Size Matters: Improving the Performance of Small Files in Hadoop

Salman Niazi[†*] Mikael Ronström [‡] Seif Haridi[†*] Jim Dowling[†*]

[†] KTH - Royal Institute of Technology [‡] Oracle AB [*] Logical Clocks AB
{smkniazi,haridi,jdowling}@kth.se mikael.ronstrom@oracle.com

Experimentation and Deployment Paper

Abstract

The Hadoop Distributed File System (HDFS) is designed to handle massive amounts of data, preferably stored in very large files. The poor performance of HDFS in managing small files has long been a bane of the Hadoop community. In many production deployments of HDFS, almost 25% of the files are less than 16 KB in size and as much as 42% of all the file system operations are performed on these small files. We have 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 HDFS. Our solution is completely transparent, and it does not require any changes in the HDFS clients or the applications using the Hadoop platform. In experiments, we observed up to 61 times higher throughput in writing files, and for real-world workloads from Spotify our solution reduces the latency of reading and writing small files by a factor of 3.15 and 7.39 respectively.

ACM Reference Format:
Salman Niazi[†*] Mikael Ronström [‡] Seif Haridi[†*] Jim Dowling[†*]. 2018. Size Matters: Improving the Performance of Small Files in Hadoop. In *Proceedings of Middleware'18.* ACM, Rennes, France, Article 3, 14 pages. https://doi.org/https://doi.org/10.1145/3274808.3274811

1 Introduction

Distributed hierarchical file systems typically separate metadata from data management services to provide a clear separation of concerns, enabling the two different services to be independently managed and scaled [1–5]. While this architecture has given us multi-petabyte file systems, it also imposes high latency on file read/write operations that must first contact the metadata server(s) to process the request and then the block server(s) to read/write a file's contents. With the advent of lower cost main memory and high-performance Non-Volatile Memory Express solid-state drives (NVMe SSDs) a more desirable architecture would be a tiered storage architecture where small files are stored at metadata servers either in-memory or on NVMe SSDs, while larger files are kept at block servers. Such an architecture would mean that reading/writing small files would save a round-trip to the block servers, as metadata server(s) would now fully manage the small files. Such an architecture should also be able to scale out by adding new metadata servers and storage devices.

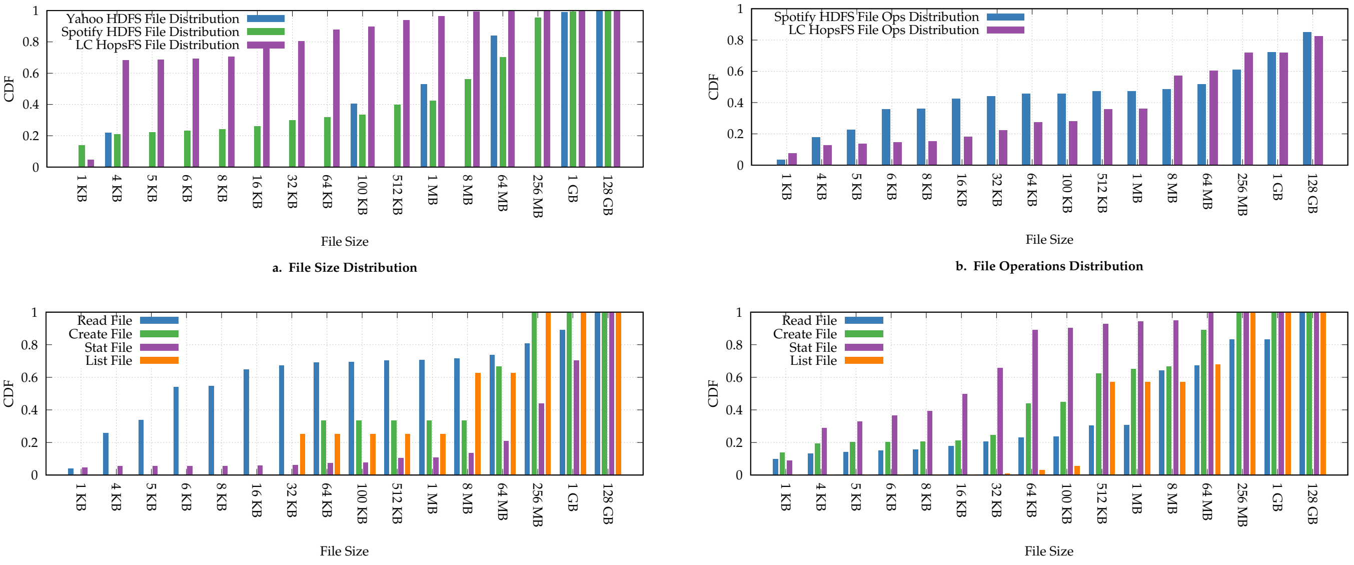
A distributed, hierarchical file system that could benefit from such an approach is the Hadoop Distributed File System (HDFS) [6]. HDFS is a popular file system for storing large volumes of data on commodity hardware. In HDFS, the file system metadata is managed by a metadata server, which stores the entire metadata in-memory on the heap of a single JVM process called the *namenode*. The file data is replicated and stored as file blocks (default size 128 MB) on block servers called the *datanodes*. Such an architecture is more suitable for providing highly parallel read/write streaming access to large files where the cost of the metadata operations, such as file open and close operations, is amortized over long periods of data streaming to/from the datanodes.

Best practices for Hadoop dictate storing data in large files in HDFS [7]. Despite this, a significant portion of the files in many production deployments of HDFS are small. For example, at Yahoo*!* and Spotify, who maintain some of the world's biggest Hadoop clusters, 20% of the files stored in HDFS are less than *4 KB*, and a significant amount of file system operations are performed on these small files. In Logical Clocks' administered Hadoop cluster majority of the files are small, that is, 68% of the files are less than *4 KB* (see Figure 1a. and section 2). Small files in HDFS affect the scalability and performance of the file system by overloading the namenode. In HDFS the scalability and performance of the file system is limited by the namenode architecture, which limits the capacity of the file system to ≈500 million files [8]. Storing the data in small files not only reduces the overall capacity of the file system but also causes performance problems higher up the Hadoop stack in data parallel processing frameworks [9]. The latency for reading/writing small files is relatively very high as the clients have to communicate with namenode and datanodes in order to read a very small amount of data, described in detail in section 3.2.

The problem with adding a tiered storage layer, that stores small files in the metadata service layer (namenode) of HDFS is that it would overload the already overloaded namenode. However, a new open-source[1] distribution of HDFS, HopsFS [8, 10], has been introduced as a drop-in replacement for HDFS that stores file system metadata in a highly available, in-memory, distributed relational

Middleware'18, December 2018, Rennes, France
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5702-9/18/12.
https://doi.org/https://doi.org/10.1145/3274808.3274811

[1] HopsFS Source Code: *https://github.com/hopshadoop/hops*



a. File Size Distribution

b. File Operations Distribution

c. Breakdown of File System Operations at Spotify

d. Breakdown of File System Operations at Logical Clocks

Figure 1: These figures show the distribution of the files and operations according to different file sizes in Yahoo!, Spotify, and Logical Clocks' Hadoop clusters. Figure a. shows the cumulative distribution of files according to different file sizes. At Yahoo! and Spotify ≈20% of the files are less than 4 KB. For Logical Clocks' Hadoop cluster ≈68% of the files are less than 4 KB. Figure b. shows the cumulative distribution of file system operations performed on files. In both clusters, ≈80% of all the file system operations are performed on files. At Spotify, and Logical Clocks ≈42% and ≈18% of all the file system operations are performed on files less than 16 KB files, respectively. Figure c. and Figure d. show the breakdown of different file system operations performed on files. At Spotify ≈64% of file read operations are performed on files less than 16 KB. Similarly, at Logical Clocks, ≈50% of file stat operations are performed on files less than 16 KB.

database. HopsFS supports multiple stateless namenodes with concurrent access to file system metadata. As HopsFS significantly increases both the throughput and capacity of the metadata layer in HDFS, it is a candidate platform for introducing tiered storage for small files.

In this paper, we introduce HopsFS++, the latest version of HopsFS, which uses a new technique for optimizing the file system operations on small files by using *inode stuffing* while maintaining full compatibility with the HDFS clients. Inode stuffing is a technique that improves the throughput and latency of file system operations for small files by collocating the metadata and data blocks for small files. We modified HopsFS to only decouple metadata and data blocks for large files. For small files, the data blocks are stored with the metadata in the distributed database. The database transactions and database replication guarantee the availability, integrity, and consistency of the small files stored in the database. We have implemented a tiered storage service where data blocks for very small files, typically ⩽1 KB, are stored in memory in the database, while data blocks for other small files, ⩽ 64 KB, are stored on-disk in the database, typically on NVMe SSDs. Larger files are stored on existing Hadoop block storage layer comprising of the datanodes. This architecture has the cost advantage that potentially hundreds of millions of files can be stored on commodity NVMe disks without the need for enough main memory in database servers to store all the blocks of the small files. The architecture is also future-proof, as higher-performance non-volatile memory (NVM), such as Intel's 3D XPoint (Optane$^{\text{TM}}$), instead of NVMe disks, could be used to store small files' data. The metadata layer can also easily be scaled out online by adding new namenodes, database nodes, and storage disks to improve throughput and capacity of the *small file storage layer*.

To the best of our knowledge, this is the first *open-source* tiered block stored solution for a hierarchical file system that uses a distributed relational database to store small files blocks. Our solution for small files has been running in production at a data center administered by Logical Clocks AB in Luleå, Sweden [11]. HopsFS++ is a drop-in replacement for HDFS, and the tiered storage for small files is implemented such that all the changes for tiered block storage are fully transparent to HDFS clients and the data processing frameworks using HopsFS++. 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, containing 9 million images (mostly small files) as well as a number of microbenchmarks. 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. 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 more disks and larger clusters. The HopsFS++ solution can be scaled out at each of the storage tiers, by adding more memory, NVMe SSDs, and servers, respectively.

2 Prevalence of Small Files In Hadoop

We have analyzed the file systems namespaces and operational logs of Spotify and Logical Clocks' administered (LC) deployments of their Hadoop clusters to find out how pervasive are the small files? Hadoop cluster statistics for Yahoo*!* are publicly available at [12], which contains information about the distribution of files according to different file sizes. For Spotify, we analyzed the entire HDFS namespace and the audit logs for HDFS operations to determine the number of different types of file system operations performed on different sized files. At Spotify, more than 100 GB of HDFS audit logs are generated every day. We analyzed multiple snapshots of the HDFS namespace (*FSImage*) and more than 10 TB of audit log files representing three months of cluster operations. For Logical Clocks, we have analyzed the audit logs representing file system operations for one week.

Figure 1a. shows the cumulative distribution of files according to different file sizes. Spotify, Yahoo*!*, and LC clusters contain ***357***, ***41***, and ***47*** million files respectively. Both at Spotify and Yahoo*!* ***20%*** of the files are less than 4 KB. In the case of LC, ***68%*** of the files are less than 4 KB. At LC there are significantly more small files because the cluster is heavily used for training and testing deep learning algorithms where usually the input consists of a large number of small files, such as images. Figure 1b. shows the cumulative distribution of file system operations performed on small files. For Yahoo*!* the distribution of file system operations performed on different sized files is not publicly available. Both at Spotify and LC, approximately ***80%*** of the file system operations are directly performed on files. Small files receive a significant portion of all the file system operations. At Spotify, ***42%*** of all the file system operations are performed on files that are less than 16 KB, while in case of LC ***27%*** of the file system operations are directly performed on small files that are less than 64 KB.

Figure 1c. and Figure 1d. show the breakdown of the percentage of different types of file system operations performed on different sized files. For example, at Spotify ***68%***, ***33%***, ***7%***, and ***25%*** of all the read file, create file, stat file, and list file operations are performed on files less than 64 KB respectively. Similarly, at LC ***22%***, ***43%***, ***88%***, and ***2%*** of all the read file, create file, stat file, and list file operations are performed on files less than 64 KB respectively.

Clearly, small files in Hadoop cluster are very pervasive and a significant number of file system operations are performed on the small files. In section 3 and in section 4, we explain the internals of HDFS and HopsFS and show why small files have poor performance. Spotify's cluster contains the most number of small files. Almost 71 million files at Spotify are smaller than 4 KB in size. Assuming if each file is exactly of size 4 KB then all the files will only take ≈800 GB of disk space with triple replication. The amount of disk space taken by small files is very small and it is feasible to store these small files on a small number of high-performance NVMe disks. In section 5 we show how we store small files in-memory and on NVMe disks in a distributed relational database to improve the performance of small files.

3 HDFS

Apache HDFS [6] is an open source Java implementation of the Google File System (GFS) [13]. HDFS is the default distributed hierarchical file system for Hadoop data processing platform [14]. HDFS stores its metadata on a single server called the Active NameNode (ANN) see Figure 2. The active namenode is responsible for handling all public file system operations, such as create, delete, rename, etc., sent by potentially thousands of concurrent file system clients. HDFS is implemented in Java and the entire metadata is stored in-memory on the heap of a single Java Virtual Machine (JVM) process. The Java garbage collection imposes a practical limit on the maximum size of the Java heap, currently, with significant ongoing tuning effort, at around 200-300 GB [15, 16]. Files in HDFS are split into large blocks of 128 MB (default), which are replicated three times (default) across the datanodes. Irrespective of the block's size, it will be stored on the datanodes. The metadata, in contrast, is stored at the namenode and includes file and directory names, the hierarchical directory structure, user permissions and ownership, time-stamps, access control lists, extended attributes, file to block mappings and other data structures related to monitoring and repairing the file system state.

HDFS uses an Active/Standby deployment model to provide high availability of the metadata service. The active namenode logs all the changes in the file system namespace to a quorum of journal nodes (usually three), and the Standby NameNode (SbNN) pulls the changes from the journal nodes and applies the changes to its in-memory copy of the file system metadata. A ZooKeeper coordination service is used to reliably fail-over from the active to the standby namenode in the case of a failure of the active namenode.

3.1 The Small Files' Problem in HDFS

In existing HDFS clusters, the ratio of datanodes to the active namenode can be as high as 4500:1 [17]. Multiple namenodes in HDFS do not improve the performance as only one namenode can be active at any given time. All file system protocols and usage patterns are optimized to reduce the memory consumption and the load on the namenode. As the namenode is involved in translating all the client file system operations into block operations on datanodes, small files are particularly problematic as they cause (1) excessive load on the namenode and (2) consume as much metadata storage space as a file of up to 128 MB in size. Note that, as there is a hard practical limit on the number of files that can be managed by the namenode (≈500 million) [8], smaller average file sizes mean smaller clusters, ceteris paribus.

The namenode uses a map-like data structure that stores file to block mappings, and its maximum size is bounded by the amount of memory that can be efficiently managed by the JVM (a few hundred GB, at most [15, 18]). Consider a simplified scenario where the largest number of blocks that can be stored in the map is one

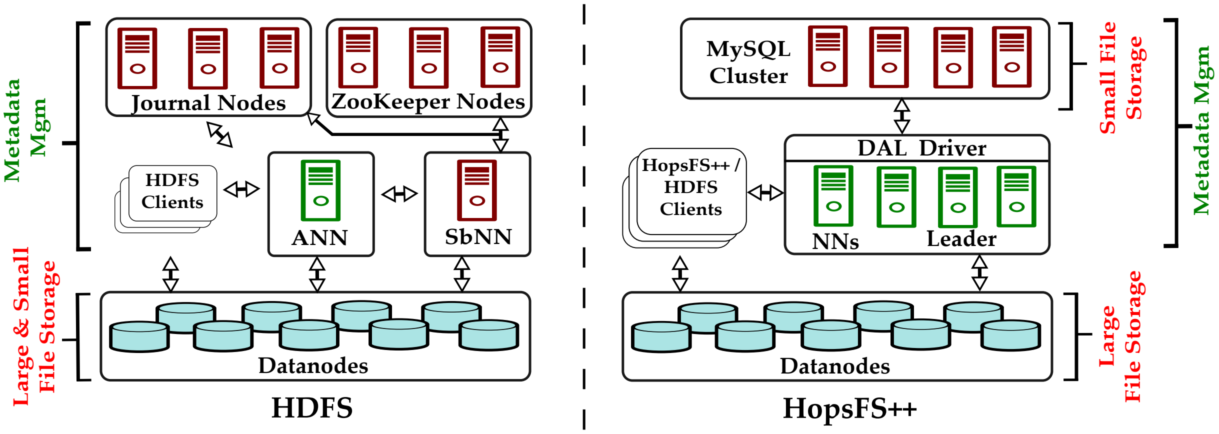


Figure 2: System architecture diagrams of HDFS and HopsFS with enhanced small files support (HopsFS++). HDFS supports only one Active Namenode (ANN) that stores the entire file system metadata in-memory and handles all the file system operations. High availability is provided using Active/Standby deployment model that requires at least one Standby Namenode (SbNN) and quorum of journal nodes. A coordination service such as ZooKeeper is used for reliable fail-over. Data blocks of files of all sizes are stored on the datanodes. HopsFS stores the metadata in MySQL Cluster distributed database; the metadata is accessed and updated by multiple stateless namenodes. The data blocks of large files are stored on the datanodes that specialize in providing streaming access to large data blocks. In HopsFS++ the data blocks of small files are stored alongside the metadata in the distributed database that specializes in providing low latency access to small amounts of data.

billion entries. If all the blocks on disk are full, that is, the blocks are exactly 128 MBs in size, then the file system can store 119.2 PB of data. However, if the blocks are only 4 KB in size, then the file system can only store 3.7 TB data (which could easily fit on a commodity hard drive). 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 larger number of blocks, and the cluster is beset by frequent pauses where the namenode garbage collects the metadata. Tuning garbage collection on the namenode for such a cluster requires significant, skilled administrator effort.

3.2 Small Files' Performance in HDFS

As HDFS separates metadata management from block management, clients have to follow a complex protocol to read a file even if the file only has a few bytes of data. When reading a file, a client first contacts the namenode to get the location of the data block(s) of the file. The namenode returns the locations of the blocks to the client after checking that the client is authorized to access the file. Upon receiving the locations of the data blocks the client establishes communication channels with the datanodes that store the data blocks and reads the data sequentially. If the client is located on the same datanode that stores the desired block then the client can directly read the data from the local disk (short-circuit read [19]). This protocol is very expensive for reading/writing small files where the time required to actually read/write the small data block is significantly smaller than the time taken by the associated file system metadata operations and data communication protocols.

The problem is even worse for writing small files, as the protocol for writing a file involves a relatively very large number of file system operations for allocating inodes, blocks, and data transfer. In order to write a file, the client first sends a request to the namenode to create a new inode in the namespace. The namenode allocates a new inode for the file after ensuring that the client is authorized to create the file. After successfully creating an inode for the new file the client then sends another file system request to the namenode to allocate a new data block for the file. The namenode then returns the address of three datanodes where the client should write the data block (triple replication, by default). The client then establishes a data transfer pipeline involving the three datanodes and starts sending the data to the datanodes. The client sends the data sequentially to the first datanode in the data transfer pipeline, and the first datanode then forwards the data to the second datanode, and so on. As soon as the datanodes start to receive the data, they create a file on the local file system to store the data and immediately send an RPC request to the namenode informing it about the allocation of the new block. Once the data is fully written to the blocks, the datanodes send another RPC request to the namenode about the successful completion of the block. The client can then send a request to the namenode to allocate a new block or close the file. Clearly, this protocol is only suitable for writing very large files where the time required to stream the data would take much longer than the combined time of all the file system operations involved in the file write protocol, that is, the cost of the metadata operations and establishing communication channels with the datanodes is amortized over the relatively long periods of time spent in reading/writing large files. In contrast, the latency of file system operations performed on small files is dominated by the time spent on metadata operations, as reading/writing a small file involves the client communicating with both the namenode and at least one datanode.

3.3 Side Effects on Hadoop Stack

Higher-level data parallel processing frameworks are designed to work more efficiently with large files [20, 21]. Poor support for small files in HDFS complicates the design of higher level frameworks in the Hadoop ecosystem. In the original MapReduce framework [20], the number of files controlled the number of mappers required to perform a job, with small files leading to lots of mappers and excessive network I/O to combine inputs and disk I/O to write intermediate results as lots of files.

Another problem in HDFS is the effect of small files on the resource-intensive block-reporting protocol. In HDFS' block-reporting protocol (default every six-hour) all the datanodes report the health of their stored blocks to the namenode, and the namenode identifies and fixes the corrupt blocks. Storing a large number of small blocks on a datanode results in huge block reports that the namenode has to process. Large block reports can cause performance degradation of the namenode [22]. In our solution based on inode stuffing, blocks stored in the database are not included as part of the block reporting protocol, as the integrity of the state of those blocks is guaranteed by the database.

3.4 Current Solutions for Small Files

In production environments where HDFS is used by a myriad of applications, making small files are unavoidable, such as storing small images, configuration files, intermediate results or logs of different data processing jobs. Current best practices for storing a large number of small files in HDFS are:

- Archiving the small files. HDFS provides an archiving mechanism, known as Hadoop Archives (HAR), as a solution to reduce the overhead on the namenode introduced by the small files [23]. HAR compacts many small files into a single file. It also builds a multilevel index to identify the location of a small file in the archive. The main objective of HAR is to reduce the number of files and thus alleviate contention on the namenode. HAR does not improve the performance of reading or writing small files. In fact, it makes the performance of small files worse as HAR maintains two indexes to locate a small file in the HAR archive. Reading the indexes and seeking to a small file in the HAR archive slows down file system operations on small files. Moreover, once a file is added to HAR archive it cannot be changed or deleted without recreating the whole archive file.
- Using alternative storage systems, such as HBase [24] and Cassandra [25] for storing small files. However, these proposed solutions significantly complicate the design of the applications higher up in the stack, that need to be aware of file size and edge cases when accessing small files stored in different storage systems. Additionally, these storage systems have different consistency semantics, for example, in HDFS the data is kept strongly consistent while Cassandra provides eventual consistency for the stored data. This adds additional complexity at the application level, which could be avoided if the storage layer handles both large and small files efficiently.

3.4.1 Heterogeneous Storage Both HopsFS and HDFS have support for heterogeneous storage volumes [26, 27], where each mounted disk on the datanodes is categorized as one of *ARCHIVE*, *DISK*, *SSD*, or *RAM_DISK* storage volumes. During file creation, a preference for a storage type can be supplied, and HDFS/HopsFS will try to ensure that blocks for the file are stored on disks of the desired storage type on the datanodes. However, it must be noted that heterogeneous storage does not solve the problem of small files as it neither reduces the load on the namenode nor it simplifies the file system operations' protocols for the small files.

4 HopsFS

HopsFS [8] is a drop in replacement for HDFS that addresses metadata scalability and performance issues caused by the monolithic architecture of the HDFS namenode. This section reviews the HopsFS architecture, and the following section describes how we extended HopsFS to support high-performance file system operations on the small files.

HopsFS provides a more scalable metadata management service comprising of multiple active namenodes and a distributed database, see Figure 2. Unlike HDFS, where the amount of the metadata is limited as the entire metadata is stored in-memory of the namenode, HopsFS stores its metadata in an external distributed database. By default, HopsFS provides support for the relational, distributed, in-memory MySQL Cluster database [28]. Removing the metadata from the namenode makes them stateless, and when using MySQL Cluster as the database, it raises by an order of magnitude both the amount of metadata that can be managed in-memory and the throughput of the cluster, in file system operations per second [8]. Despite the distributed architecture, HopsFS provides the same metadata consistency semantics as HDFS and it is fully API compatible with HDFS, that is HopsFS can be accessed using HDFS clients.

In HopsFS, the datanodes provide the block storage service for files of all sizes. HopsFS stores the data blocks for large and small files on the datanodes and the file system operations protocols for reading and writing files are the same as HDFS. Despite having higher throughput for metadata operations, the end-to-end latency for file system operations on small files was comparable to HDFS for unloaded clusters. ***Our goal***, with tiered metadata storage in this paper, is to provide HopsFS with significantly improved throughput and latency for the file system operations performed on small files.

4.1 MySQL's Network Database (NDB) Cluster

MySQL's Network Database (NDB) Cluster 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 many database storage engines. While the SQL API for the NDB engine is also available via a MySQL Server, it is not often used to build high-performance applications for NDB. Instead, the NDB storage engine can be accessed using the native (C++) NDB API or the ClusterJ (Java) API. To NDB, the MySQL Server is just

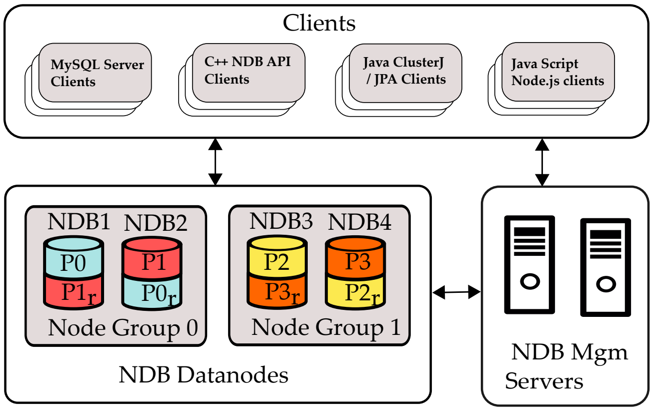


Figure 3: MySQL Cluster consists of three types of nodes: NDB datanodes, clients, and the management nodes. The NDB datanodes store the distributed database; the management nodes provide configuration information to the new NDB database nodes and the clients; and the clients nodes are active members of the cluster that access and update the data stored in the distributed database.

another client instance that uses NDB native APIs to provide a SQL interface for the data stored in NDB.

NDB Cluster consists of three types of nodes: NDB datanodes, management nodes, and clients. NDB datanodes are organized into node replication groups of equal sizes where the size of the node group is the replication degree of the database. For example, if the replication degree is set to two (default), then each node group in the MySQL Cluster will contain exactly two NDB datanodes. MySQL Cluster horizontally partitions the tables, that is, the rows of the tables are distributed among the database partitions that are uniformly distributed among the NDB datanodes. Each node group is responsible for storing and replicating all the data assigned to the NDB datanodes in the node group. For example, in the MySQL Cluster setup shown in Figure 3, there are four NDB datanodes organized into two node groups as the replication factor is set to two. The first NDB data node *NDB1* is responsible for storing the *P0* data partition while the backup/replica of the data partition, $P0_r$ is stored on the second NDB datanode *NDB2*.

By default, the database is stored in-memory at the NDB datanodes, with recovery logs and snapshots stored on disk. All transactions are committed in-memory, and transaction logs are (by default) flushed to disk every 2 seconds. The database can tolerate failures of multiple NDB datanodes as long as there is at least one surviving replica for each of the partitions. For example, in Figure 3, the database cluster will remain alive if *NDB1* and *NDB4* fail. However, if two nodes in the same node group fail, then the database will halt its operations until the unavailable node group has recovered. As such, NDB favors consistency over availability [29]. MySQL Cluster supports both node level and cluster level recovery using persistent transaction *redo* and *undo* logs and checkpointing mechanisms. Every two seconds a global checkpointing mechanism ensures that all the NDB datanodes checkpoint their logs to a 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.

4.1.1 On-Disk Data Tables in NDB Although NDB is an in-memory database, it also supports storing selected columns in on-disk tables. Updates to disk data in NDB are made *in-place*, a direct consequence of it being a relational database that uses a variant of T-trees to index data [30] (T-trees are similar to B-trees, but optimized for main-memory systems). As such, the throughput and latency of on-disk columns in NDB are not great when used with spinning disks, as they have poor performance when there are many random reads (disk seeks). In contrast, modern NVMe SSDs can perform many more random read/write operations per second making them a more suitable storage device for tables with on-disk data. In the near future, Non-Volatile Memory (NVM) technologies, such as Intel's 3D XPoint (Optane^{TM}), could also be used to store on-disk data in NDB, further improving the throughput and decreasing the latency for on-disk columns. In NDB, on-disk tables store their primary keys and indexes in-memory and there is also a page cache (of configurable size) for on-disk columns, set by a combination of the *DiskPageBufferEntries* and *DiskPageBufferMemory* configuration parameters. For read-heavy workloads, a high page cache hit-rate for on-disk data can significantly improve the performance of database transactions. Users can also configure the number of threads used for reading/writing data files that back on-disk columns, using the *DiskIOThreadPool* configuration parameter. Increasing the number of threads above 1 improves read/write throughput on the backing data file, but the practical upper limit is only a few threads, at most, to prevent side effects, such as timeouts in NDB's heartbeat and global checkpointing protocols. One limitation of NDB on-disk columns is that the storage capacity used is not easily downsized, as data files can only be removed if all data objects inside them are empty (which is highly unlikely). New data files, however, can be added on-line, as needed, to increase on-disk data capacity.

5 Tiered Block Storage in HopsFS++

HopsFS++ introduces 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, see Figure 4. and Figure 5. 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, then 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. An average file requires 1.5 KB of metadata [8] 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

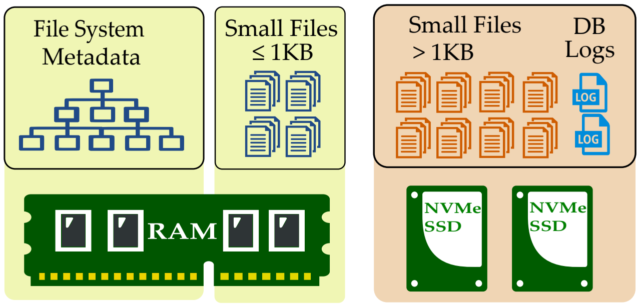


Figure 4: In HopsFS++ the entire metadata of the file system is significantly smaller than the stored file system data, and it can easily fit in-memory of the NDB datanodes. However, often it is infeasible to store all the small files in-memory. Small files that are ⩽1 KB are stored in-memory in the distributed database while larger small files can be stored in on-disk tables stored on high-performance NVMe SSDs.

is stored in-memory with the metadata. Other small files are stored in on-disk data tables. The latest high-performance NVMe SSDs 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 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. For our experiments on file read latency, we took a much more optimistic scenario where the clients always had existing, unsecured connections to datanodes, but, in practice, in secure HDFS deployments, connection setup can introduce increased latency for reading small files.

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 the 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 a *file open* request to the namenode to allocate a new inode. The client then starts writing the file data to its local buffer, see Figure 5. 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 exactly the same way as in HDFS (and HopsFS).

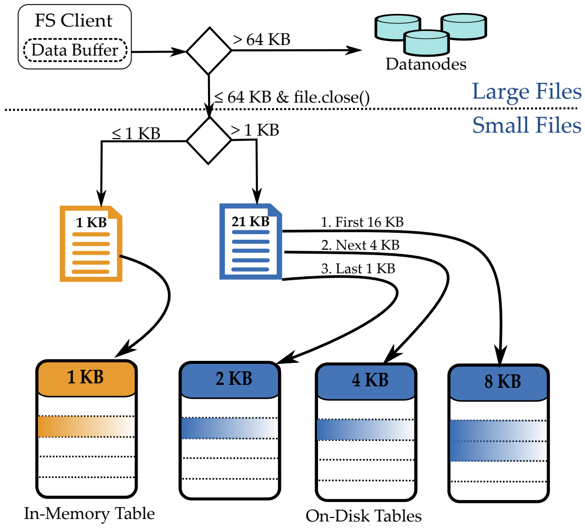


Figure 5: In HopsFS++, small files that are less than 1 KB are stored in the "1 KB" in-memory table. For larger small files the data is split into smaller chunks and stored in the corresponding on-disk data tables, for example, a file of 21 KB is split into 2 chunks of 8 KB, 1 chunk of 4 KB and 1 chunk of 1 KB. Chunking the file into table buckets gives better performance than simply storing the file as a single blob in a table.

5.1 Small Blocks in the Database

The small files' blocks are stored in the database tables in variable length data columns, such as *varchar*. The varchar columns have very low overhead for storing variable length data, as they only require one or two bytes of additional memory to store the length information of the data. A naïve solution for HopsFS++ would be to have two tables with varchar data columns to store the small files.

The first table would be stored in memory and it would contain the data blocks for files that are ⩽1 KB. The other table would store larger data blocks and the table would be stored on disk. This solution has two main problems. First, in NDB the maximum row size is 14 KB, and second, in NDB the on-disk varchar columns consume the entire space, that is, a varchar column of maximum length *n* would take *n* bytes on disk even if there is only one byte stored. *Blob* data columns are also an alternative for storing large variable length data. Using blobs any amount of data can be stored in a single row. However, in NDB the blob columns have higher overhead compared to varchar columns as the database internally splits the data into 2 KB chunks and stores the chunks in a separate blob table. In our experiments, we have observed that for large amounts of data, blob columns in NDB were significantly slower than varchar columns. In order to efficiently use the disk space, we split the data blocks into smaller chunks and store the chunks in different disk data tables using varchar columns, see Figure 5. For example, in order to store a 21 KB small file in the database the file is split into 4 chunks, that is, 2 chunks of 8 KB, 1 chunk of 4 KB and 1 chunk of 2 KB. These chunks are then stored in the corresponding disk data tables. These chunk sizes were selected experimentally. In NDB, the database disk page size for on-disk data is 32 KB. In our experiments, we have observed that for a chunk size larger than 8 KB, disk paging was less effective and the throughput dropped.

5.2 Small File Threshold Sizes

The threshold sizes at which small files are stored in-memory, on-disk in the database, or in the large file storage layer is configurable, and dependent on a number of factors, such as the cluster's distribution of file sizes, the amount of available memory and NVMe storage space at database nodes, the number of database nodes, and the number of namenodes in the system. The upper limit on the size of the different small file storage layer tiers (in-memory or on NVMe SSDs) is, in practice, determined by the characteristics of the database. Our default database, MySQL Cluster, can scale available in-memory block storage to a few tens of TBs, due to a limit of 48 database nodes in the cluster. Increasing the 48-node limit would be challenging, due to the presence of internal global agreement protocols in NDB, such as heartbeat and global checkpoints protocols. Database nodes can attach a number of NVMe SSDs for storing on-disk data, so with NDB, the NVMe SSD storage layer could potentially scale to >100 TBs. However, as motivated in the evaluation section 6.3, there is a threshold size for files, above which, for performance reasons, they should be stored in the large file storage layer.

Given this, we need to define a default small size of HopsFS++. Based on the distribution of file sizes from both Spotify Yahoo*!* and Logical Clocks' Hadoop clusters (Figure 1) and the experiments, we set the default threshold size for small files for HopsFS++ to be ⩽64 KB. The choice of the 64 KB boundary is also influenced by the default 64 KB client-side write-buffer in HDFS. 64 KB files comprise ≈30% of all files in the Spotify's HDFS cluster. These files consumed 3.4 TB of disk space with replication, which is 0.00015% of the 73 PB of disk space consumed by the entire distributed file system, and yet these files receive ≈45% of all the file system operations.

5.3 HDFS Backwards Compatibility

HopsFS++ is fully compatible with HDFS for the metadata operations. Our changes for small files have required changes in the namenode and datanode to maintain compatibility with HDFS clients. Existing HDFS clients should be able to transparently read and write small files stored in the HopsFS++ small file storage layer.

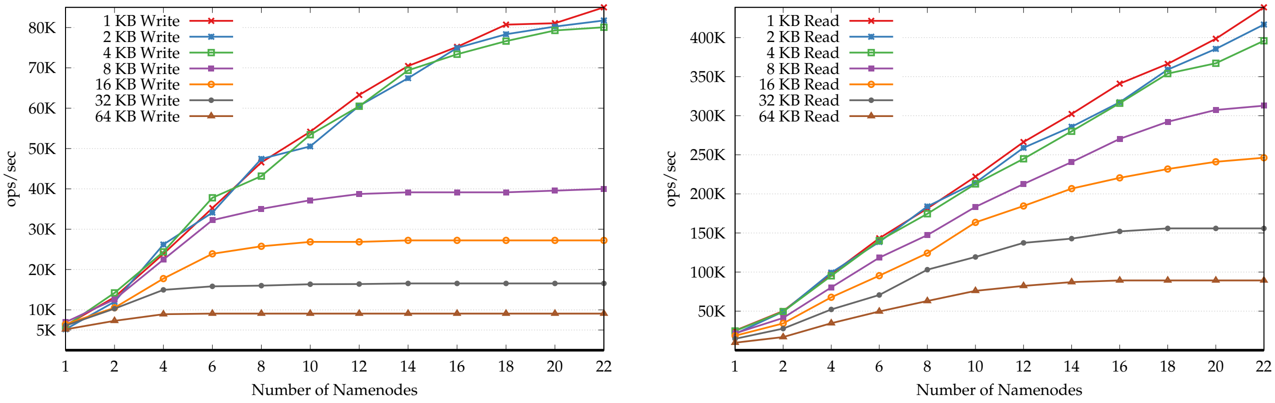
As existing HDFS clients are not aware of our new HopsFS++ protocol for writing files, all new small files created by HDFS clients will be stored on the datanodes. When an HDFS client requests to append to a small file stored in the small file storage layer, the namenode will first move the requested small file to the datanodes, before finally returning the addresses of the datanodes storing the small file's data block to the HDFS client. The HDFS client can then append to the file, following the existing HDFS write protocols.

We introduced a mechanism that allows HopsFS++ clients and datanodes easily distinguish between files stored in the small file storage layer and files stored in the large file storage layer. Data blocks stored in the small file storage layer and on the datanodes have different ID ranges. The IDs for blocks stored on the datanodes are monotonically increasing positive number, while the data blocks stored in the database have monotonically decreasing negative numbers. When HopsFS++ namenodes detect that an HDFS client wants to read a small file stored in the database, then it returns the small block's ID and a handle for a randomly selected datanode in the system to the HDFS client. The HDFS client contacts the datanode to read the data block. When HopsFS++ datanodes receive a request to read a small file's block (with a negative ID), they forward the request to small file storage layer. After reading the file's data from the small file storage layer, it relays the data to the HDFS client without breaking the HDFS data read pipeline. These changes increase the latency of file system operations on small files for HDFS clients. However, if lower latency small-file operations are desired, then the HDFS applications simply have to be linked with the HopsFS++ client libraries. Note that, while existing HDFS clients will experience higher latency than HopsFS++ clients, the system's throughput for reading/writing small files is unaffected by the choice of HopsFS++ or HDFS clients.

6 Evaluation

While vanilla HopsFS (without small-files extension) supports higher throughput for reading/writing files than HDFS, end-to-end latencies for file system operations in HDFS and HopsFS clusters are identical, therefore, all of our experiments are designed to comparatively test the performance and scalability of file system operations performed on small files in HopsFS++ and HDFS.

All the experiments were run on-premise using Dell PowerEdge R730xd servers (Intel (R) Xeon (R) CPU E5-2620 v3 2.40 GHz, 256 GB RAM, 4 TB 7200 RPM HDD) running CentOS 7.2 (Linux kernel 3.10.0-327.el7.x86_64) connected using a single 10 GbE network adapter. In the experiments, we used a six-node database cluster, NDB version 7.5.6, and the database replication degree was set to (default) 2. On each database server, an Intel 750 series 400 GB PCIe NVMe SSD was installed to store the small files in the database. According to the manufacturer's specifications, each drive is capable of performing 2200 MB/s sequential read, and 900 MB/s sequential write operations. Using the FIO benchmark [31] we have



a. Write Throughput

b. Read Throughput

Figure 6: Throughput of file write and file read operations for HopsFS++. The throughput of the file system operations linearly increases as more namenodes are added to the system. Using 22 namenodes, HopsFS++ can create more than 80 thousand 1 KB files per second and 70 thousand 4 KB files per second. Similarly, for reading the throughput of the file system linearly increases as more namenodes are added to the system. For 1 KB files, HopsFS++ is able to perform more than 400 thousand file read operations per second. In both experiments, the throughput halves when the file size is doubled beyond the 4 KB file size.

tested the drives to perform 160 thousand random read operations and 40 thousand random write operations for 20% write intensive workload using 4 KB block size. The NVMe SSDs were formatted with *Ext4* Linux file system. For testing the performance of the two file systems we used the benchmark utility published in [8], which is an extension of Quantcast File System (QFS) [32] benchmarking system. QFS is an open source implementation of HFDS, written in C++. The benchmarking utility is a distributed application that spawns tens of thousands of file system clients, distributed across many machines. The benchmark application can generate file system operations based on Spotify's traces, and it can also test the maximum throughput of any single file system operation.

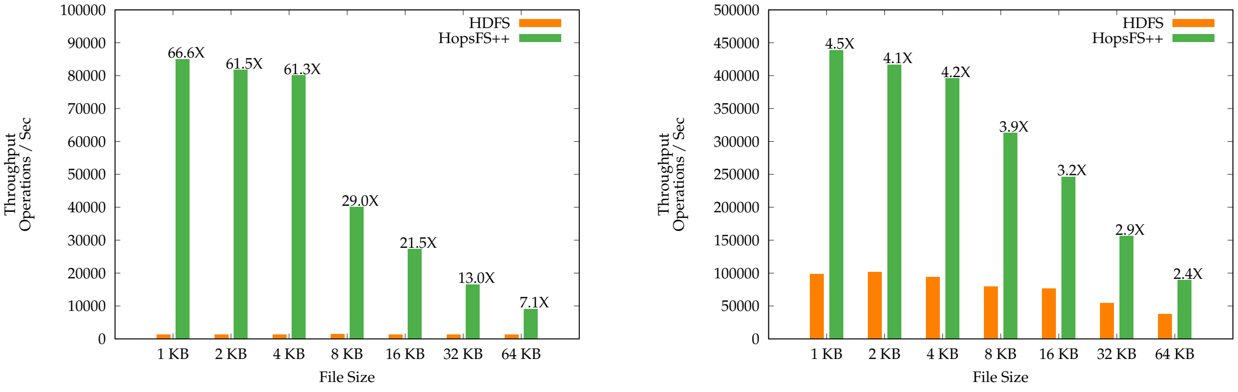
In these experiments, a total of 36 servers were used. Apache HDFS, version 2.7.3, was also run on the same servers. Apache HDFS was set up using 5 servers in the metadata service layer, which is a common practice in the production deployments of large HDFS clusters. One server was used for active namenode, one for standby namenode, and three servers were used for the ZooKeeper and HDFS journal nodes. HDFS only uses ZooKeeper during namenode fail-over, and co-locating the two services does not have any negative impact on the performance of normal HDFS operations. The remaining machines were used as HDFS datanodes to store the file blocks. The HDFS cluster was set up according to the best practices of HDFS, as described here [33]. Rest of the servers were used as Apache HDFS datanodes. For HopsFS++, the same set of machines were divided among the database, namenodes and the datanodes. Six servers where used for the NDB distributed database and the rest of the servers were divided among the datanodes and the namenodes according to the different experiment requirements.

6.1 Read/Write Throughput Benchmarks

In the first set of experiments, we investigated the scalability of file read and write operations in HopsFS++. As in HopsFS++ the namenodes access and update the data stored in the database, the throughput of file system operations that can be performed by HopsFS++ metadata layer directly depends on the number of the namenodes. Figure 6. shows the throughput of file system operations as a function of the number of namenodes and the size of the stored files.

For writing small files, the throughput of the file system linearly increases as more namenodes are added to the system. Using 22 namenodes, HopsFS++ can create more than ***80 thousand*** 1 KB files per second. For 4 KB files, HopsFS++ manages to write ***70 thousand*** files per second. We have observed that, for sizes beyond 4 KB, the performance of small files halves when the size of the small files is doubled. For these experiments, we had only six NVMe drives available, which would quickly become saturated as we were not only storing the replicated small files on the NVMe drives but also the undo and redo logs of the database on the same drives. HopsFS++ managed to write ***9 thousand*** 64 KB files per second using only 4 namenodes, and the throughput of the file system did not increase because the NVMe drives were fully saturated. Similarly, for reading the throughput of the file system linearly increases as more namenodes are added to the system. For 1 KB files, HopsFS++ is able to perform more than 400 thousand file read operations per second. Similar, to the file-write experiment, the throughput halves when the file size is doubled beyond the 4 KB file size.

Figure 7. shows the comparison of the throughput of file read and write operations, between HDFS and HopsFS++. These figures show the maximum average throughput that we achieved in our



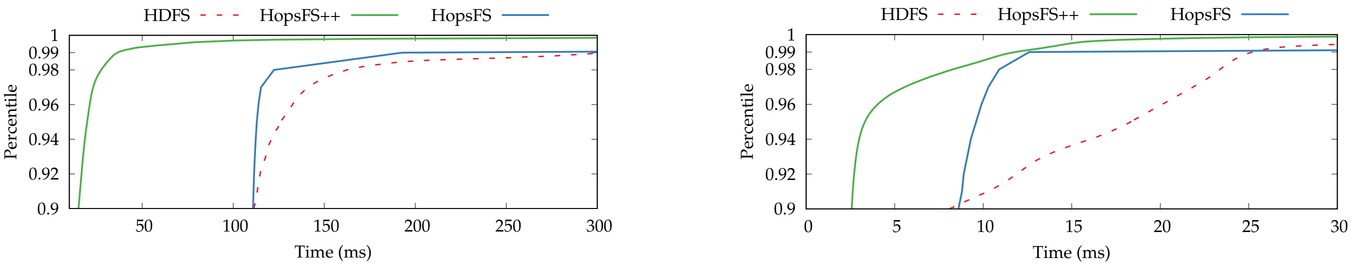
a. File Write Performance

b. File Read Performance

Figure 7: Comparison of the max throughput of HDFS and HopsFS++ for reading and writing small files. For writing HopsFS++ outperforms HDFS by *66.6 times* and *61.3 times* for small files of size 1 KB and 4 KB respectively. Similarly, for reading HopsFS++ outperforms HDFS by *4.5 times* and *4.2 times* for small files of size 1 KB and 4 KB respectively.

experiments. In our tests, for HDFS, we were unable to write more than ***1400*** files per second irrespective of the size of the small files. HDFS has poor performance for file write operations, due to its complicated write pipeline protocol that requires many internal RPCs to get set up. Additionally, in HDFS, the file system metadata consistency is maintained using multiple readers/single-writer lock, that is, each metadata operation that updates the namespace takes an exclusive lock on the entire namespace to update the metadata. This greatly impacts the performance of file write operations in HDFS. HopsFS++ does not have this problem as it does not lock the entire namespace in order to update a single file/block. In HopsFS++, all file system operations are implemented as transactions that lock only the metadata required for the file system operations. In HopsFS++, the throughput of the file system operations depends on the size of the database cluster and the number of the namenodes. For writing small files of size 1 KB and 4 KB, HopsFS++ outperforms HDFS by ***66*** and ***61 times***, respectively. Similar to previous experiments, the throughput halves when the file size is doubled. For small files of size 64 KB, HopsFS++ can write ***7.1 times*** more files per second than HDFS.

Similarly, for reading small files the performance of HDFS is limited by the single active namenode. For reading very small files, that is, for 1 KB files, HDFS operates at maximum throughput which is similar to the HDFS throughput as measured by the HDFS creators [34]. The throughput of HDFS drops to 37 thousand operations per second for 64 KB files. HopsFS++ outperforms HDFS by ***4.5*** and ***4.2 times*** for small files of size 1 KB and 4 KB, respectively. *We expect that the performance of HopsFS++ will increase with additional hardware, that is, with more NVMe drives, namenodes and database servers.*



a. Spofity Workload: File Write Op End–to–End Latency

b. Spofity Workload: File Read Op End–to–End Latency

Figure 8: End-to-end latency for file read and file write operations for real-world HDFS workload traces from Spotify. For 90th percentile, HopsFS++ has *7.39 times* and *3.15 times* lower operational latencies than HDFS/HopsFS for writing and reading small files, respectively.

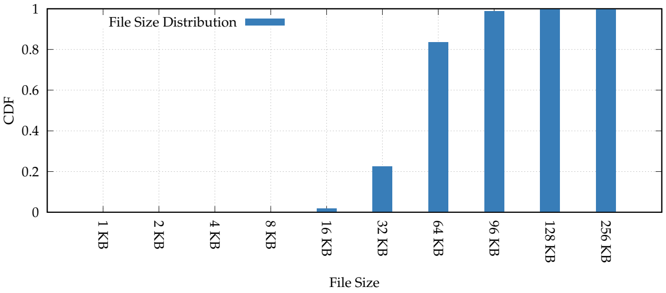


Figure 9: Open images dataset file size distribution. *83.5%* of the files in the dataset are ⩽ 64 KB.

6.2 Industrial Workloads

6.2.1 Small Files' Operational Latencies for Spotify Workloads In this experiment, we ran the workload traces from Spotify. The Spotify workload is described in detail in [8]. All the file system operations are performed on the small files. The workload created and accessed the files according to the statistics shown in Figure 1. In the Spotify workload, the percentage of file append operation is zero, that is, in case of HopsFS++ the small files' sizes did not grow and exceed the maximum small file size threshold limit to be moved from the small file storage layer to the large file storage layer. In these experiments, we only ran 20 namenodes for HopsFS++ and 25 datanodes in HDFS setup, that is, the number of servers for both the file system that stored and retrieved small files was 26 (HopsFS++ 20 NN + 6 NDB, HDFS 1 ANN + 25 DNs), and rest of the machines were used as file system clients to run the workload traces. In these experiments, the two file systems were run at 50% loads to accurately measure the end-to-end latency experienced by the file system clients.

Figure 8. shows the end-to-end latency of file read and write operations in the Spotify workload. HopsFS++ has a significantly lower end-to-end latency for reading and writing small files, this is due to simplified file system operation protocols for small files. For 90^{th} percentile, HopsFS++ has ***7.39 times*** and ***3.15 times*** lower operational latencies for writing and reading small files respectively for Spotify's workload traces.

6.2.2 Open Images Dataset: Mixed File Sizes Training machine learning models at scale is an increasing popular data-center workload. Reading/writing a mix of small and large files is a typical task in an image classification pipeline, where the images are first read and then transformed (rotated, warped) before being fed as training data to neural networks. In this experiment, we read and wrote the open images dataset containing 9 million files [35]. The Open Images Dataset, curated by Google, is frequently used to train convolutional neural networks for image classification tasks [35]. It is an extended version of the ImageNet dataset, widely used to benchmark image classification solutions. This dataset is particularly interesting as it contains both *large* and *small* files, based on our definition of a small file as being one smaller than 64 KB in size and a large file being larger than 64 KB in size. It is also of interest, because a real-world distributed deep learning application, reported by Facebook on the ImageNet dataset [36], read images at a rate of 40 thousand images/sec during training. With increased demand for even larger, more performant distributed deep learning systems, such systems will soon be able to process more files per second than is currently possible with HDFS.

The file size distribution for the Open Image Dataset is shown in Figure 9. In this experiment, the setup for HDFS is the same as the one described above. For HopsFS++, we used 10 namenodes and 12 datanodes to store large files blocks. In this experiment, HopsFS++ outperformed HDFS by a factor of ***5.9 times*** and ***4.5 times*** for both reading and writing the large dataset respectively, see Figure 10.

6.3 Small File Threshold

Although files of several megabytes in size can be stored in MySQL Cluster as blobs, in practice, a size threshold exists where small files are more efficiently stored in the large file storage layer instead of in the database. Real-time OLTP databases impose a limit on the maximum row size because large rows hog resources and can cause unrelated transactions to timeout due to starvation. In this experiment, we tested the performance of small files of different sizes to determine, for our experiment setup, the largest file that can be efficiently stored in the database than in the large file storage layer. We ran 100 file system clients that wrote varying size small files ranging from 1 KB to 1 MB. Using 100 clients both HDFS and HopsFS++ file systems were operating at approximately 50% loads. For HopsFS++, the number of namenodes was set to 20; and all the files, that is, files of size 1 KB to 1 MB were stored in the database. For HDFS, we ran 25 datanodes that stored all the files. Figure 11. shows the performance for the different file sizes, aggregated throughput and the average end-to-end latency observed by the file system clients.

For HDFS, the latency and the throughput of file write operations do not vary for small files because the file system metadata

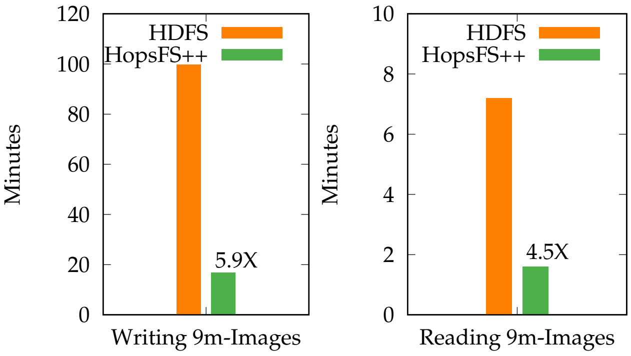
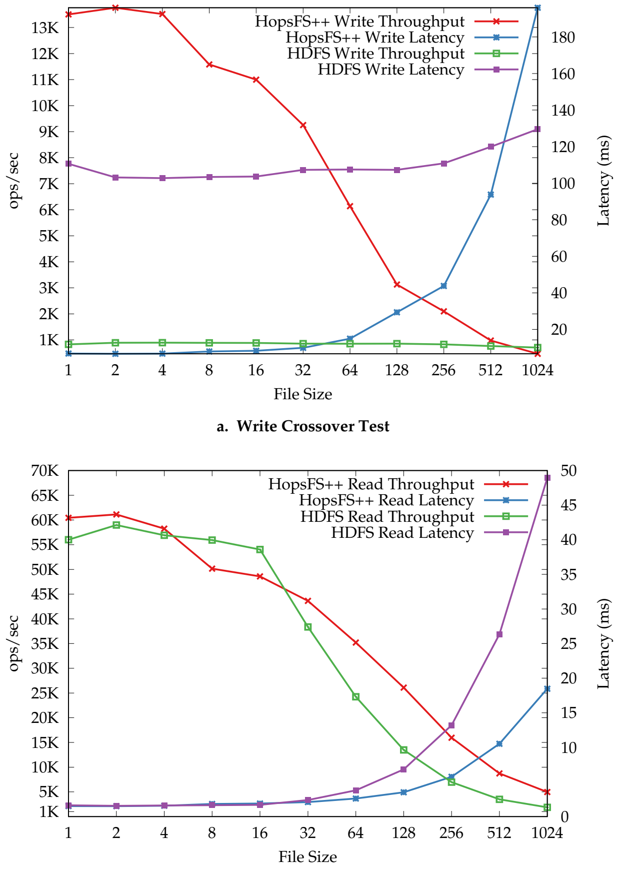


Figure 10: Performance of HDFS and HopsFS++ for reading and writing the 9 million files of the Open Images Dataset. For bulk reading the files, HopsFS++ is *4.5 times* faster than HDFS, while for bulk writing the files HopsFS++ is *5.9 times* faster than HDFS.



a. Write Crossover Test

b. Read Crossover Test

Figure 11: Establishing the small file size threshold (crossover point). At the threshold file size, it is better to store smaller files in the database, and larger files in the large file storage service of HopsFS++/HDFS.

operations take significantly longer than the time needed to write the data to the datanodes. This is due to multiple readers/single-writer concurrency mechanism in HDFS namenode that serializes all file system operations which update the namespace. At 50% load, HDFS managed to write ≈700 files per second, and the throughput remained the same for all small files of different sizes. For writing files, HopsFS++ has high throughput for very small files, such as files ranging from 1 KB to 4 KB, but the throughput gradually drops as the file sizes increases. The end-to-end latency for small files increases as the size of the files increases. Similarly, for read operations, the throughput drops and end-to-end latency increases for read operations as the file size is increased. Together, these results suggest that HopsFS++ can efficiently store small files up to 64 KB in size, using NDB as a storage layer for the small files.

7 Related Work

Walnut [37], 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 [38]. 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 MySQL Cluster to store stuffed inodes as on-disk columns in tables, WiscKey [39] 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 MySQL 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, 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 [40] 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 install on all the datanodes that manage the small files stored on the disk. TableFS [41] 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 [42], 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 [43].

HDFS provides archiving facility, known as Hadoop Archives (HAR), that compresses and stores the small files in large archives as a solution to reduce the contention on the namenode cause by the small files, see section 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 quickly access the contents of the small file stored in a large file [44]. MapR is a proprietary distributed file system that stores first 64 KB of all the files with the metadata [5], 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 [45], 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 MySQL Cluster is that, because its updates are made in-place, it has lower write-amplification than LSM-tree databases [24, 46], which can improve SSD device lifetime.

8 Conclusions

The poor performance of HDFS in managing small files has long been a bane of the Hadoop community. The main contribution of this paper is to show that a file system designed around large blocks (optimized to overcome slow random I/O on disks) can be transparently redesigned to leverage NVMe hardware (with fast random disk I/O) for small files. In this paper, we introduced a tiered file storage solution for small files in HopsFS++ that is fully compatible with HDFS. Our system 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 NVMe SSDs, and the biggest files are stored on spinning disks. We have implemented a distributed version of this architecture for HopsFS++ where very small files are stored in-memory in the back-end NewSQL database, MySQL Cluster, while other small files can be stored on NVMe SSD storage at database nodes. The large file storage service of HDFS/HopsFS++ remains unchanged, and we adapted our changes so that HDFS/HopsFS++ client compatibility has been retained. Through a mix of throughput and latency benchmarks on a Hadoop workload from Spotify, as well as a real-world workload of 9m small files, used in Deep Learning, we showed that HopsFS++ can deliver significant improvements in both throughput and latency, with the highest gains seen in writing files. We were limited in our available hardware, and are confident that with more servers, storage devices and tweaks to our software, the small file storage layer could produce even bigger performance gains.

Our small files extension to HopsFS++ is currently running in production, providing Hadoop-as-a-Service to hundreds of researchers, and is available as open-source software. We expect, also, our work to provide important feedback on improving the performance of on-disk data for MySQL Cluster. Finally, our expectation for the new improved HopsFS++ is that it will enable data parallel processing frameworks higher up in the stack (such as MapReduce, Apache Flink, and Apache Spark) to reimagine how they use the file system, now that creating and reading small files can both scale and be fast.

9 Acknowledgements

This work is funded by Swedish Foundation for Strategic Research projects "Continuous Deep Analytics (CDA) under grant agreement no. BD15-0006", "Smart Intra-body under grant agreement no. BD15-0006 RIT15-0119", and "EU H2020 Aegis project under grant agreement no. 732189".

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] "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].

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

[14] A. Foundation, "Apache Hadoop." https://hadoop.apache.org/. [Online; accessed 30-Aug-2017].

[15] 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].

[16] 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.

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

[18] K. V. Shvachko, "HDFS Scalability: The limits to growth," *login*, vol. 35, no. 2, pp. 6–16, 2010.

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

[20] 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.

[21] 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.

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

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

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

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

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

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

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

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

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

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

[32] 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.

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

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] J. Hendricks, R. R. Sambasivan, S. Sinnamohideen, and G. R. Ganger, "Improving small file performance in object-based storage," 2006.

[44] 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.

[45] 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.

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