Scaling Distributed Hierarchical File Systems Using NewSQL Databases

SALMAN NIAZI

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2018

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

TRITA-EECS-AVL-2018:79
ISBN 978-91-7729-987-5

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i informations och kommunikationsteknik fredagen den 7 december 2018 klockan 9.00 i Sal B i Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

For many years, researchers have investigated the use of database technology to manage file system metadata, with the goal of providing extensible typed metadata and support for fast, rich metadata search. However, earlier attempts failed mainly due to the reduced performance introduced by adding database operations to the file system's critical path. Recent improvements in the performance of distributed in-memory online transaction processing databases (NewSQL databases) led us to re-investigate the possibility of using a database to manage file system metadata, but this time for a distributed, hierarchical file system, the Hadoop Distributed File System (HDFS). The single-host metadata service of HDFS is a well-known bottleneck for both the size of the HDFS clusters and their throughput.

In this thesis, we detail the algorithms, techniques, and optimizations used to develop HopsFS, an open-source, next-generation distribution of the HDFS that replaces the main scalability bottleneck in HDFS, single node in-memory metadata service, with a no-shared state distributed system built on a NewSQL database. In particular, we discuss how we exploit recent high-performance features from NewSQL databases, such as application-defined partitioning, partition pruned index scans, and distribution aware transactions, as well as more traditional techniques such as batching and write-ahead caches, to enable a revolution in distributed hierarchical file system performance.

HDFS' design is optimized for the storage of large files, that is, files ranging from megabytes to terabytes in size. However, in many production deployments of the HDFS, it has been observed that almost 20% of the files in the system are less than 4 KB in size and as much as 42% of all the file system operations are performed on files less than 16 KB in size. HopsFS introduces a tiered storage solution to store files of different sizes more efficiently. The tiers range from the highest tier where an in-memory NewSQL database stores very small files (<1 KB), to the next tier where small files (<64 KB) are stored in solid-state-drives (SSDs), also using a NewSQL database, to the largest tier, the existing Hadoop block storage layer for very large files. Our approach is based on extending HopsFS with an inode stuffing technique, where we embed the contents of small files with the metadata and use database transactions and database replication guarantees to ensure the availability, integrity, and consistency of the small files. HopsFS enables significantly larger cluster sizes, more than an order of magnitude higher throughput, and significantly lower client latencies for large clusters.

Lastly, coordination is an integral part of the distributed file system operations protocols. We present a novel leader election protocol for partially synchronous systems that uses NewSQL databases as shared memory. Our work enables HopsFS, that uses a NewSQL database to save the operational overhead of managing an additional third-party service for leader election and deliver performance comparable to a leader election implementation using a state-of-the-art distributed coordination service, ZooKeeper.

Sammanfattning

I många år har forskare undersökt användningen av databasteknik för att hantera metadata i filsystem, med målet att tillhandahålla förlängbar metadata med stöd för snabb och uttrycksfull metadatasökning. Tidigare försök misslyckades dock huvudsakligen till följd av den reducerade prestanda som infördes genom att lägga till databasoperationer på filsystemets kritiska väg. De senaste prestandaförbättringarna för OLTP databaser som lagras i minnet (NewSQL databaser) ledde oss till att undersöka möjligheten att använda en databas för att hantera filsystemmetadata, men den här gången för ett distribuerat hierarkiskt filsystem, Hadoop Distributed Filesystem (HDFS). Metadata i HDFS lagras på en maskin, vilket är en känd flaskhals för såväl storlek som prestandan för HDFS kluster.

I denna avhandling redogör vi för de algoritmer, tekniker och optimeringar som används för att utveckla HopsFS, en med öppen källkod, nästa generationens distribution av HDFS som ersätter lagringen av metadata i HDFS, där den lagras enbart i minnet på en nod, med ett distribuerat system med delat tillstånd byggt på en NewSQL databas. I synnerhet diskuteras hur vi utnyttjar nyligen framtagen högpresterande funktionalitet från NewSQL-databaser, exempelvis applikationsdefinierad partitionering, partitionsskuren indexskanning och distributionsmedvetna transaktioner, samt mer traditionella tekniker som batching och skrivcache, som banar väg för en revolution inom prestanda för distribuerade filsystem.

HDFS design är optimerad för lagring av stora filer, det vill säga filer som sträcker sig från megabyte till terabyte i storlek. Men i många installationer i produktionsystem har det observerats att nästan 20 procent av filerna i systemet är mindre än 4 KB i storlek och så mycket som 42 procent av alla filsystemsoperationer utförs på filer mindre än 16 KB i storlek. HopsFS introducerar en nivåbaserad uppdelning av olika filstorlekar för mer effektiv lagring . Nivåerna varierar från högsta nivån där en NewSQL-databas lagrar i minnet mycket små filer (<1 KB), till nästa nivå där små filer (<64 KB) lagras i SSD-enheter (Solid State Drives) en NewSQL-databas, till den största delen, det befintliga Hadoop-blocklagringsskiktet för mycket stora filer. Vårt tillvägagångssätt bygger på att utöka HopsFS med en utfyllningsteknik för filer, där vi lägger in innehållet i små filer tillsammans med metadata och använder databasstransaktioner och databasreplikation för att garantera de små filernas tillgänglighet, integritet och konsistens säkerställs. HopsFS möjliggör signifikant större klusterstorlekar, mer än en storleksordning högre transaktionsgenomströmning, och signifikant lägre latens för klienter till stora kluster.

Slutligen är koordinering en central del av protokollet för distribuerade filsystemsoperationer. Vi presenterar ett nytt ledarval protokoll för delvis synkrona system som använder NewSQL databaser som delat minne. Vårt arbete möjliggör HopsFS, som använder en NewSQL-databas för att spara in på de operativa kostnader det skulle medföra att hantera en ytterligare tredjepartstjänst för ledarval. Protokollets prestanda kan jämföras med en ledarval implementation I ZooKeeper, som är en modern distribuerad koordinationsservice.

To Baba, Amma, and Hafsa.

Acknowledgements

I'm deeply indebted to my Ph.D. advisor Jim Dowling for giving me the opportunity to work under his supervision. Without his guidance and support, this thesis would not have been possible. I would also like to extend my deepest gratitude to my second advisor Seif Haridi for his constructive discussion and feedback on this thesis and the included research papers.

I also wish to thank my friend and colleague Mahmoud Ismail for his support and contributions in the development of the file system. Developing the first version of HopsFS was a painstaking job which took more than three years of continuous development. Without Mahmoud's help, HopsFS system would have taken significantly more time to develop. I also want to thank all my colleagues and friends at Logical Clocks AB, KTH, and RISE SICS for their support during the Ph.D. studies.

Finally, I would like to thank my family and my better half, Hafsa. Their love and support made it all possible. Thank you for always being there.

Included Papers

Paper *I*

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

Paper *II*

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

Paper *III*

Size Matters: Improving Small Files' Performance in Hadoop.
Salman Niazi, Seif Haridi, Jim Dowling, Mikael Ronström. *In ACM/IFIP/USENIX 19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.*

Paper *IV*

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.*

Software

The research conducted within the scope of this dissertation has resulted in three main open-source software contributions: Hops Hadoop Distributed File System, Hammer distributed file system benchmark tool and a leader election library that uses NewSQL databases as shared memory.

Hops Hadoop Distributed File System

Hops Hadoop Distributed File System (HopsFS) is a next-generation high-performance distribution of the Hadoop Distributed File System (HDFS) that replaces HDFS single node in-memory metadata service with a distributed metadata service built on a MySQL's Network Database (NDB) cluster. HopsFS provides a high-performance unified file system namespace that outperforms HDFS by an order of magnitude in terms of the size of the supported namespace and also the throughput of the file system operations. HopsFS can handle both large and small files efficiently, and it has significantly lower operational latencies for the file system operations.

Source Code: `https://www.github.com/hopshadoop/hops`

Hammer File System Benchmark

We have developed a benchmark tool for testing the throughput of the metadata operations of any Hadoop Compatible File System. The benchmark utility is a distributed application that can generate a large number of metadata operations using tens of thousands of clients spread across multiple machines. It can be used to test the throughput of individual file system operations or generate file system operations based on industrial workload traces. The benchmark utility supports testing HopsFS, HDFS, MapR, and CephFS.

Source Code: https://www.github.com/smkniazi/hammer-bench

Leader Election Library Using NewSQL Databases

We have developed a leader election library that uses NewSQL databases as shared memory. This library enables distributed systems that already use NewSQL databases to save the operational overhead of managing an additional third-party coordination service for leader election. Currently, the library only supports NDB cluster NewSQL database. It is possible to add support for additional databases by implementing the database abstraction layer of the leader election library. The library can support any NewSQL database that supports row-level locking and two-phase transactions to access and update the data.

Source Code: https://github.com/hopshadoop/hops/tree/master/hops-leader-election

Awards

HopsFS also won the 10th IEEE International Scalable Computing Challenge (SCALE 2017) sponsored by the IEEE Computer Society Technical Committee on Scalable Computing (TCSC). The contest focuses on end-to-end problem solving using concepts, technologies, and architectures (including Clusters, Grids, and Clouds) that facilitate scaling. Participants in the challenge identify significant current real-world problems where scalable computing techniques can be effectively used. The Participants design, implement, evaluate, and feature live demonstrate their solutions.

Contents

Part I

Thesis Overview

1

Introduction

'Hierarchical File Systems are Dead.' [Not!]

— Margo Seltzer, Nicholas Murphy

The Unix operating system has popularized the hierarchical namespace. In POSIX compliant file systems the files are organized in top-down hierarchical structure, where all the non-terminal nodes in the hierarchy are directories, and all the terminal nodes (leaves) can be files or directories [1]. To this day, even after five decades, it remains a popular mechanism for organizing and accessing user data. Due to the simplicity and ubiquity of the single server hierarchical file systems many distributed file systems have also adopted the hierarchical namespace abstraction [2–16]. In POSIX, all file system operations, such as *create, open, close, rm, mv, stat, etc.*, are atomic file system operations, that is, if multiple threads call one of these functions, then the threads would see either all or none of the specified effects of the file system operations. Before the file system operation is performed, the file paths are resolved, and the permissions are checked. Consider concurrent file write operations on two independent files */home/user1/file1* and */home/user2/file1*. Writing to these two files would require resolving the file paths from the *root (/)* directory all the way down to the files and acquiring locks on the files and their parent directories to perform the operations atomically. The hierarchical namespace has obvious natural hotspots, that is, the directories at the top

of the namespace hierarchy are accessed very frequently. Hierarchical namespaces are a mismatch for distributed file systems that can store significantly more data and perform vastly more file system operations per second than single server file systems [17]. The hotspots not only complicate the design of the metadata service but also limit the scalability of the distributed file systems.

Typically, distributed file systems separate metadata from data management services to provide a clear separation of concerns, enabling the two different services to be independently managed and scaled. Usually, the file system metadata is stored on a single metadata server, such as in HDFS [2], GFS [18], and QFS [3] that provides high availability of the metadata service using the active-standby failover model [19]. The files' blocks are stored on block storage servers which ensure high availability of the data blocks using techniques like forward error correction codes (such as, erasure codes) and data redundancy. Such an architecture has two main problems. First, the scalability of the file system is limited by the scalability of the single-host metadata service. Second, this architecture is only suitable for data parallel applications which use a large number of file system clients to read the data in parallel after obtaining the files' metadata from the metadata service layer. This design only works if the user data is stored in large files which would reduce the amount of metadata stored on the metadata servers and the cost of file system metadata operations would be amortized over the relatively long periods of time spent in reading/writing these large files. File systems such as NFS [10], AFS [11], MapR [12], Locus [13], Coda [14], Sprite [15] and XtreemFS [16] statically partition the namespace into sub-trees across multiple metadata servers (shards) to scale-out the metadata service layer. However, the metadata partition boundaries (shards) are visible to the file system users due to the lack of support for the atomic *link* and *rename* operations that cross the metadata partition boundaries.

Recent studies about distributed file systems' workloads in production deployments have shown that significant portions of the files in the production distributed file systems are very small in size [20]. For example, in the case of Spotify and Yahoo!, almost 20% of the files stored in their HDFS clusters are less than 4 KB in size. At Spotify, almost 42% of all the file system operations are performed on files less than 16 KB in size. The presence of a large number of small files puts pressure on distributed file systems with single-host metadata servers resulting in poor performance

of the file systems. See Chapter 9 for more details related to the small files' problem in Hadoop. Many applications, such as streaming and real-time analytics, not only generate a large number of file system metadata operations but also require low latency highly parallel file system metadata operations. Moreover, existing distributed file systems do not provide mechanisms to enable online ad-hoc analytics on the file system metadata. For example, finding all files belonging to a user that were created during a given time range, or finding the largest files/directories in the file system. Administrators often resort to writing custom tools to analyze the distributed file system metadata.

The file system metadata consists of highly structured information such as file/directory names, size, user permissions, data block locations, and lease information. In distributed file systems, the size of the metadata is very small compared to the amount of data stored. Databases are the de facto standard for storing large amounts of structured data and provide ACID transactions to access and update the stored data in an atomic manner. The conventional wisdom is that databases are ideal for storing hierarchical file system metadata, as the metadata not only easily fits a database, but also the ACID transactions provided by the database would simplify the design and implementation of the file system metadata service. Also, the databases provide a mechanism to perform ad hoc analytics using SQL which could be used to analyze the file system metadata.

For many years, researchers have investigated this idea of using database technology to manage file system metadata, with the goal of providing high-performance metadata service [21–25]. However, earlier attempts failed mainly due to the reduced performance introduced by adding database operations to the file system's critical path. To improve the path lookup performance, existing solutions that use databases to manage file system metadata store the file system metadata in denormalized form, that is, each inode stores complete pathname of the files and directories. This not only consume a significant amount of precious database storage space but also, a simple directory rename operation would require updating the file paths for all the children of the directory subtree – potentially a very expensive operation.

However, recent improvements in the performance of distributed in-memory online transaction processing databases (NewSQL databases)

has led us to re-investigate the possibility of using a database to manage file system metadata. In this thesis, we base our investigation on the popular Hadoop distributed file system (HDFS). The single-host metadata service of HDFS is a well-known bottleneck for both the size of HDFS clusters and their throughput. Migrating the HDFS metadata to a NewSQL database offered the possibility of a more scalable and simpler metadata service. We store the file system metadata in normalized form, that is, instead of storing complete file paths with each inode, we store individual file path components in the database, enabling in-place *mv*, *chown*, and *chmod* operations.

1.1 Thesis Statement

> Modern NewSQL database systems can be used to store fully normalized metadata for distributed hierarchical file systems and provide high throughput and low operational latencies for the file system operations.

1.2 Hops Hadoop Distributed File System

In this thesis we show how to build a high-performance distributed file system, using a NewSQL database, which leverages both classical database techniques such as extensive use of *primary key* operations, *batching* operations and *write-ahead* caches within transactions, as well as modern data distribution aware techniques commonly found in NewSQL databases. These distribution-aware NewSQL techniques include *application-defined partitioning* (we partition the namespace such that the metadata for all immediate descendants of a directory (child files/directories) reside on the same database partition for efficient directory listing), and *distribution aware transactions* (we start a transaction on the database partition that stores all/most of the metadata required for the file system operation), and *partition pruned index scans* (scan operations are localized to a single database partition [26]). We show how we improve the latency and throughput of the small files by collocating file system metadata and data blocks for small files. Lastly, we show how to build a coordination service using NewSQL system that simplifies the administration of the file system by reducing its reliance on third-party coordination services, such as Apache ZooKeeper [27].

We present Hops Hadoop Distributed File System (HopsFS) a new production-ready file system which is a drop-in replacement for the current state-of-the-art industry-standard the Hadoop Distributed File System. HopsFS stores the file system metadata in a NewSQL relational database, the MySQL Cluster's Network Database (NDB) storage engine in our case. In HopsFS, the metadata service layer consists of multiple stateless metadata servers that convert the file system operations into database transactions. The database ensures that the conflicting file system operations are isolated and replicates the metadata for high availability. HopsFS provides a high-performance unified file system namespace that outperforms HDFS by an order of magnitude both in terms of the size of the supported namespace and also the throughput of the file system operations. HopsFS can handle both large and small files efficiently, and it has significantly lower latencies for the file system operations.

1.3 List of Publications

The research conducted within the scope of this dissertation resulted in four research papers listed below in chronological order

- 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.*

  Author's Contributions: The thesis author is the main contributor in designing, implementing and testing the proposed leader election service. The thesis author did the main bulk of the work including writing the paper.

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

  Author's Contributions: The thesis author is one of the main contributors to HopsFS. The thesis author did the main bulk of the work

that includes, implementing HopsFS, performance optimization, benchmarking, evaluation and writing of the paper.

- Scaling HDFS to more than 1 million operations per second with HopsFS. Mahmoud Ismail, ***Salman Niazi***, Seif Haridi, Jim Dowling, Mikael Ronström. *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 played a major role in the discussions and implementations of file system operations using different types of database operations. The thesis author was responsible for an optimal configuration of the database, operating system and the network needed for high-performance of HopsFS and performed the experiments. This paper was also nominated for 10th IEEE International Scalable Computing Challenge (SCALE 2017). The thesis author also presented HopsFS in SCALE challenge and demonstrated the scalability of the file system.

- Size Matters: Improving Small Files' Performance in Hadoop. ***Salman Niazi***, Seif Haridi, Jim Dowling, Mikael Ronström. *In ACM/IFIP/USENIX 19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.*

  Author's Contributions: The thesis author is the main contributor in designing, implementing and testing the proposed solution. The thesis author also analyzed the production clusters of Hadoop at Spotify and Logical Clocks to identify the prevalence of small files, and the frequency of file system operations performed on the small files. The thesis author did the main bulk of the work including writing the paper.

1.4 Thesis Outline

This thesis consists of two parts. In the first part, Chapter 2 gives an overview of the HopsFS architecture. Chapter 3 describes in detail the system architecture of the NDB NewSQL storage engine and describes the different types of supported NewSQL database operations. Chapter 4 briefly describes the contributions of this thesis followed by an introduction of the prior research in the field of distributed file systems. Chapter 6 concludes the first part of the thesis, and it also discusses possi-

ble venues for the future research. The second part of the thesis consists of four chapters that discuss the thesis contributions in much detail.

2

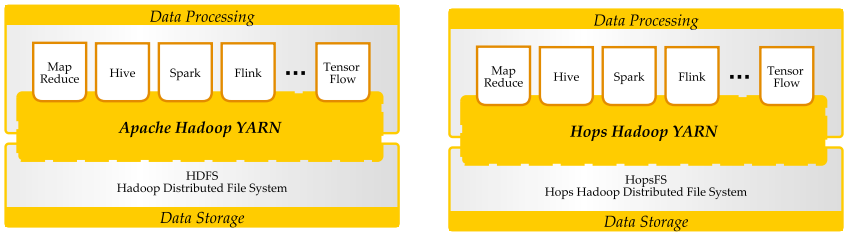
HopsFS Overview

The world's most valuable resource is no longer oil, but data.

— The Economist, 6th May 2017

This is an age of *big data*. Internet companies are storing and processing large volumes of user data in distributed file systems to understand their customers better and provide value-added services. Apache Hadoop is currently the open-source state-of-the-art industry-standard software that provides a highly-scalable, fault tolerant and extremely flexible set of frameworks to store and process large amounts of data using commodity hardware [28]. At the heart of the Hadoop is the Hadoop Distributed File System that is designed to store large volumes of data and provide streaming access to the stored data, see Figure 2.1a. The data stored in HDFS is accessed and processed by data processing frameworks such as *Pig*, *Hive*, *HBase*, *MapReduce*, *Tez*, *Spark*, and *Giraph* [29–35]. These applications are managed by Apache YARN [36], which is a resource negotiator and application scheduler for the Apache Hadoop clusters. Together, it is said that HDFS and YARN comprise the operating system for many modern data centers [37].

Hadoop Open Platform-as-a-Service (Hops) is a new open-source distribution of Apache Hadoop that is based on a next-generation, scale-out



a). Apache Hadoop b). Hops Hadoop

Figure 2.1: Software stacks of Apache Hadoop and Hops Hadoop. Hops Hadoop replaces the HDFS and YARN with the HopsFS and Hops YARN. At the data processing layer there is no significant difference with Apache Hadoop and Hops Hadoop, that is, both Hadoop distributions support the same data processing frameworks.

distributed architecture for HDFS and YARN metadata. Hops Hadoop is designed to make Hadoop more scalable and easier to use for the users. Hops Hadoop has been running in production in Sweden with over 500 users. Hops Hadoop uses HopsFS for storage which is more scalable and high-performant than the Apache Hadoop's HDFS. In this thesis, we will focus on system development and research contributions for HopsFS. At the data processing layer there is no significant difference with Apache Hadoop and Hops Hadoop, that is, both Hadoop distributions share the same data processing frameworks.

2.1 Apache Hadoop Distributed File System

Apache HDFS has five types of nodes: namenodes, datanodes, clients, journal nodes, and ZooKeeper nodes that are needed to provide a highly available file system, see Figure 2.2.

2.1.1 HDFS Namenodes

HDFS decouples file system metadata from blocks that store the file data. The data blocks for files of all sizes are stored and replicated on a large number of independent nodes, called the datanodes, while the file system metadata is stored in-memory on a special node called the Active Namenode (ANN). HDFS supports multiple standby namenodes (SbNNs), and it uses the *active-standby* model for high availability [38]. A standby namenode stores a replica of the file system metadata and takes over when the active namenode fails. The active namenode is responsible for the entire file system. It manages the file system metadata, performs

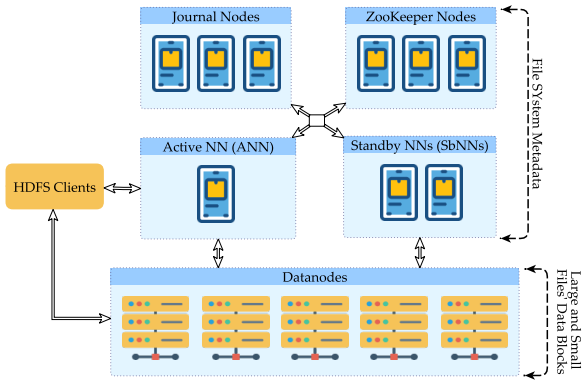


Figure 2.2: HDFS system architecture. In HDFS, a single namenode manages the metadata for the entire namespace. For high availability, the namenode logs all the changes in the metadata on a quorum of journal nodes using quorum-based replication. The log is subsequently replicated asynchronously to the standby namenodes. ZooKeeper is used to detect failed namenodes and to reliably failover from active to standby namenodes. In HDFS, the clients communicate with the active namenode and the datanodes in order to read and write files.

all the file system metadata operations, and ensures that the file system is in a healthy state. The namenode is implemented in Java, and it keeps the metadata in-memory on the heap of a single Java Virtual Machine (JVM) process for high performance. Storing the metadata in-memory is necessary because random-access to metadata on either a magnetic disk or a solid-state-disk is an order of magnitude slower than reading from main memory. The namenode uses custom map-like in-memory data structures to minimize the memory footprint. However, the maximum amount of metadata that can be handled by the namenode is bounded by the amount of memory that can be efficiently managed by the JVM garbage collector (a few hundred GBs, at most [2, 39]). At the time of writing the Spotify HDFS cluster, consisting of ≈2000 data nodes, stores 73 PBs of data in ≈0.5 billion data blocks. Further scalability of the cluster is hampered by the namenode's inability to handle more file system metadata, and the cluster is beset by frequent multi-second pauses where the JVM garbage collects the metadata. Tuning garbage collection on the namenode for such a large cluster requires significant, skilled administrator effort.

Moreover, the namespace metadata is kept strongly consistent using *multiple readers/single-writer* concurrency semantics, implemented using a global namespace lock. All the namespace changes (write-operations) are serialized, even if these operations intend to mutate different files in the

different sub-trees of the namespace hierarchy. A write-operation blocks all other operations, and a large write (that is deleting millions of files at once) operation can potentially block the namespace for a very long time. Some operators have reported that due to a sudden influx of write-operations and JVM stop-the-world garbage collection, the namenode can become unresponsive for tens of minutes [40]. Due to the *coarse-grained* global namespace lock, the namenode does not scale well on multi-core hardware, and the CPU is often underutilized for write-intensive workloads [41, 42]. For example, the HDFS throughput is limited to ≈80 thousand file system operations/sec on typical read-heavy industrial workloads, such as at Spotify. For a more write-intensive workload with 20% writes, the throughput drops to ≈20 thousand operations/sec [43].

2.1.2 HDFS Datanodes

The files in HDFS are split into blocks, with a default of 3 replicas of each block stored on different datanodes. A block placement policy determines the choice of datanodes to store the replicas of the blocks. The default block placement policy provides rack-level fault-tolerance and minimizes data transfers between racks. The client writes the first block to a local datanode if there is one, and the next two blocks to two datanodes on the same remote rack. The default block placement policy can be replaced with a custom block placement policy by an administrator. The datanodes periodically send heartbeats to both the active and standby namenodes indicating that they are alive, and the active namenode takes management actions to ensure that the file system is in a safe state. For example, when the active namenode detects that a datanode has failed, due to a number of consecutively missed heartbeats, for each block on the failed datanode, it orders one of the surviving datanodes that stores a copy of the lost block to replicate it to a different alive datanode in the system. The datanodes send periodic block-reports (default six hours) to the namenodes. The block-report contains IDs of all the blocks stored on the datanode. The block-report is also sent to the secondary namenode to keep it up-to-date with the block changes on the datanodes.

2.1.3 HDFS Clients

HDFS clients communicate with both the active namenode and the datanodes to access and update the stored data. To read a file, the client first communicates with the active namenode and retrieves the metadata for

the file. The metadata contains information about the datanodes that store the blocks of the file. The client then directly contacts the corresponding datanodes to retrieve the data blocks. Large HDFS deployments may have tens of thousands of clients [2]. For a large number of file system clients, the performance of the file system degrades as the file system operations wait in RPC call queues at the active namenode [44]. This severely affects the performance of the file system operation for small files where the cost of the metadata operations is significantly higher than the cost of writing/reading small files' data to/from the datanodes.

2.1.4 Journal Nodes

To avoid losing metadata in the eventuality of a namenode restart or failure, all changes to the metadata are logged and replicated to a set of external (at least three) Journal Nodes using a quorum-based replication algorithm. The log entries in the journal nodes are applied to the standby namenodes in batches. One of the standby namenodes takes over as the primary namenode when the active namenode fails after all the outstanding log entries in the journal nodes have been applied [45].

2.1.5 ZooKeeper Nodes

ZooKeeper is a third-party coordination service that provides services such as configuration management, naming, distributed synchronization, and group membership services [27]. HDFS uses ZooKeeper instances to allow all nodes in the cluster to reach an agreement on which namenode is currently active and which is standby.

2.2 Hops Hadoop Distributed File System

Hops Hadoop Distributed File System (HopsFS) is a new high-performance distribution of HDFS that removes the HDFS metadata bottleneck by introducing a scale-out distributed metadata layer. Hops can be scaled out at runtime by adding new nodes at both the metadata layer and the datanode layer. HopsFS has four types of nodes: namenodes, datanodes, clients, and NDB database nodes.

2.2.1 HopsFS Namenodes

In HopsFS, the metadata is stored in an in-memory, shared-nothing, open-source database called NDB cluster. This makes the namenodes in HopsFS

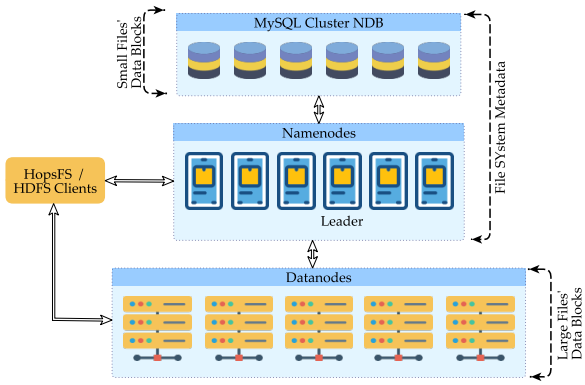


Figure 2.3: HopsFS system architecture. HopsFS supports multiple stateless namenodes and a single leader namenode. All the namenodes access and update the metadata stored in NDB cluster. HopsFS only decouples the metadata for large files. Data blocks for small files (⩽64 KB) are stored with the metadata in the distributed database, while the data blocks for large files (>64 KB) are stored and replicated on HopsFS datanodes. Unlike HDFS which uses ZooKeeper for coordination services, HopsFS solves this problem by using the database as shared memory to implement a leader election and group a membership management service.

stateless. HopsFS supports multiple redundant stateless namenodes that, in parallel, read and update the metadata stored in the external database. HopsFS only decouples the metadata for large files. Data blocks for small files (⩽64 KB) are stored with the metadata in the distributed database, while the data blocks for large files (>64 KB) are stored and replicated on the HopsFS datanodes.

HopsFS Leader Namenode

Unlike HDFS all the namenodes are active in HopsFS that simultaneously perform file system operations. The internal management (housekeeping) operations must be coordinated amongst the namenodes. Some housekeeping operations cannot be performed by multiple namenodes simultaneously. These housekeeping operations include ensuring that the file blocks are properly replicated and removing any dead datanodes. If these operations are executed simultaneously on multiple namenodes, they can potentially compromise the integrity of the namespace. For example, the replication manager service makes sure that all the blocks in the system have the required number of replicas. If any block becomes over-replicated, then the replication manager removes the replicas of the block until it reaches the required level of replication. If, however, we

had several replication managers running in parallel without coordination, they might take conflicting/duplicate decisions, and a block might end up being completely removed from the system. In HopsFS, these housekeeping operations are performed by a distinguished (leader) namenode. Unlike HDFS, which uses ZooKeeper [27] for coordination services, HopsFS solves this problem by using the database as a shared memory to implement a leader election and group membership service. The leader election service is discussed in detail in Chapter 10.

2.2.2 Network Database Cluster

Network Database Cluster (NDB) cluster belongs to the class of *NewSQL databases*: distributed, in-memory, high-performance, online transaction processing (OLTP) databases [46]. A typical NDB cluster setup includes multiple NDB database nodes that run Network Database storage engine, management nodes and optionally MySQL Server nodes. The NDB storage engine automatically partitions and replicates the data across multiple NDB database nodes. MySQL Server nodes can be used to access the data stored on the NDB database nodes using SQL queries. However, for high performance, NDB provides native APIs to bypass MySQL Servers and directly access the data stored on the NDB database nodes. Native APIs provide greater flexibility and yield higher throughput. NDB cluster has at least one management node that prevents data inconsistencies by halting the cluster when it detects a network partition. In the CAP theorem model, NDB provides consistency and partition tolerance, giving up availability. In scalability tests with 32 nodes, NDB has performed more than 200 million updates using native NDB API and 2.5 million SQL updates per second [47]. NDB cluster supports distribution aware ACID transactions. Transaction coordinators handle the transactions. Each database node has at least one (and maybe several) transaction coordinators. By default, the transactions are randomly load-balanced among all the transaction coordinators. However, the native APIs allows the client to enlist a transaction on a specific datanode. The transaction coordinator handles all the read and update queries in the transaction. The performance of a transaction greatly depends on how the data is partitioned. A transaction would run faster if all the data that is read and updated is local to the transaction coordinator, that is, the data resides in a partition stored on the same datanode where the transaction coordinator is running. This way all the read operations are handled by the local replica of the partition and updates are synchronized with the replicas

of the partition in the same replication node group, see Chapter 3 for a detailed discussion on NDB cluster.

2.2.3 HopsFS Datanodes

Unlike HDFS that store data blocks for files of all sizes on the datanodes, HopsFS only stores the data blocks of large files (>64 KB) on the datanodes. Similar to HDFS, the datanodes are connected to all the namenodes in HopsFS. Datanodes periodically send a heartbeat message to all the namenodes to notify them that they are alive. The heartbeat also contains information such as the datanode capacity, available space, and the number of active data transfer connections. This information is used by the namenodes for future block allocations and not persisted in either HDFS or HopsFS, as it is rebuilt on system restart using heartbeats from the datanodes. Both HDFS and HopsFS persist the mapping of files to blocks to stable storage so that it can be recovered in the event of failures. However, only HopsFS persists information about the datanodes on which blocks are stored. This enables faster restarts on a cluster failure compared to HDFS, as HopsFS does not strictly require block-reports from all the datanodes to rebuild the block location mappings data. In HopsFS, the datanodes also send periodic block-reports to the namenodes, however, instead of sending the block-report to all the namenodes the block-reports are sent to only one namenode in a *round-robin* manner. Moreover, the block-reports in HopsFS are more efficient. In HDFS the namenode processes all the blocks in a block-report, while, HopsFS uses a hash-based block-report system that is inspired by *Merkle trees* to reduce the computation of block-reports on the namenode side. In HopsFS, the block-report is split into a large number of sub-block-reports. The datanodes compute the hash of the sub-block-reports and send the hashes to the namenodes. The blocks in the sub-block-report are only processed if the hashes do not match on the namenode side [48].

2.2.4 HopsFS Clients

Similar to HDFS, the HopsFS clients communicate with both the namenodes and the datanodes. For example, to read a large file, the client first sends a *read-file* operation to one of the namenodes in the system. The namenode fetches the file's metadata from the database and ensures that the file path is valid and the client is authorized to read the file. After performing the metadata operation, the namenode returns the addresses

of the datanodes, where the file is stored, to the client. The client then contacts the datanodes to read the file. For reading small files, whose data blocks are stored in the database, the namenode fetches the data blocks along with the file's metadata and the namenode then returns the small files data blocks directly to the client. This reduces the number of communication steps involved in reading small files, reducing the end-to-end operational latencies for small files, see Chapter 9. HopsFS clients support *random*, *round-robin*, and *sticky* policies to distribute the file system operations among the namenodes. If a file system operation fails, due to namenode failure or overloading, the HopsFS client transparently retries the operation on another namenode after backing off if necessary. HopsFS clients refresh the namenode list every few minutes, enabling new namenodes to join an operational cluster. HDFS clients are fully compatible with HopsFS, although they do not distribute operations over namenodes, as they assume there is a single active namenode.

3 NDB Cluster NewSQL Database

In this chapter, we will briefly introduce the MySQL's Network Database (NDB) cluster architecture and the different types of database operations supported by NDB.

3.1 HopsFS Distributed Metadata

HopsFS stores the file system metadata in NDB. The file system operations are converted into distributed transactions that read and update the metadata stored in NDB cluster. In HopsFS, a non-empty file comprises one or more data blocks which are replicated (by default) three times on HopsFS datanodes. The file system metadata is stored in different tables in the database. The *inodes* table stores the name of the file/directory, permission attributes, file size, and ownership information. Other tables, such as *blocks* table stores information about the data blocks belonging to the different files and the *replicas* table stores information about which HopsFS datanodes store the replicas of the blocks.

The metadata is stored in the database in normalized form. Instead of storing entire file paths with each inode, HopsFS stores individual file path components as separate rows in an inodes table. For example, instead of storing an inode with full pathname */etc/gnupg*, we store three inodes (one for each path component): *root (/)*, *etc*, and *gnupg* with foreign keys to the parent inodes to maintain the namespace hierarchy. Storing the normalized data in the database has many advantages, such as enabling *rename* file system operation to be performed in-place by updating a single

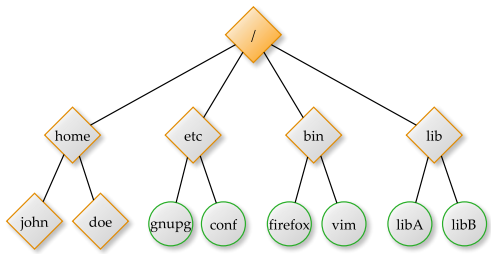


Figure 3.1: A sample file system namespace. The diamond shapes represent directory inodes and the circles represent file inodes. The inodes are stored in the Inodes *table where each row represents a single file or directory.*

row in the database that represents the inode. If complete file paths are stored alongside each inode, it would not only consume a significant amount of precious database storage space but also, a simple directory rename operation would require updating the file paths for all the children of the directory subtree. The rename operation is an important operation for higher level frameworks, such as Apache Hive that uses the atomic rename operations to commit transactions in Hive. The HopsFS schema contains around 40 tables that store the metadata of the file system. For the sake of simplicity, here we will only consider the *inodes* table. Table 3.1 shows a possible solution for how the namespace shown in Figure 3.1 can be stored in the *inodes* table.

Now we will discuss the system architecture of the NDB cluster database and show how the *inodes* table is stored in the distributed database and how the *inodes* table is accessed using different types of database operations.

3.2 NDB Cluster NewSQL Database

HopsFS uses MySQL's NDB cluster to store the file system metadata. NDB is an open source, real-time, in-memory, shared nothing, distributed database management system (and is *not* to be confused with clustered MySQL Servers based on the popular InnoDB storage engine. The MySQL Server supports different database storage engines). NDB has three types of nodes: NDB database nodes, management nodes, and database clients, see Figure 3.2. The management nodes are only used to disseminate the configuration information and to act as arbitrators in the event of a

ID	PID	Name	isDir	...
1	0	/	true	...
2	1	home	true	...
3	1	etc	true	...
4	1	bin	true	...
5	1	lib	true	...
6	2	john	true	...
7	2	doe	true	...
8	3	gnupg	false	...
9	3	conf	false	...
10	4	vim	false	...
11	4	firefox	false	...
12	5	libA	false	...
13	5	libB	false	...

Table 3.1: Normalized representation of the `inodes` table. Each path component is stored as a single row in the table. `PID` refers to the parent inode's `ID` for the given path component (for example, the `etc` directory is the parent of `gnupg` file).

network partition. The client nodes access the database using the SQL interface via MySQL Server or using the native APIs implemented in C++, Java, JPA, and JavaScript/Node.js. The SQL API is not recommended for implementing high-performance applications for NDB. Instead, the NDB storage engine can be accessed using its native (C++) NDB API or the ClusterJ (Java) API for low latency database operations.

NDB database nodes are responsible for storing the database tables. Figure 3.2 shows a NDB cluster setup consisting of four NDB database nodes. Each NDB database node has multiple transaction coordinators (TC), local data managers (LDM), send, receive, and IO threads. The transaction coordinators execute two-phase commit transactions. The local data management threads are responsible for storing and replicating the data partitions assigned to the NDB database nodes. Finally, the send and receive threads are used to exchange the data between the NDB database nodes and the clients, and the IO threads are responsible for performing disk IO operations.

NDB horizontally partitions the tables, that is, the rows of the tables are distributed among the database partitions stored on the NDB database nodes. Rows are assigned to the partitions based on an application-defined partition key or if none has been supplied, the MD5 hash of the primary key of the row. The NDB database nodes are organized into node replication groups of equal sizes to manage and replicate the data

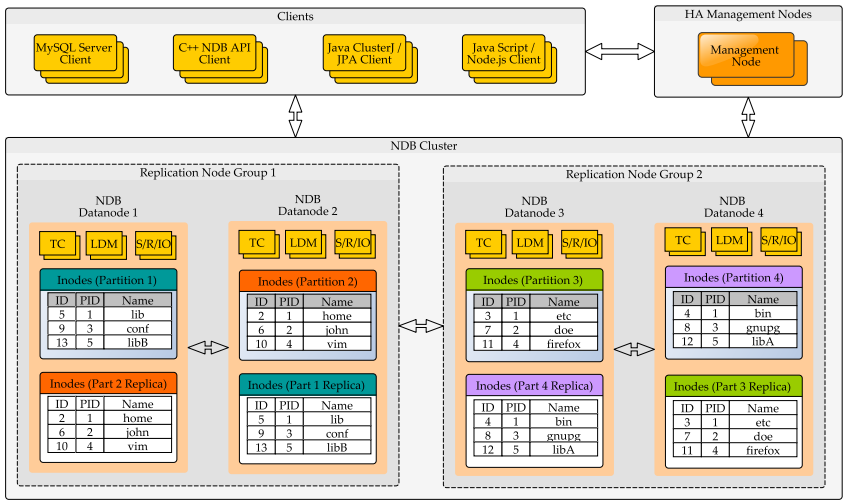


Figure 3.2: System architecture of NDB cluster. The NDB cluster has three types of nodes: database clients, NDB management nodes, and NDB database nodes. This figure also illustrates how the inodes table as shown in Table 3.1 is horizontally partitioned and replicated across the NDB database nodes.

partitions. The size of the node replication group is the replication degree of the database. In the example setup, the NDB replication degree is set to two (default value). Therefore, each node replication group contains two NDB database nodes. The first replication node group consists of NDB database node 1 and 2, and the second replication node group consists of NDB database nodes 3 and 4. Each replication node group is responsible for storing and replicating all the data partitions assigned to the NDB database nodes in the replication node group. By default, NDB hashes the primary key columns of the tables to distribute the rows among the different database partitions. Figure 3.2 shows how the *inodes* table (Table 3.1) is partitioned and stored in NDB. In production deployments, each NDB database node may store multiple data partitions, but for simplicity, the NDB database nodes shown in Figure 3.2 store only one data partition. The replication node group 1 is responsible for storing and replicating partitions 1 and 2 of the inodes' table. The primary replica of partition 1 is stored on the NDB database node 1, and the replica of the partition 1 is stored on NDB database node 2. For example, the NDB database node 1 is responsible for storing a data partition that stores the *lib*, *conf*, and *libB* inodes, and the replica of this data partition is stored on NDB database node 2.

NDB also supports an application-defined partitioning of the stored tables, that is, it can partition the data based on a user-specified table column. Application-defined partitioning provides greater control over how the data is distributed among different database partitions, which helps in implementing very efficient database operations, see section 3.4.3 for more details on application-defined data partitioning in NDB.

By default, the database is stored in-memory on the NDB database nodes, with recovery logs and snapshots stored on disk. Columns in a table may be defined as on-disk columns, but the primary key and indexes are still stored in-memory. All transactions are committed in-memory, and transaction logs are (by default) flushed to disk every two seconds. The database can tolerate failures of multiple NDB database nodes as long as there is at least one surviving replica for each of the partitions. For example, in Figure 3.2, the database cluster will stay active even if the NDB database nodes 1 and 4 fail. However, if two nodes in the same replication node group fail, then the database will halt its operations until the unavailable replication node group has recovered. As such, NDB favors consistency over availability [49]. NDB cluster supports asynchronous replication of the whole cluster, enabling data from one NDB cluster (the master) to be copied to one or more NDB clusters (the slaves). When the master database cluster is unavailable, then the database can fail-over to a slave NDB cluster database on the fly without affecting the availability of the database. NDB cluster supports both node level and cluster level recovery using persistent transaction *redo* and *undo* logs and checkpoint mechanisms. Every two seconds a global checkpoint mechanism ensures that all the NDB database nodes agree on a global *epoch ID* that is appended to the logs checkpointed to local-disk. Global checkpoints are needed as there are multiple independent transaction coordinators that need to agree on a consistent snapshot of the system when recovering, identified by the latest *epoch ID*. As such, there is a possibility of data loss on a system failure, for any transaction committed in-memory with an epoch-id higher than the last epoch persisted in the last global checkpoint. By default, a global checkpoint is performed every two seconds. Typically this means up to two seconds of data may be lost on a system failure, but some of that data may be recoverable from redo logs.

3.3 Transaction Isolation

NDB only supports *read-committed* transaction isolation, which guarantees that any data read is committed at the moment it is read. The *read-committed* isolation level does not allow *dirty* reads, but *phantom* and *fuzzy* (non-repeatable) reads can happen in a transaction [50]. However, NDB supports row-level locks, such as *exclusive* (write) locks, *shared* (read) locks, and *read-committed* locks that can be used to isolate conflicting transactions.

3.4 Types of Database Operations

HopsFS implements the file system operations as distributed database transactions that consist of three distinct phases. In the first phase, all the metadata that is required for the file system operation is read from the database, in the second phase the operation is performed on the metadata, and in the last phase all the updated metadata (if any) is stored back in the database, see Section 4.1.4 and 7.5 for more detail. As the latency of the file system operation depends on the time spent in reading and writing the data, therefore, it is imperative to understand the latency, and computational cost of different database operations used to read and update the stored data, to understand how HopsFS implements low latency scalable file system operations.

NDB supports four different types of database operations: *primary key*, *partition pruned index scan*, *distributed index scan*, and *distributed full table scan* operations for reading the data stored in the database. Below, these different types of database operations are explained in detail along with micro-benchmarks that show the throughput and latency of these operations. The micro-benchmarks were performed on a four-node NDB cluster setup running NDB Version 7.5.6. All the experiments were run using Dell PowerEdge R730xd servers (Intel Xeon ® CPU E5-2620 v3 @ 2.4 GHz, 256 GB RAM, 4 TB 7200 RPM HDD) connected using a single 10 GbE network adapter. In the experiments, we varied the amount of the data read/updated in each operation as well as the number of concurrent database clients. Up to 1.2 millions inodes were created in a table to test the performance of different database operations. The database clients were not collocated with the database nodes, and the database clients (up to 400) were uniformly distributed across 20 machines.

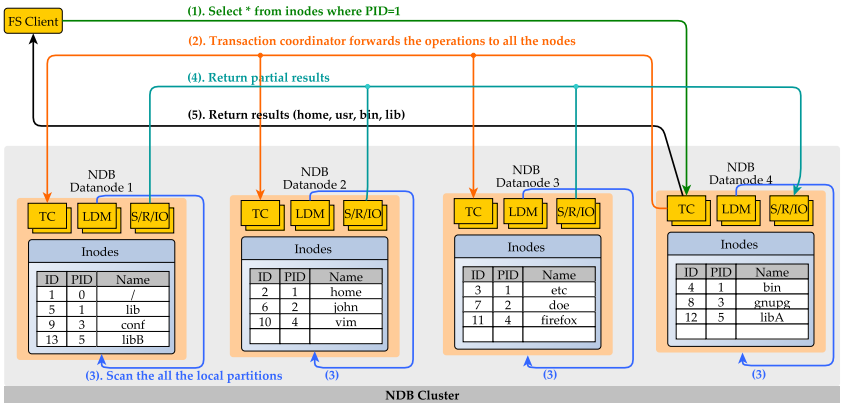
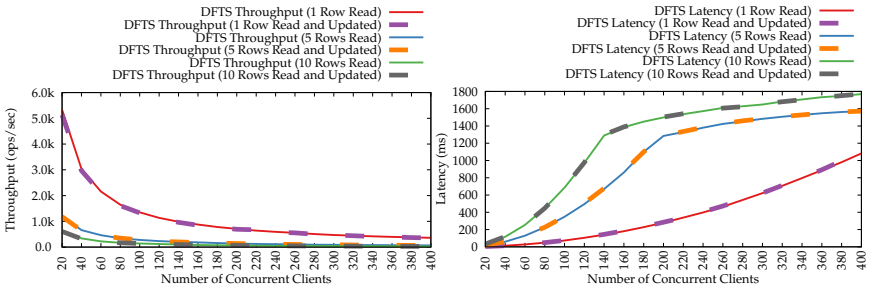


Figure 3.3: Different stages of distributed full table scan operation for reading all the immediate children of the root(/) directory. The client sends the operation to one of the transaction coordinators in the system, which forwards the operation to all the other NDB database nodes. The NDB database nodes read all the rows of the inodes table to locate the desired information, that is, rows where PID is set to 1. The partial results from each NDB database nodes are sent back to the transaction coordinator that initiated the operation. The transaction coordinator collects all the partial results and then forwards the results back to the client.

3.4.1 Distributed Full Table Scan Operations

Distributed full table scan operations read all the rows of the table stored in all the database partitions to retrieve the desired information. Distributed full table scan operations are computationally intensive and have very high latency. For example, consider a client that wants to perform a directory listing operation on the root directory of the namespace, as shown in Figure 3.1. Assume that the metadata is hash partitioned based on the *inodes'* table ID column, that is, all the inodes are uniformly distributed among the NDB database partitions, as shown in Figure 3.3. The listing operation can be performed using the query, `select * from inodes where PID = 1`. The figure also shows the different steps involved in performing the distributed table scan operation.

1. First, the user sends the transaction containing the single operation (`select * from inodes where PID = 1`) to a random transaction coordinator.
2. As the data is hash partitioned using the ID column of the *inodes* table, the immediate children of the *root(/)* directory will be spread across all the database partitions. As no single database partition



(a) DFTS Throughput *(b) DFTS Latency*

Figure 3.4: Micro-benchmarks for distributed full table scan (DFTS) operations. The experiments are repeated twice. The first set of experiments (represented by the solid lines) are read-only operations, and the second set of experiments (represented by the dotted lines) are read-write operations where all the data read is also updated and stored back in the database. For example, the solid red line shows the throughput and latency of distributed full table scan operations that read only one row. Distributed full table scans have a maximum throughput of 5.5K operations per second when only one row is read from the database. The throughput drops as more data is read in each operation and the number of concurrent clients increases. For reading 10 rows in each operation the maximum throughput drops to 650 operations per second and the maximum latency increases to 1.8 second per operation. The dotted lines for read-write experiments overlap with their corresponding read-only experiments because the amount of data that is updated is very low, and it does not have any significant impact on the performance of read-write distributed full table scan operations.

holds all the rows needed for the operation, the transaction coordinator forwards the database operation to all the database partitions to perform the scan operations.

3. All the database partitions locally scan all the stored rows for the *inodes* table to retrieve the rows with PID set to 1.

4. Upon completion of the local table scan operations, the NDB database nodes send the partial results back to the transaction coordinator that is responsible for the operation.

5. Once the transaction coordinator has received the results from all the NDB partitions it forwards the results back to the client.

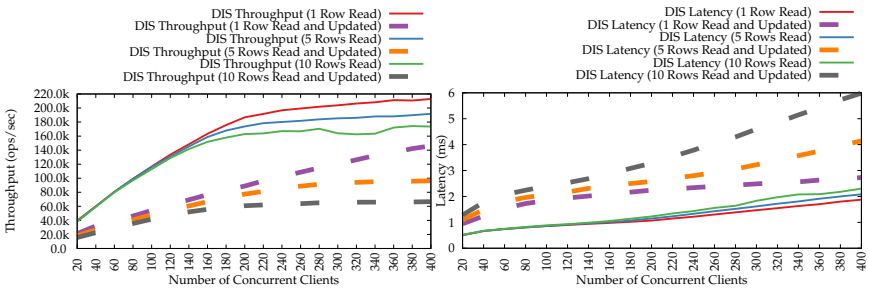
Distributed full table scan operations are the least efficient database operation for reading data from the database. The *inodes* table may contain billions of rows. Reading and searching through all the rows incurs prohibitive CPU costs. Figure 3.4 shows the throughput and latency of distributed full table scan operation as a function of the number of concurrent clients and the number of rows read/updated in each operation. The

experiments are repeated twice. The first set of experiments (represented by solid lines) are read-only operations, and the second set of experiments (represented by dotted lines) are read-write operations where all the data read is also updated and stored back in the database. Distributed full table scan operations do not scale, and the throughput of the system drops and the latency increases when the number of concurrent database clients increases. This is because the database can only handle a very limited number of concurrent distributed full table scan operations. NDB limits the number of concurrent distributed full table scan operations to 500, due to very high CPU cost of these operations. In the micro-benchmarks, we only managed to perform 5.5K distributed full table scan operations that read and update a single row from the test table, and the throughput drops to 350 operations per second when the number of concurrent database clients reaches 400. The latency of the distributed full table scan operations is also very high that increases with the increase in the number of concurrent database clients. The dotted lines for read-write experiments overlap with their corresponding read-only experiments because the amount of data updated is very small, and the time required to update the data is not visible due to a very high cost of reading the data using distributed full table scan operations. For the operations that read and update 10 rows, the average latency of the operation increases to 1.8 seconds using 400 concurrent database clients.

3.4.2 Distributed Index Scan Operations

The primary reason that distributed full table scans are very slow is that these operations read all the data stored in the tables. A simple solution to speed up the distributed full table scan operations is to create an index over the desired table columns. In the example above if we create an index for the *PID* column then the performance of the scan operation can be significantly improved. A distributed scan operation using an index becomes *distributed index scan operation*. The steps involved in a distributed index scan operation are the same as distributed full table scan operations except the datanodes use an index scan operation. Figure 3.5 shows the throughput and latency of distributed index scan operations. Distributed index scan operations have better scalability and lower latency than the distributed full table scan operations. Distributed index scan operation delivers 38X to 590X the throughput of distributed full table scan operations using 20 and 400 concurrent database clients, respectively for reading single row in each operation. In the case of

read-write experiments, the throughput drops and the latency increases due to the overhead of distributed two-phase commit protocol used by NDB to update the table rows atomically. For the single row tests, the throughput drops from 210K to 145K and the latency increases from 1.8 milliseconds to 2.7 milliseconds for reading and updating a single row in each operation. Similarly, for tests that read and update 10 rows, the throughput drops from 173K to 66K operations per second and the latency increases from 2.3 milliseconds to 5.9 milliseconds using 400 concurrent database clients.



(a) DIS Throughput *(b) DIS Latency*

Figure 3.5: Throughput and latency of distributed index scan (DIS) operations as a function of the number of database clients and the amount of data read/updated in each operation. Creating an index significantly improves the performance of the operation. The maximum throughput of distributed index scan operations reaches 210K operations per second as opposed to a maximum throughput of 5.5K operations per second for distributed full table scan operations. Updating the data generally halves the throughput and doubles the latency of the operation, this is due to the high cost of two-phase commit protocol for updating the data.

3.4.3 Partition Pruned Index Scan Operations

The performance of the distributed index scan operation can be improved by reducing the number of NDB database nodes that participate in the operation. As the default partitioning scheme spreads the inodes' rows for the immediate children of the *root(/)* directory across all data partitions, therefore, it is not possible to confine the index scan operation to a limited number of database nodes. NDB supports an application-defined partitioning mechanism that allows the users to partition the tables using user-defined columns. A limitation of NDB is that the partition key must be part of the table's primary key. If the *inodes* table is partitioned using the parent ID (PID) column, then all the immediate children of a directory will reside on same database partition. For example, in Figure 3.6, all the

immediate children of the *root(/)* directory will reside on NDB database node 1, and the immediate children of the home directory will reside on NDB database node 2, and so on, making it possible to read all the required metadata from a single database partition for the directory listing operation. A partition pruned index scan operation is an index scan operation where the index scan is performed on a single database partition. In HopsFS, the *inode ID* column acts as a candidate key column, and the composite primary key consists of the *parent ID (PID)* and *name* columns of the *inodes* table. Figure 3.6 shows different stages of the partition pruned index scan operation used to read all the immediate children of the *root(/)* directory.

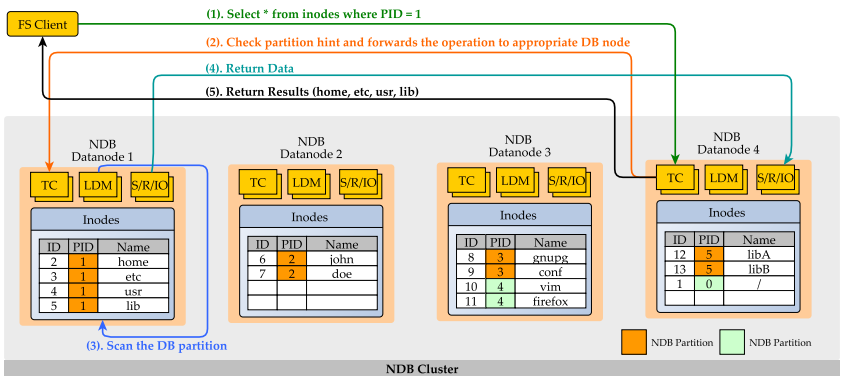


Figure 3.6: Different stages of a partition pruned index scan (PPIS) operation for reading all the immediate children of the root(/) directory. The client sends the operation to one of the transaction coordinators in the system, which forwards the operation to only one NDB database node which stores the data partition responsible for holding all the immediate children of the root(/) directory. The NDB database node performs a local index scan operation on a single database partition and sends the results back to the transaction coordinator. The transaction coordinator then forwards all the results back to the client.

1. The client sends the operation (`select * from inodes where PID = 1`) to a transaction coordinator located on a random NDB database node.

2. As the *inodes* table is partitioned using the PID column the transaction coordinator can infer where all the data for this query resides. The transaction coordinator forwards the operation to the NDB database node 1, which stores all the immediate children of the *root(/)* inode.

3. The index scan operation is performed by only one database partition on the NDB database node 1.

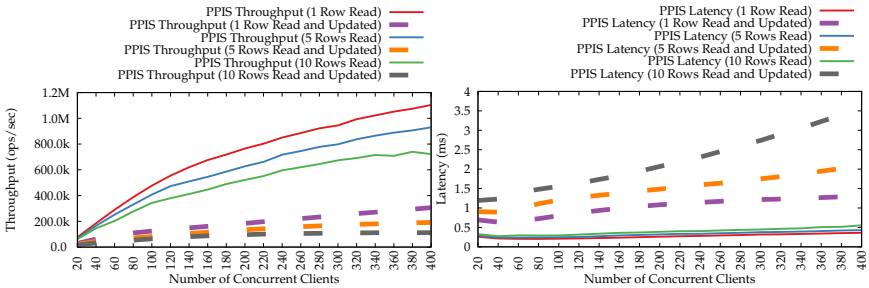
4. The results are sent back to the transaction coordinator that initiated the operation.

5. The transaction coordinator then returns the results to the client.

3.4.4 Data Distribution Aware Transactions

Apart from controlling how NDB partitions the data, the client can also specify which transaction coordinator should be used to perform the database operations. In NDB this is done by specifying a *hint* when starting a transaction. For example, in the above case, when starting the transaction, the client can provide a hint (the partition ID) to NDB to start the transaction on the NDB database node that is responsible for all the rows with *PID=1*. This will start the transaction on NDB database node 1, thus, reducing the number of messages exchanged between the NDB database nodes for the given transaction. Such types of transactions are also known as *data distribution aware transactions*.

Figure 3.7 shows the throughput and latency of partition pruned index scan operations. All the operations performed in these experiments are data distribution aware, that is, the partition pruned index scan operations were started on the datanodes that held the data required for the operation. Partition pruned index scan operations scale better than distributed index scan operations, as in each operation only one NDB data partition participates in performing the index scan operation. In our experiments with four NDB database nodes, Partition pruned index scan operations delivers ≈4-5X the throughput of distributed index scan operations depending upon the amount of data read in each operation and have lower latency than distributed index scan operations. Similar to the distributed index scan operations, the throughput and latency drop significantly for read-write operations. Using partition pruned index scan operations, we managed to perform ≈120K operations, that read and update 10 rows, every second, which is ≈1.8X the throughput of distributed index scan operations for the similar experiment. In our experiments, we only had four NDB database nodes. For clusters with even more NDB database nodes, we can expect that partition pruned index scan operations will increase their relative performance advantage after distributed index scan



(a) PPIS Throughput *(b) PPIS Latency*

Figure 3.7: Throughput and latency of partition pruned index scan (PPIS) operations as a function of the number of database clients and the amount of data read/updated in each operation. Limiting the number of database partitions that participates in the index scan operation greatly improves the performance and lowers the latency of the operations. The partition pruned index scan delivers ≈4-5X the throughput of distributed index scan operations depending upon the amount of data read in each operation and have lower latency than distributed index scan operations.

operations as transactions will have to wait for responses from even more NDB database nodes.

3.4.5 Primary Key Operations

A Primary Key operation reads/writes a single row from a table. HopsFS uses primary key operations to resolve the file and directory paths recursively. Note that the primary key of the *inodes* table in HopsFS is a composite key consisting of the *name* and *parent ID* of the inode. Moreover, the root directory is a special immutable directory whose ID and name does not change. The ID and parent ID of the *root(/)* directory are set to 1 and 0, respectively. In HopsFS, the path */home/doe* will be resolved, as following, using only primary key lookup operations. In the first primary key lookup, the *root(/)* directory will be resolved using the query `(select * from inodes where PID=0 and name='/')`. This will return the inodes' table row for the *root(/)* directory. In the second primary key lookup, the *home* directory will be resolved using the query `(select * from inodes where PID=root.ID and name='home')`. This will return the inodes' table row for the *home* directory. In the subsequent third primary key operation, the directory *doe* will be resolved using the query `(select * from inodes where PID=home.ID and name='doe')`. This will return inodes' table row for the directory *doe* stored in the *home* directory. Primary key operations are data distribution aware, as NDB uses the primary key (or part of the primary key in case of a composite primary key) to

partition the table's data. Primary key operations are the most efficient database operation for reading single rows or a small number of rows in a batch of primary key operations. Figure 3.8 shows the steps involved in performing a primary key lookup operation for resolving the *home* directory inode.

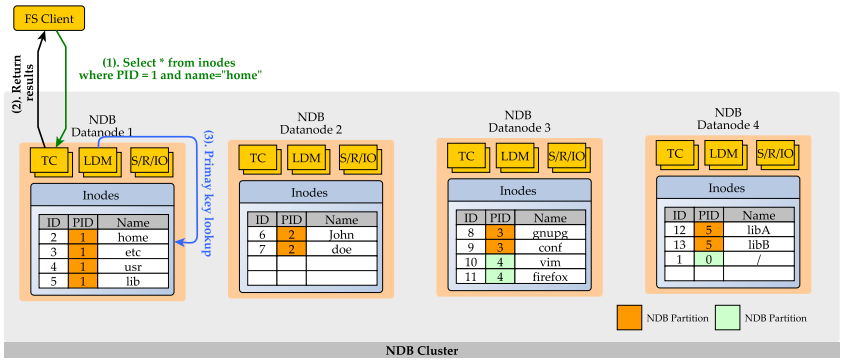
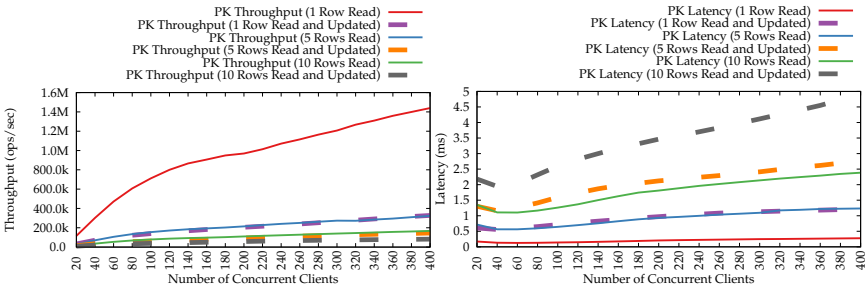


Figure 3.8: Primary key operations are data distribution aware. The client sends the operation to one of the transaction coordinators on NDB database node 1 as it is responsible for storing all inodes with PID=1. The operation is locally resolved and the result is sent back to the client.

Figure 3.9 shows the throughput and latency of primary key lookup operations. Primary key operations can read up to 1.4 million rows per second using 400 concurrent database clients. The throughput drops to 170K operations per second if 10 rows are read in each transaction. This is due to the iterative reading of the rows, that is, using a primary key operation the first row is read, and then the second row read, and so on. Primary key operations have very low latency where reading a single row takes on average 0.2 milliseconds. Similar to the partition pruned index scan operations, the throughput of the primary key operations drops for the read-write tests (represented by the dotted lines). This is due to the high cost of two-phase commit protocol used to update the replicated inode rows atomically.

3.4.6 Batched Primary Key Operations

The throughput of the database, in terms of operations/second, can be improved by batching operations in transactions. The batch size defines the number of operations in a transaction and, in general, larger batch sizes result in higher throughput in operations/second. NDB's Java API, ClusterJ (used by HopsFS) only supports batching primary key operations.



(a) PK Throughput *(b) PK Latency*

Figure 3.9: Throughput and latency of primary key (PK) operations as a function of the number of database clients and the amount of data read/updated in each operation. Primary key operations are data distribution aware, that is, these operations are sent directly to the database nodes that hold the desired data. Using primary key operations, more than 1.4 million rows can be read every second using 400 concurrent database clients. For reading multiple rows in the same transaction the performance drops significantly due to iterative operations used to read multiple rows. Primary key operations have very low latency, where a reading a single row takes on average 0.2 milliseconds.

Figure 3.10 shows a batch of two primary key operations used to resolve the two inodes that make up the path component of the path */home*. The inodes are *root (/)* inode and the *home* inode.

1. The client creates a batch of primary key operations. In this example the batch consists of two primary key read operations, that is, (PID=0, name="/") and (PID=1, name="home"). See section 7.5.1 for more details on how HopsFS discovers the primary keys for individual file path components using the *inode cache*. Primary key operations are data distribution aware. However, the data for primary key operations in a batch may reside on different partitions. The client starts the transaction on an NDB database node that holds all or most of the data required for the operations. Here the data is spread over two partitions stored on NDB database node 1 and NDB database node 4. Therefore, the client started the transaction on NDB database node 4 that holds 50% of the data required for the batch operation.

2. The transaction coordinator forwards the primary key operations to their respective NDB database nodes that store the data required for individual primary key operations. Here one of the primary key operations is resolved locally on NDB database node 4 while the other primary key operation is sent to NDB database node 1.

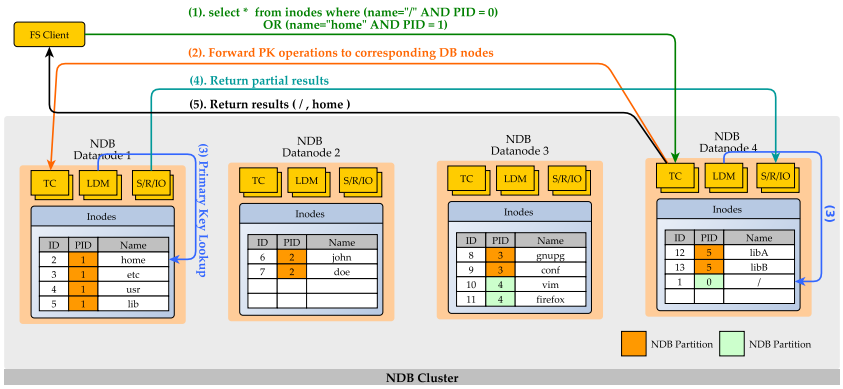
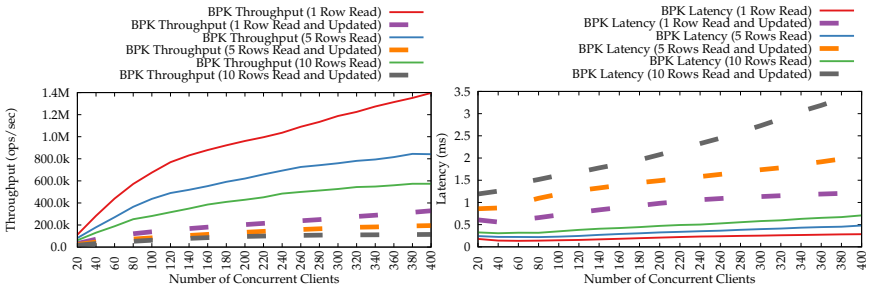


Figure 3.10: Steps involved in batched primary key operations in NDB. The client starts the transaction on the NDB database node that holds all or most of the data required for the primary key operations. The transaction coordinator forwards the primary key operations to the NDB database nodes that hold the data rows required for the primary key operations. The NDB database nodes perform local primary key lookup operations and send the results back to the transaction coordinator. The transaction coordinator then forwards the results to the client.

3. The NDB database nodes perform a local primary key lookup to locate the rows.

4. The NDB database nodes send the results back to the transaction coordinator which then forwards the results to the client.

Figure 3.11 shows the throughput and latency of the batched primary key operations. For reading a single row, the throughput of the operations (1.4 million ops/sec) matches the throughput of the primary key operations because batching a single operation is same as a single primary key operation. However, for reading multiple rows, batched operations have significantly higher throughput than iterative primary key operations. For example, for reading 10 rows, batched operations deliver 3.4X times the throughput of primary key operations and have three times lower average latency. For the read-write test, the throughput drops to 115K operations per second when ten rows are read and updated in each operation using 400 concurrent clients. While for a similar read-write test that read and updated 10 rows using (non-batched) primary key operations the throughput is 80K operations per second, see Figure 3.9.

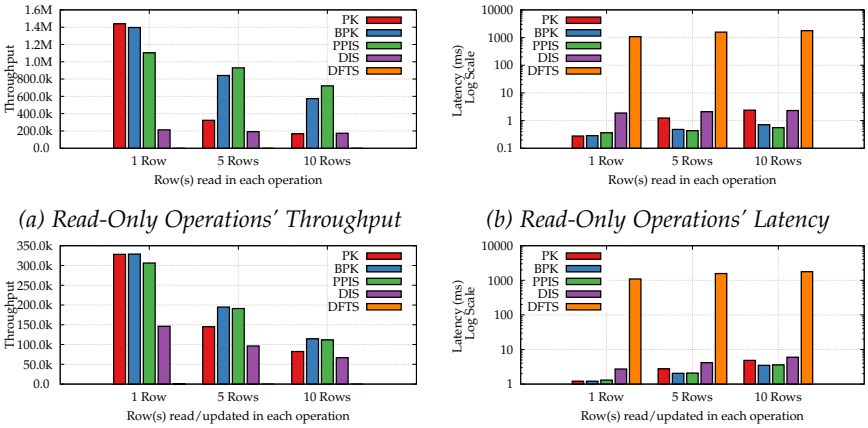


(a) BPK Throughput *(b) BPK Latency*

Figure 3.11: Throughput and latency of batched primary key (BPK) operations as a function of the number of database clients and the amount of data read/updated in each operation. Batched primary key operations have 3.4X times higher throughput and 3X times lower latency than primary key operations when ten rows are read in each operation. Similar to primary key operations tests throughput drops for read-write tests. The throughput drops to 115K operations per second when ten rows are read and updated in each operation, where for the similar test the throughput for primary key operations is 80K operations per second.

3.5 Comparing Different Database Operations

Figure 3.12 shows the comparison of throughput and latency of different types of database operations using 400 concurrent database clients. Primary key operations are the fastest database operations for reading single rows from the database. However, if multiple rows are read from the database in the same transaction, then partition pruned index scan operations and batched primary key operations are more suitable. Batched primary key operations are used in scenarios where the client knows in advance the primary keys of the rows that it wants to read. Otherwise, partitioned pruned index scan operations are more suitable for reading multiple rows from the database. Distributed full table scan operations and distributed index scan operations do not scale as the number of concurrent database clients increases. These operations have the lowest throughput and the highest end-to-end latency among all the database operations, therefore, these operations must be avoided in implementing file system operations. Primary key, batched primary key operations, and partition pruned index scan operations are scalable database operations that can be used to implement efficient file system operations. The throughput of these operations increases as the number of concurrent database clients increases. Moreover, the end-to-end latency of these operations does not increase rapidly when the number of concurrent database clients increases. The design of the database schema, that is, how the data is laid out in different tables,



(a) Read-Only Operations' Throughput

(b) Read-Only Operations' Latency

(c) Read-Write Operations' Throughput

(d) Read-Write Operations' Latency

Figure 3.12: Comparison of the performance of different database operations using 400 database clients executing concurrent database operations on a 4 node NDB cluster database. Distributed full table scan operations (DFTS) and distributed index scan operations (DIS) do not scale as the number of concurrent database clients increase. Primary key (PK), batched primary key (BPK) operations, and partition pruned index scan operations (PPIS) are scalable database operations, whose throughput increases as the number of concurrent database clients increases. For reading multiple independent rows of data, batched primary key operations have higher throughput and lower latency than the iterative primary key operations.

the design of the primary keys of the tables, types/number of indexes for different columns of the tables, and data partitioning scheme for the tables plays a significant role in choosing an appropriate (efficient) database operation to read/update the data. In HopsFS, the schema is designed such that all frequently used file system operations are implemented using high-throughput and low-latency database operations. For more detail see Chapter 7, and Chapter 8.

4
Thesis Contributions

Great scientific contributions have been techniques.

— B. F. Skinner

The contributions of this thesis are twofold. First, as a part of the research, we have implemented a production-grade distributed file system, HopsFS, which is a drop-in replacement for HDFS. HopsFS is the underlying file system in Hops Hadoop distribution[1]. Hops Hadoop has been running in production in RISE SICS ICE data center since February 2016 [51]. At RISE SICS ICE we have been providing Hadoop-as-a-Service to researchers and companies in Sweden. Currently, our system has five hundred registered users that regularly use our data processing platform. Companies like *Scania AB*[2], *Mobilaris Group*[3], and *Karolinska Institute*[4] are among the prominent users of HopsFS. HopsFS was part of the teaching platform used to teach the *ID2223: Scalable Machine Learning and Deep Learning*[5] course at KTH in the Fall 2016 and 2017 attended by more than 200 students. Second, from the research perspective, we show different algorithms and design/implementation techniques used to build a scalable,

[1] Hops Hadoop: Hadoop for humans. `http://www.hops.io/`

[2] Scania AB. `https://www.scania.com`

[3] Mobilaris Group. `http://www.mobilaris.se/`

[4] Karolinska Institute. `https://ki.se/start`

[5] ID2223: Scalable Machine Learning and Deep Learning. `https://www.kth.se/student/kurser/kurs/ID2223?l=en`

high-performance distributed metadata service using a NewSQL database system.

4.1 HopsFS Research Contributions

To the best of our knowledge, HopsFS is the first open-source distributed file system that stores normalized metadata in a NewSQL distributed relational database, NDB cluster in our case. Storing the metadata in the NewSQL database introduces many new challenges. For example, how to partition (shard) the file system metadata and implement strongly consistent file system metadata operations using only the *read-committed* transaction isolation level provided by the NewSQL distributed databases? Storing the file system metadata in an external database increases the latency of the file system operations as the external database is accessed multiple times to resolve individual file path components and to read blocks' metadata. To improve the performance of file system operations, we leverage both classical database techniques such as *batching* (bulk operations) and *write-ahead* caches within transactions, as well as distribution aware techniques commonly found in NewSQL databases. These distribution-aware NewSQL techniques include *application-defined partitioning* (we partition the namespace such that the metadata for all immediate descendants of a directory (child files/directories) reside on the same database partition for efficient directory listing), and *distribution aware transactions* (we start a transaction on the database partition that stores all/most of the metadata required for the file system operation), and *partition pruned index scans* [26]. We also introduce an inode hints cache for faster resolution of file paths. Cache hits when resolving a path of depth N can reduce the number of database round-trips from N to 1.

Recursive operations on large directories, containing millions of inodes, are too large to fit in a single transaction. Our solution is a protocol that implements subtree operations incrementally in batches of transactions. Our subtree operations protocol provides the same consistency semantics as subtree operations in HDFS. We have also designed an adaptive tiered storage using in-memory and on-disk tables stored in a high-performance distributed database to efficiently store and improve the performance of the small files in Hadoop. Lastly, we show how to avoid third-party coordination service and implement a leader election service using NewSQL databases for partially synchronous systems, which ensures at most one leader at any given time. In the rest of this chapter, we will briefly intro-

duce these research contributions that enables HopsFS to deliver an order of magnitude higher throughput and lower latency file system operations than HDFS for real word industrial workload traces, see section 4.1.9 for an overview of the performance improvements in HopsFS.

4.1.1 Distributed Metadata

Metadata for hierarchical distributed file systems typically contains information on inodes, blocks, replicas, quotas, leases, and mappings (directories to files, files to blocks, and blocks to replicas). When metadata is distributed, an application-defined partitioning scheme is needed to partition the metadata, and a consensus protocol is required to ensure metadata integrity for operations that cross partitions (shards). File system operations in HopsFS are implemented primarily using multi-partition two-phase transactions and row-level locks in MySQL Cluster to provide serializability [52] for metadata operations.

The choice of partitioning scheme for the hierarchical namespace is a key design decision for distributed metadata architectures. We base our partitioning scheme on the expected relative frequency of file system operations in production deployments and the cost of different database operations that can be used to implement the file system operations. We have observed that in production environments (such as at Spotify) *list*, *stat* and *file read* operations alone account for $\approx 95\%$ of the operations in the HDFS cluster. These statistics are similar to the published workloads for Hadoop clusters at Yahoo! [53], LinkedIn [54], and Facebook [41]. The metadata partitioning scheme for HopsFS is designed such that the metadata frequently used in file system operations resides on a single database partition. HopsFS metadata design and metadata partitioning enable implementations of common file system operations using only the low-cost database operations, that is, primary key operations, batched primary key operations, and partition pruned index scans. For example, the read and directory listing operations are implemented using only (batched) primary key lookups and partition pruned index scans. Index scans and full table scans were avoided, where possible, as they touch all database shards and scale poorly.

Figure 4.2 shows how HopsFS stores the metadata for the namespace shown in Figure 3.1 in the NDB cluster in normalized form. For readability, we only show three tables, that is, *inodes*, *blocks*, and *replicas* tables. The

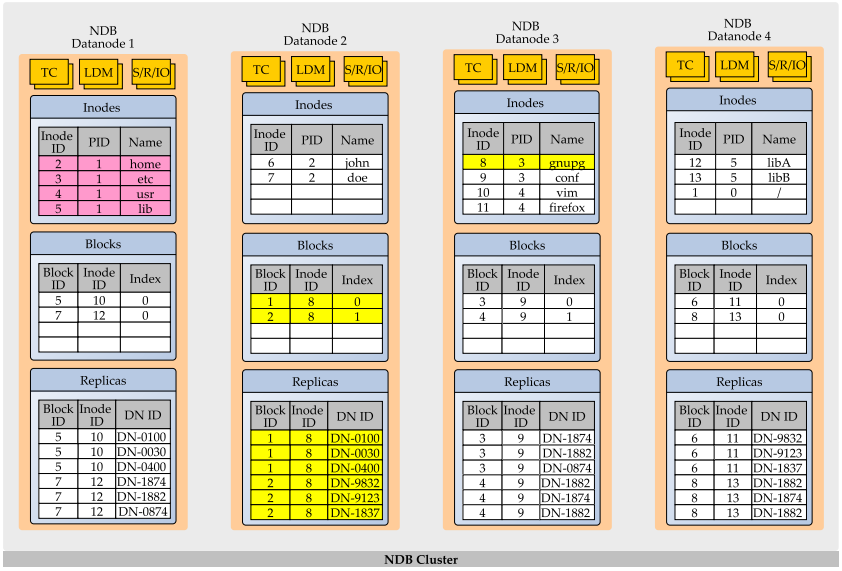
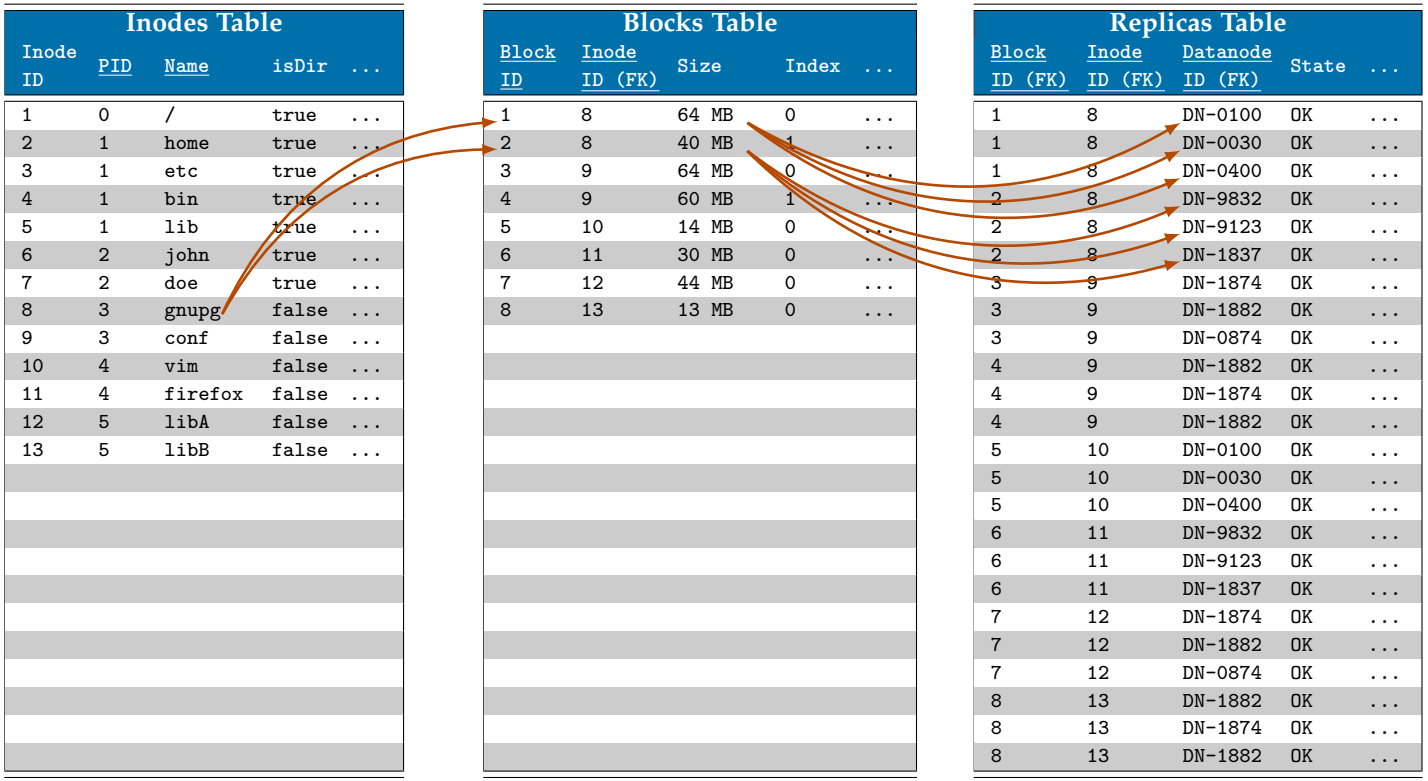


Figure 4.1: This figure shows how the metadata in HopsFS is partitioned and stored in NDB cluster. HopsFS partitions inodes by their parents' inode ID, resulting in inodes with the same parent inode being stored on the same database partition. For example, all the immediate children of the root(/) inode are stored in a partition on NDB datanode 1 which enables the efficient implementation of the directory listing operation. File inode related metadata, that is, blocks, replicas, and other tables are partitioned using the file's inode ID. This results in for a given file all the blocks and replicas being stored in a single database partition, again enabling efficient file operations. For example, the blocks and replicas for the file /etc/gnupg are stored on a database partition on NDB datanode 2.

inodes table stores the names of files and directories, permission attributes, file size, and ownership information. Each inode also stores a reference to the parent (*PID*) to maintain the hierarchical namespace structure. A non-empty file comprises one or more data blocks which are replicated (by default) three times on the HopsFS datanodes. The *blocks* table store information about the data blocks belonging to the different files. For example, the */etc/gnupg* file consists of two blocks, and the blocks' metadata is stored in the *blocks* table. Each block belonging to the */etc/gnupg* file is replicated on three different HopsFS datanodes for high availability. The location of the copies (replicas) of the data blocks is stored in the *replicas* table. The metadata for a file resides in different tables, and the metadata is linked together using foreign keys. For example, the *replicas* table contains a foreign key to the *inodes* table and to the *blocks* table.

HopsFS partitions *inodes* by their *parents' inode ID*, resulting in inodes with



Inodes Table

Inode ID	PID	Name	isDir	...
1	0	/	true	...
2	1	home	true	...
3	1	etc	true	...
4	1	bin	true	...
5	1	lib	true	...
6	2	john	true	...
7	2	doe	true	...
8	3	gnupg	false	...
9	3	conf	false	...
10	4	vim	false	...
11	4	firefox	false	...
12	5	libA	false	...
13	5	libB	false	...

Blocks Table

Block ID	Inode ID (FK)	Size	Index	...
1	8	64 MB	0	...
2	8	40 MB	1	...
3	9	64 MB	0	...
4	9	60 MB	1	...
5	10	14 MB	0	...
6	11	30 MB	0	...
7	12	44 MB	0	...
8	13	13 MB	0	...

Replicas Table

Block ID (FK)	Inode ID (FK)	Datanode ID (FK)	State	...
1	8	DN-0100	OK	...
1	8	DN-0030	OK	...
1	8	DN-0400	OK	...
2	8	DN-9832	OK	...
2	8	DN-9123	OK	...
2	8	DN-1837	OK	...
3	9	DN-1874	OK	...
3	9	DN-1882	OK	...
3	9	DN-0874	OK	...
4	9	DN-1882	OK	...
4	9	DN-1874	OK	...
4	9	DN-1882	OK	...
5	10	DN-0100	OK	...
5	10	DN-0030	OK	...
5	10	DN-0400	OK	...
6	11	DN-9832	OK	...
6	11	DN-9123	OK	...
6	11	DN-1837	OK	...
7	12	DN-1874	OK	...
7	12	DN-1882	OK	...
7	12	DN-0874	OK	...
8	13	DN-1882	OK	...
8	13	DN-1874	OK	...
8	13	DN-1882	OK	...

Figure 4.2: These tables show how the metadata for the namespace as shown in Figure 3.1 is stored in the database in normalized form. The inodes table stores the names of files and directories, permission attributes, file size, and ownership information. Each inode also stores a reference to the parent (PID) to build the hierarchical namespace structure. The blocks *table store information about the data blocks belonging to the different files. For example, /etc/gnupg file consists of two file blocks and the blocks' metadata is stored in the blocks table. Each block belonging to the /etc/gnupg file is replicated on three different HopsFS datanodes for high availability. The location of the copies (replicas) of the data blocks is stored in the replicas table.*

the same parent inode being stored on the same database partition. This has the effect of uniformly partitioning the metadata among all database partitions, and it enables the efficient implementation of the directory listing operation. Figure 4.1 shows how the metadata shown in Figure 4.2 is partitioned and stored in NDB. For example, the inodes for immediate children of the *root(/)* directory are stored on a database partition on NDB datanode 1. For listing the *root(/)* directory, we can retrieve all the required metadata by performing a partition pruned index scan operation on single database partition on NDB datanode 1. File inode related metadata, that is, blocks, replicas, and other tables are partitioned using the file's *inode ID*. This results in all the blocks and replicas for files being stored in a single database partition, again enabling efficient file operations, see Figure 4.1. For example, the metadata for all the blocks and replicas of the blocks for the files */etc/gnupg* and */etc/conf* are stored in database partition on NDB datanode 2 and 3 respectively. Using our partitioning scheme, the complete metadata for a file would reside on at-most two database partitions. One database partition would hold the single inode row for the file inode (partitioned by parent's inode ID) while the other database partition would hold all the file related metadata comprised of *blocks*, *replicas* and other file-related information (partitioned by inode's ID). For example, when the file */etc/gnupg* is read then inode row (file name and permission information) is read from database partition on NDB datanode 3, and the blocks and replicas are all read from single database partition on NDB datanode 2. In Figure 4.1 we have shown only two file inodes related tables, that is, *blocks* and *replicas* tables. Inodes related tables comprise the bulk of the metadata stored in the database and all of these tables are partitioned using the inode ID. These tables include *corrupt*, *deleted*, *excess*, *under-construction*, *under-replicated* blocks, *lease*, and *quota* tables.

4.1.2 Serialized File System Operations

Due to poor performance, NewSQL systems typically do not provide serializable as the default transaction isolation level, if they even support it at all [55]. Many NewSQL databases only support *read-uncommitted* and *read-committed* transaction isolation levels [56]. *Read-committed* transaction isolation guarantees that any data read is committed at the moment it is read while the *read-uncommitted* transaction isolation may read uncommitted data from other concurrent transactions. The *read-uncommitted* transaction isolation level allows *dirty*, *non-repeatable(fuzzy)*, and *phantom* reads, while the *read-committed* isolation level does not allow *dirty* reads

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read	Supported
Read Uncommitted	Permitted	Permitted	Permitted	YES
Read Committed	—	Permitted	Permitted	YES
Repeatable Read	—	—	Permitted	NO
Serializable Transactions	—	—	—	NO

Table 4.1: Transaction isolation levels supported by NDB. NDB only supports read-uncommitted and read-committed transaction isolation levels. In NDB, the default transaction isolation level is read-committed which is not strong enough to implement strongly consistent file system metadata operations as it permits non-repeatable and phantom reads.

but *non-repeatable* and *phantom* reads can happen in a transaction [50]. The default transaction isolation level in NDB is *read-committed* [56]. We now show why both the *read-uncommitted* and *read-committed* transaction isolation levels are not strong enough to build consistent file system operations. Consider a very simplified scenario where the file system namespace has only one directory that is the *root(/)* directory. Furthermore, assume two concurrent file system clients want to create the same */bin* directory. In POSIX it is not allowed that the two file system clients create two directories with the same name, that is, one of the clients should get an exception that the directory already exists. Figure 4.3 shows how using *read-committed* transaction isolation (strongest transaction isolation level supported by NDB) can cause inconsistent file system operations. Before the clients create the new directory, they check if there is already a directory with the same name, line 2 for client 1 & 2. Both clients see that the */bin* directory does not already exist and the clients then proceed to create the new */bin* directory. This way two clients can create the same directory simultaneously causing inconsistent file system operations. This would not have happened using the *serializable* transaction isolation level. However, *serializable* transaction isolation level is not supported by NDB.

Read-committed transaction isolation cannot be used to implement consistent HopsFS metadata operations. Similarly, weaker transaction isolation levels, such as *read-uncommitted* are also not suitable for implementing file system operations. However, NDB supports row-level locks, such as *exclusive* (write) locks and *shared* (read) locks that can be used to isolate conflicting transactions. HopsFS implements a pessimistic concurrency

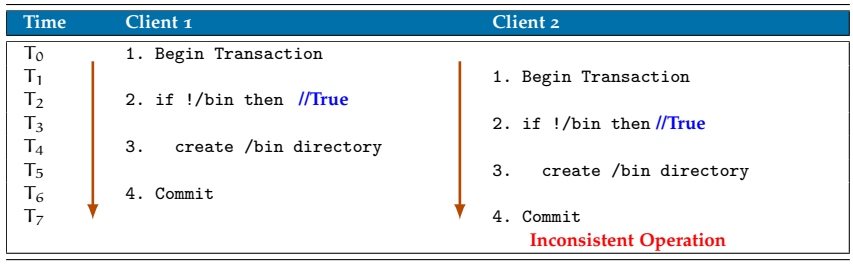


Figure 4.3: Read-committed transaction isolation can cause inconsistent file system operations when two concurrent file system operations try to create the same /bin directory. Before the clients create the new directory, they check if there is already a directory with the same name. Both clients see that the /bin directory does not exist and the clients proceed to create the /bin directory. This way two clients are able to create the directory simultaneously causing inconsistent file system operations.

model that supports parallel read and write operations on the namespace, serializing conflicting file system operations. In HopsFS read-only operations, such as reading and listing operations take *shared* locks on the metadata that is being read, while the file system operations that modify the namespace, such as creating new files/directories and adding new data blocks take *exclusive* locks on the metadata that is being updated. Figure 4.4 shows how HopsFS uses row-level locks to isolate concurrent file system clients that try to create the same directory */bin*. Before the clients create a new directory, the clients acquire an exclusive lock on the parent directory of the new directory. In this case only one of the clients, that is, client 1, manages to acquire an exclusive lock on the *root(/)* directory. Client 2 blocks and waits for the lock on *root(/)* directory. After acquiring the locks, client 1 proceeds to create the new directory and commits the transaction which also releases the locks it acquired during the transaction. After that, the client 2 would get an *exclusive* lock on the *root(/)* directory and continue. However, now the client 2 will find out that the directory */bin* already exists and it will throw an exception that the directory already exists.

4.1.3 Resolving File Paths

In hierarchical file systems, all file system metadata operations are path based. Before a metadata operation is performed the file path is resolved to make sure that the file path is valid and the user is allowed to perform the file system operation. HopsFS stores the inodes in normalized form, that is, we store individual path components not complete path name

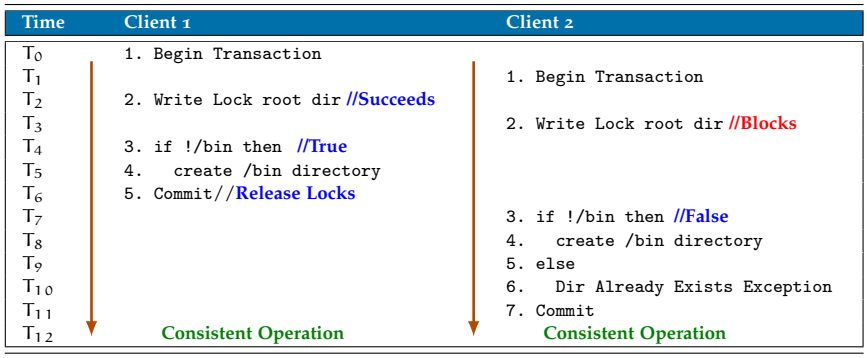


Figure 4.4: This figure shows how HopsFS uses row-level locks to isolate conflicting file system operations. Two concurrent file system operations want to create the same /bin directory. One of these file system operations should fail as in POSIX it is not permissible to create two directories with the same pathname. Before the clients create a new directory, the clients acquire an exclusive lock on the parent directory of the new directory. In this case only one of the clients, that is, client 1 manages to acquire an exclusive (write) lock on the root(/) directory. Client 2 blocks and waits for the lock on root(/) directory. After acquiring the locks, client 1 proceeds to create the new directory and commits the transaction which also releases the locks it acquired during the transaction. Then client 2 would get the exclusive lock on the root(/) directory and continue. However, now the client 2 would find out that the directory /bin already exists and it will throw an exception.

with each inode. In HopsFS for a path of depth N, it would require N roundtrips to the database to retrieve file path components, resulting in high latency for file system operations, as discussed in section 3.4.5. Similar to AFS [11] and Sprite [15], we use *hints* [57] to speed up the path lookups. Hints are mechanisms to quickly retrieve file path components in parallel (batched operations). In our partitioning scheme, inodes have a composite primary key consisting of the parent inode's ID and the name of the inode (that is, file or directory name), with the parent inode's ID acting as the partition key. Each namenode caches only the *name*, *parent inode's ID*, and *inode ID* of the inodes. Given a pathname and a hit for all path components directories, we can discover the primary keys for all the path components which are used to read the path components in parallel using a single database batch query containing only primary key read operations.

We use the inodes' hint cache entries to read the all the inodes in a pathname using a single batch query at the start of a transaction for a file system operation. If a hint entry is invalid, a primary key read operation fails and path resolution falls back to a recursive method for resolving

file path components, followed by repairing the cache. Cache entries infrequently become stale, as move and delete operations, that invalidate entries in the inodes' cache, are less than 2% of operations in typical Hadoop workloads, see industrial workload traces in Chapter 7. Moreover, typical file access patterns follow a heavy-tailed distribution (in Yahoo 3% of files account for 80% of accesses [53]) and using a *sticky* policy, for HopsFS clients, improves temporal locality and cache hit rates.

4.1.4 Transactional Metadata Operations

A transactional file system operation in HopsFS has three main stages which are discussed below with an example shown in Figure 4.5.

1. **Pre-Transaction Phase:** In the pre-transaction phase we try to resolve the file path using the inodes' hint cache. Due to the locality of reference and *sticky* client sessions with the namenodes, all or most of the file path components are resolved using the inodes' hint cache. However, this does not mean that the file path is valid or the client is allowed to access the file. The cache does not contain file or directory permissions, and it is quite possible that the cache entries are stale. For example, a client wants to read the file */etc/gnupg*. The client will first send the operation to a namenode. The namenode will try to resolve the file path locally using the inodes' hint cache. If the file is popular, then it is likely that the complete file path will be resolved locally at the namenode. This will result in the discovery of the primary keys for all the file path components. The namenode will also discover the inode ID for the *gnupg* inode. The namenode will then start a database transaction and supply the *gnupg* inode's ID as a hint to start the transaction on the NDB datanode that holds the data for the file. If the pathname resolution fails, then the transaction is started on a random NDB datanode.

2. **Metadata Retrieval Phase:** In this phase, all the metadata that is needed for the file system operation is retrieved from the database. First, the file/directory pathname will be resolved. The pathname can be resolved using a single batch of primary key operations. The namenode resolves the file path recursively if the batch operation fails or the file path is not found in the inodes' hint cache. In this case, the inodes' hint cache is updated after resolving the pathname recursively.

Then the file/directory on which the file system operation is performed is locked using row-level locks. Read-only file system operations take a *shared* lock on the inode(s) and the file system operations that update the metadata take an *exclusive* lock on the inodes that are updated by the transactions. In this case of reading the file */etc/gnupg*, a *shared* lock is acquired on the *gnupg* inode. After this, all the file inode related metadata is read from the database. In this simplified example, only the file blocks and the replicas information are needed to read the file. The majority of the metadata needed for this operation is stored on the database partition on NDB datanode 2 which is read using efficient partition pruned index scan operations. Other operations may require additional metadata which will also reside on the same partition as all the tables other than the *inodes* table are partitioned using the inode ID, see Figure 4.5 for the steps involved in a transactional file system operation.

3. **Execute and Update Phases:** All the metadata that is read in the previous phase is stored in a per-transaction cache. The inode operation is performed by processing the metadata stored in the per-transaction cache. Updated and new metadata generated during the second phase is stored in the cache which is sent to the database in batches in the final *update* phase, after which the transaction is either committed or rolled back.

4.1.5 Recursive Subtree Operations

Recursive operations on large directories, containing millions of inodes, are too large to fit in a single transaction, that is, locking millions of rows in a transaction is not supported in NDB or any other existing open-source online transaction processing system. These operations include *move*, *delete*, *change owner*, *change permissions*, and *set quota* operations. The *move* operation changes the absolute paths of all the descendant inodes, while *delete* removes all the descendant inodes, and the *set quota* operation affects how all the descendant inodes consume disk space or how many files/directories they can create. Similarly, changing the permissions or owner of a directory may invalidate operations executing at the lower subtrees.

Our solution is a protocol that implements subtree operations incrementally in batches of transactions. Instead of row-level database locks, our

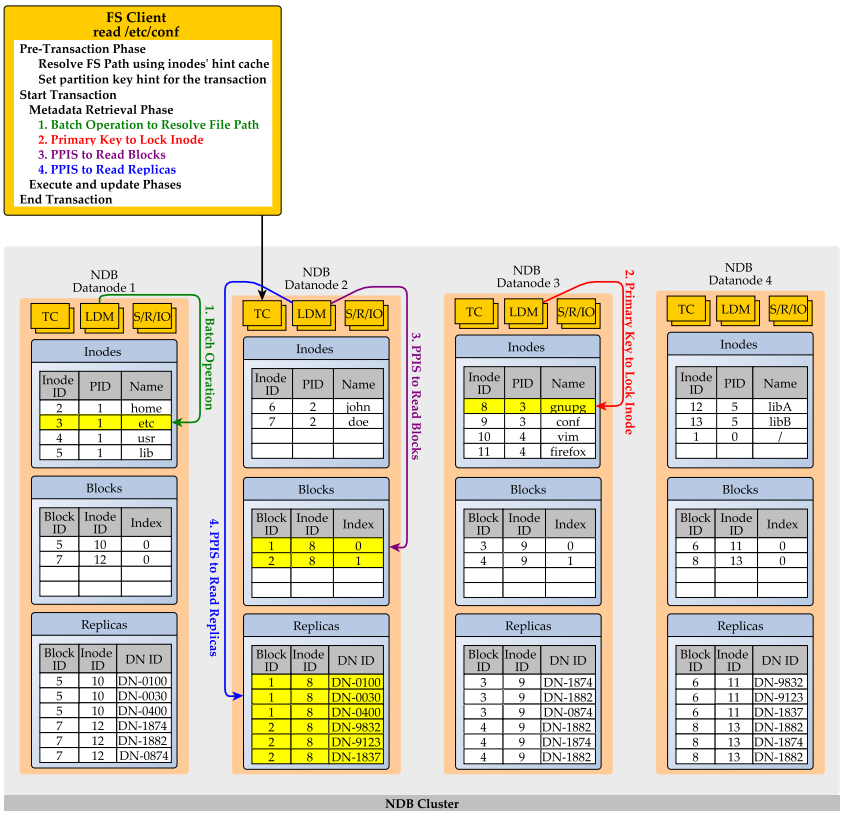


Figure 4.5: Different operations performed in a transactional file system operation for reading the file /etc/gnupg. For this file system operation ten metadata rows (highlighted in yellow color) are needed. Using the inodes' cache the file system path is resolved and the database partition that holds the metadata for the file is determined. The transaction is started on the database node that holds the metadata for the gnupg inode. 1) All the file path components upto the penultimate file path component are retrieved using a single batch operation of primary key operations. 2) Then the gnupg file inode is locked and read. 3 & 4) All the file related metadata, such as blocks and replicas is read using efficient partition pruned index scan operations. All the data that is read from the database is stored in the per-transaction cache. The file system operation is performed and all the changes in the metadata (if any) are stored back in the per-transaction cache. In the end all the changes are transferred to the database in batches and the transaction is committed.

subtree operations protocol uses an application-level distributed locking mechanism to mark and isolate the subtrees. We serialize subtree operations by ensuring that all ongoing inode and subtree operations in a subtree complete before a newly requested subtree operation is executed. We implement this serialization property by enforcing the following invariants: (1) no new operations access the subtree until the operation

completes, (2) the subtree is quiesced before the subtree operation starts, (3) no orphaned inodes or inconsistencies arise if failures occur.

Our subtree operations protocol provides the same consistency semantics as subtree operations in HDFS. For the delete subtree operation, HopsFS provides even stronger consistency semantics. Failed *delete* operations in HDFS can result in orphaned blocks that are eventually reclaimed by the block reporting subsystem (hours later). HopsFS improves the semantics of the delete operation, as failed operations do not cause any metadata inconsistencies, see section 7.6.2.

Subtree operations have the following phases

Phase 1: In the first phase, an exclusive lock is acquired on the root of the subtree, and a *subtree lock* flag (which also contains the ID of the namenode that owns the lock) is set and persisted in the database. The flag is an indication that all the descendants of the subtree are locked with exclusive (write) lock.

Before setting the lock, it is essential that there are no other *active subtree operations* at any lower level of the subtree. Setting the *subtree lock* could fail active subtree operations executing on a subset of the subtree. We store all active subtree operations in a table and query it to ensure that no subtree operations are executing at lower levels of the subtree. Checking the subtree operations table and setting the lock on the root of the subtree is done in a single transaction, similar to the mechanism discussed in section 4.1.4. In the transaction, we acquire an exclusive lock on the root inode of the subtree and then query the subtree operations table to see if there is any active subtree operation that conflicts with the new subtree operation. If there are no concurrent conflicting subtree operations then a new row containing information about the subtree operation is inserted in the subtree operations table, the lock on the subtree root is set and the transaction is committed. However, this is not enough to guarantee the isolation of subtree operations. Assume there are no subtree operations running on the namespace as shown in Figure 3.1. Two concurrent subtree operations, that is, *rm -rf /home* and *rm -rf /home/doe* are started. Both operations will acquire exclusive locks on their respective subtree root inodes and concurrently check the subtree operations table. These operations will find that no conflicting subtree operation is currently running. Both subtree operations will then set the subtree lock on the

root inodes of their subtrees, that is, *rm -rf /home* operation will set the subtree lock on the *home* inode and the *rm -rf /home/doe* operation will set the subtree lock on the *doe* inode. Then the operations will add their information to the subtree operations table and commit the transactions. This way two concurrent subtree operations may execute on the same subtree which may lead to inconsistent file system operations. To isolate subtree operation, we also take shared locks on the ancestor directories of the subtree root directory. For example, the operation *rm -rf /home/doe* will take shared locks on the *root(/)*, and *home* directory inodes and an exclusive lock on the *doe* inode. While the operation *rm -rf /home* will take a shared lock on the *root(/)* inode and then it will try to take an exclusive lock on the *home* inode. However, as the *home* inode is shared lock by the other operation, therefore, the operation will wait until the first operation completes. After the first operation completes and releases the locks, the second operation will continue and discover that there is already a conflicting subtree operation running on the same subtree. For setting the subtree lock, taking shared locks on the ancestors of the subtree root isolates conflicting subtree operations. In a typical workload, the subtree operations table does not grow too large as subtree operations are usually only a tiny fraction of all file system operations. It is important to note that during path resolution, inode and subtree operations that encounter an inode with a subtree lock turned on voluntarily abort the transaction and wait until the *subtree lock* is removed.

Phase 2: To quiesce the subtree we wait for all ongoing inode operations to complete by taking and releasing database write locks on all inodes in the subtree in the same total order used to lock inodes. To do this efficiently, a pool of threads in parallel execute partition pruned index scans that write-lock child inodes. This is repeated down the subtree to the leaves, and, a tree data structure containing the inodes in the subtree is built in memory at the namenode, see Figure 7.5. The tree is later used by some subtree operations, such as *move* and *delete* operations, to process the inodes.

Phase 3: In the last phase the file system operation is broken down into smaller operations that execute in parallel. For improved performance, large batches of inodes are manipulated in each transaction. The delete operation uses the in-memory tree to delete the subtree incrementally using parallel transactions, starting from the leaves and traversing the tree upwards. The move operation uses the in-memory tree to calculate

the size of the tree for maintaining the quota information. All subtree operations other than the delete operation can be performed using a single transaction as the amount of data that is updated is small and is not a function of the size of the subtree, that is, they only update the root of the subtree. For example, *mv* operation only updates the subtree's root. Similarly, the quota operation also only updates the *quota* values for the root of the subtree after checking that the new quota values do not violate the quota requirements for any of the descendants of the subtree.

4.1.6 Handling Failed Subtree Operations

HopsFS takes a lazy approach to cleanup subtree locks left by the failed namenodes [58]. Each namenode maintains a list of the active namenodes provided by our leader election and group membership service. If an operation encounters an inode with a subtree lock set and the namenode ID of the *subtree lock* belongs to a dead namenode, then the *subtree lock* is cleared. However, it is important that when a namenode executing a subtree operation fails, it should not leave the subtree in an inconsistent state. The in-memory tree built during the second phase plays an important role in keeping the namespace consistent if the namenode fails. For example, in case of a *delete* operation, the subtree is deleted incrementally in *post-order* tree traversal manner using transactions. If halfway through the operation the namenode fails then the inodes that were not deleted remain connected to the namespace tree. HopsFS clients will transparently resubmit the file system operation to another namenode to delete the remainder of the subtree.

Other subtree operations (*move*, *set quota*, *chmod*, and *chown*) do not cause any inconsistencies as the actual operation where the metadata is modified is done in the third phase using a single transaction that only updates the root inodes of the subtrees, and the inner inodes are left intact. In the case of a failure, the namenode might fail to unset the *subtree lock*. However, this is not a problem as other namenodes can easily remove the *subtree lock* when they find out that the *subtree lock* belongs to a dead namenode.

See Chapter 7 for further details and experimental results for HopsFS file system operations. Chapter 8 presents more detail on the techniques we used in the design of HopsFS, including the main features and configuration parameters we used in NDB to scale HopsFS. Also, we discuss different optimization for the operating system used by HopsFS and the

impact of these optimizations on the performance of HopsFS.

4.1.7 Improving the Performance of Small Files

To make HopsFS a drop-in replacement for HDFS, all file system operations protocols were implemented exactly as in HDFS. HopsFS outperformed HDFS under load due to high parallelism and scalable design of the metadata service layer. Despite having very high throughput for file system operations, the end-to-end latency of the file system operations in unloaded HopsFS and HDFS clusters were identical. Similar to HDFS, HopsFS had relatively high end-to-end latency for file system operations performed on small files due to intricate file system operations protocols that involved hopping between the database, namenodes, datanodes, and the clients to read/write small amounts of data.

We have redesigned HopsFS to introduce two file storage layers, in contrast to the single file storage service in HopsFS (and HDFS). The existing *large file storage layer* is kept as is, consisting of datanodes specialized in handling large blocks, and a new *small file storage layer* has been designed and implemented where small blocks are stored in the distributed database. The new small file storage layer is tiered where very small blocks are stored in tables that reside in-memory, while other small blocks are stored in on-disk tables in the database. Our approach benefits from the fact that HDFS is an append-only file system, so we avoid dealing with complex scenarios where small files could keep changing between large files and small files states. In our system, when a small file is appended and it becomes a large file, and it stays a large file.

Our small file storage layer is based on an inode stuffing technique that brings the small files' data blocks closer to the metadata for efficient file system operations. In HopsFS, an average file requires 1.5 KB of metadata [43] with replication for the high availability of the metadata. As a rule-of-thumb, if the size of a file is less than the size of the metadata (in our case 1 KB or less), then the data block is stored in-memory with the metadata. Other small files are stored in on-disk data tables. The latest high-performance NVMe solid-state drives are recommended for storing small files data blocks as typical workloads produce a large number of random reads/writes on disk for small amounts of data.

Inode stuffing has two main advantages. First, it simplifies the file system

operations protocol for reading/writing small files, that is, many network round trips between the client and datanodes (in the large file storage layer) are avoided, significantly reducing the expected latency for operations on small files. Second, it reduces the number of blocks that are stored on the datanodes and reduces the block reporting protocol traffic on the namenode. For example, when a client sends a request to the namenode to read a file, the namenode retrieves the file's metadata from the database. In case of a small file, the namenode also fetches the data block from the database. The namenode then returns the file's metadata along with the data block to the client. Compared to HDFS this removes the additional step of establishing a validated, secure communication channel with the datanodes (Kerberos, TLS/SSL sockets, and a block token are all required for secure client-datanode communication), resulting in lower latencies for file read operations.

Similar to reading small files, writing a small file in our system avoids many communication round trips to the datanodes for replicating the small files' blocks, as well as the time required by HDFS to set up the replication pipeline for writing the file. In HopsFS, we take advantage of the fact that, when writing files, both HDFS and HopsFS clients buffer 64 KB of data on the client side before flushing the buffer and sending the data to the datanodes. The 64 KB buffer size is a default value and can be configured, but for backward compatibility with existing HDFS clients, in HopsFS, we keep the 64 KB size buffer. The 64 KB buffer size was established experimentally by the Hadoop community as a reasonable trade-off between the needs of quickly flushing data to datanodes and optimizing network utilization by sending larger network packets.

For HopsFS, when writing a file, the client first sends an *open file* request to the namenode to allocate a new inode. The client then starts writing the file data to its local buffer. If the client closes the file before the buffer fills up completely (64 KB), then the data is sent directly to the namenode along with the close file system operation. The namenode stores the data block in the database and then closes the file. In case of a large file, the client sends an RPC request to the namenode to allocate new data blocks on the datanodes, and the client then writes the data on the newly allocated data blocks on the datanodes. After the data has been copied to all the allocated data blocks, then the client sends a *close file* request to the namenode. In HopsFS, all file system operation protocols for large files are performed the same way as in HDFS. Detailed discussion on the

solution of improving the performance of small files and the experimental results are discussed in detail in Chapter 9.

4.1.8 HopsFS Leader Election Service

Distributed file system operations such as housekeeping operations can only be performed by a single metadata server. For example, the file system operations of removing the dead nodes and recovering the data stored on the dead nodes could lead to inconsistent file system state if multiple metadata servers try to perform the same housekeeping operation. Such conflicting operations should only run on a designated (leader) metadata server, chosen using a coordination service. In partially synchronous systems, designing a leader election algorithm, that does not permit multiple leaders while the system is unstable, is a complex task. As a result, many production systems use third-party distributed coordination services, such as ZooKeeper and Chubby, to provide a reliable leader election service. However, adding a third-party service such as ZooKeeper to a distributed system incurs additional operational costs and complexity. ZooKeeper instances must be kept running on at least three machines to ensure its high availability.

We show how we avoid third-party coordination service and implement our leader election service using NewSQL databases for partially synchronous systems, which ensures at most one leader at any given time. The leader election protocol uses the database as a distributed shared memory. Our work enables distributed systems that already use NewSQL databases to save the operational overhead of managing an additional third-party service for leader election. Logically, all processes communicate through shared registers (implemented as rows in a database table). Each process stores its descriptor in the database. Process descriptor contains *ID*, *heartbeat counter*, *IP*, and *port* information. Periodically each process updates its heartbeat counter (in a transaction) to indicate that it is still alive. Each process also maintains a local history of all the processes' descriptors. Using the local history, a process is declared dead if it fails to update its counter in multiple consecutive rounds. A process declares itself to be the leader when it detects that it has the smallest *ID* among all the alive processes in the system. The leader evicts failed processes, and it is also responsible for increasing the heartbeat *round time* to accommodate slow processes.

All processes run in parallel. Concurrency control could be handled with a transaction isolation level set to *serializable*, ensuring that conflicting transactions will execute one after another. For example, if two processes, P_a and P_b, want to become leader simultaneously then the transactions will automatically be ordered such that if P_a manages to execute first, then P_b is put on hold. The transaction P_b waits until transaction P_a has finished. However, as discussed before due to poor performance, NewSQL systems typically do not provide serializable as the default transaction isolation level, if they even support it at all [55]. The strongest isolation level supported by NDB is the *read-committed* isolation level, guaranteeing that any data read is committed at the moment it is read. However, it is not sufficient for implementing a reliable leader election service. We use row-level locking to implement stronger isolation levels for transactions. Row-level locking complicates the design, but allows for more fine-grained concurrency control and thus, higher throughput. We also use lease-based mechanisms to ensure that at any given time there is at most one leader in the system. Our leader election algorithm is discussed in detail in Chapter 10.

4.1.9 Results Overview

The poor performance of HDFS has long been a bane of the Hadoop community. In this thesis, we introduced HopsFS which is an open-source, highly available file system that scales out in both capacity and throughput by adding new namenodes and database nodes. We have evaluated our system with real-world workload traces from Spotify and with experiments on a popular deep learning workload, the Open Images dataset [59], containing 9 million images (mostly small files) as well as a number of microbenchmarks. We show that for real-world workload traces HopsFS delivers 16 times higher throughput than HDFS, and HopsFS has no downtime during failover. For a more write-intensive workload, HopsFS delivers 37 times the throughput of HDFS. Our results show that for 4 KB files, HopsFS could ingest large volumes of small files at *61 times* and read 4 KB files at *4.1 times* the rate of HDFS using only six NVMe disks in the *small file storage layer*. Our solution has *7.39 times* and *3.15 times* lower operational latencies for writing and reading small files respectively for Spotify's workload traces. For files from the Open Images dataset, and a moderate-sized hardware setup, HopsFS's throughput exceeds HDFS' by *4.5 times* for reading and *5.9 times* for writing files. Further scalability is possible with larger clusters. Our architecture supports a

pluggable database storage engine, and other NewSQL databases could be used. Finally, HopsFS makes metadata tinker friendly, opening it up for users and applications to extend and analyze in new and creative ways. Evaluation of HopsFS is discussed in detail in Chapters 7, 8, 9, and 10.

5 Related Work

Earlier file systems, such as the Unix time-sharing file system [60] and BSD's fast file system [61], have popularized hierarchical namespaces for single server file systems. Implementing distributed metadata for hierarchical file systems is a challenging task as file system operations may require data from multiple metadata servers to resolve the file path and check user permissions. This increases the latency and complexity of the file system operations, as it introduces the need for agreement between the servers on how concurrent metadata operations are processed and when metadata is replicated, an agreement is needed between metadata servers on the value of the current replicated state. Distributed hierarchical file systems is a well-studied field. In this chapter, we draw upon the vast body of literature about hierarchical file systems that are most relevant to HopsFS. More specifically, here we focus on distributed metadata management, small files optimization in distributed hierarchical file systems, and coordination services used by different distributed file systems.

5.1 Client-Server Distributed File Systems

In the client-server file systems, the server provides network access to its files. Client-Server distributed file systems are also known as network attached storage (NAS) systems. NAS typically uses the existing network infrastructure and provides file-level access to the storage system. NFS [10] and CIFS [62] protocols are used to access the NAS. NFS [10] is the most popular client-server file system. The clients can mount a directory from the NFS server on their local file system, and access it as if it were a local

directory. CIFS [62] is another popular protocol to access NAS. CIFS is an open-source implementation of Server Message Block (SMB) protocol. In contrast to NFS which transparently mounts the remote directories onto the clients' local file system, CIFS makes requests to the server to access the remote file. The server receives the requests and performs the operations and return the results to the client.

NFS relaxes the file system consistency semantics as it was primarily designed as a stateless application (NFS version 3 and earlier). For example, in NFS the clients write changes to the files, without waiting for confirmation that the file blocks have been successfully written to the NFS server. This leads to better client-side file write performance. However, concurrent file read operations by multiple clients may not return the same copy of the data. Similarly, the NFS clients also cache the file system metadata for some file system operations, such as *list* and *stat* to limit the number of file system operations on the NFS server. Although the clients cache the metadata for a short period, this can still lead to application problems due to inconsistent file system operations.

In the client-server model, the central server is the scalability bottleneck as all file system operations are performed by the centralized server. Moreover, managing a client-server file system is not trivial as these file systems grow in size the administrators have to re-partition the namespace manually and mount multiple NFS servers on the client side. Also, this strategy is not practical as it does not take into account varying file system workloads.

5.2 Shared-Disk Distributed File Systems

Storage area network (SAN) is a special type of storage network system that connects computers and disk devices the same way as SCSI cables connect disk drives to a computer. SAN typically requires a dedicated storage data network infrastructure. SAN is accessed using SCSI over TCP/IP, InfiniBand, ATA over Ethernet (AoE), Fiber Channel Protocol (FCP) or similar protocols. Storage area Networks (SAN) only provide block-level access to the storage disks, and the management of the file system typically resides with the file system clients, that is, the clients implement the shared file system.

GPFS, Farangipani, XFS, GlobalFS, StorageTank and Panasas [6–9, 63, 64]

are SAN based distributed file systems that store the file systems metadata on shared-disks. Multiple nodes can access the metadata stored on the shared-disks. Locking mechanisms are needed to support a single-node equivalent POSIX compliant file system semantics for the file system operations. In the case of centralized lock management, all conflicting file system operations are sent to a designated node, which performs the requested file system operation. While in the case of distributed locking each file system operation acquires appropriate locks on the inodes and their corresponding blocks to serialize conflicting operations. Distributed locking scales better, however, in the presence of frequent conflicting operations on a subset of inodes the overhead of distributed locking may exceed the cost of sending the conflicting operations to a dedicated node.

5.3 Monolithic Metadata Servers

Many distributed file system, such as GFS, HDFS, and QFS use monolithic metadata servers that store and process the entire metadata [2, 3, 6–9, 18]. Although monolithic metadata servers are easier to implement, they limit the scalability and performance of the entire distributed file system. For example, in GFS the metadata is stored on a single metadata server, and the contention on the metadata server is alleviated using read-only replicas of the metadata server.

5.4 Subtree Metadata Partitioning

The file system namespace can be statically/dynamically split into multiple sub-trees and assigned to the metadata servers for high-performance and better scalability.

5.4.1 Static Subtree Metadata Partitioning

Distributed file systems such as AFS, NFS, LOCUS, Coda, MapR, Sprite, and federated HDFS [10–15, 65] statically partition the namespace using directory sub-trees (also called volumes). Subtree partitioning has an advantage that metadata objects within a directory subtree are located on a single metadata server. However, this technique fails to divide the workload evenly among the metadata servers. Typically, these sub-trees are treated as independent structures, and the subtree boundaries are visible to the file system users due to lack of support for file system operations, such as *rename* and *link* that cross multiple subtree partitions. Another

problem with this approach is that it often requires an administrator effort to decide how the file system would be partitioned. As different file system sub-trees grow at a different rate, it is difficult to partition the namespace optimally. The unpredictable growth of the namespace may require repartitioning the sub-trees. If the file system workload is not evenly balanced across all the metadata servers, then the hotspots (frequently accessed files and directories) can overload the metadata servers storing the popular files and directories.

5.4.2 Dynamic Subtree Metadata Partitioning

When the metadata is stored on multiple metadata servers, it is possible that the file system would not be balanced across the metadata servers. Moreover, distributed transactions are needed to perform file system operations. Ursa Minor, Farsite and CephFS [4, 5, 66–69] use dynamic subtree partitioning to distribute the file system namespace across multiple metadata servers. As the file system workload changes or the number of metadata servers change, the file system sub-trees are dynamically reassigned to the metadata servers. In the case of Ursa Minor, the main objective of dynamic partitioning is to avoid distributed transactions. Whenever it encounters an atomic operation that spans multiple partitions, it repartitions the metadata by collocating the metadata that needs to be atomically updated [5]. The directory service in Farsite dynamically partitions the sub-trees according to the file identifiers which are immutable and have a tree structure. Atomic rename operations are implemented using two-phase locking. The server that manages the destination directory acts as the leader and the servers that hold the source files act as followers. Similar to Farsite, CephFS also dynamically partitions the file system tree and replicates hotspot directories on multiple metadata servers for better performance [68, 69].

5.5 Hash-Based Metadata Distribution

File systems like GlusterFS [70] Vesta [71], Lustre [72], InterMezzo [73], and RAMA [74] use hash-based techniques to distribute file system metadata. Partial or complete file paths are hashed to locate the metadata servers storing the file information. Using hash-based metadata distribution it is easier for the file system clients to locate the metadata. For a well-balanced file system namespace tree, the metadata is evenly distributed across the metadata servers. However, expanding the metadata service

could be expensive as metadata has to be relocated because of the changes in the output of the hash function. Using the file paths for hashing loses the metadata locality for file system operations, such as directory listing. Moreover, renaming large directories could be very expensive as all the dependent files and directories have to be updated and relocated to the new metadata servers. Some distributed file systems use partial file paths, such as using the file path up to the penultimate file path component to hash the metadata to preserve the metadata locality for file listing operations.

HopsFS also relies on a hash-based metadata partitioning to distribute the metadata across the database partitions. In HopsFS, we store only one path component with each inode, and we hash the metadata using the *immutable* inode ID. For example, the files are partitioned using the parent inode's ID, which stores all the immediate descendants of the directory on the same database partition for efficient directory listing. While the file metadata (such as the blocks and replicas information) is partitioned using the file inode ID to read the metadata efficiently for file read operations. Using the inode ID to partition the metadata also makes it possible to implement in-place rename operations. Renaming (moving) directories in HopsFS only changes the parent directory ID of the root of the directory subtree. Thus, only one inode is moved between the database partitions.

One of the most significant limitations of hash-base metadata distribution is that the file path permission checks are very slow as the clients have to contact multiple metadata servers to obtain the metadata for the individual file path components. Lazy Hybrid (LH) [75] tries to solve this issue by using a dual-entry access control list structure that merges the net effect of the permission check into each file metadata which allows file permission to be determined directly without traversing the entire path. LH also uses complete file paths to hash the metadata. Inspired by peer-to-peer file sharing applications file systems like CFS [76] and PAST [77] use scalable distributed hash tables to distribute file system metadata that are based on Chord [78] and PASTRY [79].

5.6 Using a Database to Store Metadata

For years researchers have tried to use databases to store file system metadata as database transactions would provide simple and correct handling of file system metadata operations. Earlier systems used local databases or

used transactions provided by the operating system to implement the file system metadata services [21–23]. The conventional wisdom has been that it is too expensive (in terms of throughput and latency) to store hierarchical file system metadata in an external database [17,80]. For example, file systems such as WinFS [24] and InversionFS [25] that store the metadata in an external database have poor performance.

Recently, high performance distributed databases such as HBase [31,81], Cassandra [82], and CalvinDB [55] have enabled the development of new distributed metadata management architectures in file systems. Unlike HopsFS which stores metadata in the external database in normalized form, all of these file systems store the file system metadata in denormalized form. GiraffaFS [83] is Hadoop compatible distributed file system stores the file system metadata in HBase in denormalized form, that is, the identifier for each inode is byte array representing full path of the file. For example, a file, */home/user/file.txt* is stored in the HBase as a single row identified by a row key that is the file full path name. The metadata is sorted using a *lexicographical* order to ensures locality of reference in HBase, that is, the immediate children of a directory will be stored in the same database partition. This strategy is very similar to HopsFS. We partition the immediate children of a directory based on the parent inode ID, which puts all the immediate children of the directory on the same database partition. However, HopsFS stores fully normalized metadata. GiraffaFS does not support in-place rename operations, and a rename operation changes the row identifiers of all the children of the renamed directory.

CassandraFS [84] is another Hadoop compatible distributed file system that stores the metadata in Cassandra key-value store. The entire metadata is modeled with two column families in Cassandra that represent two distinct metadata services. The first column family stores the file/directory inodes, and the second column family stores the block locations. Like GiraffaFS, CassandraFS also stores the metadata in denormalized form, and it does not support in-place rename operations.

CalvinFS partitions the metadata by hashing the complete directory/file path names and stores it in the Calvin database – a log replicated distributed transactional key-value store [55,85]. CalvinFS avoids multiple network round-trips to the database to resolve the file path by storing the entire path information along with all permissions attributes with each

metadata entry. This metadata partitioning approach requires CalvinFS to update all the descendants of a subtree for all subtree operations, making it prohibitively expensive. CavlinFS relies on CalvinDB to perform large file system operations atomically, such as renaming and deleting large directories. CalvinDB runs large transactions in two phases. In the first phase the lock set is identified, and in the second phase all the locks are acquired, and the operation is performed, provided that the lock set has not changed. However, it is not clear whether CalvinFS' proposed use of multiphase optimistic locking to perform directory operations is viable in current OLTP databases, as potentially millions of locks may be acquired to update the metadata atomically.

Wave file system [86] is another file system that uses HyperDex [87], a transactional key-value store, for storing the file system metadata. It also avoids file path traversal by storing complete file paths with the inodes. As a side effect, the Wave file system does not support permission checks on the full path from the root. For file system operations, such as, listing directories which require a scan operation, the wave file system maintains special files that store the children of the directories which are atomically updated using HyperDex transactions.

5.7 Improving the Performance of Small Files

Walnut [88], from Yahoo! in 2012, described a hybrid storage system that stores large files in a file system and small files in a Log-Structured Merge-Tree (LSM-tree) database, BLSM [89]. They identified an object size threshold of 1 MB for SSD storage, where objects under 1 MB in size could be stored with higher throughput and lower latency in the database, while objects larger than 1 MB were more efficient to store in a file system. Although they chose 1 MB as the crossover region, the results showed that between 100 KB and 1 MB, there was no clear winner.

Although we use NDB cluster to store stuffed inodes as on-disk columns in tables, WiscKey [90] recently showed how separating the storage of keys from values in an LSM-tree database can help improve throughput and reduce latency for YCSB workloads on SSDs. This tells us there is still significant potential for performance improvements when using SSDs for disk-based columns in NDB cluster.

File systems like HDFS and GFS store the data block on the datanodes

as files. The files are managed by local file systems such as Ext4, ZFS, XFS, and Btrfs. These local file systems often provide functionalities, such as erasure coding, journaling, encryption, and hierarchical file system namespace, that may not be directly required by the distributed file systems. For small files, the overhead introduced by the local file systems is considerable compared to the time required to actually read/write the small files. In distributed file systems these features, such as encryption, replication, erasure coding, etc., are provided by the distributed metadata management system. The iFlatLFS [91] improves the performance of the handling of the small files by optimally storing the small files on the disk of the datanodes using a simplified local file system called the iFlatLFS. The iFlatLFS is a local file system installed on all the datanodes that manage the small files stored on the disk. TableFS [92] has shown that better performance can be achieved if the metadata and the file data is stored in a local key-value store such as LevelDB [93], however, TableFS is a not a distributed file system. James Hendricks et al. has shown that the performance of small files can be improved by reducing the interactions between the clients and the metadata servers, and by using caching and prefetching techniques [94].

HDFS provides an archiving facility, known as Hadoop Archives (HAR), that compresses and stores small files in large archives as a solution to reduce the contention on the namenode caused by the small files, see section 9.3.4 for more details. Similar to HAR, Xuhui Liu et al. group the small files by *relevance* and combines them into a large file to reduce the metadata overhead. It creates a hash index to access the contents of the small files stored in a large file [95]. MapR is a proprietary distributed file system that stores first 64 KB of all the files with the metadata [12], which improves the performance of small files.

In industry, many companies handle different client requirements for fast access to read/data by using multiple scale-out storage services. Typically, this means using a NoSQL database, such as Cassandra or HBase, for fast reading/writing data, as done by Uber [96], while an archival file system, such as HDFS, is used for long-term storage of data. This approach, however, complicates application development, as applications need to be aware of where data is located. In contrast, our small file storage layer solution ensures that HopsFS clients are unaware of whether a file is stored in the database or on a HopsFS datanode.

Finally, one advantage of using NDB cluster is that, because its updates are made in-place, it has lower write-amplification than LSM-tree databases [31, 97], which can improve SSD device lifetime.

Finally, alternative in-memory storage engines to NDB could be investigated. MemSQL supports high throughput transactions, application-defined partitioning, and partition pruned queries [98]. However, VoltDB is not currently a candidate as it serializes cross-partition transactions [99].

5.8 Coordination Services In Distributed File Systems

Consensus algorithms are a core building block of distributed systems. Many commercial distributed storage systems use third-party services like ZooKeeper [27], *etcd* [100, 101], and Chubby [102] to provide consensus among the nodes for different parts of the storage system. In many cases, these third-party solutions can be avoided by exploiting the properties, such as locking and transactions that are often supported by the underlying storage systems.

CassandraFS [84] uses *Phi* Accrual Failure Detector [103] to detect node failures [104]. Cassandra takes in to account the network performance, workload, and application data to dynamically adjusts the *Phi* threshold for each node. Cassandra also supports lightweight transactions that are restricted to single partitions using the Paxos algorithm [105] which can also be used to implement a leader election algorithm [106]. DynamoDB [107] also supports locking and transactions that can be used to implement a lease-based leader election algorithm [108].

CephFS [109] uses a few monitor nodes that provide group membership services and maintain the master copy of the cluster map. The monitors' cluster uses Paxos [110] to reach an agreement on which object storage devices are active. One of the monitors node is elected as a leader, which ensures that the monitor nodes are active and all the monitor nodes have the most recent map *epoch*. The leader monitor also issues short-term leases to other monitors. Leases grant permissions to the active monitors to distributed their copy of the cluster map to the OSDs as long as their lease is valid.

Chubby [102] is a distributed lock service developed at Google that provides coarse-grained locking and reliable storage for the loosely-coupled distributed systems. At Google, Chubby is used by many internal services

including the Google File System (GFS) [18]. ZooKeeper is an open-source alternative for the Google's Chubby distributed locking service. HDFS uses both the ZooKeeper and a heartbeat-based mechanism to detect failed nodes. ZooKeeper is used to detect namenodes failures and to switch from active to standby namenodes reliably. The datanodes in HDFS send heartbeat messages to the namenodes to indicate that the datanodes are alive. Similar to HDFS, the datanodes in HopsFS send heartbeats to the namenodes, however, instead of using ZooKeeper the namenodes use the database as a shared memory to implement a leader election and a group membership service for the namenodes. Other distributed file systems that use ZooKeeper are Apache Ignite file system [111], and MapR [12]

Some notable leader election protocols in the message passing paradigm using timers are [112–114]. In these protocols, the processes send messages to each other to indicate they are alive. A process is suspected if it fails to send a heartbeat message within a time bound. If a heartbeat is received from a suspected process the timer is increased to accommodate slow processes. Eventually, time bounds for processing and communication latencies are determined for the given system by successively increasing the timer upon receiving a message from a suspected process.

6

Conclusions and Future Work

We believe that distributed metadata in a commodity database is a significant new enabling technology and it can become a reliable source of ground truth for metadata applications built on top of distributed file systems. In this thesis, we introduced HopsFS, that is, to the best of our knowledge, the first open-source production-grade distributed hierarchical file system that stores its metadata in an external NewSQL relational database in a normalized form. We show how to build a highly available file system that scales out in both capacity and throughput by leveraging traditional database techniques as well as modern NewSQL techniques, such as, data distribution aware transactions and partition pruned index scan operations. We also show how the performance of small files can be improved by using a tiered file storage solution which naturally matches the storage hierarchy typically seen on servers, where small fast data is stored in-memory, and larger frequently accessed files are stored on SSDs, and the biggest files are stored on spinning disks. Lastly, we show how to build a leader election service using NewSQL databases, which simplifies the administration of the distributed file system as it does not have to rely on third-party coordination services for leader election and group membership services.

Storing the file system metadata in an external database has opened up many new research and development opportunities. HopsFS enables online ad-hoc analytics on the metadata using SQL. HopsFS metadata can be selectively and asynchronously replicated to either a backup cluster or a MySQL slave server, enabling complex analytics without affecting the

performance of the active cluster.

The metadata architecture of HopsFS makes it easier to implement geo-replicated distributed hierarchical file systems for the cloud that spans multiple data centers in different *availability zones* and *regions* to provide high availability in the eventuality of a data center failure. A region is a geographically independent set of data centers that consists of multiple availability zones connected through low-latency links with sub-millisecond inter-availability zone network latencies. The inter-region network latencies vary between 25 to 500 milliseconds [115]. We are building the next version of HopsFS which can be deployed across multiple data centers. With very few changes, HopsFS can be deployed across two availability zones in the same region, such that, the database nodes, namenodes, and the datanodes are split across the two data centers, and still provide low latency file system operations as the inter-availability zone latencies are very low. In such a setup, the NDB nodes in each node-group will be split across the two availability zones, and the data replication policy for the file system is modified such as there is at least one replica of each block in the two availability zones. HopsFS can scale to more than two availability zones if the replication factor of the database (default 2) is increased. However, this setup would not work across data centers in different zones due to high network latencies. In order to enable HopsFS across data centers across multiple regions, we propose that each data center will have its own complete database setup and the namenodes would perform the file system operations using the local database in the data center. The changes in the metadata will be asynchronously replicated across the data centers using MySQL asynchronous replication [116] and the conflicting file system operations will be detected and resolved by the namenodes [117].

We are also looking to making HopsFS cloud-native, by enabling blocks in the file system be stored on external cloud-storage, such as the S3 block-store [118]. HopsFS will treat the blocks stored in external block-stores the same as the blocks stored on HopsFS datanodes. This introduces problems of ensuring the availability, consistency, and integrity of block data stored in external block-stores. Currently, HopsFS has a periodic synchronization protocol that ensures that the blocks stored at datanodes are kept consistent with the metadata describing the availability and location of the blocks. Supporting external block-stores would require extending the blocks' metadata to describe which blocks are stored in

external block-stores, designing a new periodic block synchronization protocol that also ensures that the blocks stored in the external block-stores are not tampered with unknowingly, and implementing a solution that has limited impact on the throughput and latency of HopsFS. An additional constraint is to design a solution that is transparent to existing HDFS clients.

Part II

Publications

7

HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases

Salman Niazi, Mahmoud Ismail, Seif Haridi,
Jim Dowling, Mikael Ronström, Steffen Grohsschmiedt

In 15th USENIX Conference on File and Storage Technologies, Santa Clara, California, USA, 2017.

8

Scaling HDFS to more than 1 million operations per second with HopsFS

Mahmoud Ismail, **Salman Niazi**, Seif Haridi,
Jim Dowling, Mikael Ronström

In proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '17, Madrid, Spain.

9

Size Matters: Improving Small Files' Performance in HDFS

Salman Niazi, Seif Haridi, Jim Dowling, Mikael Ronström

ACM/IFIP/USENIX 19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.

10

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

Bibliography

[1] P. A. S. C. of the IEEE Computer Society and T. O. Group, "Draft Standard for Information Technology Portable Operating System Interface (POSIX)." `http://www.open-std.org/jtc1/sc22/open/n4217.pdf/`, 2007. [Online; accessed 26-February-2018].

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2010.

[3] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," *Proc. VLDB Endow.*, vol. 6, pp. 1092–1101, Aug. 2013.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 1–14, Dec. 2002.

[5] M. Abd-El-Malek, W. V. Courtright, II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie, "Ursa Minor: Versatile Cluster-based Storage," in *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2005.

[6] Schmuck, Frank and Haskin, Roger, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, (Berkeley, CA, USA), USENIX Association, 2002.

[7] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, (New York, NY, USA), pp. 224–237, ACM, 1997.

[8] K. W. Preslan, A. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, and et al., "Implementing Journaling in a Linux Shared Disk File System," 2000.

[9] Welch, Brent and Unangst, Marc and Abbasi, Zainul and Gibson, Garth and Mueller, Brian and Small, Jason and Zelenka, Jim and Zhou, Bin, "Scalable Performance of the Panasas Parallel File System," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, (Berkeley, CA, USA), USENIX Association, 2008.

[10] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3 - Design and Implementation," in *In Proceedings of the Summer USENIX Conference*, pp. 137–152, 1994.

[11] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Commun. ACM*, vol. 29, pp. 184–201, Mar. 1986.

[12] M. Srivas, P. Ravindra, U. Saradhi, A. Pande, C. Sanapala, L. Renu, S. Kavacheri, A. Hadke, and V. Vellanki, "Map-Reduce Ready Distributed File System," 2011. US Patent App. 13/162,439.

[13] G. Popek and B. J. Walker, *The LOCUS distributed system architecture*. MIT Press, 1985.

[14] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, and C. Steere, "Coda: A Highly available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, vol. 39, pp. 447–459, 1990.

[15] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The sprite network operating system," *IEEE Computer*, vol. 21, pp. 23–36, 1988.

[16] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The XtreemFS architecturea case for object-based file systems in Grids," *Concurrency and computation: Practice and experience*, vol. 20, no. 17, pp. 2049–2060, 2008.

[17] M. Seltzer and N. Murphy, "Hierarchical File Systems Are Dead," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2009.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29–43, Oct. 2003.

[19] D. R. Jeske and X. Zhang, "Some successful approaches to software reliability modeling in industry," *The Journal of Systems and Software*, vol. 74, no. 1, pp. 85–99, 2005.

[20] K. Ren, *Fast Storage for File System Metadata*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2017.

[21] M. I. Seltzer and M. Stonebraker, "Transaction support in read optimizied and write optimized file systems," in *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, (San Francisco, CA, USA), pp. 174–185, Morgan Kaufmann Publishers Inc., 1990.

[22] B. Liskov and R. Rodrigues, "Transactional file systems can be fast," in *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, (New York, NY, USA), ACM, 2004.

[23] L. F. Cabrera and J. Wyllie, "Quicksilver distributed file services: an architecture for horizontal growth," in *[1988] Proceedings. 2nd IEEE Conference on Computer Workstations*, pp. 23–37, Mar 1988.

[24] "WinFS: Windows Future Storage." `https://en.wikipedia.org/wiki/WinFS`. [Online; accessed 30-June-2015].

[25] M. A. Olson and M. A, "The Design and Implementation of the Inversion File System," 1993.

[26] M. Zait and B. Dageville, "Method and mechanism for database partitioning," Aug. 16 2005. US Patent 6,931,390.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pp. 11–11, 2010.

[28] A. Foundation, "Apache Hadoop." `https://hadoop.apache.org/`. [Online; accessed 30-Aug-2015].

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 1099–1110, ACM, 2008.

[30] Thusoo, Ashish and Sarma, Joydeep Sen and Jain, Namit and Shao, Zheng and Chakka, Prasad and Anthony, Suresh and Liu, Hao and Wyckoff, Pete and Murthy, Raghotham, "Hive: A Warehousing Solution over a Map-reduce Framework," *Proc. VLDB Endow.*, vol. 2, pp. 1626–1629, Aug. 2009.

[31] L. George, *HBase: The Definitive Guide*. Definitive Guide Series, O'Reilly Media, Incorporated, 2011.

[32] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[33] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1357–1369, ACM, 2015.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[35] "Apache Giraph." `http://giraph.apache.org/literature.html`. [Online; accessed 30-June-2017].

[36] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

[37] T. Wolpe, "Hortonworks founder: YARN is Hadoop's datacentre OS." https://www.zdnet.com/article/hortonworks-founder-yarn-is-hadoops-datacentre-os/. [Online; accessed 26-February-2018].

[38] J. Yates, "Support more than 2 NameNodes." https://issues.apache.org/jira/browse/HDFS-6440. [Online; accessed 26-February-2018].

[39] S. Pook, "Pilot Hadoop Towards 2500 Nodes and Cluster Redundancy." http://events.linuxfoundation.org/sites/events/files/slides/Pook-Pilot%20Hadoop%20Towards%202500%20Nodes%20and%20Cluster%20Redundancy.pdf. [Apache Big Data, Miami, 2017. Online; accessed 28-Sep-2017].

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

[41] "The Curse of the Singletons The Vertical Scalability of Hadoop NameNode." http://hadoopblog.blogspot.se/2010/04/curse-of-singletons-vertical.html. [Online; accessed 30-Aug-2015].

[42] "Improve namenode scalability by splitting the FSNamesystem synchronized section in a read/write lock." https://issues.apache.org/jira/browse/HDFS-1093. [Online; accessed 30-Aug-2015].

[43] 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.

[44] "RPC Congestion Control with FairCallQueue." https://issues.apache.org/jira/browse/HADOOP-9640. [Online; accessed 1-January-2016].

[45] "Quorum-Journal Design." https://issues.apache.org/jira/browse/HDFS-3077. [Online; accessed 30-Aug-2015].

[46] F. Özcan, N. Tatbul, D. J. Abadi, M. Kornacker, C. Mohan, K. Ramasamy, and J. Wiener, "Are We Experiencing a Big Data Bubble?," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 1407–1408, 2014.

[47] "MySQL Cluster Benchmarks." http://www.mysql.com/why-mysql/benchmarks/mysql-cluster/. [Online; accessed 30-June-2015].

[48] A. Bonds, "Hash-based Eventual Consistency to Scale the HDFS Block Report." Masters thesis at KTH (TRITA-ICT-EX ; 2017:150), http://kth.diva-portal.org/smash/get/diva2:1181053/FULLTEXT01.pdf, 2017.

[49] E. Brewer, "Pushing the cap: Strategies for consistency and availability," *Computer*, vol. 45, pp. 23–29, Feb. 2012.

[50] 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.

[51] "HOPS, Software-As-A-Service from SICSS new datacenter." https://www.swedishict.se/hops-software-as-a-service-from-sicss-new-datacenter. [Online; accessed 23-May-2016].

[52] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 463–492, July 1990.

[53] C. L. Abad, *Big Data Storage Workload Characterization, Modeling and Synthetic Generation*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

[54] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 237–248, IEEE Press, 2014.

[55] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast Distributed Transactions for Partitioned Database Systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 1–12, ACM, 2012.

[56] M. Ronström, *MySQL Cluster 7.5 inside and out*. O'Reilly Media, Incorporated, 2018.

[57] B. W. Lampson, "Hints for computer system design," *SIGOPS Oper. Syst. Rev.*, vol. 17, pp. 33–48, Oct. 1983.

[58] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[59] I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy, "Openimages: A public dataset for large-scale multi-label and multi-class image classification.," *Dataset available from https://github.com/openimages*, 2017.

[60] D. M. Ritchie and K. Thompson, "The unix time-sharing system," *Commun. ACM*, vol. 17, pp. 365–375, July 1974.

[61] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix," *ACM Trans. Comput. Syst.*, vol. 2, pp. 181–197, Aug. 1984.

[62] "CIFS VFS - Advanced Common Internet File System for Linux ." `https://linux-cifs.samba.org/`. [Online; accessed 30-Jan-2018].

[63] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless network file systems," in *ACM TRANSACTIONS ON COMPUTER SYSTEMS*, pp. 109–126, 1995.

[64] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "Ibm storage tank: A heterogeneous scalable san file system," *IBM Systems Journal*, vol. 42, no. 2, pp. 250–267, 2003.

[65] "HDFS Federation." `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html`, 2017.

[66] J. R. Douceur and J. Howell, "Distributed Directory Service in the Farsite File System," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 321–334, USENIX Association, 2006.

[67] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger, "A transparently-scalable metadata service for the ursa minor storage system," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2010.

[68] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 307–320, USENIX Association, 2006.

[69] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, (Washington, DC, USA), pp. 4–, IEEE Computer Society, 2004.

[70] "Docs - Getting started with GlusterFS - Architecture." `http://gluster.readthedocs.org/en/latest/Quick-Start-Guide/Architecture/`, 2011. [Online; accessed 30-June-2015].

[71] P. Corbett, D. Feitelson, J.-P. Prost, and S. Baylor, "Parallel access to files in the Vesta file system," in *Supercomputing '93. Proceedings*, pp. 472–481, Nov 1993.

[72] "The Lustre Storage Architecture." `http://wiki.lustre.org/manual/LustreManual20_HTML/UnderstandingLustre.html`. [Online; accessed 30-Aug-2015].

[73] Peter Braam Braam and Michael Callahan, "The InterMezzo File System," in *In Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*, (Monterey, CA, USA), 1999.

[74] E. L. Miller and R. Katz, "RAMA: An Easy-To-Use, High-Performance Parallel File System," *Parallel Computing*, vol. 23, pp. 419–446, July 1997.

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

[76] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, (New York, NY, USA), pp. 202–215, ACM, 2001.

[77] A. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 188–201, Oct. 2001.

[78] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[79] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, (London, UK, UK), pp. 329–350, Springer-Verlag, 2001.

[80] E. Levy and A. Silberschatz, "Distributed file systems: Concepts and examples," *ACM Computing Surveys*, vol. 22, pp. 321–374, 1990.

[81] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[82] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.

[83] Y. C. Konstantin V. Shvachko, "Scaling Namespace Operations with Giraffa File System," *login: The Magazine of USENIX*, vol. 42, no. 2, pp. 27–30, 2017.

[84] "Cassandra File System Design." `http://www.datastax.com/dev/blog/cassandra-file-system-design`. [Online; accessed 1-January-2016].

[85] A. Thomson and D. J. Abadi, "CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, (Santa Clara, CA), pp. 1–14, USENIX Association, Feb. 2015.

[86] R. Escriva and E. G. Sirer, "The design and implementation of the wave transactional filesystem," *CoRR*, vol. abs/1509.07821, 2015.

[87] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, (New York, NY, USA), pp. 25–36, ACM, 2012.

[88] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, "Walnut: A unified cloud object store," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 743–754, ACM, 2012.

[89] R. Sears and R. Ramakrishnan, "blsm: A general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 217–228, ACM, 2012.

[90] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, p. 5, 2017.

[91] S. Fu, L. He, C. Huang, X. Liao, and K. Li, "Performance optimization for managing massive numbers of small files in distributed file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 3433–3448, Dec 2015.

[92] K. Ren and G. Gibson, "TABLEFS: Enhancing Metadata Efficiency in the Local File System," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 145–156, USENIX, 2013.

[93] "LevelDB." http://leveldb.org/. [Online; accessed 1-January-2016].

[94] J. Hendricks, R. R. Sambasivan, S. Sinnamohideen, and G. R. Ganger, "Improving small file performance in object-based storage." http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-06-104.pdf, 2006.

[95] X. Liu, J. Han, Y. Zhong, C. Han, and X. He, "Implementing webgis on hadoop: A case study of improving small file i/o performance on hdfs," in *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–8, Aug 2009.

[96] L. E. Li, E. Chen, J. Hermann, P. Zhang, and L. Wang, "Scaling machine learning as a service," in *Proceedings of The 3rd International Conference on Predictive Applications and APIs* (C. Hardgrove, L. Dorard, K. Thompson, and F. Douetteau, eds.), vol. 67 of *Proceedings of Machine Learning Research*, (Microsoft NERD, Boston, USA), pp. 14–29, PMLR, 11–12 Oct 2017.

[97] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb.," in *CIDR*, 2017.

[98] "MemSQL, The World's Fastest Database, In Memory Database, Column Store Database." http://www.memsql.com/. [Online; accessed 30-June-2015].

[99] "VoltDB Documentation." http://docs.voltdb.com/ReleaseNotes/. [Online; accessed 30-June-2015].

[100] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 305–320, USENIX Association, 2014.

[101] "Distributed reliable key-value store for the most critical data of a distributed system ." https://coreos.com/etcd/docs/latest/. [Online; accessed 30-March-2018].

[102] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 335–350, USENIX Association, 2006.

[103] Naohiro Hayashibara, Xavier Défago, Rami Yared, Takuya Katayama, "The accrual failure detector," in *Reliable Distributed Systems, IEEE Symposium on(SRDS)*, vol. 00, pp. 66–78, 10 2004.

[104] "Cassandra WiKi: Architecture Internals." `https://wiki.apache.org/cassandra/ArchitectureInternals`. [Online; accessed 30-March-2017].

[105] "Lightweight transactions in Cassandra 2.0." `https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0`. [Online; accessed 30-March-2017].

[106] "Consensus on Cassandra." `https://www.datastax.com/dev/blog/consensus-on-cassandra`. [Online; accessed 30-March-2017].

[107] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[108] "Building Distributed Locks with the DynamoDB Lock Client." `https://aws.amazon.com/blogs/database/building-distributed-locks-with-the-dynamodb-lock-client/`. [Online; accessed 30-March-2018].

[109] S. A. Weil, *Ceph: Reliable, Scalable, and High-Performance Distributed Storage*. PhD thesis, University of California Santa Cruz, 2007.

[110] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[111] "Apache Ignite: Database and Caching Platfom ." `https://ignite.apache.org/`. [Online; accessed 30-March-2018].

[112] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "On implementing omega in systems with weak reliability and synchrony assumptions," *Distributed Computing*, vol. 21, no. 4, pp. 285–314, 2008.

[113] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Communication-efficient Leader Election and Consensus with Limited Link Synchrony," in *Proceedings of the Twenty-third Annual ACM*

Symposium on Principles of Distributed Computing, PODC '04, (New York, NY, USA), pp. 328–337, ACM, 2004.

[114] M. Larrea, A. Fernandez, S. Arevalo, J. Carlos, and J. Carlos, "Optimal Implementation of the Weakest Failure Detector for Solving Consensus," pp. 52–59, IEEE Computer Society Press.

[115] "AWS Inter-Region Latency." `https://www.cloudping.co/`. [Online; accessed 30-March-2017].

[116] "MySQL 5.0 Reference Manual :: 16 Replication." `http://dev.mysql.com/doc/refman/5.0/en/replication.html`. [Online; accessed 30-June-2015].

[117] R. Bampi, "LenticularFS: scalable hierarchical filesystem for the cloud." Masters thesis at KTH (TRITA-ICT-EX ; 2017:147), `http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1164146&dswid=6704`, 2017.

[118] " Amazon S3 Object storage built to store and retrieve any amount of data from anywhere ." `https://aws.amazon.com/s3/`. [Online; accessed 26-February-2018].

[119] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), pp. 59–72, ACM, 2007.

[120] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere Platform for Big Data Analytics," *The VLDB Journal*, vol. 23, pp. 939–964, Dec. 2014.

[121] E. Corporation, "HADOOP IN THE LIFE SCIENCES:An Introduction." `https://www.emc.com/collateral/software/white-papers/h10574-wp-isilon-hadoop-in-lifesci.pdf`, 2012. [Online; accessed 30-Aug-2015].

[122] A. J. Peters and L. Janyst, "Exabyte Scale Storage at CERN," *Journal of Physics: Conference Series*, vol. 331, no. 5, p. 052015, 2011.

[123] K. V. Shvachko, "HDFS Scalability: The Limits to Growth," *login: The Magazine of USENIX*, vol. 35, pp. 6–16, Apr. 2010.

[124] "MySQL Cluster CGE." `http://www.mysql.com/products/cluster/`. [Online; accessed 30-June-2015].

[125] M. Ronström and J. Oreland, "Recovery Principles of MySQL Cluster 5.1," in *Proc. of VLDB'05*, pp. 1108–1115, VLDB Endowment, 2005.

[126] I. Polato, R. Ré, A. Goldman, and F. Kon, "A comprehensive view of Hadoop research – A systematic literature review," *Journal of Network and Computer Applications*, vol. 46, pp. 1–25, 2014.

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

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

[129] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pp. 223–234, 2011.

[130] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-distributed Database," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 251–264, USENIX Association, 2012.

[131] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Trans. Database Syst.*, vol. 12, pp. 609–654, Nov. 1987.

[132] 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.

[133] "Hammer-Bench: Distributed Metadata Benchmark to HDFS." https://github.com/smkniazi/hammer-bench. [Online; accessed 1-January-2016].

[134] "Hadoop Open Platform-as-a-Service (Hops) is a new distribution of Apache Hadoop with scalable, highly available, customizable metadata.." https://github.com/smkniazi/hammer-bench. [Online; accessed 1-January-2016].

[135] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of hadoop usage in scientific workloads," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 853–864, 2013.

[136] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte, "GIGA+: Scalable Directories for Shared File Systems," in *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, (New York, NY, USA), pp. 26–29, ACM, 2007.

[137] K. Shvachko, "Name-node memory size estimates and optimization proposal." https://issues.apache.org/jira/browse/HADOOP-1687, August 2007. [Online; accessed 11-Nov-2014].

[138] "MySQL Cluster: High Availability." https://www.mysql.com/products/cluster/availability.html. [Online; accessed 23-May-2016].

[139] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SQCK: A Declarative File System Checker," in *Proc. of OSDI'08*, pp. 131–146, USENIX Association, 2008.

[140] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson, "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, (New York, NY, USA), pp. 236–249, ACM, 2015.

[141] R. Latham, N. Miller, R. B. Ross, and P. H. Carns, "A Next-Generation Parallel File System for Linux Clusters," *LinuxWorld Magazine*, January 2004.

[142] S. Yang, W. B. Ligon III, and E. C. Quarles, "Scalable distributed directory implementation on orange file system," *Proc. IEEE Intl. Wrkshp. Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011.

[143] O. Rodeh and A. Teperman, "zFS - a scalable distributed file system using object disks," in *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 207–218, April 2003.

[144] C. Johnson, K. Keeton, C. B. Morrey, C. A. N. Soules, A. Veitch, S. Bacon, O. Batuner, M. Condotta, H. Coutinho, P. J. Doyle, R. Eichelberger, H. Kiehl, G. Magalhaes, J. McEvoy, P. Nagarajan, P. Osborne, J. Souza, A. Sparkes, M. Spitzer, S. Tandel, L. Thomas, and S. Zangaro, "From research to practice: Experiences engineering a production metadata database for a scale out file system," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, (Berkeley, CA, USA), pp. 191–198, USENIX Association, 2014.

[145] "Elasticsearch." `https://www.elastic.co/products/elasticsearch`. [Online; accessed 1-January-2016].

[146] "Interrupt and process binding." `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_for_Real_Time/7/html/Tuning_Guide/Interrupt_and_process_binding.html`. [Online; accessed 1-January-2017].

[147] "Linux: scaling softirq among many CPU cores ." `http://natsys-lab.blogspot.se/2012/09/linux-scaling-softirq-among-many-cpu.html`. [Online; accessed 1-January-2017].

[148] P. Schwan, "Lustre: Building a File System for 1000-node Clusters," in *Proc. of OLS'03*, 2003.

[149] Cindy Gross, "Hadoop Likes Big Files." `https://blogs.msdn.microsoft.com/cindygross/2015/05/04/hadoop-likes-big-files/`. [Online; accessed 30-Jan-2017].

[150] Tom White, "The Small Files Problem." `http://blog.cloudera.com/blog/2009/02/the-small-files-problem/`. [Online; accessed 1-March-2017].

[151] Ismail, Mahmoud and Niazi, Salman and Ronström, Mikael and Haridi, Seif and Dowling, Jim, "Scaling HDFS to More Than 1 Million Operations Per Second with HopsFS," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid*

Computing, CCGrid '17, (Piscataway, NJ, USA), pp. 683–688, IEEE Press, 2017.

[152] "Yahoo Research. S2 - Yahoo Statistical Information Regarding Files and Access Pattern to Files in one of Yahoo's Clusters." https://webscope.sandbox.yahoo.com/catalog.php?datatype=s. [Online; accessed 30-Jan-2017].

[153] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for openjdk," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, p. 13, ACM, 2016.

[154] M. Asay, "http://www.techrepublic.com/article/why-the-worlds-largest-hadoop-installation-may-soon-become-the-norm," *Tech Republic*, vol. Sep, 2014.

[155] "HDFS Short-Circuit Local Reads." https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html. [Online; accessed 30-March-2017].

[156] "Hadoop Archives Guide." https://hadoop.apache.org/docs/r1.2.1/hadoop_archives.html. [Online; accessed 30-Jan-2017].

[157] A. Agarwal, "Heterogeneous Storages in HDFS." https://hortonworks.com/blog/heterogeneous-storages-hdfs/, 2014. [Online; accessed 26-February-2018].

[158] B. Leenders, "Heterogeneous storage in hopsfs." Masters thesis at KTH (TRITA-ICT-EX, 2016:123), 2016.

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

[160] "Flexible IO Tester." https://webscope.sandbox.yahoo.com/catalog.php?datatype=s. [Online; accessed 30-Jan-2017].

[161] Arpit Agarwal, "Scaling the HDFS NameNode." https://community.hortonworks.com/articles/43838/scaling-the-hdfs-namenode-part-1.html. [Online; accessed 30-Jan-2017].

[162] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017.

[163] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *J. ACM*, vol. 43, pp. 685–722, July 1996.

[164] F. P. Junqueira and B. C. Reed, "The life and times of a zookeeper," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pp. 4–4, ACM, 2009.

[165] M. Ronström and J. Oreland, "Recovery Principles of MySQL Cluster 5.1," in *Proc. of VLDB'05*, pp. 1108–1115, VLDB Endowment, 2005.

[166] R. Guerraoui and M. Raynal, "A Leader Election Protocol for Eventually Synchronous Shared Memory Systems," (Los Alamitos, CA, USA), pp. 75–80, IEEE Computer Society, 2006.

[167] A. Fernandez, E. Jimenez, and M. Raynal, "Electing an Eventual Leader in an Asynchronous Shared Memory System," in *Dependable Systems and Networks, 2007. DSN '07.*, pp. 399–408, June 2007.

[168] A. Fernandez, E. Jimenez, M. Raynal, and G. Tredan, "A Timing Assumption and a t-Resilient Protocol for Implementing an Eventual Leader Service in Asynchronous Shared Memory Systems," in *ISORC '07*, pp. 71–78, May 2007.

[169] "SQLServer." `http://www.microsoft.com/en-us/server-cloud/products/sql-server/`. [Online; accessed 30-June-2015].

[170] "MySQL :: The world's most popular open source database." `http://www.mysql.com/`. [Online; accessed 30-June-2015].

[171] "White paper: XA and Oracle controlled Distributed Transactions." `http://www.oracle.com/technetwork/products/clustering/overview/distributed-transactions-and-xa-163941.pdf`. [Online; accessed 30-June-2015].

[172] "Riak, Basho Technologies." `http://basho.com/riak/`. [Online; accessed 30-June-2015].

[173] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The End of an Architectural Era: (It's Time for a Complete Rewrite)," in *Proceedings of the 33rd International VLDB Conference*, pp. 1150–1160, 2007.

[174] "Multi-Model Database – FoundationDB." `https://foundationdb.com/`. [Online; accessed 30-June-2015].

[175] "VoltDB – In-Memory Database, NewSQL and Real-Time Analytics." `http://voltdb.com/`. [Online; accessed 30-June-2015].

[176] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.

[177] V. Sanz-Marco, M. Zolda, and R. Kirner, "Efficient Leader Election for Synchronous Shared-Memory Systems," in *Proc. Int'l Workshop on Performance, Power and Predictability of Many-Core Embedded Systems (3PMCES'14)*, Electronic Chips and Systems Design Initiative (ECSI), Mar. 2014.

[178] A. Mostefaoui, E. Mourgaya, and M. Raynal, "Asynchronous implementation of failure detectors," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pp. 351–360, June 2003.