2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

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

Mahmoud Ismail*, Salman Niazi*, Mikael Ronström†, Seif Haridi*, Jim Dowling*
*KTH - Royal Institute of Technology, † Oracle
{maism, smkniazi, haridi, jdowling}@kth.se, mikael.ronstrom@oracle.com

***Abstract*—HopsFS is an open-source, next generation distribution of the Apache Hadoop Distributed File System (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. By removing the metadata bottleneck in Apache HDFS, HopsFS enables significantly larger cluster sizes, more than an order of magnitude higher throughput, and significantly lower client latencies for large clusters.**

In this paper, we detail the techniques and optimizations that enable HopsFS to surpass 1 million file system operations per second - at least 16 times higher throughput than HDFS. 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. Together with more traditional techniques, such as batching and write-ahead caches, we show how many incremental optimizations have enabled a revolution in distributed hierarchical file system performance.

***Keywords*-File System Design; Distributed File System; High-performance file systems; NewSQL**

I. Introduction

During the last decade, many distributed file systems have been developed to cope with the data deluge resulting from huge drops in the cost of storage [1], [2]. The Hadoop Distributed File System (HDFS) [2] is the most popular open-source platform for storing large volumes of data, with cluster sizes of up to 100PB being reported [3]. However, HDFS' design introduces two scalability bottlenecks when processing metadata [4]. Firstly, metadata, relating to the file system namespace and housekeeping functions, is managed in-memory on a single server, called the Namenode. The Namenode's Java Virtual Machine (JVM) architecture places practical limits on the size of the namespace (files/directories), with maximum heap sizes of around 200GB being reported[1]. Above this size, garbage collection effects can cause severe performance degradation [5]. HDFS' second main bottleneck is a single global lock on the namespace that ensures the consistency of the file system by limiting concurrent access to the namespace to a single-writer or multiple-readers. This limits HDFS' usage in high throughput scenarios to read-heavy workloads (such as 94.74% read only metadata operations in Spotify's workload, see figure 3).

Researchers have previously investigated the use of single-node and distributed databases to store file system metadata, and the conventional wisdom has been not to store the file system's metadata fully normalized for performance reasons [6], [7]. The drawback of denormalized metadata, however, is that some file system operations require vastly increased processing time to handle all the duplicate data. For example, file system read and write operations can be optimized at the cost of impractically slow subtree operations, such as, renaming a directory containing millions of files.

However, recent advances in shared-nothing, transactional, in-memory NewSQL [8] databases, as well as huge drops in the cost of main memory, have changed the trade-offs needed to build a distributed metadata service for hierarchical distributed file systems. Large databases with tens of terabytes of in-memory capacity that can handle millions of operations per second are now available as open-source platforms [9].

HopsFS [10] is a new distribution of HDFS that decouples the file system metadata storage and management services, exploiting the capacity and performance properties of NewSQL databases. HopsFS stores the file system metadata fully normalized in a scale-out, in-memory, distributed, relational database called Network Database (NDB), a NewSQL storage engine for MySQL Cluster [9]. HopsFS supports up to at least 24 TB of metadata (37 times HDFS' metadata capacity) and at least 16 times the throughput of HDFS.

HopsFS has been running in production since April 2016, providing Hadoop-as-a-Service for researchers at the SICS ICE data center in Luleå, Sweden.

This paper extends our previous work in [10] to present 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. As well as leveraging classical database techniques such as *batching*, and *write-ahead* caches within transactions, and exploiting the scalability properties of distribution aware techniques found in NewSQL database, such as *application defined partitioning(ADP)*, *distribution aware transactions(DAT)*, and *partition-pruned index scans (PPIS)*.

II. Background

A. Hadoop Distributed File System (HDFS)

HDFS [2] is an open-source implementation of the Google File System (GFS) [1]. HDFS splits files into blocks (default 128 MB in size), and the blocks are replicated across multiple Datanodes for high availability (default of 3 replicas). A HDFS cluster consists of a single Active Namenode (ANN), which is responsible for managing the file system namespace, and serving file system requests sent by the clients, as well as many Datanodes (up to 5-10K) that store and manage the file blocks, see figure 1.

[1]Recent figures from Spotify's HDFS cluster

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.117

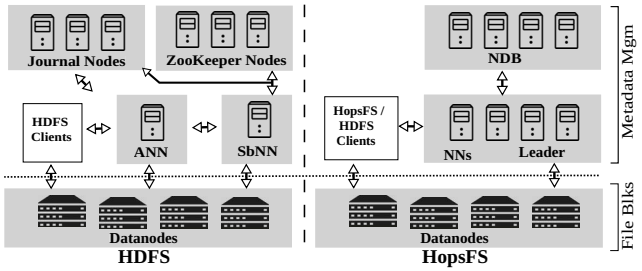


Figure 1: Architecture diagram for HDFS and HopsFS. HopsFS provides highly available stateless Namenodes, with one leader Namenode for housekeeping activities, using NDB cluster. On the other hand, HDFS requires an Active Namenode (ANN), at least 1 Standby Namenode (SbNN), at least 3 Journal nodes, and at least 3 Zookeeper nodes to enable a highly available setup.

The Namenode design suffers from metadata scalability bottlenecks. For instance, the HDFS metadata is stored on the JVM heap of the Namenode which limits the scalability of the namespace [4]. Moreover, the Namenode uses a global lock on the entire file system metadata to execute the file system operations atomically on the namespace, limiting the Namenode semantics to single-writer, multiple-reader concurrency semantics. To ensure high availability of the metadata, the Namenode logs the changes to the metadata into journal servers using quorum-based replication. The logs are later read and reapplied asynchronously by the Standby Namenode (SbNN). ZooKeeper [11] is used to coordinate failover from the active Namenode to the standby Namenode, as well as providing agreement on which Namenode is the active and which is standby.

B. HopsFS

HopsFS [10] is an open-source, drop-in replacement for HDFS. HopsFS overcomes the shortcomings of HDFS by replacing the Active and Standby Namenodes with a distributed metadata service built on a NewSQL database, see figure 1. HopsFS' supports a scalable number of stateless Namenodes that still serve file system requests sent by the clients, but instead of processing the metadata locally in-memory, they process the metadata stored in the database. Each Namenode has a Data Access Layer (DAL) driver that encapsulates all database operations, allowing HopsFS to use a variety of NewSQL databases, even with different licensing models. Currently, only MySQL Cluster is supported. HopsFS Namenodes can serve requests from both HDFS and HopsFS clients, but HopsFS clients also provide load balancing between the Namenodes using *random*, *round-robin*, and *sticky* policies. HopsFS' Namenodes uses the database as a shared memory to elect one of the Namenodes as a leader [12]. The leader Namenode is responsible for housekeeping activities, such as replication of under-replicated-blocks and handling failures of the Datanodes.

C. Network Database (NDB)

MySQL Cluster [9] is a shared-nothing, in-memory, highly-available, distributed, relational NewSQL database. Network Database (NDB) is the storage engine of MySQL Cluster. MySQL Cluster consists of at least one management node for monitoring and configuring the cluster, and multiple NDB datanodes for storing the tables' data and handling transactions to access/process the stored data.

NDB datanodes, not to be confused with HDFS/HopsFS datanodes, are organized into node groups, where the number of NDB datanodes in a group is determined by the replication factor of the cluster. NDB stores its data in tables in row-oriented format, sharding the rows in a table over NDB datanodes using the hash value of the table's partition key, which is by default the primary key. Each partition (shard) is replicated and stored by a node group, that is, all the NDB datanodes in a node group contain a complete copy of all the shards assigned to the node group. NDB provides a transaction coordinator (TC) at every NDB datanode and they run in parallel, supporting concurrent cross-partition transactions. New NDB datanodes and node groups can be added to a running cluster without affecting ongoing operation, with the rebalancing of data among node groups handled transparently by MySQL cluster. Moreover, NDB supports the following features:

Application Defined Partitioning (ADP)*:* This enables developers to override the default NDB partitioning scheme, for fine grained control over how the tables' data is distributed across the NDB datanodes.

Distribution Aware Transactions (DAT)*:* The latency of database operations can be reduced by specifying a *transaction hint*, based on the *application defined partitioning* scheme, to start the transaction on the NDB datanode containing the data to be read/updated by the transaction. Incorrect transaction hints may incur additional network traffic (as a Transaction Coordinator (TC) may route the requests to a different NDB datanode holding the data), but otherwise correct system operation.

D. Different Types of NDB Read Operations

NDB supports different types of operations to read the data from the database. Each of these operations has a different cost as shown in figure 2. A description of these operations is as follows:

Primary Key (PK) Operation*:* PK operations read/write/update a single row stored in a database shard, and NDB is designed to support high throughput and low latency PK operations.

Batched Primary Key (B) Operations*:* Batching PK operations can enable higher throughput at the cost of increased latency by making more efficient use of network bandwidth.

Partition-Pruned Index Scan (PPIS)*:* A $PPIS$ is an index scan operation that is local to a single database shard, making it scalable (NDB clusters can have hundreds of database shards). A $PPIS$ is a distribution aware

operation that exploits the *application defined partitioning* and *distribution aware transactions* features of NDB.

Index Scan (IS): An index scan operation causes an index scan operation to be executed on all the database shards, causing it to become increasingly slow for increasingly larger clusters.

Full Table Scan (FTS): Similar to index scan operations, FTS operations are not distribution aware. FTS operations also do not use any index, thus, reading all the rows of a table stored on all the database shards. In distributed databases, FTS operations should be avoided due to their high cost.

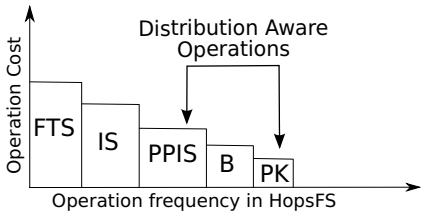


Figure 2: Cost of different NDB read operations. FTS is the least efficient operation, while PK is the most efficient operation. PK, B, and PPIS are scalable operations, as they exploit the distribution aware transactions feature of NDB.

III. HopsFS Key Design Decisions

In this section, we describe the key decisions taken for HopsFS. We reimplemented most of the file system operations such that these operations only use scalable database access operations such as *primary key (PK)*, *batched primary key (B)*, and *partition-pruned index scan (PPIS)*. We tried to avoid *index scans (IS)* and *full table scans (FTS)* as much as possible since these operations are not as scalable.

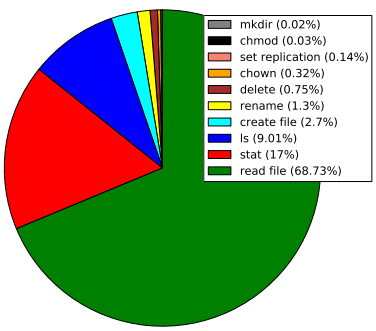


Figure 3: Frequency of file system operations for Spotify's HDFS cluster.

A. Partitioning Scheme

In HDFS, a file/directory is identified as an *inode*. An *inode* has zero or more *blocks*, where each *block* is replicated into multiple *replicas* (usually three). To keep track of the files' blocks and replicas status, multiple queues are used such as *under-replicated-blocks*, *excess-replicas*, *invalidated-blocks*, etc. HopsFS translates all the aforementioned queues into tables in the database. For example, a file/directory is identified as a row in *inodes* table, while each of its blocks is a row in *blocks* table.

We base our choice of the partitioning scheme on the relative frequency of file system operations reported by production deployments, where $\approx 95\%$ of the operations comprise of *read*, *list*, and *stat* operations, see figure 3. We choose the *inode's parent ID* as the partition key for the *inodes* table that enables fast list operation for directories as all the immediate descendants of a directory are stored on the same database shard. Similarly, we shard the *blocks* and *replicas* tables using *inode's ID* which enables fast file read operations as all the *blocks* and *replicas* information for a file is stored on the same database shard.

Hierarchical namespaces are inherently imbalanced, as all full file paths include one of a small number of top-level directories. The root directory participates in all file system operations, that is, all file path resolution operations start at the root. HopsFS caches the *root inode* ($/$) to solve this problem, also see section III-D2. In HopsFS, all path resolution operations start at the second path component, .e.g., $/home$. As mentioned above we use the *inode's parent ID* as the partition key for the inodes table. This causes all immediate children of the root directory to be stored on the same database shard. This particular database shard becomes a bottleneck during path resolution operations as all the path resolution operations start at this shard. HopsFS uses an adaptive pseudo-random partitioning mechanism to uniformly distribute the top level directories among the database shards.

B. Fine Grained Locking

HopsFS uses hierarchical locking for *inode* operations. That is, taking a lock on an inode implicitly locks all its associated metadata. We implemented hierarchical locking using *row-level locking* that is provided by the NDB. To maintain the global order, HopsFS ensures that all operations start on the *inodes* table. We also implement a novel *subtree* locking mechanism to handle operations on large directories containing potentially millions of files. Unlike HDFS which provides single-writer, multiple-reader concurrency semantics, HopsFS provides multiple-writer, multiple-reader concurrency semantics.

C. Optimizing File System operations

All the file system operations start by resolving the file path. Therefore, we choose the primary key of the *inodes* table to be a combination of the *inode's parent ID* and *inode's name*. This ensures that the path resolution is done using primary key (*PK*) operations. Our partitioning scheme, as described earlier, enables reading the *blocks* and *replicas* of a file, as well as listing directories using efficient partition-pruned index scan (*PPIS*) operations. Each file system operation translates into one or more database transactions. In table I, we show the corresponding database operations for the common file system operations.

FS Ops	Without Inode Hint Cache at the namenodes	Inode Hint Cache Hits
mkdirs(d_N)	$(N-3)PK_{rc} + 2PK_w$	$2PK_w + B_{N-3}$
addBlk(f_N)	$(N-2)PK_{rc} + PK_w + PK_r + B_1 + (f_s = 0\ ?\ PPIS : 6PPIS)$	$PK_w + PK_r + (2B)_{N-1} + (f_s = 0\ ?\ PPIS : 6PPIS)$
create(f_N)	$(2N-5)PK_{rc} + 5PK_w + (f_s = 0\ ?\ PPIS : 8PPIS) + \sum_{b=b_0}^{b_x}$ addBlk(f_N)	$5PK_w + (2B)_{2N-5} + (f_s = 0\ ?\ PPIS : 8PPIS) + \sum_{b=b_0}^{b_x}$ addBlk(f_N)
read(f_N)	$(N-2)PK_{rc} + PK_w + (f_s = 0\ ?\ 0 : 5PPIS)$	$PK_w + B_{N-2} + (f_s = 0\ ?\ 0 : 5PPIS)$
ls(i_N)	$(N-2)PK_{rc} + PK_r + (i$ is a dir $?\ PPIS : 0)$	$PK_r + B_{N-2} + (i$ is a dir $?\ PPIS : 0)$
chmod(i_N)	$(2N-4)PK_{rc} + 2PK_w + (i$ is a dir $?\ IS : PPIS)$	$2PK_w + (2B)_{2N-4} + (i$ is a dir $?\ IS : PPIS)$
stat(i_N)	$(N-2)PK_{rc} + PK_r$	$PK_r + B_{N-2}$
del(f_N)	$(2N-4)PK_{rc} + 2PK_w + (f_s = 0\ ?\ 0 : 8PPIS)$	$2PK_w + (2B)_{2N-4} + (f_s = 0\ ?\ 0 : 8PPIS)$
mv(f_N,f_M)	$(2M+2N-S-7)PK_{rc} + (f_s = 0\ ?\ 0 : 9PPIS) + (S = M-1\ \&\ M = N\ ?\ 3PK_w : 4PK_w)$	$(3B)_{2M+2N-S-7} + (f_s = 0\ ?\ 0 : 9PPIS) + (S = M-1\ \&\ M = N\ ?\ 3PK_w : 4PK_w)$

f_N,d_N,i_N : f=file, d=dir, i=file/dir path at depth N
PK_w,PK_{rc} : a primary key read operation with a write or a read-committed lock (i.e., no lock)
xB_N : x primary key batch operations that read in total N rows from the database
S : number of shared inodes in the paths between (f_N and f_M)
f_s : size of a file f
$PPIS$: a partition-pruned index scan
IS : Index scan (hits all shards)

Table I: The cost of file system (metadata) operations in terms of database accesses for (1) no inode hint caching and (2) cache hits.

D. Caching

1) Per-Transaction Cache: HopsFS implements a pessimistic concurrency control model, where we use *row-level locking* to serialize conflicting operations. We encapsulate the operations in a transaction [10]. Inside the transaction, usually the rows are read and updated multiple times. So to avoid many round trips to the database, we implement a per-transaction cache (snapshot) which sends the updated rows to the database as a single batch at the end of the transaction.

2) Root Inode Cache: The *root inode* is shared among all valid file system paths. Therefore, it is read by all the file system operations. As a result, the database shard responsible for the *root inode* becomes a bottleneck. The *root inode* is immutable, so to alleviate the load on the database shard that stores the root inode, we cache the *root inode* on all the Namenodes.

3) Inode Hints Cache: Resolving a path is the first step in every file system operation. Assuming a path of length *N*, resolving the path will take *N* network round trips to the database. This suggests that our performance will degrade as the path length increases. As a solution, each Namenode caches only the primary keys of the inodes. Given a path and a cache hit on all the path components, we can read all the path inodes from the database using a single batched primary key operation (*B*). The cost reduction provided by the inode hints cache is detailed in table I.

IV. Configuring HopsFS and NDB

This section introduces the most important configuration parameters for tuning the performance of HopsFS and NDB together.

A. Optimizing NDB cluster setup

HopsFS performance is directly affected by how the NDB cluster is setup. Each NDB datanode runs a multi-threaded application called "ndbmtd", which is responsible for handling all the tables' data assigned to that NDB datanode, as well as transaction handling, node recovery, check-pointing to disk, and online backup. Table II describes the key NDB datanode threads and the recommended number of threads to enable a high performance HopsFS setup.

Task	Count	Responsibility
LDM	12	Local Data Manager threads handle the data in one or more shards on the NDB datanode. For best performance one LDM thread per database shard is recommended.
TC	4	Transaction Coordinator threads handle ongoing transactions on the NDB datanode. TC to LDM ratio is determined by the type of the workload, for example, for read-heavy workloads a 1 to 4 ratio is used, while for update-heavy a 1 to 1 ratio should be used.
MAIN	1	The main thread handles schema management, and it can also act as a transaction coordinator, if needed.
RECV	2	Receive threads handle inbound network traffic.
SEND	1	Send threads handle outbound network traffic.
REP	1	Handles asynchronous replication operations to other database clusters.
IO	1	Handles I/O operations on the underlying file system on the NDB datanode.

Table II: Important NDB configuration parameters.

In our NDB setup each LDM thread handled exactly one database shard. In the large scale experiments, involving 12 NDB datanodes, this resulted in $12 * 12 = 144$ shards.

B. Thread locking and Interrupt Handling

Together HopsFS and NDB generate millions of network interrupts every second. Good interrupt handling is imperative for high performance of NDB and HopsFS. Usually the Linux operating system distributes network interrupts among the CPUs. However, sharing CPUs among interrupt handlers and user processes can affect the performance of both the interrupt handlers and the user processes due to CPU instruction cache misses. Moreover, binding a process to a CPU reduces instruction cache misses and improves the performance. We configured each NDB datanodes to run with a total of 22 threads. All NDB datanodes threads were bound to their own CPU. In our experiments, the remaining CPUs and the CPUs assigned to MAIN and REP threads were used to handle the hardware and software interrupts. HopsFS Namenodes were configured to run 200 client request handler threads and a dedicated thread for communication with the NDB cluster. The Namenode thread responsible for communication with the NDB was bound to a dedicated CPU, while the other threads shared the remaining CPUs [13], [14]. For high performance, we recommend at least 10 GbE with sub millisecond ping latency between the NDB datanodes and the Namenodes. Additionally, increasing the network maximum transmission unit (MTU) size to 9000 (jumbo frames) also increases both the network throughput and HopsFS throughput.

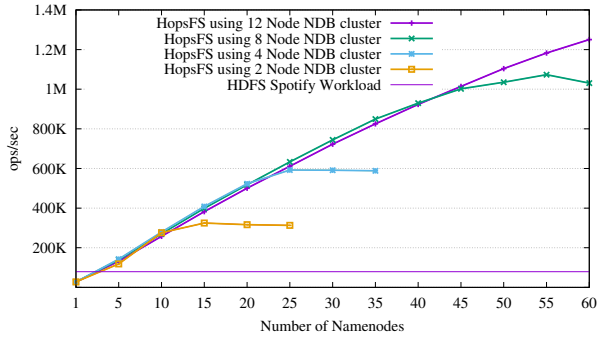


Figure 4: HopsFS and HDFS throughput for Spotify workload.

V. Scalability Evaluation

We ran all the experiments on premise using Dell PowerEdge R730xd servers (Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz, 256 GB RAM, 4 TB 7200 RPM HDDs) connected using a single 10 GbE network adapter. Unless stated otherwise, NDB, version 7.5.3, was deployed on 12 nodes configured to run using 22 threads each and the data replication degree was 2. We extended the QFS [15] benchmark utility[2] to test the metadata scalability of HopsFS [10].

A. Throughput Scalability

We used workload traces from Spotify that operates a Hadoop cluster containing 1600+ nodes and storing 60 PB of data. As shown in figure 4, we compared the performance of HopsFS with HDFS. Also, we varied the setup of the NDB cluster from 2 to 12 nodes, to show the effect of scaling out the NDB cluster on HopsFS. Using a 2-node NDB cluster, HopsFS scales linearly up to 10 Namenodes, then it reaches a plateau since the database becomes overloaded. Similarly, 4-node and 8-node NDB clusters help HopsFS to scale linearly up to 25 and 45 Namenodes respectively. HopsFS can perform **1.25 million** operations per second that is **16** times the throughput of HDFS using Spotify's workload, as shown in figure 4.

As shown in figure 3, 2.7% of the operations in the Spotify's workload are file create operations. We derived a write-heavy synthetic workload by increasing the frequency of file create operations to 20% and decreasing the frequency of the file read operations accordingly. Then, we ran both HopsFS and HDFS on that synthetic workload. HopsFS outperforms HDFS throughput by **37** times, due to the use of fine grained locking compared to the global single lock used in HDFS.

B. Metadata Scalability

Assuming a file with two blocks that are triple replicated, and with a file name of 10 characters long, a file would require **458** bytes to store its metadata in memory in HDFS. While, in HopsFS, the same file would require **1552** bytes, however, the metadata is replicated twice. Thus, by considering the high availability of HDFS, where active and standby Namenodes participate, and the twice replication of NDB, HopsFS requires $\approx$ **1.5** times more memory than HDFS.

As shown in figure 5, HDFS scales up to 460 million files with a JVM heap size of 200 GB, while HopsFS can scale up to 17 billion files using 48-node NDB cluster with 512 GB of RAM on each NDB datanode. HopsFS can store **37 times** more metadata than HDFS.

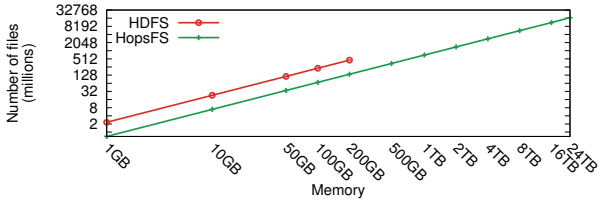


Figure 5: HopsFS and HDFS Metadata Scalability plotted in Log Scale.

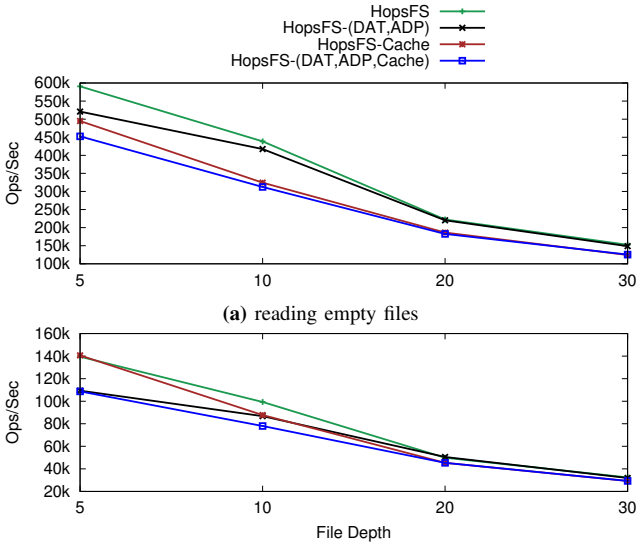
C. Effect of the Database Optimization Techniques

In this experiment, we focus on three optimizations; (1) *distribution aware transactions (DAT)*, (2) *application defined partitioning (ADP)*, and (3) the *inode hint cache* at the Namenodes. We ran file creation and file read experiments on a 30 Namenode HopsFS cluster while varying the file depth, as shown in figures 6a and 6b.

File system operations in HopsFS are implemented as one or more transactions, where each transaction performs multiple round trips to the database to read and update the metadata. In Table I, we show the number of round trips needed for each operation in the case of (1) no *inode hint cache* and (2) *inode hint cache* hits. For example, if we create an empty file */d1/d2/.../d9/f* at depth $N = 10$, then in the case of no caching, we substitute in the equation from Table I and derive the cost of the operation as $15PK_{rc}+5PK_{w}+PPIS$, that is 15 *primary key* reads with read-committed lock, 5 *primary key* reads with write lock, and 1 *partition-pruned index scan (PPIS)*. In contrast, in the case of cache hits for all inodes hints, we derive the cost of the operation as $5PK_{w}+PPIS+2(B)_{15}$ irrespective of the depth of the file in the namespace hierarchy. Cache hits save 15 *primary key (PK)* round trips to the database and instead uses 2 *batched primary key operations (B)* to fetch the desired rows.

We ran experiments for both cases, that is, with the cache enabled *(HopsFS)* and with the cache disabled *(HopsFS-Cache)*, as shown in figure 6b. Our experiments show that at depth $N = 10$ the *inode hint cache* improves HopsFS' throughput for file creation by $\approx 11\%$. Similarly, in the case of file read, the cache improves HopsFS by $\approx 26\%$, see figure 6a. Moreover, in figure 6b, the difference between different optimizations is not apparent while increasing the depth of the file, since we also have a PPIS in the equation for file create which adds a higher cost than PK, and B operations. On the other hand, in figure 6a, the difference is still apparent since a file read consists of only PK operations (that scale linearly). Fortunately, average file depths in production workloads are

[2]https://github.com/smkniazi/hammer-bench



(a) reading empty files

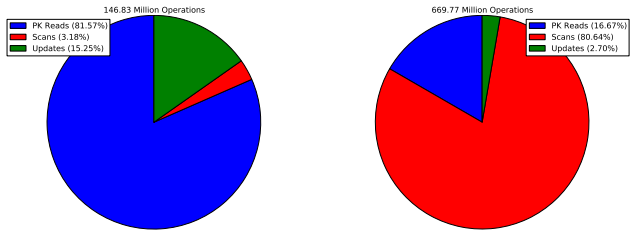
(b) creating empty files

Figure 6: Effect of different optimizations on file create and file read operations with respect to depth of the file in the namespace hierarchy. We ran HopsFS with 30 Namenodes.

not that high - Spotify's workload reported an average file depth of **7**.

VI. Load on the Database

NDB provides a set of counters to display how the cluster is loaded with different operations. These counters include primary key reads, scans per partition, and update counters. We ran Spotify Workload on HopsFS with 30 namenodes twice; first with HopsFS running with all optimizations (DAT, ADP, Cache), and the second run with no optimizations at all. After each run, we collected the counters from the database as shown in figure 7. In the unoptimized version, see figure 7b, we see higher numbers of scans compared to the optimized version, see figure 7a. HopsFS' throughput drops by only 5%, since the Spotify's workload is read heavy, see figure 3, which translates into mainly primary key operations on the database. However, the unoptimized version incurs ≈ 4.5 times more operations than the optimized version due to the increased number of scans in the unoptimized version.



(a) Optimized **(b)** Unoptimized

Figure 7: Load on the NDB cluster while running Spotify Workload on HopsFS with 30 Namenodes.

VII. Conclusions

In this paper, we presented HopsFS, a more scalable version of HDFS with a distributed metadata layer. We detailed the different techniques we used to optimize the performance of HopsFS, and their relative performance gain, including well-known techniques such as *batching*, *a hint cache*, and *a write-ahead cache*, as well as less well-known NewSQL features such as *application defined partitioning*, *distribution aware transactions*, and *partition-pruned index scans*. The cumulative effect of our optimizations is a distributed file system that surpasses 1 million file system operations per second on Spotify's Hadoop workload. That is, HopsFS supports at least 16 times higher throughput than HDFS, and scales to significantly larger clusters, storing at least 37 times more metadata.

VIII. Acknowledgements

This work is funded by Swedish Foundation for Strategic Research project "E2E-Clouds", and by EU FP7 project "Scalable, Secure Storage and Analysis of Biobank Data" under Grant Agreement no. 317871.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies, 2010*, May 2010, pp. 1–10.

[3] 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.

[4] K. V. Shvachko, "HDFS Scalability: The Limits to Growth," *login: The Magazine of USENIX*, vol. 35, no. 2, pp. 6–16, Apr. 2010.

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

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

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

[8] 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*, 2014, pp. 1407–1408.

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

[10] 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: USENIX Association, 2017, pp. 89–104.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[12] S. Niazi, M. Ismail, G. Berthou, and J. Dowling, "Leader Election using NewSQL Systems," in *Proc. of DAIS*. Springer, 2015, pp. 158 –172.

[13] "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].

[14] "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].

[15] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1092–1101, Aug. 2013.