FileScale: Fast and Elastic Metadata Management for Distributed File Systems

Gang Liao*
ByteDance Infrastructure System Lab
San Jose, California, USA
gangliao@bytedance.com

Daniel J. Abadi
University of Maryland
College Park, Maryland, USA
abadi@umd.edu

ABSTRACT

File systems that store metadata on a single machine or via a shared-disk abstraction face scalability challenges, especially in contexts demanding the management of billions of files. Recent work has shown that employing shared-nothing, distributed database system (DDBMS) for metadata storage can alleviate these scalability challenges without compromising on high availability guarantees. However, for low-scale deployments – where metadata can fit in memory on a single machine – these DDBMS-based systems typically perform an order of magnitude worse than systems that store metadata in memory on a single machine. This has limited the impact of these distributed database approaches, since they are only currently applicable to file systems of extreme scale.

This paper describes FileScale, a three-tier architecture that incorporates a DDBMS as part of a comprehensive approach to file system metadata management. In contrast to previous approaches, FileScale performs comparably to the single-machine architecture at a small scale, while enabling linear scalability as the file system metadata increases[1].

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.

KEYWORDS

Distributed File System, Metadata Management, Elastic Computing, Distributed Database

*Work performed during PhD at University of Maryland, College Park
[1]The code is currently available at https://github.com/umd-dslam/FileScale

SoCC '23, November 7–9, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624784

ACM Reference Format:
Gang Liao and Daniel J. Abadi. 2023. FileScale: Fast and Elastic Metadata Management for Distributed File Systems. In *ACM Symposium on Cloud Computing (SoCC '23), November 7–9, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624784

1 INTRODUCTION

As the data stored by organizations rapidly expands, both the structured metadata and unstructured byte contents of the files managed within file systems scale commensurately. In general, it is easier to scale the unstructured data than the structured data, since there is no requirement to perform atomic transactions that update the unstructured bits across multiple files. Therefore, unstructured data can simply be placed in blocks that are partitioned across a shared-nothing cluster of nodes (machines), and all operations over this data can be done in parallel across this cluster, with little-to-no coordination across nodes except for replication.

However, scaling the structured data is more challenging: First, there is a requirement for atomic, isolated, and durable transactions that may access data in multiple partitions. For example, recursively deleting or changing the permissions of a directory affect that directory and all its sub-directories, and must occur atomically. Similarly, moving or copying directories, may span multiple partitions, and also must occur atomically and serializably. Second, metadata is repeatedly accessed throughout file system requests for verifying paths, checking permissions, and finding relevant data, and cannot afford excessive delays for multi-node coordination.

The first generation of scalable file systems, such as GFS, HDFS, Lustre, Ursa Minor, Farsite, and XtreemFS [19, 20, 28, 31, 45, 46], focused on scaling the unstructured data linearly, but stored metadata in memory on a single machine. They scaled to petabytes of data by using block sizes on the order of megabytes or gigabytes, and limiting the number of unique files and directories under management, so that information about blocks, files and directories can fit in memory on the metadata node. These restrictions are acceptable for data processing and large scale analysis workloads, which typically involve large scans and prefer large block sizes anyway. However, they are problematic for workloads that access

data in smaller quantities. In addition, even those workloads that use large blocks sizes are reaching the metadata limits of existing scalable file systems with increasing frequency.

Furthermore, this single metadata server becomes a bottleneck when it is overwhelmed by many concurrent client requests, along with processing heartbeats from the increasingly large numbers of block-store servers in the system [47]. It also becomes a single point of failure unless a fail-over machine that has identical provisions of copious memory and processing runs alongside it. Therefore, solutions that remove the memory limitations by incorporating fast external storage attached to the metadata node (e.g. [14, 21, 22, 34, 43, 44, 49, 52, 54]) will not be sufficient in the long run.

One approach to scaling metadata is to partition it, but restrict atomicity and isolation guarantees to only those requests that can be processed by a single partition. This is the approach taken by HDFS's federation option [5, 10] where the file system namespace is statically partitioned across completely independent "NameNode" servers that store disjoint partitions of file system metadata, with optional client-side routing tables [11] or a routing layer [4, 8, 9, 38] that direct metadata requests to the correct NameNode. Nonetheless, preventing multi-partition requests limits the general applicability of these approaches, and reduces the functionality of the file system. In one case study, Facebook stated that they needed "tens of HDFS clusters per datacenter to store analytics data", a situation that was "operationally inefficient" as even "single data warehouse datasets are often large enough to exceed a single HDFS cluster's [metadata] capacity" [40]. Similarly, ByteDance ran into HDFS metadata scalability problems, and likewise rejected HDFS federation due to the lack of atomic transactions across namespaces [26].

An alternative approach is to store the metadata in a distributed database system (DDBMS) that manages the partitioning, and guarantees atomicity, isolation, and durability of all transactions — even those that span partitions [39, 42, 50][2]. These approaches have demonstrated that scalable DDBMSs can successfully scale all aspects of file system metadata management. However, performance and efficiency can be a problem. When the file system logic is running outside of the DDBMS, there are typically many round trips between the file system logic and database layer for each file system request. These round trips can add up to substantial increased latency, and reduced efficiency of system resources. In one case, it was reported that it took 3 NameNodes and 2 database servers to match the throughput that the single active HDFS NameNode is able to achieve [39]. On the other hand, building the file system logic into the DDBMS requires rip-and-replace upgrades of existing file system technology and has yet to be shown to be a generally applicable approach.

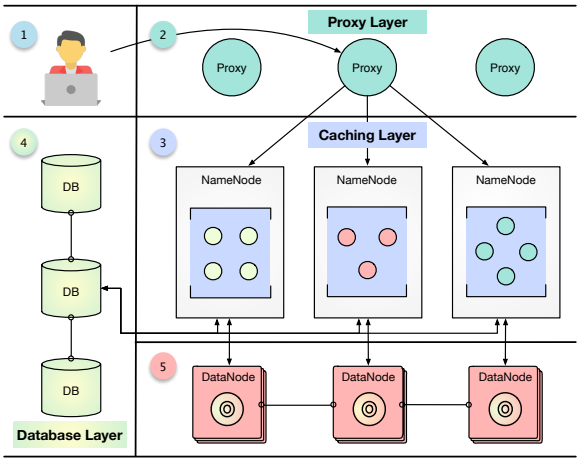


Figure 1: System Architecture of FileScale.

In this paper, we describe the design of FileScale, an HDFS-based file system that replaces metadata management in HDFS with a three-tiered distributed architecture that incorporates a DDBMS at the lowest layer, along with distributed caching and routing functionality above it, so that most requests can be served with asynchronous, batched interactions with the DDBMS. This architecture enables a simple drop-in upgrade of existing HDFS implementations in which all interfaces — both internally and externally — remain the same, and the performance on a single node is nearly identical to the original HDFS implementation. However, as the metadata scales, the architecture partitions the metadata over a shared-nothing cluster, achieving linear scalability relative to performance on a single node.

2 HDFS BACKGROUND

HDFS is perhaps the most widely deployed distributed file system today for machine learning and data analytics [18, 24, 53]. It uses a leader/follower architecture in which a NameNode manages all file system metadata and regulates data access on behalf of clients. Files are split into one or more blocks, and these blocks are replicated across a set of DataNodes in a shared-nothing architecture.

NameNode durability is implemented via a write-ahead log called the `EditLog`. Recovery is performed by loading a checkpoint called a `FSImage` and then replaying the `EditLog` over this image file. This process can be time consuming. Therefore, for improved high availability, HDFS allows for the deployment of a hot-standby that continuously, asynchronously, keeps the `FSImage` merged with the `EditLog`, so that it can take over with only minor delay when the primary NameNode crashes or temporarily goes down.

[2]Although Colossus[29] and Giraffa [48] use scalable data stores (BigTable [23] and HBase [2]), they do not support multi-partition requests because they lack strongly consistent distributed transactions.

inode2block

block-id	id	index
1073741825	16386	0
1073741826	16386	1
1073741827	16386	2
1073741828	16388	0
1073741829	16389	0

datablocks

block-id	kbytes	stamp	replica
1073741825	131072	1001	1
1073741826	131072	1002	1
1073741827	45056	1003	1
1073741828	6.6	1004	1
1073741829	1628.2	1005	1

block2storage

block-id	idx	storage-id
1073741825	0	DS-e3d5de23
1073741826	0	DS-e3d5de23
1073741827	0	DS-08989547
1073741828	0	DS-dc8aa54e
1073741829	0	DS-dc8aa54e

inodes

id	pid	pname	name	access-time	update-time	header	permission
16385	0	null	/	0	1545261571024	0	1099511693805
16386	16385	/	event_data	1545267685278	1545264231090	281474976710672	1099511693823
16387	16385	/	dnn_model	0	1545267685104	0	1099511693805
16388	16386	/dnn_model	graph.ckpt.pbtxt	1545267685125	1545267685125	281474976710672	1099511693823
16389	16386	/dnn_model	model.ckpt.data0	1545267685224	1545267685224	281474976710672	1099511693823

Table 1: Data model in FileScale.

3 SYSTEM ARCHITECTURE

FileScale is designed to serve as a drop-in replacement for HDFS, maintaining an identical client API, and intercepting communication with the HDFS NameNode and redirecting it to FileScale's more scalable, distributed NameNode implementation. FileScale uses a three-tiered architecture that implement routing, caching, and stable storage of metadata.

The high level architectural design of FileScale is illustrated in Figure 1. When a client (1) makes a request, a proxy server (2) receives the request and routes it to a NameNode based on requested file paths. The NameNode (3) functions as a cache of a subset of metadata. If the metadata relevant to the request is currently in the cache of the NameNode that receives it, it can respond immediately. Otherwise, either the relevant data is brought into cache, or (4) this request is forwarded and processed as a transaction in the database layer. The results of the request are then returned to the client, which typically contain locations of DataNodes where the raw data is stored. The DataNode (5) code in FileScale is identical to the DataNode code in HDFS. The following sections 4, 5 and 6 provide more detail on each layer.

4 DATABASE LAYER

FileScale stores all file system metadata in a DDBMS that is partitioned and replicated across a shared-nothing cluster. Metadata operations are performed as atomic transactions over the DDBMS. FileScale uses a modular architecture such that any ACID-compliant SQL DDBMS could be used. The metadata component of most file system commands can be transformed into a series of simple INSERT, UPDATE, or SELECT statements over the database system. However recursive operations are more complicated. FileScale requires that the DDBMS either directly supports recursive operations, or otherwise supports generic stored procedures so that these recursive operations can be implemented inside the DDBMS without paying an additional round trip to the DDBMS for each recursive step of the operation. Both of these options typically require some new code to be added to be able to support a new DDBMS. The codebase currently supports VoltDB [17] and Apache Ignite [3].

File systems typically store metadata as a tree of inodes with a root corresponding to the root directory, and children corresponding to directories and files located in the parent directory. Files are leaves of this tree (i.e. they have no children) and they point to data block references from which the data associated with this file can be read. In HDFS this entire tree is stored in the memory of the NameNode.

FileScale transforms the inode tree into a relational schema that contains 14 tables. This includes tables for the two main entities: `inodes` and `datablocks`, along with several relationship tables such as mappings from inodes to blocks, and blocks to storage locations (DataNodes). Table 1 shows a simplified version of the FileScale schema. The *pid* and *pname* attributes of the `inodes` table enables the reconstruction of the parent-child relationships from the original tree.

Recent work that scales metadata management in file systems by storing data in database systems or LSMs do not store the full path within an inode tuple. Instead, parents and children are referred to by their unique inode IDs [39, 43, 44, 54]. This approach has two advantages: (1) space savings and (2) faster rename operations (only the root of the renamed branch needs to be modified). In contrast, FileScale not only stores full path names in each inode tuple, but even makes the path (`pname`, `name`) the primary key. This approach yields a different set of advantages. First, it improves modularity since it requires less support for recursion in the underlying database system. By storing full paths in the inodes, simple WHERE clauses containing prefix matching operations on

the full path can be used to directly find all nodes that are part of a particular branch. This avoids a recursive traversal of the inode tree. Second, there is no need to maintain an in-memory mapping between inode IDs and paths, since the paths themselves are the IDs. Although storing full paths requires more space, FileScale's horizontal scalability makes the additional storage requirements less problematic. Renames in FileScale are implemented by performing the above described prefix matching in a WHERE clause.

All tables that have 1:n relationships with the `inodes` table, such as `datablocks`, are partitioned based on their association with `inodes`, in order to maximize locality. The remaining (small) tables are replicated across the cluster.

5 CACHING LAYER

In theory, purpose-building a new DDBMS that can be naively integrated with FileScale would yield optimal performance. In practice, however, it is well-known that purpose-building anything for a specific application yields big performance benefits in the short term, but generally fails to keep up with technology developments as research progresses. In general it is preferable to use commodity components that can be switched out and replaced with newer and more advanced versions as they become available. This is especially important in the context of FileScale in which the DDBMS performs multi-partition transactions. Multi-partition transactions are notoriously challenging to perform with high performance and high isolation and consistency guarantees simultaneously and research in this area is currently very active with new developments being made on an ongoing basis. FileScale is thus designed to use off-the-shelf distributed database systems instead of using a purpose-built native system.

However, the downside of building on top of an external system is the overhead involved in forming and sending a request to the external system and receiving, parsing, and processing the response. FileScale must thus be designed to avoid excessive calls to the database. This is done via implementing a cache layer in each NameNode's memory that enables a copy of a set of metadata objects (such as inodes) to be stored in local memory, which can be accessed directly by metadata operations and thereby avoid communication with the database system upon a cache hit. Updates to metadata stored inside a FileScale cache are not propagated to the underlying database system until an event occurs that requires propagation, such as an expiration, periodic flush, or distributed transaction. Thus, the database layer lags behind the cache layer, and up-to-date access to records in the database layer may require synchronization activities with the cache layer prior to serving those accessed records.

5.1 Object Cache

The mappings of files to blocks and blocks to DataNodes in HDFS's namespace are implemented as a light-weight hash table in HDFS whose primary goal is to optimize memory usage within the NameNode [1] so that the entire metadata can fit in memory. In contrast, in FileScale, these mappings are stored in an object cache in which no assumption is made that all data fits in memory.

An important advantage of HDFS's assumption that all metadata fits in memory on the NameNode is that this metadata can be given a permanent location in memory that can be directly referenced by other metadata. For example, in HDFS, the children of a directory inode (the files and directories stored inside it) can be stored as an in-memory list of direct pointers to the location of the inodes for these children. In order to resolve a complete path, HDFS simply needs to start at the root, and follow the series of direct pointers from root to the next child, and from there to the next child, etc.

In contrast, in FileScale's cache-based design, metadata cannot live in a permanent location in memory, since each cached object may be evicted according to the cache eviction algorithm. Therefore, each object is given a globally unique identifier, and references to objects, such as the children of a directory, are done via specifying the identifier instead of via a direct pointer. A separate lookup must occur to find the current location of the identified object in memory.

Although the extra lookups can cause increased latency, they can often be performed in parallel, which ends up in latency savings rather than costs. For example, each metadata operation in a file system must resolve path components recursively to validate the entire path and check user permissions and quota configuration. The direct pointer approach requires a search at each level of the path being validated to find the next child in the path. This is implemented in HDFS via a binary search within the list of children of a directory. The process of traversing these pointer connections is thus fundamentally a sequential operation. FileScale eliminates the need for this search at each level since the reference to the child is derived directly from the path name of the child. For example, to resolve the path (`/tmp/logs/data`), separate hash lookups for `</>`, `</tmp>`, `</tmp/logs>`, and `</tmp/logs/data>` are performed separately and **in parallel** instead of traversing the tree. This resolution technique is faster than the pointer-based traversal technique for long paths (because of its parallel execution) or paths with fat directories (because of its avoidance of the binary search). These differences will be explored further in Section 7.2.3.

5.2 Durability

Since updates are not necessarily immediately propagated to the DBMS, the cache layer implements a write-ahead logging

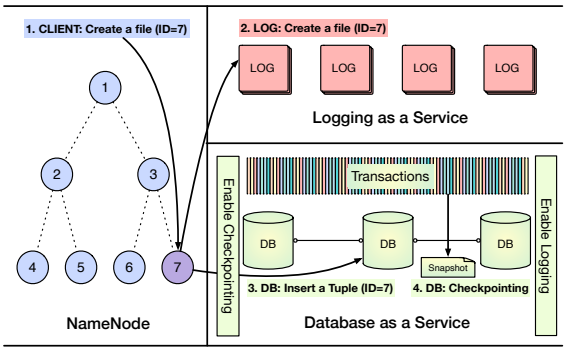


Figure 2: Workflow of file-create metadata operation.

mechanism based on an extension of HDFS's `EditLog`. Each NameNode logs all modifications it makes to a separate log file stored remotely in a network file system, similar to how HDFS stores edit logs on Quorum Journal Machines [7] or NFS [6] for high availability. Locks are not released until the logging service acknowledges the writes.

A periodic process asynchronously flushes recent writes to the database layer in batches. This limits the staleness of database state. A background process in the DDBMS takes periodic durable checkpoints of a snapshot of transaction-consistent state. Recovery starts from the most recent checkpoint, and plays forward any log records found in the logging service that were not incorporated in the database state, which are merged with log records found in the DDBMS log.

Log records that are reflected in any database layer checkpoint can be safely removed from the logging service.

Figure 2 shows the workflow of file-create operation. When FileScale receives a request to create a file with ID = 7, the NameNode writes a log record to the remote server and creates an inode object in the cache. After it receives a success message from the logging service, it makes the inode visible to subsequent requests prior to flushing the write to the DDBMS. Eventually the write is flushed to the DDBMS and is incorporated into a database snapshot, after which the log record associated with that write can be safely truncated.

If the DBMS fails during flush or multi-partition operations, the namenode employs an exponential backoff for future retries, anticipating database recovery.

6 PROXY LAYER

FileScale horizontally scales the name service through the creation of multiple, independent NameNodes in the caching layer. Each NameNode manages a disjoint partition of the namespace. However, the union of all the partitions need not cover the entire namespace. Requests over partitions of the namespace not covered by a NameNode are routed to a default NameNode that forwards the request directly to the database layer. FileScale implements a proxy layer to route requests to the appropriate NameNodes that will process those requests. Unlike HDFS, FileScale supports multi-partition (multi-NameNode) transactions.

6.1 Request Routing

FileScale stores the namespace partitioning across NameNodes in a "mount table" stored in Zookeeper [30]. Specific file path prefixes are assigned to NameNodes, which become the only eligible location for caching metadata associated with those path prefixes. The mount table is updated when new NameNodes are added or removed from the cluster, or when partitions need to be combined or split for improved load balancing. In practice it is read far more frequently than it is updated. Therefore, routing paths can be cached at the individual servers of the proxy layer for improved performance. However, this results in the proxy layer occasionally routing a request to the wrong NameNode, and that NameNode must then forward the request to the correct one (see below).

FileScale supports two modes to route user requests: (1) proxy mode and (2) watch mode. In proxy mode, the proxy layer consists of multiple routers that use the same communication protocols as HDFS. The router acts as a middleware layer that includes an upstream manager that maintains communication sessions for different clients, and intercepts client requests/responses to manipulate them as needed. The proxy layer can share hardware with the caching layer, such that there exists a router on each NameNode. When a client request is received by a router, the file paths associated with that request are extracted, and longest prefix matching is performed to locate the mount table entries relevant to that request. If all items accessed by the request are managed by a single NameNode, the request is forwarded there. Otherwise, the protocol described in Section 6.2 is used.

Watch mode works identically to proxy mode, except that the client watches ZooKeeper and caches the mount table at the client-side to save a network hop. The performance benefits of watch mode will be explored in Section 7.3.1.

Figure 3 shows an example request being routed to the appropriate NameNode. The two different sets of blue lines correspond to the proxy and watch modes described above.

In both the watch mode and proxy mode, mount table data is cached locally and a listener receives notifications when changes occur. On occasion, a name space partition may be moved from one NameNode to another, or an existing partition may be split or combined with a partition located on a different NameNode, temporarily rendering these caches stale, and causing misrouting of requests. Each NameNode maintains a recent-memory of paths that it formally managed, but part or all of it was moved[3] This enables the NameNode to immediately forward requests that were

[3]Adding a new path is performed via a split of the existing root path.

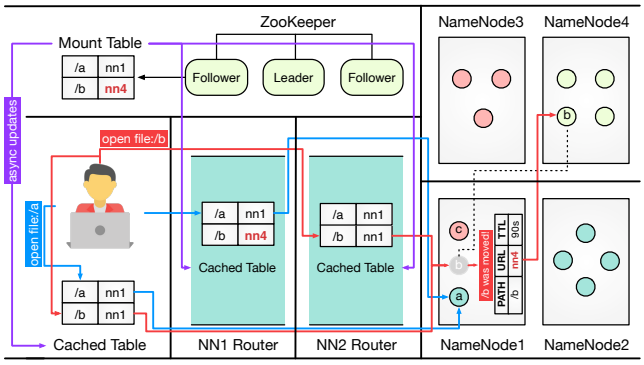


Figure 3: Request Routing in FileScale.

misrouted to it to the NameNode that took over the management of that partition of the namespace. This recent memory of moved paths is maintained with a short Time to Live (TTL) for each entry, since the cache of the mount table at each location is typically updated with short delay after it becomes stale. In the rare occasion where a NameNode receives a misrouted request for which it has no entry in its list of recent moves (because the TTL for that entry was too short and the entry already expired), the NameNode must look up the correct routing information in ZooKeeper to properly reroute the request.

In Figure 3, the two red lines illustrate this process. The requests are forwarded to the wrong NameNode because of outdated routing information and are then forwarded to the correct location directly from the old NameNode.

6.2 Multi-partition requests

File systems that partition by path prefixes reduce the frequency of multi-partition transactions; however, they still occur. The main source of multi-partition requests are 'move' or 'copy' operations in which data from the source partition must be read (for 'copy' operations) or removed (for 'move' operations) and inserted into the destination partition. Additionally, recursive operations such as 'chmod' (change the file permissions) or 'rm' (delete) that starts high in the directory tree (close to the root) may span partitions.

In FileScale, all multi-partition requests are performed by the database system after all data accessed by the transaction are removed from cache (dirty data is written to the database prior to removal) and prevented from being brought into cache while the transaction is ongoing.

Figure 4 shows the control flow between the NameNodes and associated services when a directory move operation spans multiple partitions. An example directory with an inode ID of 3 (along with its children) is being moved to a destination partition managed by a different NameNode. (1) The source NameNode writes back all relevant dirty inodes in batches and removes the subtree from the cache layer.

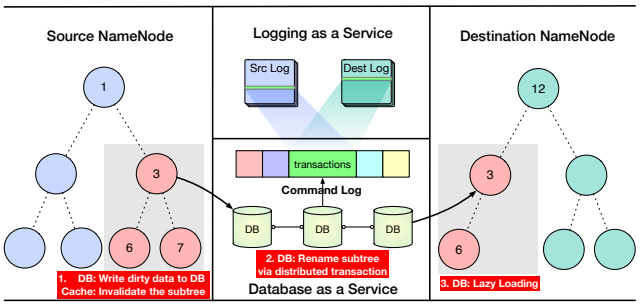


Figure 4: Move a folder across NameNodes.

(2) The database layer is updated synchronously via a (distributed) transaction that updates all affected inodes' names and their parent names. The precise implementation of the transaction depends on the underlying system, but can often be implemented via the SQL `LIKE` or `STARTS WITH` clause. (3) The destination NameNode can choose to load the entire new subtree or lazily load it as needed.

FileScale's cache layer log appends the offsets of the database log of the multi-partition transaction after it completes. This is only done for book-keeping purposes and is never relevant for recovery. This is because, as described in Section 5.2, only those log records not already incorporated in DBMS state are replayed during recovery. Since the processing of a multi-partition transaction is preceded by a DBMS flush, only cache-layer log records after the multi-partition transaction are potentially relevant to recovery.

7 PERFORMANCE EVALUATION

The implementation of FileScale directly inside the HDFS codebase was a large engineering effort and produced a total of 40k lines of code in HDFS 3.3.0. This effort allows for direct comparison of the metadata scalability and performance of FileScale with standard HDFS along with a state-of-the-art HDFS alternative that also stores data in a distributed database system (HopsFS) [39].

We initially use VoltDB [17] for FileScale's database layer. VoltDB is an in-memory DBMS that implements durability via a combination of asynchronous checkpointing and synchronous command logging that can be deterministically replayed to arrive at the state prior to a crash. In Section 7.6, we investigate the performance consequence of replacing VoltDB with Apache Ignite [3].

7.1 Experimental Setup

Previous attempts to scale metadata management within HDFS have succeeded in scaling file system throughput far beyond what a single HDFS NameNode is able to achieve. However, this comes at a cost of efficiency. For example, the

HopsFS paper reported that it took 3 NameNodes and 2 database servers to match the throughput that the single active NameNode is able to achieve [39] (see Figure 6 from that paper). A major goal of FileScale's architecture is to enable file system scalability with a higher amount of efficiency, so that it can be used from the early stages of an application up through the later stages as the application scales over time.

To that end, our experiments focus on **both small and large deployments**, ranging from running on a single server to large clusters of servers running on Amazon Web Services (AWS) EC2 instances. All experiments are run on EC2 t3a.2xlarge[4] instances for NameNodes and database servers. Each EC2 instance attached a EBS volume optimized for transactional workloads, and the volume is a 128 GiB of Provisioned IOPS (io1) SSD that can provision up to 64000 IOPS. Optimal NameNode heap size depends on many factors, such as the number of files, the number of blocks, and the load on the system, and generally requires tweaking since each workload has a unique byte-distribution profile. To reach the NameNode memory bottleneck quickly for our experiments, we use 16 GB for heap memory and garbage collection.

The NNThroughoutBenchmark [13] is used to generate test workloads. NNThroughoutBenchmark runs a series of client threads against a NameNode. However, the benchmark code out of the box runs on a single node without end-to-end network latency, so we extended the client workload generation in the benchmark codebase to run in the large-scale environments required for our analysis.

7.2 Single-node Experiments

We start by comparing FileScale with HDFS version 3.3.0 and HopsFS on a single AWS EC2 instance. All systems use a single NameNode, and the database servers used by FileScale (VoltDB) and HopsFS (NDB) run on the same machine as the NameNode.

7.2.1 Basic Operations. Figure 5 shows the throughput of delete, directory create and file create, open, and rename operations while varying the total number of these operations run (i.e., the number of files created, opened, etc.), and the number of client threads. NNThroughoutBenchmark introduces some fixed end-to-end overhead per run which is amortized across all operations in that run. Therefore, throughput for all systems improves as the number of operations increases.

For all types of operations, the performance of FileScale and HDFS is similar. This is because both systems store all metadata in memory when it fits on a single node and the performance of their respective in-memory data structures are similar. HopsFS ran out of memory after operations on over 100,000 files (a standard HopsFS deployment would divide the metadata across many machines in order to avoid running out of memory). At smaller scales, the throughput of HopsFS was approximately one tenth of HDFS and FileScale for create and rename operations, and one fifth for other operations. These results are consistent with the numbers reported in the HopsFS paper in which it took 5 servers—3 NameNodes and 2 database servers—to match the throughput that the single active NameNode [39]. The main reason for the performance difference is that FileScale avoids a round trip to the database system on the critical path during request processing when all data fits in cache. Instead of relying on the database system for durability, FileScale persists all updates to its write-ahead log (which has similar performance as writes to HDFS's write-ahead log). This allows FileScale to propagate updates to the DBMS asynchronously, in batches (in Section 7.5, we find that changing the frequency of these batch updates does not effect throughput, but does effect recovery time). In contrast, every HopsFS request requires at least one synchronous round trip to the DBMS.

7.2.2 Recursive Delete Operations. FileScale caches inodes in memory in a format that enables a one-to-one mapping with relational tuples in the database system. In contrast, HDFS stores all file system metadata in memory, and parent nodes can store direct, in-memory pointers to children nodes. As was explained in Section 5, this makes operations that traverse the directory structure, such as recursive file system operations, slower in FileScale relative to HDFS. To understand this tradeoff in more detail, we ran an experiment in which we measured the latency of performing a recursive delete operation, while varying the number of files and the depth of the tree being deleted. The results are shown in Figure 6.

The results show that the primary bottleneck is the process of deleting each individual file. The latency of all systems therefore increased as the number of files being deleted increased. To delete a file, HDFS needs to remove the file in memory (along with writing a log record to stable storage), while FileScale invalidates nodes in its cache, writes a log record, and asynchronously deletes related tuples in the database system. Since the latency of the individual deletes was the bottleneck, the latency numbers were only slightly impacted when height of the tree changed (when keeping the number of deleted files constant). Nonetheless, as expected, the overall latency of HDFS was slightly faster than FileScale. This is because FileScale's caching layer must be robust to situations in which child inodes are removed from the cache. Therefore, FileScale uses identifiers instead of direct pointers to children, and require a hashmap lookup the current location in memory of a particular ID.

[4]Each instance contains 32 GiB of memory, 8 VCPUs feature the 2.5 GHz AMD EPYC 7000 series processors and 5 Gbps of network burst bandwidth.

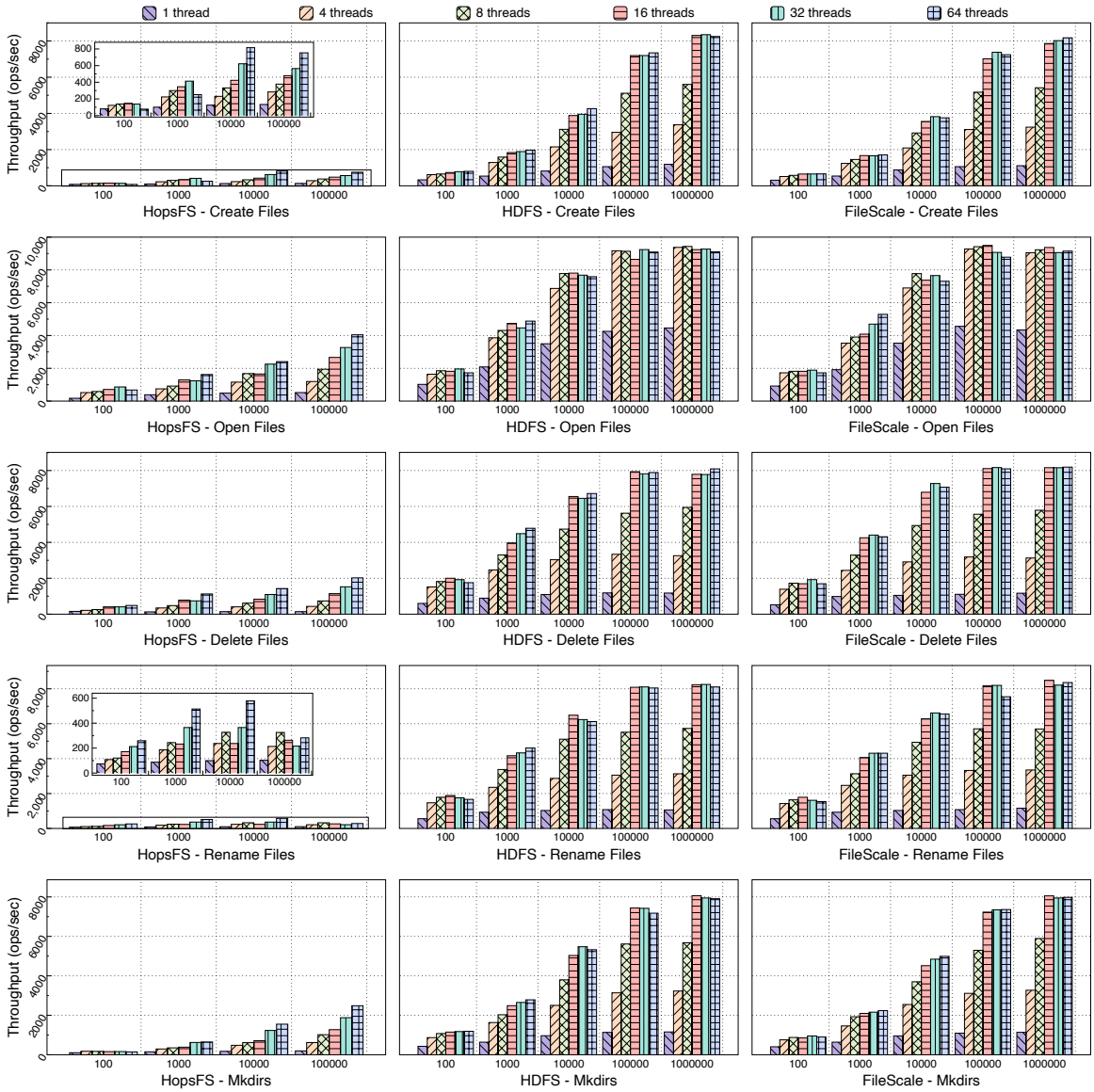


Figure 5: The throughput of basic operations: *create*, *open*, and *rename*.

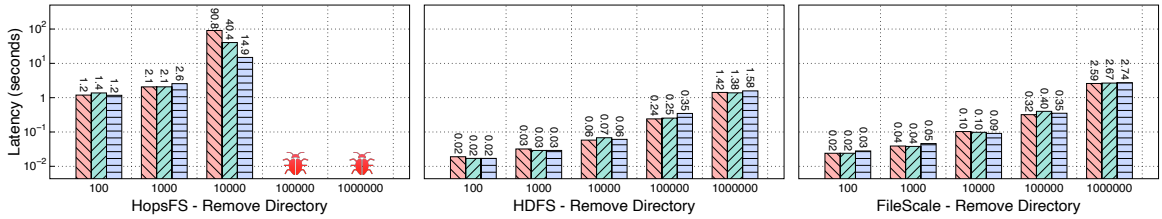
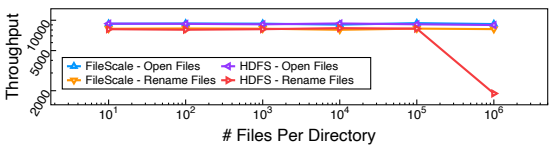


Figure 6: Recursive delete all files under the root directory.

We found that HopsFS transactions continuously time-out at more than or equal to 100,000 files (that appear to be caused by deadlocks). For the smaller experiments, we found that the latency of HopsFS are between one and two orders of magnitude worse than HDFS and FileScale (the figure uses a log scale y-axis). The relative performance between



(a) Total throughput varying the depth of 10^6 files.

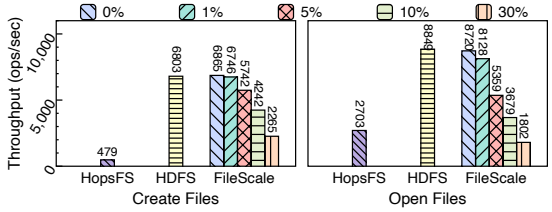
Figure 7: Large directory experiment.

HopsFS and HDFS is consistent with the results from Section 7.4.1 of the HopsFS paper [39] where it is explained that HopsFS performs poorly on these types of workloads because they are executed in many separate small transaction batches. Surprisingly, the performance of HopsFS *improved* when the depth of the tree being deleted increased. This is because deleting directories that contained a large number of files resulted in increased lock contention for the directory lock. By increasing the depth of the tree being deleted, there were fewer files per directory to delete, which reduced lock contention in the system.

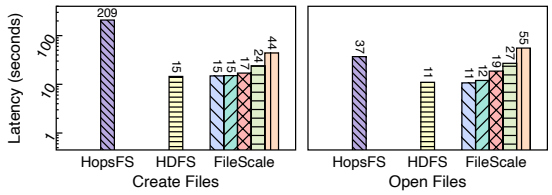
7.2.3 Large Directories. Figure 7(a) shows the throughput of 64 threads performing file open and rename operations within a directory that varies in size from 10 to 1,000,000 files. For most directory sizes, the performance of HDFS and FileScale are similar. However, in extreme cases, when directories contain a million files, the performance of the rename operation in HDFS drops substantially. This is because the list of children of a directory are stored as an `ArrayList`, and renaming files requires deleting elements from this list and reinserting them in order to keep the list in sorted order. Over time, these deletes and insertions require the entire `ArrayList` to be copied to a new location to improve the efficiency of how it is laid out in memory. However, copying a list that contains 1,000,000 causes a noticeable increase in latency, which drags down system throughput. In contrast, FileScale does not require the list of children be kept in sorted order, since path validation does not require a binary search at each level (Section 5).

7.2.4 Cache Misses. HDFS requires that all metadata fits in memory, whereas FileScale treats memory as a cache of the underlying database and remains available even when there is not enough aggregate memory across the nodes in the deployment to store all metadata in memory. To understand the extent of the performance drop at reduced memory deployments, we ran an experiment in which we increase the cache miss rate of FileScale from 0% to 30%[5] and measure the throughput reduction and latency increase on file create and open operations. The results are shown in Figure 8, and the original results for HDFS and HopsFS (from the previous experiments) are shown for comparison (even though HDFS

[5]Cache miss rates above (or even close to) 30% is extremely rare in practice.



(a) The throughput of creating and opening files.



(b) The latency of creating and opening files.

Figure 8: Cache miss penalty.

cannot run in this scenario). For each experiment, we used 16 threads to operate on 100,000 files concurrently.

Overall, the throughput and latency of FileScale degrades gracefully as the cache miss rate increases. The performance of open file operations degrades faster than for create operations because opening files can be served entirely from memory when data is in cache. (Section 7.6 shows that switching the database system from VoltDB can reduce the performance drop.) In contrast, creating files always has to push a log record to stable storage before the operation can commit regardless of whether the relevant directory data is already in cache, so the relative cost of a cache miss is smaller.

In practice, the number of cache misses can be monitored and action taken to alleviate performance problems. Specifically, the proxy server in FileScale can leverage Hadoop's existing monitoring component to immediately re-balance the mount table in FileScale's state store when performance decreases due to cache misses.

7.3 Multi-server Experiments

We next investigate the scalability of the different system architectures using multi-server deployments. We start with relatively small five-node deployments. HDFS does not support partitioning the same file system namespace across multiple NameNodes[6], but it can use the additional nodes to support HA (high availability). Therefore we set it up to use two NameNodes (in an active/standby configuration) and three JournalNodes. The journal nodes are used by HDFS to share logs between the active and standby NameNodes.

[6]HDFS does support partitioning metadata across NameNodes for different namespaces. We will investigate the performance of this alternative architecture in Section 7.3.1.

When a NameNode writes a log record, it must be written to a majority of JournalNodes before it is considered durable.

FileScale and HopsFS use a similar configuration: two of the five nodes are used for NameNodes (but unlike HDFS, the metadata can be partitioned across them), and the remaining three nodes for the database system — VoltDB for FileScale and NDB for HopsFS. For HA, log records written by FileScale and HopsFS are written to EBS volumes so that they remain persistent beyond the life span of AWS EC2 instances (which use only ephemeral storage). For fairness, we also run a version of HDFS in which log files are written to EBS (which we call HDFS-HA^{ebs}) as an alternative to the version in which log files are written to journal nodes as described above (which we call HDFS-HA^{jns}).

Figure 9 shows the throughput and latency of file-create and file-open operations under this deployment. The performance of HopsFS is almost two orders of magnitude slower than FileScale for the file-create benchmark, so the figure uses a log scale. This difference is consistent with the results presented in Figure 5 that show a large efficiency gap between FileScale and HopsFS. HopsFS approximately doubles its performance as the number of NameNodes doubles from one NameNode (HopsFS^{1nn}) to two NameNodes (HopsFS^{2nn}). However, FileScale's performance also doubles. Since opening files does not require writes to the DDBMS, a disadvantage of HopsFS (that it must synchronously write data to the underlying database) is not present, and the performance gap between FileScale and HopsFS is more narrow for the 'file open' workload.

Writing to EBS instead of the journal nodes improves HDFS performance for the file create workload, but makes no difference for the file open workload which does not require log records to be written. This enables the performance of HDFS and FileScale to be equivalent when running on one NameNode as they were in Figure 5. However the performance of FileScale doubles when doubling the number of NameNodes since it can partition data across them, whereas HDFS does not partition data and performance remains constant when adding the additional NameNode.

7.3.1 Scalability. We next increase the scale of our experiment by varying the number of NameNodes from 1 to 32 while keeping the number of database nodes constant. The workload consists of 50% file-create operations and 50% file-open operations that are uniformly spread across the namespace. We run two distinct HDFS architectures. The first is the default HDFS architecture in which the entire file system belongs to a unified namespace, so that there are no restrictions in the file system operations that can be run. However, this version must store all file system metadata on a single NameNode, as we described above. We also experiment with HDFS's router-based federation (RBF) capability, in which

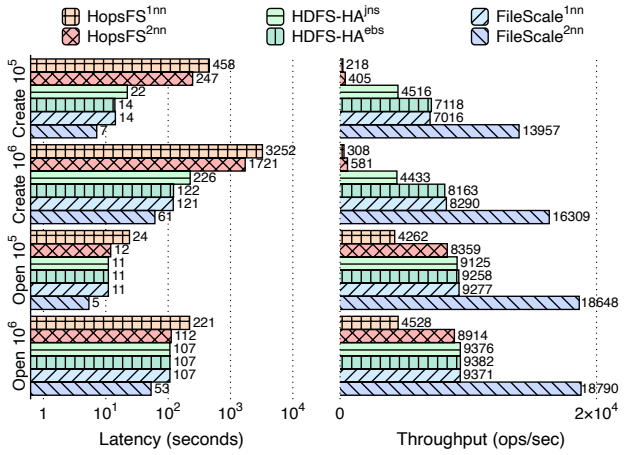


Figure 9: A multi-server deployment.

the namespace is partitioned, and the associated metadata for each partition can be managed by different NameNodes. Although the functionality of HDFS RBF is severely limited — for example, the multi-partition rename operations we run in Section 7.3.2 cannot be supported — it can support the simple file open and create operations used for this benchmark.

HopsFS ran into a bottleneck at the database layer at 16 NameNodes. Therefore we ran two versions of HopsFS — one with only three database nodes where the bottleneck is observed, and one with eight database nodes that avoids the bottleneck. FileScale did not experience the same bottleneck since it puts less pressure on the database layer by writing to the database in batches asynchronously instead of issuing synchronous writes for each new file created. Furthermore, HopsFS issues a batch query to the database layer at the beginning of every request to retrieve all the relevant inodes for the file path components of the request. This can be avoided in FileScale when the relevant data is in cache. Therefore, FileScale only requires 3 database nodes.

The results of this experiment are presented in Figure 10. HopsFS-8 NDB, HDFS-RBF, and both versions of FileScale are able to scale linearly — as the number of NameNodes double, so too does the total system throughput. Therefore, the original relative differences in performance between HopsFS, HDFS, and FileScale observed when running on a single NameNode (see Figure 5) remain present as the system scales. However, HDFS-RBF has in effect half has many NameNodes as FileScale and HopsFS since HDFS requires one hot standby for every NameNode for high availability. As expected, HDFS-RBF outperforms HDFS's default implementation, since the default implementation cannot partition metadata across the additional NameNodes and thus cannot scale. Nonetheless, HDFS's default implementation (along with HopsFS and FileScale) is able support the full range of

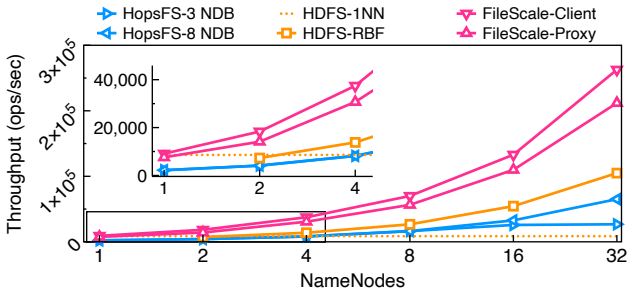


Figure 10: Throughput when scaling NameNodes.

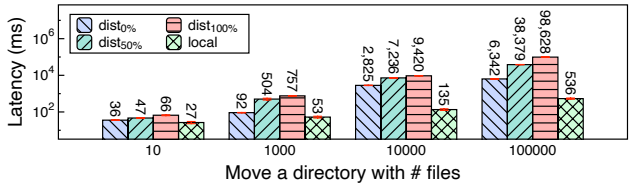


Figure 11: Local vs. distributed move operations.

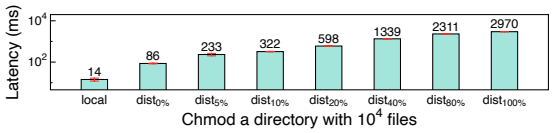
file system operations over all metadata, while HDFS-RBF must partition the namespace.

FileScale-Client corresponds to the watch mode configuration of FileScale described in Section 6, while FileScale-Proxy uses proxy mode. As expected, watch mode performs better since it is able to save a network hop during request processing, and avoid the overhead of processing and forwarding network messages at the proxy layer which shares physical hardware with the cache layer in the FileScale-Proxy deployment for this experiment.

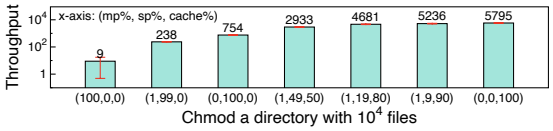
7.3.2 Multi-Partition Transactions. Figure 11 shows the average latency of multi-partition move requests in FileScale[7] as the percent of dirty data that must be written back to the database is varied. When the move operation is single-partition ("local"), latency is limited by the time to generate and write the log records to the logging service. For multi-partition move requests, log records need to be written to the log files for **both** the source and destination NameNodes. Furthermore, the move operation requires updating the primary key (the full path of the file), which is an expensive operation in the database layer since it partitions by the primary key. Nonetheless, when significant amounts of data need to be written back, it becomes the bottleneck. Supplemental experiments in Section 7.6 indicate that this bottleneck can be alleviated by changing the DDBMS.

Figure 12(a), shows the same experiment for distributed chmod operations. The database layer can process the distributed chmod transactions with much lower latency since

[7]In this experiment we omit HopsFS, since it also uses a DDBMS for multi-partition transactions, and differences in performance across systems are usually attributable to differences in the underlying DDBMS.



(a) The latency of changing a directory's permission.



(b) The total throughput of `chmod` operations.

Figure 12: Local vs. distributed chmod operations.

they do not require updating the primary key. Nonetheless, the writing of dirty data prior to transaction processing still dominates the latency. Figure 12(b) shows the throughput under varying mixes of multi-partition (MP) transactions, single-partition (SP) transactions and cache operations. Pure cache operations (100%) are 7x faster than SP transactions (100%). As soon as there are any MP transactions in a workload, even SP transactions that access the same data must be performed by the database layer. Therefore, there is more than a 1% drop in performance between 0%MP and 1%MP.

7.4 Hotspot mitigation

Figure 13 shows FileScale throughput as a hotspot is mitigated by re-balancing the mount table and distributing workloads across multiple NameNodes. We used benchmark utilities to create 4 subdirectories, all of which are initially mounted to NameNode 0. The proxy layer is triggered every 60 seconds to modify the mount table and assign each subdirectory to a new NameNode. The total throughput rises linearly as the hotspot workloads are re-distributed.

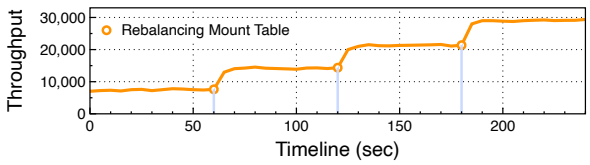
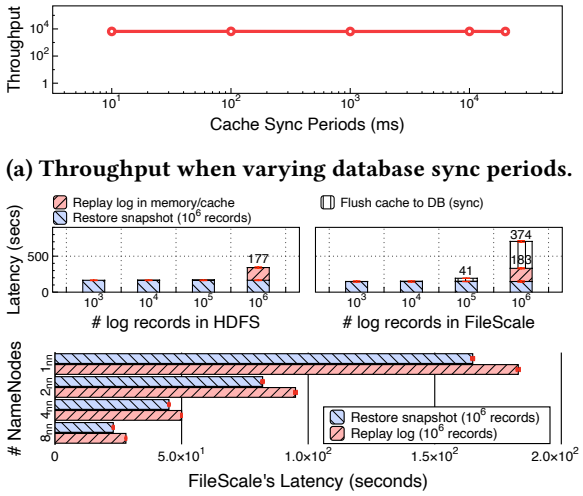


Figure 13: Hotspot Mitigation.

7.5 Flush intervals and disaster recovery

Figure 14(a) shows that size of flush intervals do not impact system performance, since the writes to the database layer are asynchronous. In essence these overheads are pushed to recovery time. Therefore we experiment with system restore to investigate this overhead, using the same experimental setup as for Figure 5. As described in Section 2, HDFS HA keeps its states (`FSImage` and `EditLog`) in a quorum-based



(a) Throughput when varying database sync periods.

(b) Latency when recovering from its backups.

Figure 14: System restore operations.

storage so that a standby can take over quickly if the active NameNode fails. Similarly, the database snapshot of FileScale provides a transactional point-in-time consistent copy of all file metadata, and the separated logging system records every changes from the last snapshot.

Figure 14(b) compares the latency of restoring snapshots and replaying logs for the different architectures. FileScale achieves comparable performance to HDFS when restoring 10^6 records from a snapshot. In HDFS, the NameNode only needs to replay edit logs in memory. However, FileScale must also update the database system after replaying logs in the cache. Doing this synchronously can add substantial latency to recovery, while doing this asynchronously works similarly to how the database layer lags behind the cache layer in normal operations. In this experiment, as the number of NameNodes increases from 1 to 8 and file metadata spreads more evenly, FileScale's restore time decreases linearly.

7.6 The Impact of Database System Choice

FileScale uses a modular architecture such that any database system could be used for the database layer as long as it supports ACID transactions and provides an interface in which transactions can be submitted in SQL (with additional support for stored procedures). Most of FileScale's functionality is implemented using SQL at the database layer, which makes adding support for a new database system straightforward. For example, the original version of FileScale was built over VoltDB's open source community edition, but when we were denied access to their enterprise version, we added support for Apache Ignite within a few weeks. In contrast, the rest of the FileScale codebase took two years of graduate student work.

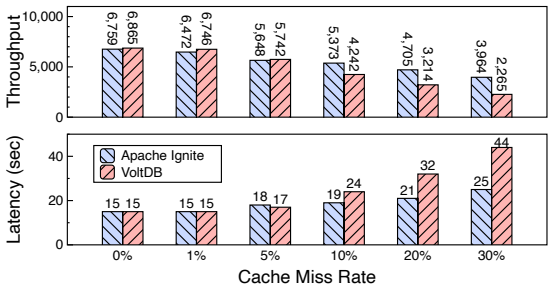


Figure 15: Cache miss penalty.

Most metadata operations require asynchronous interaction with the database layer for which the database choice does not affect performance. Therefore, the choice of database system to use in the database layer often makes no runtime performance difference. However, when the cache layer does not have sufficient memory and cache misses are frequent, the performance of the database layer starts to matter. Furthermore, multi-partition transactions always require synchronous interactions with the database layer. We therefore explore the impact of different database systems under these scenarios in which the choice of database system becomes important. Specifically, our experiments explore the performance impact of replacing VoltDB with Apache Ignite [3]: an open-source high throughput distributed database. Ignite stores data in memory by default, but also includes an optional disk tier which we enabled for these experiments. Apache Ignite provides key-value APIs as well as MapReduce-like computations in addition to ANSI-99 SQL and ACID transactions.

7.6.1 Cache Misses. To understand the extent of the performance difference between using VoltDB vs. Ignite as the database layer for FileScale, We ran an experiment in which we measure performance of file create operations as we increase the cache miss rate of FileScale from 0% to 30% and measure the throughput reduction and latency increase on file create operations. For each experiment, we used 16 threads to operate on 100,000 files concurrently, and the results are shown in Figure 15. The throughput of FileScale with VoltDB and Ignite falls smoothly as the cache miss rate rises, and the latency rises gracefully. VoltDB's performance declines faster than Ignite's after 10% cache misses. This is because Ignite's Key-value API `get()` can access the needed data from the storage using simple, light-weight access methods. In VoltDB, this was implemented using a standard SQL statement. Although VoltDB supports pre-compiling these SQL statements within a stored procedure, the performance of Ignite's lighter-weight key-value access methods is faster.

7.6.2 Multi-Partition Requests. As detailed in Section 6.2, a breadth-first search is used in FileScale to find all dirty

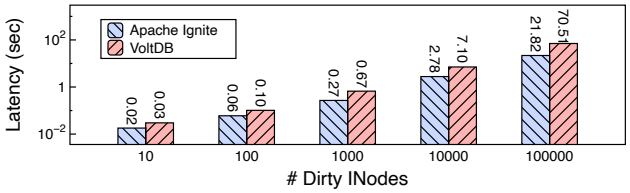


Figure 16: Dirty data flush penalty.

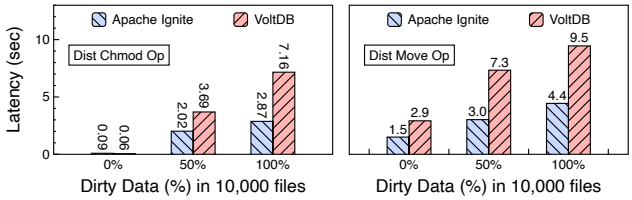


Figure 17: Distributed chmod and move operations.

inodes of the relevant data in the cache layer and bulk writes them to the database layer. After the dirty inodes are written to the database, The entire multi-partition transaction is then performed there. We saw in Section 7.3.2 that this write-back of dirty inodes is often the bottleneck for multi-partition transactions. Therefore, in this section, we investigate the performance of this cache flushing within the context of distributed transactions in more detail.

To understand the performance of cache flushing, we ran an experiment in which the amount of dirty inodes to be flushed as part of the transaction was increased from 10 to 100,000. Figure 16 shows the latency of writing dirty data to VoltDB and Ignite. Ignite supports a `putAll()` interface for bulk writes to the database, while the the same process was done in VoltDB with a bulk `INSERT` statement. Here again, avoiding the heavier-weight SQL API allows Ignite to achieve better performance.

For the transaction itself, we ran two types of operations: chmod and move. The pseudo code in Listings 1 and 2 shows how the distributed chmod operation is implemented in VoltDB and Ignite, where the standard SQL APIs of both systems is used. The query updates all of the children's permissions in a directory by matching the common ancestor path of the files and is implemented using the `STARTS WITH <string -expression>` expression in VoltDB. Ignite doesn't provide `STARTS WITH <string-expression>` in its SQL API, so the syntactically equivalent `LIKE <string-expression>%` is used instead. The use of the `STARTS WITH` clause enables the utilization of available indexes, whereas `LIKE` requires a sequential scan, since the compiler cannot tell if the replacement text ends in a percent sign or not and must plan for any possible string value. This allows VoltDB to be slightly faster than Ignite in Figure 17 when no dirty data needs to be flushed. However, when the amount of dirty data that must be written back grows, cache flushing dominates query time, for which Ignite is faster, as we saw above.

```
1  -- 1. Update all children in the subtree
2  UPDATE inodes SET permission = ? WHERE parent_name
3    STARTS WITH ?;
4  -- 2. Update the root inode of the subtree
5  UPDATE inodes SET permission = ?
6    WHERE parent_name = ? AND inode_name = ?;
```

Listing 1: Distributed chmod operation VoltDB.

```
1  IgniteCache<Object, Object> cache =
2    ignite.cache("inodes").withKeepBinary();
3  // 1. Update all children in the subtree
4  cache.query(new SqlFieldsQuery("UPDATE inodes SET
5    permission = ? WHERE parent_name LIKE ?%")
6   .setArgs(permission, subtree_path)).getAll();
7  // 2. Update the root inode of the subtree
8  cache.query(new SqlFieldsQuery("UPDATE inodes SET
9    permission = ? WHERE parent_name = ? AND inode_name = ?")
10   .setArgs(permission, parent_name, inode_name)).getAll();
```

Listing 2: Distributed chmod in Apache Ignite.

The move operation can be broken down into three steps: 1) Match the common ancestor path to obtain descendant inodes in the subtree. 2) For each child obtained from step (1) change the file path (parent name), inode name, and inode id. Since this involves updating the primary key, which also serves as the partitioning key, both VoltDB and Ignite require the updated inodes to be inserted anew into the database system 3) Delete the old subtree from the database using the old primary key and commit the transaction.

Since step 2 involves a bulk insert operation, Ignites better bulk insert performance that we saw in the cache flushing experiment helps it perform better than VoltDB for this step. Therefore, Figure 17 shows Ignite outperforming VoltDB at all data points, even when there is no dirty data to flush.

8 RELATED WORK

Merging file systems and database systems. There has been a large body of work which focuses on creating a hybrid system out of file system and database system components. For example, DeltaFS, TableFS, IndexFS and ShardFS [43, 44, 52, 54] store metadata in the local LevelDB instance [12]. However, these systems do not leverage distributed database systems to support multi-node atomic transactions. Instead they scale metadata via partitioning the file system namespace. In contrast, FileScale supports atomic modifications of all file system metadata in a global unified namespace.

There also exists a body of work in building file systems on top of distributed database systems in order to scale file system metadata. CalvinFS [50] uses a deterministic database system called Calvin [51] to store metadata, which supports high throughput distributed transactions. HopsFS [39]

uses a MySQL NDB cluster instead of Calvin, and shares FileScale's focus on being a drop-in replacement for HDFS. GiraffaFS [48] uses HBase for a similar purpose. As described in Sections 1 and 7, these systems have successfully scaled metadata management, but suffer from performance and efficiency problems that are especially noticeable when scaling down to a single node because of the frequent communication round trips between the file system and database system. In contrast, FileScale is designed to avoid synchronous interactions with the database system for most operations.

In industry, Facebook Tectonic [40] delegates file system metadata storage to ZippyDB [36], a linearizable, fault-tolerant, sharded key-value store. However, ZippyDB only supports strong consistency for single-shard operations and does not support cross-shard transactions. Thus Tectonic cannot provide cross-partition directory move operations. ADLS [42], (Microsoft Azure Data Lake Store) uses Paxos [32, 33] to maintain metadata that is stored in replicated Hekaton tables [25] and indexes. However, ADLS is similar to HopsFS in that it is designed from the beginning for extreme scales, and cannot be scaled down to efficiently run on a single node. Colossus, the next-generation of GFS [28, 37], introduced a distributed metadata model using BigTable [23] which does not allow distributed transactions and thus does not allow multi-partition metadata operations [27, 29]. WinFS [16], Microsoft's replacement for NTFS, stores file system data inside a DBMS. However, this integration was not performed for scalability reasons, but rather in order to enhance search capabilities by integrating SQL with file system metadata [16].

Federation. Giga+ [41] includes code similar to FileScale's Proxy layer in that it divides each directory into a number of partitions that are distributed across servers. Giga+ uses a bitmap to map filenames to directory partitions and to a specific server. However, Giga must implement their own version of atomic multi-partition transactions and high availability, whereas FileScale gets these "for free" by leveraging the DB layer. Furthermore, Giga+ is designed for large scale deployments, and suffers similar efficiency problems as HopsFS when scaling down to single-node deployments. The View File System (ViewFs) [11] uses the client-side mount points to split HDFS into multiple physical namespaces and presents a single virtual namespace to users. However, client configuration changes are required every time a new mount point is added or replaced, and it is difficult to roll out these adjustments without affecting production workflows [15]. HDFS Router based Federation [8, 38] and ByteDance NameNode-Proxy [4] are extensions to ViewFS-based partitioned federation that uses routers forward client calls to the correct NameNode. However, all these systems suffer from the limitation of HDFS Federation we discussed above: no support for multi-partition atomic requests.

System	Metadata	Multi-Partition Operations	Single-node In-memory Performance
DeltaFS [55]	LevelDB	No	Yes
TableFS [43]	LevelDB	No	Yes
IndexFS [44]	LevelDB	No	Yes
ShardFS [52]	LevelDB	No	Yes
GiraffaFS [48]	HBase	No	No
Colossus [29]	BigTable	No	No
Tectonic [40]	ZippyDB [36]	No	No
ADLS [42]	Hekaton [25]	Yes	No
HopsFS [39]	MySQL NDB	Yes	No
CalvinFS [50]	Calvin [51]	Yes	No
ViewFS [11]	In-Memory	No	Yes
Giga+ [41]	LevelDB	Yes	No
HDFS RBF [8]	In-Memory	No	Yes
FileScale	Ignite, VoltDB	Yes	Yes

Table 2: Comparison of related scalable file systems.

Table 2 compares all the systems discussed above in this section. All systems we have discussed either support high-efficiency in-memory performance on a single node or atomic multi-partition requests across a distributed/partitioned deployment (or neither), but not both. FileScale's novelty [35] is that it leverages distributed database system technology to support atomic multi-partition requests, while avoiding synchronous interactions with the database system so that it can gracefully scale down to single-node deployments.

9 CONCLUSIONS

FileScale's architecture enables comparable performance to file systems that store all data in memory on a single machine, and yet can also scale linearly as the size of the metadata scales. Our experiments showed that FileScale can achieve multiple orders of magnitude superior performance relative to other approaches for scaling file system metadata. FileScale's architecture also enables elastic scaling of each layer in the architecture independently.

ACKNOWLEDGMENTS

We thank Zhichao Liu for his early contributions to the codebase for this project, our shepherd Ashvin Goel for his guidance, and the reviewers of this paper for their helpful feedback. This work is supported by the National Science Foundation under grant IIS-1910613.

REFERENCES

[1] [n.d.]. Alternate Hash Table for NameNode Memory Optimization. https://issues.apache.org/jira/browse/HDFS-1114.

[2] [n.d.]. Apache HBase. https://hbase.apache.org.

[3] [n.d.]. Apache Ignite. https://ignite.apache.org.

[4] [n.d.]. ByteDance NNProxy. https://github.com/bytedance/nnproxy.

[5] [n.d.]. HDFS Federation. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html.

[6] [n.d.]. HDFS High Availability Using NFS. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html.

[7] [n.d.]. HDFS High Availability Using the Quorum Journal Manager. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html.

[8] [n.d.]. HDFS Router-based Federation. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs-rbf/HDFSRouterFederation.html.

[9] [n.d.]. HDFS Router-based Federation. https://issues.apache.org/jira/browse/HDFS-10467.

[10] [n.d.]. HDFS scalability with multiple namenodes. https://issues.apache.org/jira/browse/HDFS-1052.

[11] [n.d.]. HDFS ViewFs Guide. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ViewFs.html.

[12] [n.d.]. LevelDB. https://github.com/google/leveldb.

[13] [n.d.]. NNThroughputBenchmark. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/Benchmarking.html.

[14] [n.d.]. Removing Name-node's memory limitation. https://issues.apache.org/jira/browse/HDFS-5389.

[15] [n.d.]. Scaling Uber's Apache Hadoop Distributed File System for Growth. https://eng.uber.com/scaling-hdfs/.

[16] [n.d.]. WinFS: Windows Future Storage. https://en.wikipedia.org/wiki/WinFS.

[17] 2010. VoltDB. https://www.voltdb.com.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[19] Michael Abd-El-Malek, William V. Courtright, II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. 2005. Ursa Minor: Versatile Cluster-based Storage. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4* (San Francisco, CA) *(FAST'05)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1251028.1251033

[20] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. 2002. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 1–14. https://doi.org/10.1145/844128.844130

[21] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 353–369.

[22] Andrew Audibert. 2019. Scalable Metadata Service in Alluxio: Storing Billions of Files. https://www.alluxio.io/blog/scalable-metadata-service-in-alluxio-storing-billions.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267308.1267323

[24] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

[26] Huang Dongfa. 2022. DanceNN: Overview of Byte Self-developed 100 Billion Scale File Metadata Storage System. https://bafybeigahnjknx333gpi6uoftsoohvdssb5pyalaoavv6l2wbxicqorxwu.ipfs.infura-ipfs.io/.

[27] Pavan Edara and Mosha Pasumansky. 2021. Big Metadata: When Metadata is Big Data. *Proc. VLDB Endow.* 14, 12 (2021), 3083 – 3095. https://doi.org/10.14778/3476311.3476385

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[29] Dean Hildebrand and Denis Serenyi. 2021. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system.

[30] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[31] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Martí, and Eugenio Cesario. 2008. The XtreemFS architecture - a case for object-based file systems in Grids. *Concurrency and Computation - Practice and Experience* (2008).

[32] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[33] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[34] Haoyuan Li. 2018. Alluxio: A Virtual Distributed File System. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.pdf.

[35] Gang Liao. 2022. The Evolution of Cloud Data Architectures: Storage, Compute, and Migration. https://drum.lib.umd.edu/items/e591f36a-a240-42db-8252-196ed4facee9.

[36] Sarang Masti. [n.d.]. ZippyDB: A Distributed key value store. https://engineering.fb.com/2021/08/06/core-data/zippydb/.

[37] Marshall Kirk McKusick and Sean Quinlan. 2009. GFS: Evolution on Fast-forward. *Queue* 7, 7, Article 10 (Aug. 2009), 11 pages. https://doi.org/10.1145/1594204.1594206

[38] Pulkit A. Misra, Íñigo Goiri, Jason Kace, and Ricardo Bianchini. 2017. Scaling Distributed File Systems in Resource-harvesting Datacenters.

In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 799–811. http://dl.acm.org/citation.cfm?id=3154690.3154765

[39] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using newSQL Databases. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Santa clara, CA, USA) *(FAST'17)*. USENIX Association, Berkeley, CA, USA, 89–103. http://dl.acm.org/citation.cfm?id=3129633.3129642

[40] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*. 217–231.

[41] Swapnil V Patil, Garth A Gibson, Sam Lang, and Milo Polte. 2007. GIGA+ scalable directories for shared file systems. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*. 26–29.

[42] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 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*. 51–63.

[43] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 145–156. https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren

[44] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 237–248.

[45] P. Schwan. 2003. Lustre: Building a file system for 1000-node clusters. *In Proceedings of the 2003 Linux Symposium* (2003).

[46] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[47] Konstantin V Shvachko. 2010. HDFS Scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE* 35, 2 (2010), 6–16.

[48] Konstantin V Shvachko and Yuxiang Chen. 2017. Scaling Namespace Operations with Giraffa File System. *USENIX; login* (2017).

[49] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. 2010. BabuDB: Fast and Efficient File System Metadata Storage. *2010 International Workshop on Storage Network Architecture and Parallel I/Os* (2010), 51–58. https://doi.org/10.1109/snapi.2010.14

[50] Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=2750482.2750483

[51] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[52] Lin Xiao, Kai Ren, Qing Zheng, and Garth A. Gibson. 2015. ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) *(SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 236–249. https://doi.org/10.1145/2806777.2806844

[53] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[54] Qing Zheng, Charles D Cranor, Danhao Guo, Gregory R Ganger, George Amvrosiadis, Garth A Gibson, Bradley W Settlemyer, Gary Grider, and Fan Guo. 2018. Scaling embedded in-situ indexing with deltaFS. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 30–44.

[55] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*. 1–6.