# HopsFS-S3: Extending Object Stores with POSIX-like Semantics and more (industry track)

Mahmoud Ismail[†*] Salman Niazi[*] Gautier Berthou[*] Mikael Ronström[‡]
Seif Haridi[†*] Jim Dowling[†*]

[†] KTH - Royal Institute of Technology [*] Logical Clocks AB [‡] Oracle AB
{maism,haridi,jdowling}@kth.se {mahmoud,salman,gautier,seif,jim}@logicalclocks.com mikael.ronstrom@oracle.com

## Abstract

Object stores have become the de-facto platform for storage in the cloud due to their scalability, high availability, and low cost. However, they provide weaker metadata semantics and lower performance compared to distributed hierarchical file systems. In this paper, we introduce HopsFS-S3, a hybrid distributed hierarchical file system backed by an object store while preserving the file system's strong consistency semantics. We base our implementation on HopsFS, a next-generation distribution of HDFS with distributed metadata. We redesigned HopsFS' block storage layer to transparently use an object store to store the file's blocks without sacrificing the file system's semantics. We also introduced a new block caching service to leverage faster NVMe storage for hot blocks. In our experiments, we show that HopsFS-S3 outperforms EMRFS for IO-bound workloads, with up to 20% higher performance and delivers up to 3.4*X* the aggregated read throughput of EMRFS. Moreover, we demonstrate that metadata operations on HopsFS-S3 (such as directory rename) are up to two orders of magnitude faster than EMRFS. Finally, HopsFS-S3 opens up the currently closed metadata in object stores, enabling correctly-ordered change notifications with HopsFS' change data capture (CDC) API and customized extensions to metadata.

**ACM Reference format:**
Mahmoud Ismail, Salman Niazi, Gautier Berthou, Mikael Ronström, Seif Haridi, Jim Dowling. 2020. HopsFS-S3: Extending Object Stores with POSIX-like Semantics and more (industry track). In *Proceedings of 21st International Middleware Conference Industrial Track, Delft, Netherlands, December 7–11, 2020 (Middleware '20 Industrial Track),* 8 pages.
DOI: 10.1145/3429357.3430521


*Middleware '20 Industrial Track, Delft, Netherlands*

DOI: 10.1145/3429357.3430521

## 1 Introduction

Cloud service providers such as Amazon [9], Google [28], and Microsoft [37] offer object stores as a cheap, scalable, and highly available storage alternative to distributed hierarchical file systems [21, 25]. However, to yield scalability, these object stores offer a flat namespace and a weaker metadata semantics compared to POSIX-like distributed hierarchical file systems such as HDFS [45], and HopsFS [40]. These object stores typically provide REST APIs where objects (files) are identified by a key (path). For instance, Amazon S3 provides an eventually consistent semantics for operations such as read after write, directory listing, and rename objects or directories [2, 10, 22]. Alternative object stores such as Google Cloud Storage and Microsoft Azure blob store have strengthened the consistency semantics of the S3 APIs using a horizontally scalable and strongly consistent metadata layer [20, 27, 29]. However, they still lack the native support for atomic directory rename [15, 24, 44], which is a crucial operation for scalable SQL systems on Hadoop/Spark [11, 12, 23, 34]. Moreover, object stores offer change notification services that allow applications to react to change events on objects [3, 19, 26]. However, there are no ordering guarantees for events across objects, requiring application developers to implement their ordering on top [5, 16].

On the other hand, distributed hierarchical file systems offer POSIX-like file system semantics that ensures atomic operations such as directory rename. HopsFS [40] is a next-generation distribution of HDFS [45] with a distributed metadata layer allowing HopsFS to scale to more massive clusters with more metadata than HDFS. A HopsFS cluster consists of three main layers: the metadata storage layer, the metadata serving layer, and the block storage layer. The block storage layer is responsible for storing the file system's blocks on local disk volumes on the block storage servers. HopsFS provides a change data capture (CDC) API that produces correctly-ordered file system operations [36]. HopsFS lacks native support for object stores as a storage backend for the block storage layer, increasing its cost and limiting its adoption in cloud environments.

In this paper, we introduce HopsFS-S3 as a hybrid distributed hierarchical file system that offers POSIX-like file

system semantics while transparently using object stores as the backend storage for the file system's data. We leverage the heterogeneous storage APIs provided by HopsFS to implement a cloud storage type enabling fine-grained control over which part of the file system namespace to be stored in the cloud. Our implementation allows the block storage servers to be used as proxy servers to handle access to object stores in the cloud. To improve performance by avoiding unnecessary round trips to the object stores, we implemented a block cache on the block storage servers and a block selection policy on the metadata servers to ensure the locality of block data reads. Our caching layer exploits the relative higher performance of NVMe drives compared to object stores (such as S3). We also examined the different file system operations and redesigned them to ensure high performance and strong consistency on object stores. To the best of our knowledge, HopsFS-S3 is the first distributed hierarchical filesystem that supports tiered storage from small files in metadata [41], cached blocks on NVMe storage, and other blocks in object storage. Moreover, HopsFS-S3 provides a pluggable architecture allowing different object stores such as Amazon S3, Google cloud storage, and Azure blob store to be used as a storage backend. In this paper, we focus our discussion on Amazon S3 since it is the most widely used object store in the cloud. Finally, HopsFS-S3 leverage HopsFS to open up the currently closed metadata in object stores, enabling correctly-ordered change notifications with HopsFS' change data capture (CDC) and customized extensions to metadata [36, 40].

## 2 Background and Related work

HopsFS is an open-source next-generation distribution of HDFS that mitigates the HDFS scalability bottlenecks by storing the file system metadata in a distributed database [40]. In HopsFS, the metadata storage layer is a distributed database that is responsible for storing the file system's metadata, see Figure 1. HopsFS provides a pluggable architecture using the data access layer (DAL), allowing different distributed databases to be used, however, the default and recommended distributed database is NDB [38]. NDB is an in-memory, shared-nothing, distributed database. The metadata serving layer is responsible for executing parallel file system requests from potentially thousands of clients. The metadata servers are stateless, and they communicate only through the leader election protocol to elect a leader that is responsible for housekeeping operations of the file system [39, 40]. HopsFS leverages NVMe disks to store the small files, $< 128KB$, embedded in the metadata in the metadata storage layer [41]. The block storage layer is responsible for storing large files, $> 128KB$, where the files are split into blocks typically of $128MB$ size and replicated across block storage servers. HopsFS lacks support for object stores as a storage backend for the file system's data.

Amazon S3 does not implement a POSIX file system API. For scalability, it provides eventual consistency guarantees for many operations: objects (files) may not be immediately available after creation due to negative caching, and older versions of the objects may be available after an update or a delete operation [10]. Also, directory rename and directory delete operations are not atomic. In fact, Amazon S3 does not have directories per se. To address these limitations, connectors have been developed to strengthen the S3 API with POSIX-like semantics. Hadoop offers file system connectors to different object stores such as Amazon S3 using the S3A connector [32]. S3A is a file system connector that allows reading and writing to Amazon S3. S3A uses the S3Guard [44] to mitigate the issues introduced by the relaxed semantics of Amazon S3. Internally, S3Guard uses a consistent database (Amazon DynmaoDB [1]) to keep track of the metadata for the objects stored in Amazon S3 to improve performance of operations such as directory listing and file status. The S3A connector mimics a directory by adding it to the S3Guard, and for any upcoming operation, it will check all the keys that have this directory path as a prefix in the key. Directory rename is an essential operation that is used as part of the commit protocols for scalable SQL systems on Hadoop/Spark. That is why S3A implements a commit protocol that can work with S3 without introducing inconsistencies [31]

Amazon offers Elastic MapReduce (EMR) as a cloud-native big data platform to simplify running of big data frameworks such as Apache Hadoop and Apache Spark on AWS [8]. The storage layer of EMR supports the use of different file systems, including HDFS, the local file system connected to the instances, and the EMR file system (EMRFS) [6]. EMRFS is an implementation of HDFS that stores the files in Amazon S3. Similar to S3A, EMRFS implements a consistent metadata layer on top of S3 to mitigate the S3 relaxed consistency semantics [4]. Also, EMRFS implements an S3 optimized commit protocol, however, it only supports running Spark jobs that use Spark SQL, DataFrames, or Datasets to write Parquet files. [7].

Microsoft offers Azure Data Lake Store (ADLS) [17, 43] as a cloud storage service that builds upon features from hierarchical file systems and object stores, more specifically Azure Blob Storage. Similar to HopsFS, ADLS provides a hierarchical namespace, tiered storage, and atomic metadata operations such as directory rename and delete. Also, similar to HopsFS, ADLS uses a strongly consistent, relational, distributed database to manage the file system's metadata. ADLS provides a small append service to improve the performance of small appends (a few bytes to a few hundred KB), however, it does not tackle the storage problem of small files [18]. On the other hand, HopsFS embeds small files ($< 128Kb$, a configurable limit) within the file system's metadata [41].

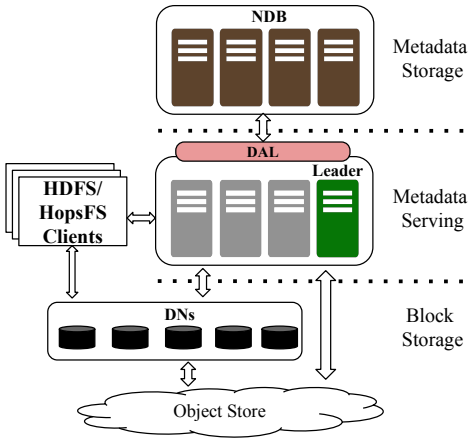


**Figure 1.** An architecture diagram of HopsFS-S3. The main difference between HopsFS and HopsFS-S3 is that the block storage servers can act as proxy servers for cloud object stores.

## 3 HopsFS-S3

We designed HopsFS-S3 as an extension to HopsFS to enable the use of object stores as a storage backend for the file system's data. There are two design alternatives either to allow the HopsFS clients to interact directly with the object store APIs or to use the block storage servers as proxy servers to object stores. The former design breaks the compatibility with HDFS clients, requires more maintenance, and raises some security concerns regarding giving the clients access to the object store directly. We favor the latter approach since it does not break the comparability of the current HDFS clients, and it is easier to implement and maintain its consistency using the metadata servers. Also, it enables the use of a pluggable architecture allowing plugging other object stores easily. HopsFS implements heterogeneous storage APIs similar to HDFS [30] to treat block storage servers (datanodes) as a collection of storage types such as DISK, SSD, and RAM_DISK. In HopsFS-S3, we implement a new storage type called "CLOUD" that leverages the heterogeneous storage APIs allowing users to set the storage policy to be "CLOUD" on a directory in the file system namespace. That is, all files under that directory will be stored in the cloud. Currently, Amazon S3 and Azure Blob Storage are supported; however, HopsFS-S3 offers a pluggable architecture allowing implementations of other object stores such as Google Cloud Storage.

Amazon S3 introduces the concepts of buckets, objects, and keys. A bucket is a container for objects stored in S3, and it has a unique global name. Objects are the basic entities in S3 that represent the users' uploaded data. An object is uniquely identified by its bucket name, key, and version, while the key is the unique identifier for an object within the bucket [10]. In HopsFS-S3, we added configuration parameters to allow users to provide their Amazon S3 bucket to be used as the block data store. Similar to HopsFS, HopsFS-S3 stores the small files, $< 128KB$, associated with the file system's metadata. For large files, $> 128KB$, HopsFS-S3 will store the files in the user-provided bucket. We identified the file system operations in HopsFS, see Section 3.1, then we redesigned those operations to work with object stores, more specifically Amazon S3, see Section 3.2

### 3.1 HopsFS operations

We divided the file system operations into two categories, *metadata operations*, and *data operations*. Metadata operations only interact with the file system's metadata without the need to read or write the actual data compared to the data operations. For example, *mkdir* is a metadata operation that creates a directory while *file read* is a data operation that reads file's content. HopsFS provides a strongly consistent file system semantics through the use of primitive locking and application-defined locking [40].

### 3.2 HopsFS-S3 operations

HopsFS-S3 uses the same metadata storage and serving layers as HopsFS. Therefore, the metadata operations are not affected by the new changes regarding data storage. However, we extended the file system's metadata to include information about each block, whether stored locally or in the cloud, and in which bucket it is stored. On the other hand, we redesigned the data operations to work with Amazon S3. S3 offers read-after-write consistency semantics for uploading new objects given that there was no get operation on the same key that happened shortly before uploading, otherwise eventual consistency semantics hold. That is any subsequent get operation to read the object might not return the object. Moreover, overwriting an existing object, deleting an object, and listing objects are eventually consistent. In HopsFS-S3, we maintain the same strong consistency as HopsFS. Therefore, we designed the file system data operations to enforce strong consistency on Amazon S3. That is, we ensure that all the objects stored in the S3 bucket are immutable. HopsFS-S3 implements variable-sized block storage to allow for any new appends to a file to be treated as new objects rather than overwriting existing objects.

In HopsFS-S3, the writing process proceeds similarly to HopsFS, where the file is split into blocks of fixed size, and the blocks are written to the block storage servers. However, instead of using the chain replication to replicate the blocks across 3 servers, we set the replication factor to 1 and use only one server that will transparently write the block to S3. The object stores maintain the fault tolerance and high availability of the stored objects. If the block storage server fails during a write operation, the client reschedules the write on a different live server. To read a file, clients in HopsFS-S3 proceed as in HopsFS by first requesting the set of block storage servers from a metadata server to start reading the file. However, since the file is stored in S3, metadata servers return either a block storage server with the cached blocks or a random block storage server that will transparently connect to S3 to return the requested blocks for the file to

the client. Each block storage server maintains a block cache where it saves the downloaded blocks to reduce the network overhead, see Section 3.2.1. The metadata servers keep track of the cached blocks on the block storage servers to allow for faster reads by sending the block storage servers with cached blocks to the client instead of a random block storage server. We also implement a synchronization protocol to ensure the consistency between the blocks stored in the cloud and the metadata stored in HopsFS-S3.

### 3.2.1 Block cache

The block storage servers act as proxy servers that transparently read/write objects to/from object stores (Amazon S3). We implement a least recently used (LRU) cache per block storage server to cache the blocks read from Amazon S3. The objects are immutable, and all the file system operations go through the metadata servers that use the metadata stored in the metadata storage layer to evaluate the operation. Thus, ensuring the strong consistency of the file system metadata. The block storage servers ensure the validity of the cache by first checking the existence of the block in the cloud before returning the cached block to the client. The metadata servers implement a block selection policy that ensures the locality of block read operations. The selection policy always favors choosing the block storage servers where the blocks are cached then random block storage servers.

## 4 Evaluation

In this section, we evaluate the performance of HopsFS-S3 against Amazon EMRFS. All the experiments were run on virtual machines of type c5d.4xlarge on Amazon EC2 with 16 vCPUs, 32 GB of memory, and 1 NVMe SSD disks (400 GB). HopsFS-S3 is based on HopsFS version 3.2.0, which is compatible with Hadoop version 3.2. We used Amazon EMR version 6.0.0 with Hadoop version 3.2.1. We created a cluster with 5 nodes (1 master node and 4 core nodes). The master node runs the metadata and resource management services, while the core nodes run the block storage and node management services. We used three different benchmarks to compare the performance of HopsFS-S3 and EMRFS. The Terasort benchmark provided by Hadoop [13, 42], TestDFSIOEnh provided by Hibench [33, 35], and the hdfs command-line tool. To ensure a fair comparison, we matched the configurations of the resource and node management services between HopsFS-S3 and EMRFS.

### 4.1 Terasort

The Terasort benchmark is a MapReduce based sorting algorithm that has been used by Hadoop clusters to compete in the annual sorting competition [46] to sort 1 TB of data. The program consists of three main MapReduce jobs; Teragen, Terasort, and Teravalidate. The Teragen program is responsible for generating the input data to be sorted. The Terasort program does the actual sorting of the input data and then

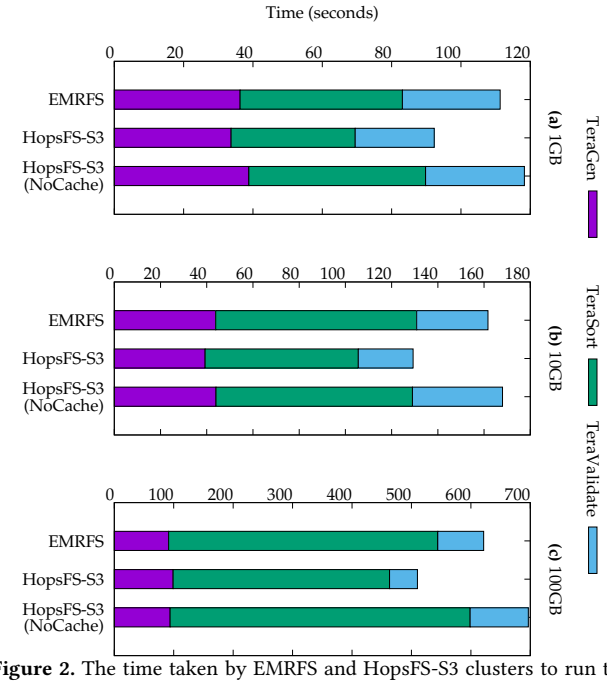


**Figure 2.** The time taken by EMRFS and HopsFS-S3 clusters to run the Terasort benchmark for input data sizes (1GB, 10GB, and 100GB)

writes the sorted data back. The Teravalidate program checks the sorted results to ensure the total order of the data. We ran the Terasort benchmark on EMRFS and HopsFS-S3 clusters for different input data sizes (1GB, 10GB, and 100GB). For HopsFS-S3, we ran two configurations enabling and disabling the block cache on block storage servers. Then, we calculated the time taken by each stage of the Terasort benchmark, as shown in Figure 2. HopsFS-S3 with block cache enabled delivers lower running time compared to EMRFS by 17% for 1GB, 20% for 10GB, and 18% for 100GB. The main reason for the performance increase of HopsFS-S3 is that the block reads will only go to S3 if the block is not present locally in the block cache of the block storage servers, which we can confirm using the utilization figures in Section 4.1.1. HopsFS-S3 ensures the selection of block storage servers with locally cached blocks when serving client read requests. Figure 2 also shows that HopsFS-S3 with cache disabled have a higher running time compared to EMRFS by 6% for 1GB, 4% for 10GB, and 12% for 100GB. The main reason for the higher running time is the indirection introduced by HopsFS-S3 to read the blocks through the block storage servers, which act as proxy servers for S3.

#### 4.1.1 Utilization

We collected the master and core nodes' utilization data when running the Terasort benchmark using an input data size of 100GB. Figure 3(a) shows the average CPU utilization on the master node. We can see that the master nodes are hardly doing any work since all the work happens on the core nodes. Figure 3(b) shows the average CPU utilization on the core nodes for different stages of the Terasort benchmark.

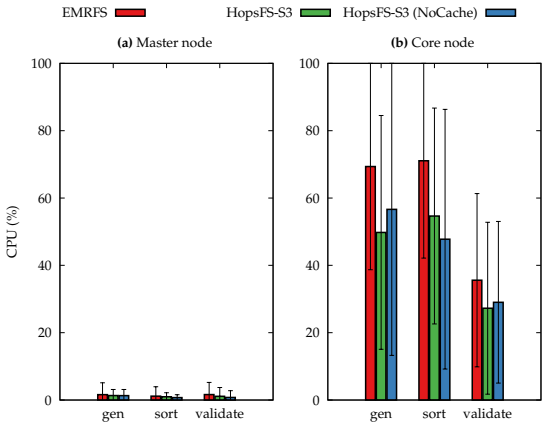


**Figure 3.** The average CPU utilization on master and core nodes for different stages of the Terasort benchmark using 100GB of input data.

EMRFS has a higher CPU load on the core nodes compared to HopsFS-S3, whether the block cache is enabled or disabled.

Figure 4(a) shows the average network write throughput on the core nodes. HopsFS-S3 and EMRFS have a similar write throughput for all three stages. On the other hand, HopsFS-S3 with cache enabled have a lower network read throughput compared to EMRFS due to the use of the block cache on the block storage servers of HopsFS-S3, as shown in Figure 4(b). Figure 4(c) shows the average disk write throughput on the core nodes for HopsFS-S3 and EMRFS. HopsFS-S3(NoCache) has significantly higher throughput on the Teravalidate stage compared to EMRFS and HopsFS-S3 with cache enabled. The reason is that when HopsFS-S3(NoCache) reads blocks, it always downloads the blocks from S3 and writes them to disk before sending them back to the client. HopsFS-S3 has a higher disk read throughput compared to EMRFS and HopsFS-S3(NoCache), as shown in Figure 4(d), due to the use of the block cache on the block storage servers. Figure 5 shows the average disk and network reading/writing throughput on the master node for all different stages of the Terasort benchmark. Both HopsFS-S3 and EMRFS have a low network and disk utilization, less than $1MB/sec$.

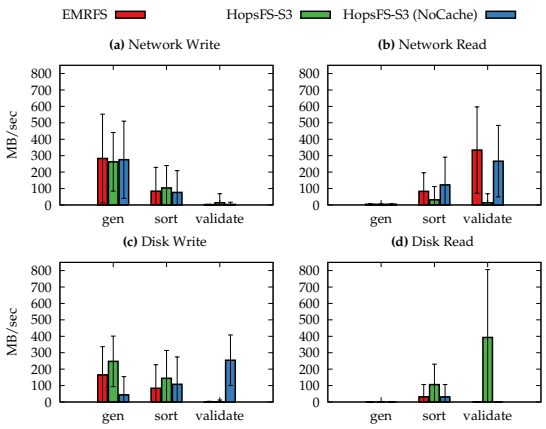


**Figure 4.** The average Disk and Network utilization on core nodes for different stages of the Terasort benchmark using 100 GB of input data.

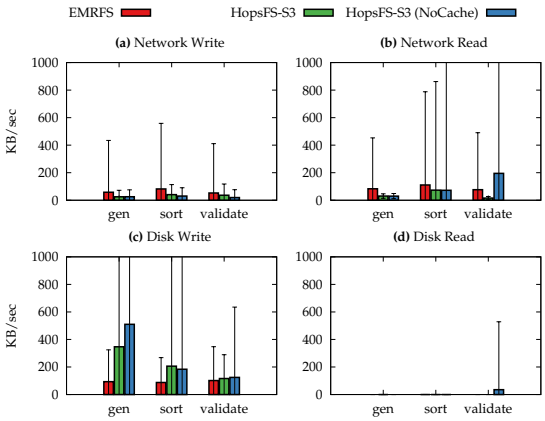


**Figure 5.** The average Disk and Network utilization on master node for different stages of the Terasort benchmark using 100 GB of input data.

## 4.2 TestDFSIOEnh

In this experiment, we used the enhanced DFSIO benchmark provided by the Hibench benchmarking suite [35]. The benchmark creates a set of map tasks that are, in parallel, writing/reading files to/from HopsFS-S3 and EMRFS clusters, and then records the total time taken, average throughput per map task, and the average aggregated throughput of the cluster. We ran the TestDFSIOEnh using $1GB$ files while varying the number of concurrent map tasks (16, 32, 64).

HopsFS-S3 takes almost the same amount of time as EMRFS to write files at a low concurrency level (16); however, the time increases by 20% when running 32 concurrent tasks, and 10% when running 64 concurrent tasks, as shown in Figure 6(a). The reason for the time increase is that the metadata server in EMRFS writes the data directly to S3 while, in the case of HopsFS-S3, the metadata server redirects the write to the block storage servers, which in turn writes the data to S3. On the other hand, HopsFS-S3 takes less time to read files, by up to 54% compared to EMRFS, as shown in Figure 6(b). Figure 7(a) shows that HopsFS-S3 has a lower average aggregated throughput, by up to 39% when writing files, compared to EMRFS. However, for HopsFS-S3 (NoCache), we show that HopsFS-S3 average aggregated throughput is almost the same as EMRFS and even higher when running 64 concurrent tasks. That is due to the high variability of the benchmark aggregated throughput results, as shown in the error bars, which is not that case when looking at the actual average

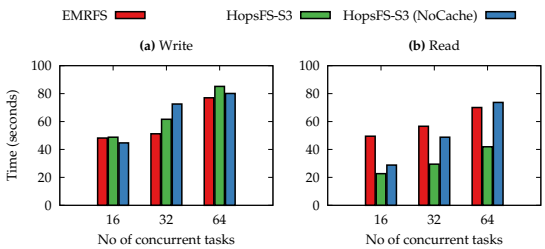


**Figure 6.** The total execution time taken by enhanced DFSIO tasks to concurrently write and read files of size 1GB to HopsFS-S3 and EMRFS clusters.

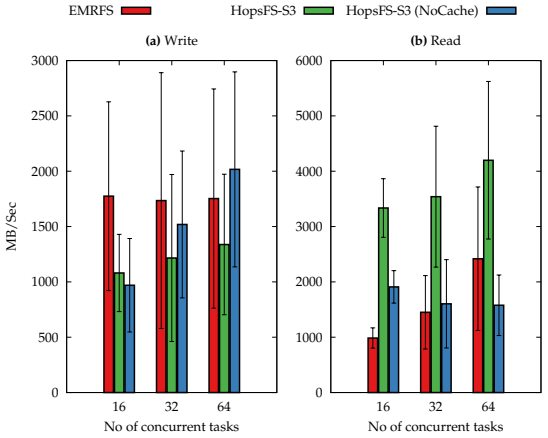


**Figure 7.** The average aggregated throughput of HopsFS-S3 and EMRFS clusters to write and read files of size 1 GB using the enhanced DFSIO benchmark.

write throughput per map task as shown in Figure 8(a). On the other hand, Figure 7(b) shows that HopsFS-S3 has a higher average aggregated throughput when reading files by up to 3.4$X$ times compared to the throughput of EMRFS at low concurrency levels, and it decreases to 1.7$X$ at higher concurrency levels. Similarly, Figure 8(b) shows the average read throughput per map task for both clusters.

### 4.3 Metadata operations

In this experiment, we used the enhanced DFSIO to create directories with 1000 and 10,000 files. Then, we used the HDFS command line tool [14] to run directory listing and rename on those directories, and recorded the average time taken by each operation. Notice that the time reported includes the startup time of the JVM. Figure 9(a) shows that HopsFS-S3 executes directory rename in two orders of magnitude lower time than EMRFS. The main reason for the huge performance gap is that EMRFS does not support directory rename. Instead, it does an expensive move operation to rename all the directory's descendent children and their

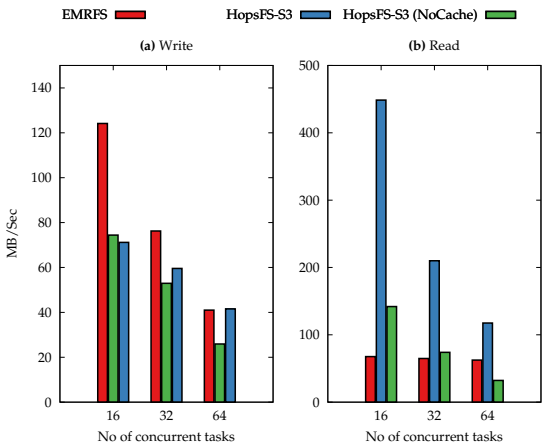


**Figure 8.** The average throughput per map task of HopsFS-S3 and EMRFS clusters when writing and reading files of size 1 GB using the enhanced DFSIO benchmark.

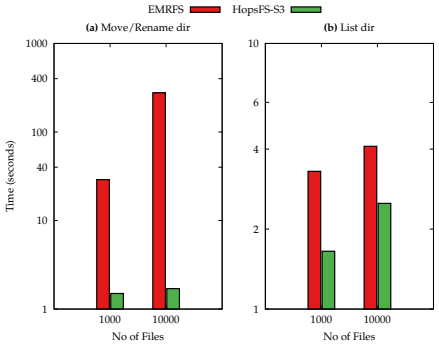


**Figure 9.** The average time taken by HopFS-S3 and EMRFS to execute directory listing and directory rename on directories with different number of children. The Y-axis is in log scale base 10.

transitive children. However, in HopsFS-S3, the directory rename operation is a metadata operation that only changes the metadata of the directory itself. Figure 9(b) shows that HopsFS-S3 takes, on average, 50% the time taken by EMRFS to execute the directory listing. The directory listing operation is a metadata operation in HopsFS-S3 and EMRFS; that is, it does not require contacting S3. HopsFS-S3 retrieves the directory's children from the metadata storage layer, while EMRFS retrieves this information from the metadata table in DynamoDB. Due to space considerations and the existence of previous published performance figures [41], we have not included experiments comparing small file performance in HopsFS-S3 and EMRFS. Given that small file operations in HopsFS-S3 are metadata operations, they again significantly outperform small file operations in S3.

## 5 Conclusions

We introduced HopsFS-S3, a hybrid cloud-native distributed hierarchical file system that allows the use of object stores as a storage backend for the file system's data without sacrificing the file system's consistency. HopsFS-S3 provides fine-grained APIs allowing users to enable cloud storage policy on a per-directory basis. Also, HopsFS-S3 offers a pluggable architecture allowing implementations using other object stores. In our experiments, we show that HopsFS-S3 outperforms EMRFS by up to 20% when running Tereasort benchmarks. Also, we show that HopsFS-S3 delivers up to 3.4$X$ the aggregated read throughput of EMRFS. Moreover, we demonstrate that directory listing operations on HopsFS-S3 are up to 50% faster than on EMRFS, and directory rename operations are two orders of magnitude faster than EMRFS. To the best of our knowledge, HopsFS-S3 is the first hierarchical distributed POSIX-like filesystem with multi-tiered file storage at metadata, block cache, and object store layers, as well as customizable metadata.

## Acknowledgements

This work was funded by the EU Horizon 2020 project Human Exposome Assessment Platform (HEAP) under Grant Agreement no. 874662.

## References


[1] Amazon 2020. Amazon DynamoDB. https://aws.amazon.com/dynamodb. (2020). [Online; accessed 12-Mar-2020].

[2] Amazon 2020. Amazon S3. https://aws.amazon.com/s3. (2020). [Online; accessed 12-Mar-2020].

[3] Amazon 2020. Configuring Amazon S3 event notifications. https://docs.aws.amazon.com/AmazonS3/latest/dev/NotificationHowTo.html. (2020). [Online; accessed 4-Sep-2020].

[4] Amazon 2020. EMRFS Consistent View. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-consistent-view.html. (2020). [Online; accessed 12-Mar-2020].

[5] Amazon 2020. Event message structure. https://docs.aws.amazon.com/AmazonS3/latest/dev/notification-content-structure.html. (2020). [Online; accessed 4-Sep-2020].

[6] Amazon 2020. Use EMR File System (EMRFS). https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-fs.html. (2020). [Online; accessed 12-Mar-2020].

[7] Amazon 2020. Using the EMRFS S3-optimized Committer. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-s3-optimized-committer.html. (2020). [Online; accessed 4-Sep-2020].

[8] Amazon 2020. What Is Amazon EMR? https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html. (2020). [Online; accessed 12-Mar-2020].

[9] Amazon EC2 2019. Amazon EC2. https://aws.amazon.com/ec2/. (2019). [Online; accessed 5-Jan-2019].

[10] Amazon S3 Consistency Model 2019. Amazon S3 Consistency Model. https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html. (2019). [Online; accessed 5-Jan-2019].

[11] Apache 2019. Apache Hudi: Upserts And Incremental Processing on Big Data. http://hudi.apache.org/. (2019). [Online; accessed 12-Sep-2019].

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

[13] Apache 2020. Apache hadoop examples: Terasort benchmark. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html. (2020). [Online; accessed 4-Sep-2020].

[14] Apache 2020. HDFS Commands Guide. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html. (2020). [Online; accessed 4-Sep-2020].

[15] Azure 2019. Hadoop Azure Support: Azure Blob Storage. https://hadoop.apache.org/docs/r3.1.2/hadoop-azure/index.html#Atomic_Folder_Rename. (2019). [Online; accessed 12-Sep-2019].

[16] Azure 2020. Azure Blob Storage as an Event Grid source. https://docs.microsoft.com/en-us/azure/event-grid/event-schema-blob-storage. (2020). [Online; accessed 4-Sep-2020].

[17] Azure 2020. Introduction to Azure Data Lake Storage Gen2. https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction. (2020). [Online; accessed 4-Sep-2020].

[18] Azure 2020. Optimize Azure Data Lake Storage Gen2 for performance. https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-performance-tuning-guidance#file-size. (2020). [Online; accessed 4-Sep-2020].

[19] Azure 2020. Reacting to Blob storage events. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-event-overview. (2020). [Online; accessed 4-Sep-2020].

[20] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, and others. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 143–157.

[21] Databricks 2020. HDFS vs. Cloud Storage: Pros, cons and migration tips. Top5ReasonsforChoosingS3overHDFS. (2020). [Online; accessed 11-Mar-2020].

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.

[23] Delta 2019. Delta Lake: Reliable Data Lakes at Scale. https://delta.io/. (2019). [Online; accessed 12-Sep-2019].

[24] Google 2019. Google Cloud Storage: mv - Move/rename objects. https://cloud.google.com/storage/docs/gsutil/commands/mv. (2019). [Online; accessed 12-Sep-2019].

[25] Google 2020. HDFS vs. Cloud Storage: Pros, cons and migration tips. https://cloud.google.com/blog/products/storage-data-transfer/hdfs-vs-cloud-storage-pros-cons-and-migration-tips. (2020). [Online; accessed 11-Mar-2020].

[26] Google 2020. Pub/Sub notifications for Cloud Storage. https://cloud.google.com/storage/docs/pubsub-notifications. (2020). [Online; accessed 4-Sep-2020].

[27] Google Cloud Storage Consistency 2019. Google Cloud Storage Consistency. https://cloud.google.com/storage/docs/consistency. (2019). [Online; accessed 5-Jan-2019].

[28] Google Compute Engine 2019. Google Compute Engine. https://cloud.google.com/compute/. (2019). [Online; accessed 5-Jan-2019].

[29] GoogleCloudPlatform 2019. How Google Cloud Storage offers strongly consistent object listing thanks to Spanner. https://cloud.google.com/blog/products/gcp/how-google-cloud-storage-offers-strongly-consistent-object-listing-thanks-to-spanner. (2019). [Online; accessed 1-Jul-2019].

[30] Hadoop 2020. Archival Storage, SSD, and Memory. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html. (2020). [Online; accessed 12-Mar-2020].

[31] Hadoop 2020. Committing work to S3 with the "S3A Committers". https://hadoop.apache.org/docs/current/hadoop-aws/tools/hadoop-aws/committers.html. (2020). [Online; accessed 12-Mar-2020].

[32] Hadoop 2020. Hadoop-AWS module: Integration with Amazon Web Services. https://hadoop.apache.org/docs/current/hadoop-aws/tools/hadoop-aws/index.html. (2020). [Online; accessed 12-Mar-2020].

[33] HiBench 2020. HiBench Github repository. https://github.com/Intel-bigdata/HiBench. (2020). [Online; accessed 4-Sep-2020].

[34] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 1235–1246.

[35] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 41–51.

[36] M Ismail, M Ronstrom, S Haridi, and J Dowling. 2019. ePipe: Near Real-Time Polyglot Persistence of HopsFS Metadata. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 92–101.

[37] Microsoft Azure 2019. Microsoft Azure. https://azure.microsoft.com. (2019). [Online; accessed 5-Jan-2019].

[38] MySQL Cluster CGE 2018. MySQL Cluster CGE. http://www.mysql.com/products/cluster/. (2018). [Online; accessed 5-Jan-2018].

[39] Salman Niazi, Mahmoud Ismail, Gautier Berthou, and Jim Dowling. 2015. Leader Election Using NewSQL Database Systems. In *Proceedings of the 15th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 9038*. 158–172.

[40] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 89–104.

[41] Salman Niazi, Mikael Ronström, Seif Haridi, and Jim Dowling. 2018. Size Matters: Improving the Performance of Small Files in Hadoop. In *Proceedings of the 19th International Middleware Conference (Middleware '18)*. 26–39.

[42] Owen O'Malley. 2008. *Terabyte sort on apache hadoop.* Technical Report. Yahoo.

[43] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 51–63. DOI: https://doi.org/10.1145/3035918.3056100

[44] S3Guard 2019. S3Guard: Consistency and Metadata Caching for S3A. https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/s3guard.html. (2019). [Online; accessed 5-Jan-2019].

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 1–10.

[46] Sort 2020. Sort Benchmark. http://sortbenchmark.org/. (2020). [Online; accessed 4-Sep-2020].