortcomings.

2.1 Hadoop Distributed File System (HDFS)

HDFS [1] is a distributed hierarchical file system that decouples the storage of the file system's metadata from the file system's data, following the design of the google file system (GFS) [6]. HDFS typically consists of a single active metadata server (Namenode) and a set of block storage servers (Datanodes). The metadata server stores the file system's metadata in memory, while the file data is split into blocks that are replicated across the block storage servers for fault tolerance using a replication pipeline protocol [1]. HDFS is an append-only file system that provides single writer multiple reader semantics. Internally, the metadata server uses a global lock to synchronize access to the file system's metadata. The metadata server is a single point of failure (SPOF). That is, if the metadata server goes down, the whole file system becomes unavailable. To avoid such a scenario, HDFS provides high availability (HA) setup based on an active-standby failure model [7,8], as shown in Figure 2.1.

An HA HDFS cluster typically consists of a single active metadata server, a set of standby metadata servers, a set of Zookeeper nodes, and a set of

journal nodes. The active metadata server logs the file system operations in a transaction log called the edit log, where each file system operation is assigned with a monotonically increasing `transaction_id`. The edit logs are written to a quorum of journal nodes for durability. The standby metadata servers asynchronously read the journal nodes' edit logs and apply the changes locally in-memory. The Zookeeper nodes are used for failover, to detect when the active metadata server fails and reliably elect one of the standby metadata servers as the new active metadata server. This architecture, however, does not address the limitations of the single active metadata server. The number of files/directories stored on the file system is limited by the size of the metadata server's memory, upper limited by the maximum size of the JVM heap [9]. Moreover, the metadata server runs as a Java process where garbage collection pauses are inevitable, which would potentially disrupt the metadata server's operations, which will likely affect the performance of the metadata server [10, 11]. The failover is not instantaneous, depending on how up-to-date the journal nodes are and how fast the zookeeper nodes detect a failure.

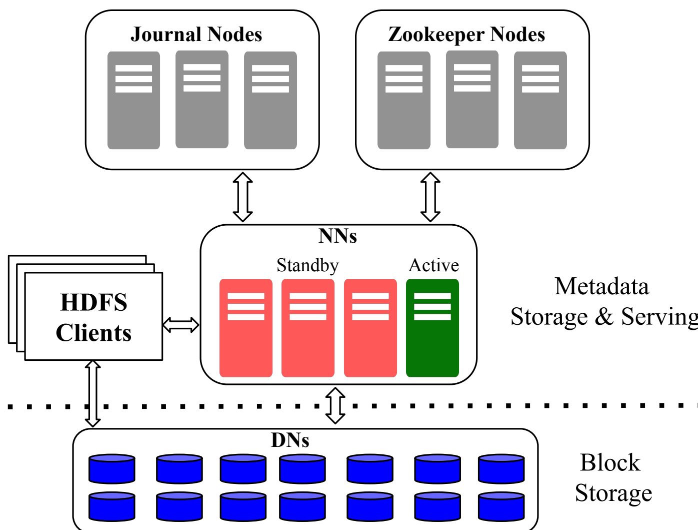


Figure 2.1: An architecture diagram of HA HDFS. A typical cluster consists of a single active metadata servers, a set of standby metadata servers, a set of journal nodes, a set of Zookeeper nodes, and a set of datanodes.

2.1.1 File system operations

Writing a file

To write a file to HDFS, the client first sends a request to the metadata server (Namenode) to create the file; the metadata server adds the file to the file system's namespace and responds to the client. Then, the client splits the file into blocks of configurable size (Default: 128 MB). For each block, the client sends a request to the metadata server to add a new block. The metadata server then randomly chooses a list of block storage servers (Datanodes) based on the block placement policy and responds with the list of the selected block storage servers, typically three servers. HDFS uses chain replication where the client starts writing the block sequentially to the first datanode in the list, then the datanode writes to the second datanode in the list, and so on until reaching the last datanode in the chain. The acknowledgment is sent in reverse order until it reaches the client, and if more blocks need to be written, the client requests another new block from the metadata server. The process repeats until all blocks are written, and the file is closed.

Block reporting

HDFS uses a block reporting protocol to synchronize the state of the blocks between the metadata server and the block storage servers to ensure the file system's consistency. Each block storage server periodically, every 6 hours, sends a block report containing information about all the block replicas it has. The metadata server cross-checks the received block report with the replicas map for that block storage server. Then, in case of mismatches, the metadata server issue commands to account for these mismatches, such as creating a new replica of an under replicated block or removing a corrupt block.

Reading a file

To read a file from HDFS, the client sends a request to the metadata server to locate the file. The metadata server responds with the list of the file's blocks and for each block, the list of block storage servers where the block is stored. The client then starts reading the blocks in order while using any of the replicas for each block.

2.1.2 File system notifications

HDFS implements a file system notification service (*inotify*) in the name-node that provides notifications about changes on the file system to clients [12, 13]. Clients will periodically poll the metadata server for new transactions (file system operations) that happened after a given transaction_id. HDFS INotify could be used to implement different polyglot persistence scenarios for the file system metadata besides the notification functionality, such as replicating the file system metadata into different stores. Still, in practice, inotify introduces excessive overhead on the metadata server. For this reason, the HDFS community has started exploring alternative solutions such as starting a shadow metadata server that responds only to inotify requests [12]. The metadata server also keeps only a subset of the edit log in an in-memory cache, so reading an older transaction will impose extra network round trips to read the data from the journal nodes. As a solution, the metadata server should write the edit logs locally and write to the journal nodes, but that will add a storage constraint on an already overburdened metadata server. Another solution would be to implement a caching mechanism for the most requested transactions in memory to save some round trips to the journal nodes.

HDFS INotify currently has poor security support, working only with superuser privileges, and it doesn't support fine-grained watches over a specific directory. The HDFS INotify service is intrusive by design and could potentially overload the metadata server. Therefore, alternative systems such as Trumpet [14] were developed to provide a non-intrusive inotify service for HDFS. Trumpet periodically polls the edit logs from the local file system of the metadata or a journal node. Then, it publishes the transactions as events into a Kafka topic, that is used by the clients to consume the events as it comes. However, polling the local file system and publishing to Kafka adds non-negligible overhead that increases the replication lag compared to the native inotify service provided by HDFS. HDFS provides a *find* operation to search through the file system's namespace based on the file/directory name. This operation is inefficient by nature since it scans the whole file system namespace without any indexes.

2.1.3 Authorization systems

HDFS supports traditional POSIX file system permissions and Access control lists (ACLs) to protect and limit access to files and directories from non-designated users and groups. Other systems in the Hadoop ecosystem, such as Hive [5], Hbase [15], Kafka [16], etc., support different methods to authorize users. That is why systems such as Apache Ranger [17] and Apache Sentry [18] were developed to provide a centralized authorization service for different systems and components of the Hadoop ecosystem. Ranger provides a centralized web interface where administrators can create policies that manage access to various resources (could be a file, directory, table, or column). These policies are stored in an internal relational database in the Ranger server. Ranger provides a set of plugins for each system/component of the Hadoop ecosystem. The plugin is loaded as part of the running system. For instance, the HDFS plugin is loaded as part of the metadata server (namenode). The plugin periodically, every 30 seconds by default, polls the policies from the Ranger server. The plugin intercepts the user request and checks if the user has the necessary access rights to proceed as defined in the user's policies. In the HDFS plugin case, if policies are not specified, the authorization proceeds with normal file permissions and ACLs checks. Due to the plugins' polling nature, inconsistencies may arise between the Ranger server and the Ranger plugins. There is also the problem that users need to understand the precedence rules between Ranger policies and HDFS permissions and ACLs - a user may see in HDFS that a file should be readable but not realize a ranger policy prevents file access.

2.1.4 S3A connector

Object stores have become the defacto standard for storage in the cloud due to their low cost, scalability, and high availability. However, they provide relaxed metadata semantics and lack critical file system operations such as atomic directory rename. HDFS provides file system connectors for different object stores such as Amazon S3 using S3A connector [19]. S3A mitigates the relaxed semantics of Amazon S3 by using S3Guard [20]. S3Guard is an experimental feature of the S3A connector that keeps track of the objects (files) stored in S3 and its associated metadata in a highly available consistent database (Amazon DynamoDB [21]). Internally, for each operation on S3A that modifies the S3 objects, S3Guard writes the changes to the database. Then, S3Guard consults the records from

the database for any subsequent operation before proceeding with the operation. That allows for improved performance and a consistent view of the metadata, especially for operations such as directory listing and file status. Still, due to the relaxed semantics of Amazon S3, newly created or updated files may not be immediately available to read after create or update operations. That is, readers may read stale data. Moreover, directory rename operation is still not atomic, affecting systems that use this operation for implementing commit protocols on top of HDFS such as big data lake frameworks [2–4], and SQL-on-Hadoop frameworks [5]. For that reason, S3A implements a commit protocol that can consistently work with Amazon S3 [22]

2.2 HopsFS

HopsFS [23,24] is a next-generation distribution of the Hadoop distributed file system (HDFS) [1]. We designed HopsFS to tackle the limitations and shortcomings of HDFS by replacing the single metadata architecture with a distributed metadata architecture. As such, we decouple the storage of the file system's metadata from its access. The file system's metadata is stored fully normalized in a shared-nothing, in-memory, distributed database, while multiple stateless metadata servers serve client requests. HopsFS consists of three main layers, the metadata storage layer, the metadata serving layer, and the block storage layer, see Figure 2.2. In this section, we present a description of each of the layers that constitute HopsFS. For more in-depth details about HopsFS refer to ***Paper I*** [24] and ***Paper II*** [23].

2.2.1 The metadata storage layer

The metadata storage layer is a NewSQL database [25] that is responsible for the storage of the file system's metadata. NewSQL databases are a class of OLTP (Online transaction processing) relational databases that typically offer SQL APIs while maintaining the ACID semantics at the scale of NoSQL databases. These databases usually employ shared-noting, in-memory, and distributed architecture. The default and recommended database in HopsFS is NDB (Network database), the storage engine of MySQL Cluster [26, 27]. NDB horizontally partitions the tables' data across datanodes in the NDB cluster. NDB provides high throughput, high availability, and real-time performance through the use of features such as row-level locking, application-defined partitioning (ADP), distributed

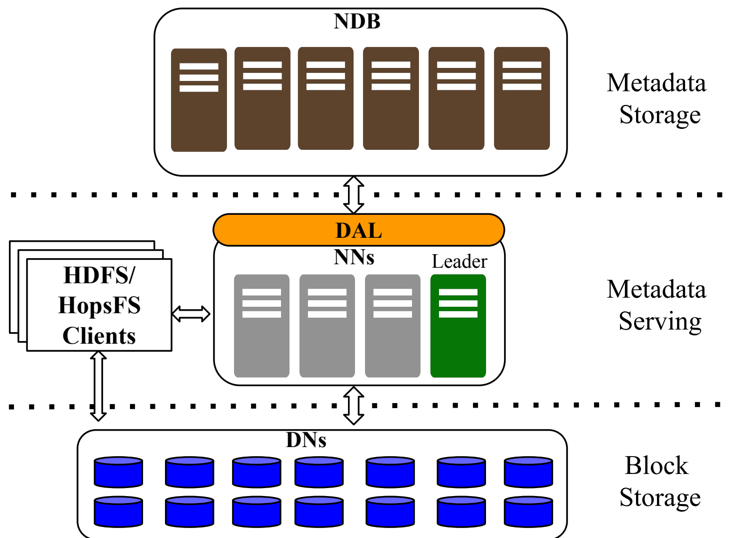


Figure 2.2: An architecture diagram of HopsFS. HopsFS consists of three layers; the metadata storage layer, the metadata serving layer, and the block storage layer. The default metadata storage layer is NDB. Small files, files with size less than 128 KB, are stored with the file's metadata in NDB.

aware transactions (DAT), as well as parallel cross-partition transactions. Currently, we only support NDB as the metadata storage layer in HopsFS. However, we provide a pluggable architecture using the DAL layer, as shown in Figure 2.2, to allow the implementation of different database connectors. Alternative NewSQL databases such as MemSQL [28] and SAP Hana [29] can be used to implement the metadata storage layer, while others, like VoltDB [30] are not recommended since they serialize cross-partition transactions.

NDB Architecture

A typical NDB cluster consists of at least one management node, multiple NDB datanodes, and one or more MySQL servers, as shown in Figure 2.3. The management nodes are responsible for configuration and arbitration in case of network partitions. The MySQL servers are responsible for handling SQL queries submitted to the cluster. Besides SQL, NDB provides a high-performance native APIs to interact with the cluster. Nodes that use the native APIs are called API nodes (such as HopsFS metadata servers). For fault tolerance, NDB replicates the tables' data across NDB datanodes. The NDB datanodes are organized into replication groups called node

groups. The number of node groups is calculated as N/R, where R is a configurable parameter to control the number of replicas in the cluster, and N is the total number of datanodes in the cluster. The number of datanodes within a node group is the NDB replication factor (R). For example, as shown in Figure 2.3, given a default replication factor set to 2, there are 4 NDB datanodes in the cluster organized into 2 node groups, where each node group has 2 NDB datanodes. Each NDB datanode within a node group contains a full replica of the tables' data assigned to that node group. That is, if an NDB datanode fails, the other nodes in the node group could take over. The NDB cluster can survive as long as there is still at least one alive NDB datanode in every node group, where a datanode is declared dead if it misses four heartbeat intervals in a row (20 seconds by default). For each partition in every node group, one NDB datanode will be assigned as a primary replica while the other nodes will be backup replicas. For durability, NDB datanodes record the in-memory data changes to REDO and UNDO transaction logs on disk. Also, the NDB datanodes asynchronously checkpoint all the in-memory data to disk and then truncate the REDO and UNDO logs to bound the size of the logs and shorten the datanode recovery time. Moreover, NDB provides a global checkpointing protocol to write all on-going transactions and their state to disk to survive cluster-level and datanode failures.

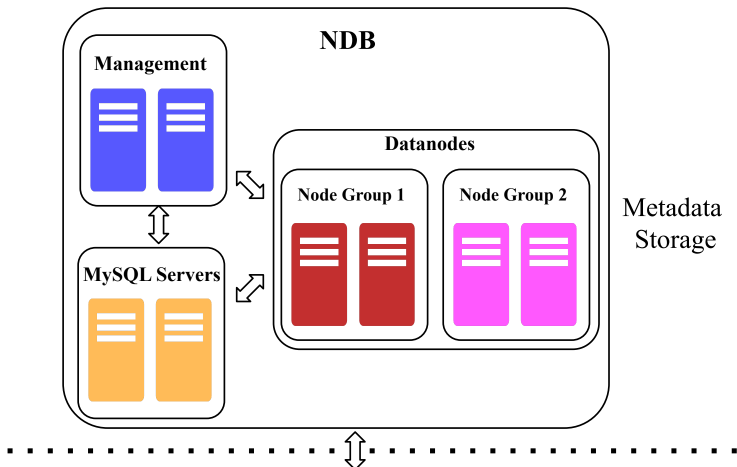


Figure 2.3: An architecture diagram of NDB. A typical cluster consists of at least one management node, a set of datanodes, and one or more MySQL server nodes. The NDB datanodes are grouped into different node groups depending on the configured replication level (2 by default).

NDB Transactions

NDB supports only *read-committed* transaction isolation level to yield high performance. Non-repeatable reads could occur in a transaction with read-committed isolation level [31]. However, NDB provides a row-level locking primitive, which can be used by applications to provide stronger consistency requirements. In HopsFS, we use the row-level locking, supplemented by an application-defined lock for subtree operations, see Section 2.2.2, to ensure the strong consistency of the file system. Each NDB datanodes runs a group of threads categorized according to their functionality, such as local data manager (LDM) and transaction coordinator (TC) threads. The number of these threads is configurable and can be tuned to improve the performance of NDB. The LDM threads are responsible for handling the different table's partitions assigned to this datanode. In contrast, the TC threads are responsible for handling all transactions, including cross partition transactions, in the cluster.

NDB horizontally partitions a table across NDB datanodes based on the MD5 hash of the primary key. Application-defined partitioning (ADP) is a feature provided by NDB that allows application developers to override the default NDB partitioning and define their partitioning scheme for their tables. NDB also provides distribution-aware transactions (DAT), enabling application developers to supply a transaction hint based on the partitioning scheme to enforce starting the transaction on an NDB datanode that potentially has the required data for that transaction. Incorrect transaction hints will not cause any inconsistencies since the transaction coordinator will reroute the request to the appropriate local data manager (LDM) that holds the required data.

For concurrency control, NDB uses the Strict Two-Phase locking protocol [32]. The transaction protocol starts by acquiring all the locks necessary and does not release these locks until the transaction commit point is reached. NDB always locks the primary replica first, then the backup replicas to avoid deadlocks. NDB implements a non-blocking, distributed commit protocol that uses the Two-Phase commit protocol (2PC) [33] across rows while using a Linear Two-Phase Commit protocol (L2PC) for each row [26]. The protocol works as depicted in Figure 2.4. First, the transaction coordinator (TC) sends the *Prepare* message to the primary replica, which propagates it to the backup replicas until it reaches the last backup replica, which sends the *Prepared* message to the TC. Then, the

Commit message is sent in the reverse order (to the last backup replica). Once the *Commit* message reaches the primary replica, the locks are released on the primary replica, and a *Committed* message is sent to the TC. Once the TC receives all the *Committed* messages from all the primary replicas for all rows, the transaction is committed and an acknowledgement is sent to the application. In parallel, to yield high performance, the TC sends *Complete* message to the primary that forwards it to the backup replica to release the locks on the backup replicas. Then, the backup replicas respond with a *Completed* message to the primary replicas, which in turn forward the *Completed* message to the TC. The backup replicas are asynchronously updated after the transaction is committed, and the application is acknowledged. Therefore, NDB ensures that all read requests are routed to the primary replicas to avoid any inconsistencies.

A TC may fail while transactions are running. Therefore, NDB implements a transaction take-over protocol that elects another TC, which reads the state of all ongoing transactions and proceed with the transactions as usual. An NDB cluster will survive as long as at least one datanode in each node group is alive. NDB implements a node failure detection and heartbeat protocols that are used by the surviving nodes on the cluster to agree on which nodes have already failed. Moreover, to ensure liveness, NDB provides timeouts such as *TransactionInactiveTimeout* to abort the transaction in case of client failure or abandoning, and *TransactionDeadlockDetectionTimeout* to abort the transaction in case of node failures, high load, and deadlocks. HopsFS uses these timeouts, as mentioned earlier, to implement a transaction retry mechanism.

NDB Operations

NDB supports different types of operations to access the database. We order the operations based on their cost in NDB. This cost entails the operation latency and throughput, as well as overhead on NDB. The order is as follows:

1. **Primary Key (PK)**: This operation reads/writes/updates a single row in/from a table in NDB using appropriate locks (read, write, or read-committed). NDB provides low latency and high throughput for this type of operation. This operation is the most efficient type of operation in NDB since it touches a single partition where the primary key infers that partition. NDB, by default, distributes the

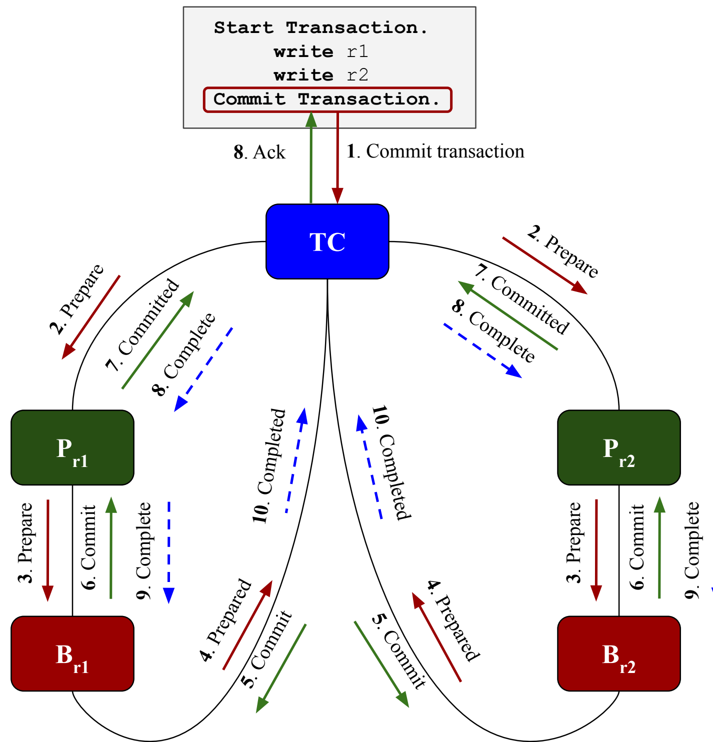


Figure 2.4: A simple description of the NDB commit protocol while committing a transaction to write two rows (r1, r2) to two different partitions. P_{r1} *is the primary replica for r1, while* B_{r1} *is the backup replica for r1. Similarly for r2,* P_{r2} *is the primary replica and* B_{r2} *is the backup replica.*

rows in a table based on their primary key unless if an application-defined partitioning is used, then NDB uses the partition key instead. The partition key must be part of the primary key. Moreover, by leveraging the distribution-aware transaction (DAT) feature, we can ensure that the operation runs on the NDB datanode that holds the rows for that partition.

2. **Batched Primary Key (B)**: This operation is an extension to the PK operation that leverages the batching technique used in the traditional databases to provide higher throughput at the cost of higher latency by making efficient use of the network bandwidth. Batches of PK operations are executed as part of a single transaction.

3. **Partition Pruned Index Scan (PPIS)**: This operation is an index scan operation that is local to a single NDB partition. This operation

requires defining an application-defined partitioning scheme for the tables. NDB will ensure that rows with the same partition key will always reside on the same partition. Moreover, by leveraging the distribution-aware transaction (DAT) feature, we can ensure that the operation runs on the NDB datanode that holds the partition.

4. **Index Scan (IS)**: This operation is a regular index scan operation that hit all partitions in NDB. Its cost scales linearly with the number of partitions, compared to **PPIS** operation, that have a constant cost.

5. **Full Table Scan (FTS)**: This operation reads all rows in all the partitions in NDB without using any indexes.

HopsFS was designed such that most of the operations are **PK**, **B** and **PPIS** operations while the **IS** operations are used in very limited non-critical path situations, and **FTS** operations are avoided.

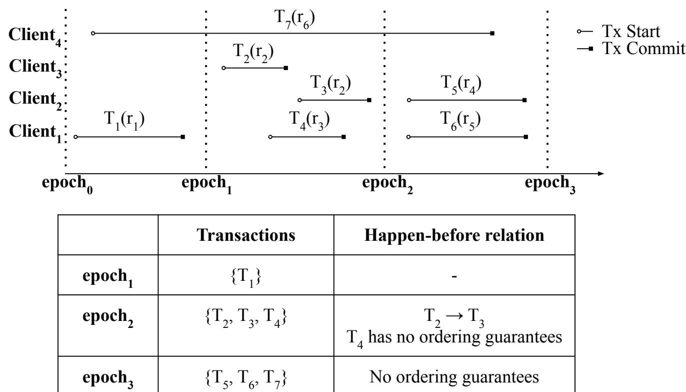
NDB Event API

NDB provides a publish-subscribe API called the Event API that allows applications to subscribe for change events (write, update, or delete) on tables on NDB. Then, NDB streams the committed changes on that table to the corresponding subscribers. Each NDB datanode independently generates a list of change events for the transactions it was involved in. Then, using the Event API, the changes from all NDB datanodes are grouped to form a consistent batch using their *epochs*, a logical clock periodically incremented across all NDB datanodes. Once all the change events from an *epoch* are received, the Event API sends these change events to the subscriber's application.

NDB implements a logical clock known as an epoch that is periodically and atomically incremented across all nodes in the cluster using a leader driven protocol [34]. Epochs are used to maintain a total order of events on the cluster, that is required by various internal functions such as grouping sets of committed transactions together for later use by the NDB Event API. The epoch itself is a 64-bit number that consists of two parts; the Global Checkpoint Index (GCI) and microGCI. The GCI is the 32 most significant bits, incremented during a checkpoint by the Global Commit Protocol (GCP), every 2 seconds by default. The microGCI is the 32 least significant bits, which is incremented quite frequently, every 100 milliseconds by default, by a GCP variation. The difference between any two successive

epoch numbers is not guaranteed to be always one since every time the GCI is incremented the microGCI is reset to zero.

An epoch contains zero or more committed transactions, and a transaction can appear only in one epoch. Long-running transactions can span multiple epochs, but they will only be recorded at the epoch in which they are committed, for example, T_7 as shown in Figure 2.5. Transactions committed in epoch $n+1$ have happened after transactions committed in epoch n. That is, row changes in epoch $n+1$ have happened after row changes in epoch n. But within the same epoch, there is no guarantee over the order of transactions. Concurrent transactions can only interact using row locks, which enforces ordering over the transactions operating on the same rows within the same epoch, for example, T_2 and T_3, as shown in Figure 2.5. However, it is not the case for concurrent transactions working on independent rows, for example, T_5 and T_6 as shown in Figure 2.5. Due to these ordering properties, epochs only guarantee a partial order on the events that make up the database changelog. The NDB Event API leverages epochs to deliver a partially ordered stream of change events happening on the rows in any table. Each NDB database node guarantees that events happened in epoch n to be delivered before events happened in epoch $n+1$.



	Transactions	Happen-before relation
epoch$_1$	$\{T_1\}$	-
epoch$_2$	$\{T_2, T_3, T_4\}$	$T_2 \rightarrow T_3$ T_4 has no ordering guarantees
epoch$_3$	$\{T_5, T_6, T_7\}$	No ordering guarantees

Figure 2.5: Ordering of events within and across epochs. $\mathtt{epoch}_1$ *contains* T_1 *operating on row* r_1, $\mathtt{epoch}_2$ *contains* T_2, T_3, T_4 *operating on rows* r_2, r_2, r_3 *respectively, and* $\mathtt{epoch}_3$ *contains* T_5, T_6, T_7 *operating on rows* r_4, r_5, r_6 *respectively.* T_2 *and* T_3 *are operating on the same rows, but we can induce the desired ordering due to the lock* T_2 *managed to acquire the lock first in this case. However, in the case of* T_5 *and* T_6 *the transactions could appear in any order in the database changelog.*

NDB uses a global commit protocol (GCP) to increment the epoch number

across all cluster nodes. The GCP is a variant of a 2-phase commit protocol that piggybacks the new epoch in the second phase of the protocol. The GCP follows a similar approach to the Single Mark algorithm described in [35]. In the NDB cluster, one NDB node is elected as master. If the current elected master fails, then another node is elected to take over as master. NDB guarantees in-order (FIFO) delivery of messages between nodes in the cluster. The protocol runs periodically at each NDB node. It operates in two phases, a preparation phase where each NDB data node freezes the current prepared transactions from beginning to commit. In contrast, already committing transactions can continue to commit normally, and ongoing transactions can continue working normally. In the second phase, the master increments the current GCI and piggybacks the newly assigned epoch number on the commit message to all nodes. Then, each node unfreezes the transactions so they can start committing and assign the new epoch to them. The GCP runs periodically every *TimeBetweenGlobalCheckpoints*, 2 seconds by default. Another variation of the GCP is the microGCP, which is used to increment the microGCI, and it runs more frequently every *TimeBetweenEpochs*, 100 milliseconds by default. The main difference between both protocols is that microGCP doesnt cause the transaction logs that were updated during that microGCI to be flushed to disk. A consequence of only flushing transaction logs at the end of a GCP is that, on catastrophic system failure, only transactions up to the last the completed GCI are recoverable.

HopsFS metadata

HopsFS stores the file system metadata fully normalized in NDB. The file system metadata is represented as rows in tables on NDB. Each inode (file/directory) is represented as a row in the *inodes* table. A file consists of zero or more data blocks where the metadata for each block is represented as a row in the *blocks* table. The data blocks are replicated where the metadata for each replica is represented as a row in the *replicas* table. Also, there are different tables to represent metadata for the different states of the blocks and replicas, such as under replicated blocks, over replicated blocks, and excess replicas. HopsFS has in total ≈ 90 tables to represent the file system's metadata and its supported features, such as access control lists (ACLS). HopsFS leverages the ADP (Application-defined partitioning) feature provided by NDB to ensure the locality of the file system's metadata. All the tables except the *inodes* table are partitioned by the inodeId. That is, all the metadata related to an inode

(file/directory) will reside on the same NDB partition (on a single NDB datanode). However, the *inodes* table is partitioned by the parent inodeId. That is, all the immediate children inodes will reside on the same NDB partition. To avoid hotspots, HopsFS implements a random partitioning scheme for the top-level directories.

2.2.2 The metadata serving layer

The metadata serving layer is responsible for handling file system requests from potentially thousands of HopsFS and HDFS clients. In HopsFS, metadata servers (namenodes) are stateless servers that, in parallel, access and mutate the file system metadata stored on the metadata storage layer (NDB). The metadata servers use the data access layer (DAL) to operate on the file system's metadata stored on NDB, see Figure 2.2.

HopsFS Transactions

The file system's metadata operations are encapsulated in HopsFS transactions that internally map to NDB transactions. HopsFS divides the file system metadata operations into two categories; *inode operations* and *subtree operations*. Inode operations operate on a single inode (file/directory) such as create, read, and delete a file, while subtree operations are operating on a subtree of the namespace such as move, rename and delete a directory, which could potentially have millions of children inodes. HopsFS implements a concurrency control mechanism based on the pessimistic concurrency control. It ensures that all file system metadata needed for a transaction are locked with a shared or exclusive lock before the transaction starts operating on those metadata [32]. HopsFS uses the row-level locking primitive provided by NDB to serialize conflicting transactions, which offers a fine-grained locking mechanism. On the other hand, HDFS uses a single global lock to lock the whole namespace when processing any file system operation (with a single concurrent write and multiple concurrent readers). Moreover, HopsFS implements an application-defined lock for subtree operations to serialize conflicting subtree operations on the namespace.

HopsFS uses hierarchical locking (implicit locking) mechanism to limit the number of locks taken by the transaction without sacrificing the correctness of the transaction [36]. HopsFS ensures that all file system operations always start with an inode that is locked according to the file

system operation type. Simultaneously, all the other associated metadata are read with no lock (using read-committed). That is, the associated metadata is guarded by the lock of their inode. All file system operations are path-based, and in any given file system operation, only a subset of the path components are mutated. Therefore, HopsFS only locks the last component (inode) of the path, and in some cases, it also locks the second last component while reading the rest of the path with no locks (using read-committed).

For example *'mkdir /usr/local/mydir'* is a file system operation to create a directory under the parent directory *'/usr/local'*. In HopsFS, each path component is represented in NDB as a row in the inodes table where the path component name, parent inode id, and partition key are the composite primary key of the table. Therefore, to create the directory '/usr/local/mydir', HopsFS will first traverse the path components and reads their corresponding metadata from NDB. It will then evaluate the metadata against the operation logic, such as checking the permissions and quotas. Every metadata server in HopsFS implements an *inodes cache* to cache the primary keys for each of the path components to avoid multiple round trips to the database, that otherwise would be needed to traverse the primary keys for the inodes.

Moreover, HopsFS uses the distribution-aware transaction (DAT) feature provided by NDB to ensure the locality of HopsFS transactions. That is, a HopsFS transaction runs on the NDB node with all/most of the needed metadata for that transaction. To use the DAT feature, HopsFS requires to provide the partition key for the inodes table at the start of the transaction. For that, we leverage the *inodes cache* to get the required partition key for the transaction to ensure the locality of the metadata.

HopsFS ensures that the file system metadata are locked in the same total order in all transactions to avoid deadlocks. That is, HopsFS traverses the file system namespace using a left-ordered depth-first search. Every HopsFS transaction runs in four phases, pre-transaction phase, lock-acquire phase, transaction processing phase, and final phase to commit or rollback the transaction. Figure 2.6 shows a template of a HopsFS transaction. A more detailed description of each of the HopsFS transaction phases is as follows:

Pre-Tranasction phase: HopsFS gets the partition key associated with

```
Pre-Transaction:
  1. Get the partition for the path from the inodes cache.
  2. Set the partition key for the transaction.
Begin Transaction
Acquire Locks:
  3. Traverse the path locally in memory using the inodes cache.
  4. Validate the path components read from the cache up to the
     penultimate inode using a batch primary key operation.
  5. If cache miss or invalid path components then
       Traverse the path recursively up to the penultimate inode.
  6. If ongoing subtree operation on the path then
       Abort transaction and retry transaction after timeout.
  7. Lock the last inode and/or the penultimate inode depending on
     the operation.
  8. Read all associated metadata for the resolved inodes and store
     in the transaction cache.
Processing:
  9. *Do the work needed by the file system operation*
Update
  10. Commit the changes captured by the transaction cache
      or rollback if the transaction is aborted
End Transaction
```

Figure 2.6: A template of a HopsFS transaction.

the file system path from the *inodes cache*. If there is no partition key in the cache for the provided path, then a random partition key is used. Note that using a random partition key will not compromise the transaction's correctness, see Section 2.2.1. HopsFS sets the transaction partition key to be the key provided by the cache or fallback to a random partition key.

Acquire Locks phase: HopsFS traverses the path locally in memory using the *inodes cache*. It then validates the correctness of the cache by reading the path components using batched primary key operation. If some or all path components are invalid, then HopsFS updates these components in the *inodes cache* accordingly and then falls back to the recursive traversal mechanism. If an existing directory along the path has an ongoing subtree operation, the transaction will abort and retry again later to ensure the consistency of the file system's

metadata. So, for subtree operations, HopsFS follows an optimistic concurrency control model. The last inode and in some cases, the second last inode is locked based on the lock type required by the operation (shared or exclusive). Then, all the metadata associated with the resolved inodes is read as defined by the transaction. For instance, if the last inode in the path is a file, then the file blocks, their replicas, and the other associated metadata tables are read. The resolved inodes and their associated metadata are kept in memory in a transaction local cache.

Processing phase: HopsFS does the work required for processing the file system operation. For instance, if a *mkdir* operation is being processed, then HopsFS checks the permissions and quotas of the parent directories and checks if the requested directory does not exist, and then updates the inodes table to add a new directory. During the processing phase, the transaction will read and update the file system metadata in the transaction local cache.

Update phase: HopsFS commits the changes in the transaction local cache to the database. If the commit succeeds, the transaction is done. Otherwise, the transaction is aborted, and the changes are rolled back from the database.

Leader Election

HopsFS implements a leader election service on top of the metadata storage layer [37]. The metadata servers use the leader election service to elect one of them as the leader. The leader election service runs every 2 seconds by default and ensures there is only one active leader at a time. The leader metadata server is responsible for housekeeping operations, such as sending commands to the datanodes to delete invalidated or over replicated blocks or replicate under replicated blocks.

HopsFS Clients

HopsFS is compatible with HDFS APIs. Hence, both HDFS and HopsFS clients can be used to access HopsFS. However, HopsFS clients are recommended since they provide load balancing and faster failover. A HopsFS client can connect to any of the metadata servers to execute file system operations and queries for all active metadata servers in the cluster. HopsFS clients provide three different policies to connect to the metadata servers.

First, a *round robin* policy where the client selects a new metadata server in round-robin every time it receives a request to execute a file system operation. Second, a *random* policy, where the client selects a random metadata server for every file system operation. Third, a *sticky* policy, where the client selects a random metadata server and sticks with it until it fails, then it selects another random metadata server. The sticky policy is the default and recommended policy since it improves the hit-rate for the *inodes cache* in the metadata servers.

2.2.3 The block storage layer

The block storage layer is responsible for storing the actual data of the file system. The files are split into blocks of configurable size, typically 128 MB, which are replicated across different nodes in the blocks storage layer, usually 3 nodes, similar to HDFS. The number of files and directories an HDFS cluster can store is limited due to the metadata server's scalability bottlenecks. For that reason, HDFS recommends using even bigger block sizes and discourages storing small files in the cluster. However, in reality, the percentage of small files, < 1 MB, in production clusters is typically high ($> 50\%$) [38]. On the other hand, HopsFS does not have the same limitation on the number of files due to the use of the distributed metadata storage layer. Moreover, HopsFS leverages NVMe disks and the NDB on-disk column feature to store the small files,< 1 MB, with their metadata in NDB [38].

2.3 Current Limitations of HopsFS and HDFS

We introduced HDFS, the de facto standard distributed file system. HDFS offers a single metadata server architecture that suffered from scalability bottlenecks limiting the number of files/directories in a cluster, the size of a cluster, increasing failover time during failures, and introducing garbage collection pauses in the JVM. Then we introduced HopsFS, a version of HDFS that uses a distributed metadata architecture to overcome the scalability bottlenecks suffered by HDFS. HopsFS provides a consistent distributed hierarchical file system metadata that can scale to much bigger and larger clusters than HDFS without compromising the file system's consistency. However, HopsFS lacks support for efficient and fast search as well as metadata replication functionality. Also, HopsFS implements the same block reporting protocol as HDFS, which is inefficient by design since its overhead grows linearly with the number of blocks stored in the

block storage nodes. HopsFS lacks support for scalable and high available deployments in cloud environments. Moreover, the externalization of the file system metadata enables the development of new applications and protocols to improve user experience and even further improve the file system availability and scalability. In the next chapter, we provide our contributions to tackle the limitations mentioned above.

3
Thesis Contributions

In this chapter, we summarize the contributions of this thesis. We developed consistent distributed metadata in HopsFS by leveraging NewSQL distributed databases, see Section 2.2, ***Paper I*** [24], and ***Paper II*** [23]. Then, we leveraged the extensibility of the metadata storage layer in HopsFS to allow for the development of applications and protocols that benefit from the underlying consistent file system metadata. First, we leveraged the consistent metadata to improve the scalability and availability of HopsFS for on-promise and cloud environments, see Section 3.1, ***Paper III*** [39], ***Paper IV*** [40], and ***Paper V*** [41]. Second, we developed new classes of applications that are tightly coupled with HopsFS, see Section 3.2, ***Paper VI*** [42], and ***Paper VII*** [43].

3.1 Improving HopsFS high availability and scalability

The consistent distributed hierarchical file system metadata offered by HopsFS became a building block for us to improve the performance of HopsFS even further by enabling higher availability and scalability. Firstly, we developed *hbr* a highly scalable block reporting protocol for HopsFS to mitigate the scalability bottlenecks imposed by the existing block reporting protocol. In *hbr*, we extended the file system's metadata by adding more tables for the new protocol while ensuring the consistency and integrity of the file system's metadata by redesigning the HopsFS file system's transactions related to block reporting. Secondly, we developed *HopsFS-CL*, a highly available distribution of HopsFS in the cloud, to enable deployments of HopsFS in the cloud across availability zones. In

HopsFS-CL, we modified the underlying metadata storage layer (NDB) to add awareness for availability zones, also we extended the file system's metadata to be availability zone aware. Thirdly, we developed *HopsFS-S3*, a highly available distribution of HopsFS that is backed by object stores in the cloud, to enable consistent usage of object stores such as Amazon S3 as a backend for storing the file system's blocks. In *HopsFS-S3*, we extended the file system's block storage layer to use different object stores as its backend without compromising the consistency of the file system.

3.1.1 hbr: Scalable block reporting for HopsFS

Decoupling the file system's metadata from the actual file blocks has helped distributed file systems to be more scalable. However, it introduced other problems when it comes to ensuring synchronization between the metadata and the blocks. HDFS and HopsFS use a synchronization protocol called "block reporting" to ensure the consistency between the block storage layer and the metadata layer (storage and serving). However, the block reporting protocol is inefficient by design and limits the cluster's scalability due to its high network and processing overhead. Each block storage server can potentially host millions of replicas. All the servers in the block storage layer send state information for each of the replicas they have to the metadata servers, which in turn cross-checks these replicas with their stored state on the metadata storage layer. The burden of the block reporting protocol is even more apparent with HopsFS since the file system's metadata resides in a distributed database (NDB) rather than in-memory in the case of HDFS, leading to even higher overhead in HopsFS [23].

We developed ***hbr***, a hash-based scalable block reporting protocol that removes the scalability bottlenecks of the existing block reporting protocol by decreasing the network and processing overhead. We introduced the concept of *buckets* as a logical collection of block replicas in the cluster. Then, we formally defined the following three functions as part of the *hbr* protocol:

1. *assignment function:* dynamically maps replicas to a specific bucket. A replica can be only part of one bucket, and all replicas of the same block should be part of the same bucket.

2. *hash function:* generates a fixed size hash that identifies a replica.

3. *hash combiner function:* combines hashes of replicas in the same bucket to generate a unique hash code identifying the whole bucket.

Moreover, we leverage the incremental block reporting protocol to build up the state of buckets independently across the metadata layer and the block storage layer, where each side will keep a hash for each bucket using the three functions that we have introduced. For example, if a replica state has changed on the block storage layer side. The *hbr* protocol first uses the *assignment function* to identify to which bucket this replica belongs then, using the *hash function* it calculates a unique hash identifying the current replica state, then using a *hash combiner function* it calculates the hash for the newly updated bucket. Now that both sides (the metadata layer and the block storage layer) have built their state information in the buckets, the block reporting protocol is as simple as comparing the buckets hash between the metadata layer and the block storage layer. In the case of invalid buckets, the *hbr* fallback to the default block reporting for that specific bucket. The *hbr* protocol functions are overridable as long as they satisfy the properties defined for each function.

In experiments based on real-world workloads, we show that *hbr* scales up to three orders of magnitude compared to the vanilla block reporting while reducing the block report size and latency by up to three orders of magnitude than the vanilla block reporting protocol. This work was awarded the best student paper award at IEEE BigDataCongress, 2019, in Milan. For more details about ***hbr***, refer to ***Paper III*** [39].

3.1.2 HopsFS-CL: Highly Available HopsFS in the cloud

Cloud service providers offer cloud infrastructures built around the concepts of regions and availability zones (AZs). Regions are data center locations around the globe with large geographical distances between regions. A single region consists of one or more availability zones (typically three) that are physically separated data centers with independent power, networking, and cooling. Availability Zones within the same region are connected with low-latency links. Cloud service providers recommend deploying applications across availability zones to ensure high availability during availability zone failures.

We introduce ***HopsFS-CL***, a redesign of HopsFS that is highly available across AZs in the cloud. We redesigned all the HopsFS layers from the

metadata storage layer to the metadata serving layer to the block storage layer to be AZ-aware and favor local AZ communications. The changes we made to the different layers of HopsFS are as follows:

- *The metadata storage layer:* We implemented AZ-awareness by introducing a new configuration parameter to NDB called *'LocationDomainID'* that is used to tag nodes in the cluster with their AZ. Nodes within the same AZ should use the same LocationDomainID. We also added two new features to NDB as table options *'Read Backup'* and *'Fully Replicated'* options to allow read requests to be served from both the primary and the backup replicas. Internally, we changed the NDB commit protocol, see Section 2.2.1, to implement these two features. Then, we changed the node ordering and transaction coordinator selection algorithm to take into account the newly introduced LocationDomainID and the two table features favoring nodes within the same AZ. We changed all the HopsFS-CL tables to be Read Backup enabled to ensure that read requests will be routed to both the primary and backup replicas.
- *The metadata serving layer:* We introduced a new configuration parameter to the metadata servers similar to the LocationDomainID to mark nodes with their corresponding AZ. Metadata servers within the same AZ should have the same LocationDomainID. We also changed the leader election service to include the locationDomainId as part of the metadata server description. Moreover, we changed the metadata server selection policy to favor metadata servers within the same AZ as the client.
- *The block storage layer:* We implemented a new block placement policy to ensure that at least one replica of every block resides in every AZ.

Figure 3.1 shows an example deployment of HopsFS-CL across three AZs. We use the LocationDomainId to identify nodes within the same AZ. For example, N1, N2, M2 are in the same zone, and their locationDomainId will be the same. The metadata replication factor is set to 3, and we place each NDB datanode from the same node group in a different zone. The nodes N1, N5, and N3 form a node group. That is, each node has a full copy of the tables' data assigned to that node group, see Section 2.2.1, similarly, the nodes N2, N4, and N6 form the other node group. Each AZ

has an NDB management node, while one of the nodes is elected as an arbitrator to break the ties in case of split-brain scenarios during network partitions. Each AZ has a group of metadata servers and block storage servers. One of the metadata servers is elected as the leader to perform housekeeping operations. If it fails, a new one will be elected using the leader election service in HopsFS, see Section 2.2.2.

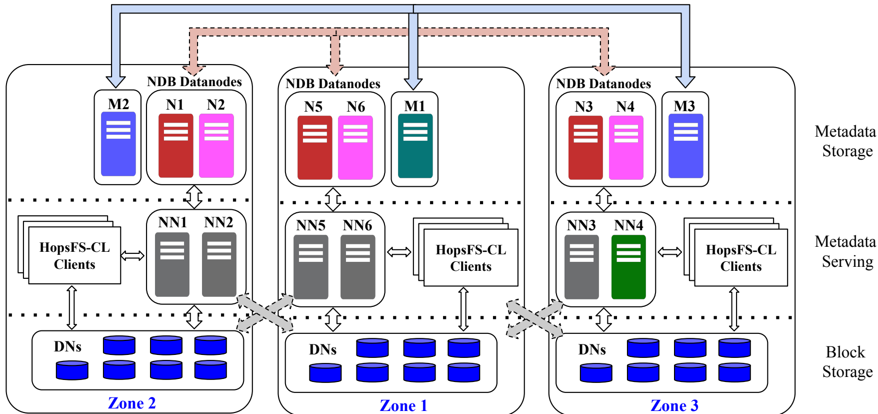


Figure 3.1: A diagram of HopsFS-CL deployed across three AZs. The metadata replication factor is set to 3. The nodes N1, N3, and N5 form a node group while N2, N4, and N6 forms the other node group. M1 is the management node in Zone 1, M2 is the management node in Zone 2, and M3 is the management node in Zone 3. M1 acts as an arbitrator. NN4 is the leader metadata server in the cluster.

In experiments based on a real-world workload from Spotify, ***HopsFS-CL***, deployed in HA mode over 3 AZs, delivers up to 1.66 million ops/s (100 million ops/min), up to 36% higher than vanilla HopsFS, while preserving the same HopsFS semantics. For more details about ***HopsFS-CL***, refer to ***Paper IV*** [40].

3.1.3 HopsFS-S3: Extending Object Stores with POSIX-like semantics

Object stores have become the defacto standard for storage in the cloud due to their low cost, scalability, and high availability. However, they have adopted eventual consistency semantics with no support for critical atomic operations such as directory rename and listing. On the other hand, HopsFS provides strong consistency with a POSIX-like semantic where directory rename and listing are atomic operations.

We developed ***HopsFS-S3*** as a hybrid distributed file system that leverages

the benefits of both HopsFS and object stores. In HopsFS-S3, we redesign the block storage layer to allow object stores to be used as a storage backend for the file system's blocks. We leveraged the heterogeneous storage APIs in HopsFS to implement a new storage type named *Cloud*, allowing for fine-grained control over which subtrees of the namespace to be stored in the cloud. We then added new configurations to HopsFS, allowing users to specify their object store credentials and name (bucket name in case of S3). There are two alternative approaches to redesign HopsFS to work with object stores. The first approach is client-based, where the client has direct access to the object stores skipping the block storage layer. The second approach is block storage proxy, where the block storage servers act as a proxy server to the object store. In this work, we favored the latter approach since it requires minimal changes to the writing and reading protocols in HopsFS, allowing for a pluggable architecture, see Figure 3.2. Our implementation of HopsFS-S3 is pluggable, allowing connecting different types of object stores to HopsFS-S3. Currently, we fully support Amazon S3 and have limited support for the Azure blob store.

We store the files' blocks as objects in the object store in a two-level tree structure. The first level, we call block-containers, is the directories (prefixes in S3). Each directory has a fixed number of objects, the number of objects per block-container is configurable, and the default is set to 500. One of the main problems in S3 is that overwriting an object is eventually consistent meaning, that it can lead to an inconsistent state while reading the object (returning an out-dated version rather than the last written). For that, we investigated the different types of file system operations in HopsFS and identified which of those operations are affected by the eventual consistency semantics of the object store. The file system operations in HopsFS can be categorized into *metadata operations* and *data operations*. The metadata operations change the file system metadata, which is not affected by the object store's eventual consistency semantics. On the other hand, we need to redesign the data operations such that we can ensure strong consistency semantic on top of the object stores. More specifically, for an append operation in HopsFS, first, it checks for the last block of the file if it is complete or not (if the block size is less than the configured block size, 128 MB by default). If the last block is not complete, HopsFS reopens the last block and appends the new data until complete. Otherwise, a new block is created. In such a scenario, inconsistencies may

arise. For that reason, we implemented variable-sized blocks per file. That is, for every new append operation, a new block is created. Another type of operation that needs redesign is the block reporting protocol to ensure a consistent view of the files' blocks between the metadata storage layer in HopsFS-S3 and the object store. We implemented a simple protocol where the leader metadata server periodically creates block reporting tasks for each block-container in the object store (prefix in S3). Then every metadata server will pick up one of the tasks to cross-checks its metadata in both HopsFS-S3 and the object store. We use a timeout mechanism to tackle the eventual consistency in directory listing in the object store APIs.

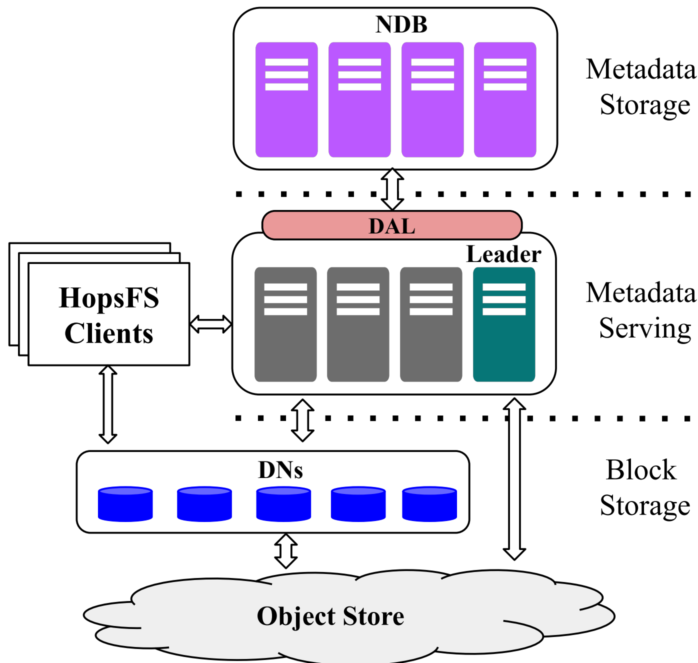


Figure 3.2: A diagram of HopsFS-S3. It consists of three main layers, the metadata storage layer, the metadata serving layer, and the blocks storage layer. The block storage servers act as proxy server to the object stores in the cloud.

In experiments based on a real-world workload, ***HopsFS-S3*** delivers up to 20% higher performance than EMRFS and up to 3.4X the aggregated read throughput than EMRFS. We also demonstrate that metadata operations in HopsFS-S3 are up to two orders magnitude faster than EMRFS. For more details about ***HopsFS-S3***, refer to ***Paper V*** [41].

3.2 New Classes of applications

The consistent distributed hierarchical file system metadata offered by HopsFS enabled the development of new classes of applications that are both extend file system's metadata and have strongly consistent lifecycles for the files/directories they extend. Firstly, we developed *ePipe* a replicated metadata as a service to enable consistent replication of the file system's metadata and its extended metadata, with low replication lag and negligible overhead on the file system. ePipe enables different new use cases for the file system, such as efficient and fast metadata search, notifications services, and extended metadata. In ePipe, we leveraged the event API provided by NDB. We also extended the file system's metadata and transactions to ensure the correctness and consistency of the file system changelog. Secondly, we developed *Hopsworks*, a multi-tenant and collaborative big data platform, to improve the user experience with big data platforms. In Hopsworks, we developed a completely new application that leverages the file system's metadata's extensibility while ensuring integrity and correctness.

3.2.1 ePipe: Polyglot Persistence for HopsFS metadata

The metadata storage layer (NDB) of HopsFS offers a SQL API that allows administrators to issue queries on the file system metadata without affecting the metadata serving layer. However, NDB is an OLTP (Online transaction processing) database that does not efficiently support complex queries and aggregations. Conventional wisdom says that there is no single database that can efficiently process all query patterns on metadata [44]. Therefore, we designed ePipe, a replicated-metadata as a service that replicates the file system's metadata from NDB to different data stores with low replication lag and negligible overhead on the metadata storage and serving layers. Moreover, ePipe provides notification services similar to *inotify* in conventional file systems.

The main challenge with ePipe is to ensure the consistency of the replicated metadata to external databases. Therefore, we formally defined a system model for HopsFS metadata operations and a consistency requirement that ePipe has to maintain while replicating to external databases. We ensure that all operations happening on a single file/directory will always be replicated in the same order as seen by HopsFS.

We extended HopsFS to log file system operations into a special logging

table (*frol*) within the same file system transaction. Then, we leveraged the NDB Event API, see Section 2.2.1, to stream the changelog from the logging table to ePipe. We implemented an efficient re-ordering algorithm to ensure the ordering of the file system operations with respect to our consistency requirement. ePipe offers a pluggable architecture allowing it to connect to different types of downstream data stores such as Elasticsearch, as well as applications that can receive notifications about changes happening on the file system. Moreover, we presented different use cases of ePipe, such as enabling fast metadata search using Elasticsearch, attaching extended metadata to files and directories, and synchronizing metadata changes between different data stores.

In experiments based on a real-world workload, we show that ePipe has a negligible overhead on HopsFS, and delivers up to 56X higher throughput than existing solutions such as HDFS Inotify, see Section 2.1, with constant low replication lag. For more details about ***ePipe***, refer to ***Paper VI*** [42].

3.2.2 Hopsworks: Improving user experience and data management

Data management was not a consideration in the original design of Apache Hadoop. Moreover, over the past years, many new systems have been added to the Hadoop ecosystem, such as Hive, HBase, and Impala. All of these systems have their metadata system introducing metadata silos between these systems. That motivated the development of new centralized systems such as Apache Ranger and Apache Sentry to control the shared metadata (permissions and polices) between different services in the Hadoop ecosystem, see Section 2.1.3. However, to keep all systems synchronized, they typically use polling techniques, potentially introducing metadata inconsistencies giving access to unauthorized users, or revoking access from authorized users.

We introduce Hopsworks, a secure, scalable, multi-tenant, and collaborative big data platform built on top of HopsFS. Using a distributed database to store the file system's metadata allows for the development of centralized metadata that can be shared and extended by different systems without sacrificing the metadata integrity. Hopsworks leverages the extensibility of HopsFS to introduce three new data management concepts *Projects*, *Datasets*, *Users* to implement a project-based multi-tenant security model. Projects are separate entities that contain datasets, users, and other Hopsworks enabled services, see Figure 3.3. Internally, projects

and datasets map to subtrees in the HopsFS's namespace. Hopsworks enables secure sharing of datasets between projects and supports dynamic role-based access control to sensitive data on a shared cluster. For every project a user is a member of, they have a role. Any application running in Hopsworks only assumes a single role, so users can be prevented from accessing data outside a given project based on their role in that project. Hopsworks provides a user-interface where users can authenticate and manage their projects and datasets. Users can then run jobs using different data-parallel processing frameworks such as Spark and Flink, use streaming services such as Kafka, use interactive notebook frameworks such as Jupyter. For more details about ***Hopsworks***, refer to ***Paper VII*** [43].

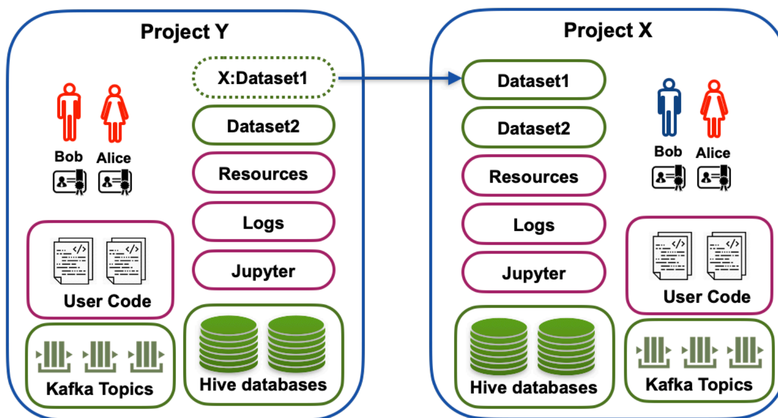


Figure 3.3: A logical diagram of a Hopsworks cluster with two projects (Project X and Project Y), and two users (Alica and Bob). Each project has resources such as datasets, users, kafka topics, and hive databases. Project Y has two data owners (Bob and Alice) and has a shared dataset with Project X (X:Dataset1). Project X has one data owner (Alice) and a data scientist (Bob).

4 Conclusions and Future Work

The consistent distributed hierarchical file system metadata proved to be an essential asset that enables the development of new applications and protocols tightly coupled with the file system. Our work in this thesis focused on two aspects of the file system: (1) improving scalability and availability for on-premise and cloud environments, (2) developing new applications that extend the file system metadata, and provide consistency guarantee with respect to files and/or directories.

Firstly, we investigated the different issues with HopsFS that limits its scalability and availability, and we identified three different issues. First, the block reporting protocol is limiting the scalability of HopsFS due to its high network and processing overhead imposed on the metadata storage and metadata serving layers. Second, HopsFS was not developed with availability zone awareness in mind, which prevented highly available deployments in the cloud. Third, the block storage layer of HopsFS uses only local disk volumes in the block storage servers to store the file system's blocks, which prevented highly available deployments in the cloud. To overcome the aforementioned issues, we developed *hbr*, *HopsFS-CL*, and *HopsFS-S3*. hbr is a new scalable block reporting protocol for HopsFS that improves the network and processing overhead by up to three orders of magnitude than the vanilla block reporting. hbr enables HopsFS clusters to scale to 10s of thousands of block storage nodes. HopsFS-CL is an extension of HopsFS that is highly available across availability zones. HopsFS-CL deployed across three availability zones provides similar performance to HopsFS deployed in one availability zone.

Furthermore, it improves the throughput over the vanilla HopsFS across three availability zones by up 36%. HopsFS-S3 is an extension of HopsFS that allows plugging object stores such as Amazon S3 as a backend for the block storage servers. HopsFS-S3 improves the availability of HopsFS in the cloud by using object stores instead of local volume disks and delivers up to 20% better performance than the Amazon Elastic MapReduce File system (EMRFS).

Secondly, we investigated the different applications that are needed to improve the usability of the file system. We identified the need for a consistently replicated metadata-as-service and multi-tenant big data platform that builds upon HopsFS to improve the user experience. First, we developed *ePipe* a replicated metadata-as-service that consistently replicates the file system's metadata with negligible overhead and low replication lag. ePipe enables different use cases for the file system's metadata, such as fast metadata search and notification services. Second, we developed *Hopsworks*, a multi-tenant and collaborative big data platform that builds upon HopsFS. Hopsworks provides a project-based, multi-tenant security model where a project is a sandbox that holds a collection of datasets, Kafka topics, users, and jobs (data-parallel processing frameworks such as Spark), interactive notebooks.

In the future, we are planning to take the HopsFS high availability one step further by supporting deployments across regions besides availability zones. First, we plan to investigate different deployment modes, such as active-active and active-standby modes. The active-active mode is more challenging to implement and maintain file system consistency across regions while providing acceptable performance (lower latency). It requires implementing a coordination service in HopsFS to coordinate between different writes happening in different regions. The active-standby mode is more viable to develop and maintain. For that, we are looking at leveraging ePipe and NDB Events API to replicate the file system metadata with fine-grained control across regions. We are also currently investigating alternative designs for HopsFS-S3 to provide dual access to HopsFS and S3. Instead of storing the files' blocks in the object store, we will store the whole files to maintain the same namespace structure allowing users to view the same file system namespace from HopsFS and the object store.

Moreover, in Hopsworks, we are currently leveraging the HopsFS's ACLs (Active Control Lists) to provide more fine-grained access control on

Datasets. We are also currently investigating adding support for provenance and governance APIs by leveraging the metadata's extensibility in HopsFS and the CDC APIs of ePipe. These APIs allow users to track the application's ownership and the associated usage graph of all of its artifacts (files), easing debugging and allowing reproducibility of applications running on top of Hopsworks. Another sought-after feature is fine-grained snapshotting and versioning of *Datasets*. We plan to investigate efficient techniques to support this feature on top of HopsFS by leveraging the metadata's extensibility. We are also exploring the possibility of using the metadata storage layer to store the application state of Apache Flink instead of local RocksDB instances, allowing for efficient and fast reconfiguration and recovery.

Bibliography

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pp. 1–10, Ieee, 2010.

[2] "Delta lake: Reliable data lakes at scale." https://delta.io/. [Online; accessed 12-Sep-2019].

[3] "Apache iceberg: open table format for huge analytic datasets." https://iceberg.incubator.apache.org/. [Online; accessed 12-Sep-2019].

[4] "Apache hudi: Upserts and incremental processing on big data." http://hudi.apache.org/. [Online; accessed 12-Sep-2019].

[5] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1235–1246, ACM, 2014.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.

[7] "High availability framework for hdfs nn." https://issues.apache.org/jira/browse/HDFS-1623. [Online; accessed 14-01-2020].

[8] "Support more than 2 namenodes." https://jira.apache.org/jira/browse/HDFS-6440. [Online; accessed 14-01-2020].

[9] "Java Heap Size." http://www.azulsystems.com/technology/java-heap-size. [Online; accessed 3-September-2015].

[10] A. Kagawa, "Hadoop Summit 2014 Amsterdam. Hadoop Operations Powered By Hadoop." https://www.youtube.com/watch?v=XZWwwc-qeJo. [Online; accessed 30-Aug-2015].

[11] "Hadoop JIRA: Add thread which detects JVM pauses.." https://issues.apache.org/jira/browse/HADOOP-9618. [Online; accessed 1-January-2016].

[12] "Hdfs inotify." https://issues.apache.org/jira/browse/HDFS-6634. [Online; accessed 15-Aug-2017].

[13] "Support for large-scale multi-tenant inotify service." https://issues.apache.org/jira/browse/HDFS-8940. [Online; accessed 15-Aug-2017].

[14] "Trumpet." http://verisign.github.io/trumpet/. [Online; accessed 20-April-2018].

[15] "Apache HBase." http://hbase.apache.org/. [Online; accessed 21-February-2017].

[16] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, pp. 1–7, 2011.

[17] "Apache ranger." https://ranger.apache.org/. [Online; accessed 14-01-2020].

[18] "Apache sentry." https://sentry.apache.org/. [Online; accessed 14-01-2020].

[19] "Hadoop-aws module: Integration with amazon web services." https://hadoop.apache.org/docs/current/hadoop-aws/tools/hadoop-aws/index.html. [Online; accessed 12-Mar-2020].

[20] "S3guard: Consistency and metadata caching for s3a." https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/s3guard.html. [Online; accessed 5-Jan-2019].

[21] “Amazon dynamodb.” https://aws.amazon.com/dynamodb. [Online; accessed 12-Mar-2020].

[22] “Committing work to s3 with the âĂIJs3a committersâĂİ.” https://hadoop.apache.org/docs/current/hadoop-aws/tools/hadoop-aws/committers.html. [Online; accessed 12-Mar-2020].

[23] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, “HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases,” in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, (Santa Clara, CA), pp. 89–104, USENIX Association, 2017.

[24] M. Ismail, S. Niazi, M. Ronström, S. Haridi, and J. Dowling, “Scaling hdfs to more than 1 million operations per second with hopsfs,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 683–688, May 2017.

[25] M. Stonebraker, “Newsql: An alternative to nosql and old sql for new oltp apps,” *Communications of the ACM. Retrieved*, pp. 07–06, 2012.

[26] M. Ronström, *MySQL Cluster 7.5 Inside and Out*. Books on Demand, 2018.

[27] “MySQL Cluster CGE.” http://www.mysql.com/products/cluster/. [Online; accessed 30-June-2015].

[28] “MemSQL, The World’s Fastest Database, In Memory Database, Column Store Database.” http://www.memsql.com/. [Online; accessed 30-June-2015].

[29] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, “Sap hana database: data management for modern business applications,” *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[30] M. Stonebraker and A. Weisberg, “The voltdb main memory dbms.,” *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[31] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ansi sql isolation levels,” *SIGMOD Rec.*, vol. 24, pp. 1–10, May 1995.

[32] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[33] J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*, (London, UK, UK), pp. 393–481, Springer-Verlag, 1978.

[34] A. Davies and H. Fisk, *MySQL Clustering*. MySQL Press, 2006.

[35] H. Garcia-Molina, C. A. Polyzois, and R. B. Hagmann, "Two epoch algorithms for disaster recovery," in *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, (San Francisco, CA, USA), pp. 222–230, Morgan Kaufmann Publishers Inc., 1990.

[36] J. Gray, R. Lorie, G. Putzolu, and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database," in *IFIP Working Conference on Modelling in Data Base Management Systems*, pp. 365–394, IFIP, 1976.

[37] G. B. Salman Niazi, Mahmoud Ismail and J. Dowling, "Leader Election using NewSQL Systems," in *Proc. of DAIS 2015*, pp. 158 –172, Springer, 2015.

[38] S. Niazi, M. Ronström, S. Haridi, and J. Dowling, "Size matters: Improving the performance of small files in hadoop," in *Proceedings of the 19th International Middleware Conference*, pp. 26–39, ACM, 2018.

[39] M. Ismail, A. Bonds, S. Niazi, S. Haridi, and J. Dowling, "Scalable block reporting for hopsfs," in *2019 IEEE International Congress on Big Data (BigDataCongress)*, pp. 157–164, IEEE, 2019.

[40] M. Ismail, S. Niazi, M. Ronström, S. Haridi, and J. Dowling, "Distributed hierarchical file systems strike back in the cloud," in *Distributed Computing Systems (ICDCS), 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2020.

[41] M. Ismail, S. Niazi, G. Berthou, M. Ronström, S. Haridi, and J. Dowling, "Hopsfs-s3: Extending object stores with posix-like semantics and more," in *Proceedings of the 21th International Middleware Conference Industrial Track*, Middleware '20, (Delft, The Netherlands), Association for Computing Machinery, 2020.

[42] M. Ismail, M. Ronstrom, S. Haridi, and J. Dowling, "epipe: Near real-time polyglot persistence of hopsfs metadata," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 92–101, 2019.

[43] M. Ismail, E. Gebremeskel, T. Kakantousis, G. Berthou, and J. Dowling, "Hopsworks: Improving user experience and development on hadoop with scalable, strongly consistent metadata," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pp. 2525–2528, IEEE, 2017.

[44] M. Stonebraker and U. Cetintemel, ""one size fits all": an idea whose time has come and gone," in *21st International Conference on Data Engineering (ICDE'05)*, pp. 2–11, IEEE, 2005.

Part II

Included Papers