ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems

Lin Xiao, Kai Ren, Qing Zheng, Garth A. Gibson
Carnegie Mellon University
{lxiao, kair, zhengq, garth}@cs.cmu.edu

Abstract

The rapid growth of cloud storage systems calls for fast and scalable namespace processing. While few commercial file systems offer anything better than federating individually non-scalable namespace servers, a recent academic file system, IndexFS, demonstrates scalable namespace processing based on client caching of directory entries and permissions (directory lookup state) with no per-client state in servers. In this paper we explore explicit replication of directory lookup state in all servers as an alternative to caching this information in all clients. Both eliminate most repeated RPCs to different servers in order to resolve hierarchical permission tests. Our realization for server replicated directory lookup state, ShardFS, employs a novel file system specific hybrid optimistic and pessimistic concurrency control favoring single object transactions over distributed transactions. Our experimentation suggests that if directory lookup state mutation is a fixed fraction of operations (strong scaling for metadata), server replication does not scale as well as client caching, but if directory lookup state mutation is proportional to the number of jobs, not the number of processes per job, (weak scaling for metadata), then server replication can scale more linearly than client caching and provide lower 70 percentile response times as well.

Categories and Subject Descriptors D.4.3 [*File Systems Management*]: Distributed file systems

General Terms Design, performance, measurement

Keywords metadata management, caching, replication

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA.

ACM 978-1-4503-3651-2/15/08.
http://dx.doi.org/10.1145/2806777.2806844

1. Introduction

Modern distributed storage systems commonly use an architecture that decouples metadata access from file read and write operations [21, 25, 46]. In such a file system, clients acquire permission and file location information from metadata servers before accessing file contents, so slower metadata service can degrade performance for the data path. Popular new distributed file systems such as HDFS [25] and the first generation of the Google file system [21] have used centralized single-node metadata services and focused on scaling only the data path. However, the single-node metadata server design limits the scalability of the file system in terms of the number of objects stored and concurrent accesses to the file system [40]. Federating independent metadata services, common among network attached storage systems (e.g. NFS, CIFS), employs multiple server nodes but does not ensure load balancing among them. Moreover, all too often the data-to-metadata ratio is not high, since most files even in large file system installations are small [16, 24, 45]. Scalable storage systems should expect the numbers of small files to soon achieve and exceed billions, a known challenge for many existing distributed file systems [32]. While distributed file system metadata servers perform many functions (such as block management and data server fail-over), our focus in this paper is on the management of the file system namespace.

Hierarchical file systems traditionally enforce transactional verification of hierarchical access control (permissions) on any pathname argument; that is, access to an object specified by a pathname requires permission to lookup that object's name in its parent directory, and permission to lookup that parent directory's name in the grandparent directory, recursing as specified by the pathname back to either the file system's root or a directory currently open in the caller's process. We call this information directory lookup state. The multi-lookup resolution of a pathname is traditionally serializable with respect to all file system operations on pathnames (not necessarily for commands on open files such as data access) and is the basis of most users' understanding of file system access control. Metadata operations with the same prefix of their absolute pathnames may con-

flict with each other, so pathname lookup is one main bottleneck for scaling.

Focusing on throughput and latency of metadata operations, we only consider designs that dynamically distribute metadata with fine granularity in this paper because static subtree partitioning of the namespace cannot provide good load balancing when the workload only accesses a small portion of the namespace. We examine three design approaches that reduce pathname lookup overheads differently, including *replicated directories with sharded files*, *dynamically partitioned namespace with client caching*, and *table partitioned namespace*. The measured systems implementing these approaches are ShardFS, IndexFS and Giraffa respectively. These design approaches distribute metadata in different ways. Giraffa stores file system metadata as entries in a table. Metadata distribution and load balancing is delegated to the underlying table system, HBase [20]. Unlike Giraffa, ShardFS and IndexFS manage metadata distribution themselves by hashing or dynamic directory allocation. ShardFS replicates directory lookup states to support server local pathname lookups. ShardFS reduces replication cost by specializing distributed transaction implementation based on categorizing metadata operations. Alternatively, IndexFS [37] follows traditional file system design and partitions the entire namespace with dynamically balanced directory distribution. To reduce the number of pathname lookups on metadata servers, IndexFS uses coherent client caching of directory lookup states. IndexFS's clients observe higher latency due to cache misses on some directories while ShardFS's clients experience slow directory metadata mutation operations. These systems are compared across several dimensions such as load balance and RPC amplification.

There are two common notions of scalability measurements: strong scalability and weak scalability. Strong scalability measures how well a system scales with increasing processing resources for a fixed workload, while weak scalability measures the scalability with a fixed workload per unit of resource. They are well defined for data dominant workloads. Consider, however, the impact of a weak scaling experiment on metadata workload. For example, a checkpoint job for high performance computing (HPC) applications saves status of running processes to files. It may create a file for each processing unit in one directory, which produces the same number of directories independent of the system size. We define weak scalability for metadata workloads as linear speedup with increasing amount of processing resources when the amount of directory metadata mutation operations is proportional to jobs while the amount of file metadata operations is proportional to processing units. If there are a fixed number of jobs, directory metadata mutation operations are constant with scale.

This paper makes four contributions. The first contribution is a characterization of metadata workloads in cloud storage systems. Secondly, we analyze and pin-point the pathname lookup trade-offs in metadata service design. We compare scale-out client caching (IndexFS), server-side replicated state (ShardFS) and table partitioned namespace (Giraffa) for pathname lookups. Thirdly, ShardFS's novel design uses directory lookup state replication and optimizes directory metadata operations with optimistic concurrency control schemes to efficiently scale workloads that are intensive with file metadata operations. The code of ShardFS is also available as open source here [39]. Finally, we propose definitions of strong and weak scalability for metadata workloads and devise a benchmark suite to systematically compare different metadata processing systems.

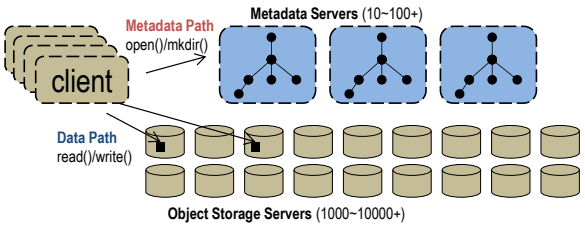


Figure 1: Typical distributed file system architecture.

In the rest of this paper, a detailed comparison of these approaches is presented. We evaluate design choices made by these systems using a set of micro-benchmarks and trace replay tests on a 256-node cluster.

2. Background

Before comparing different metadata service designs, we present an overview of file system architecture and design requirements. We also examine file system traces generated by production clusters to demonstrate the characteristics of real metadata workloads. Then we briefly discuss previous work on distributed metadata management.

2.1 System Architecture

The architecture used by all three design approaches is similar to many modern distributed file systems [21, 23, 25]. It decouples metadata from data management by having separate metadata servers and data servers, as shown in Figure 1. In such an architecture, metadata servers are responsible for managing the file system namespace and metadata associated with files and directories. Clients first communicate with metadata servers to locate data and then interact directly with data servers. This allows efficient parallel data transfers among a large number of clients and data servers. Metadata server and data servers can colocate on the same physical nodes. To simplify the implementation and design, previous file systems mostly use only a centralized single-node metadata server with backup servers in case of failure, or statically partition the namespace, suffering when a workload concentrates all its work to one server.

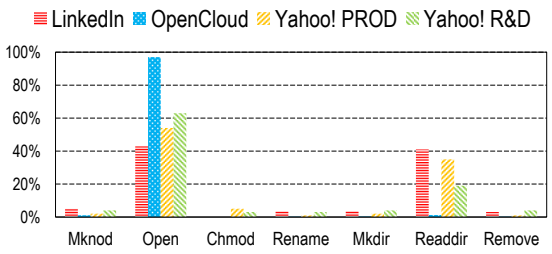


Figure 2: *Distribution of file system operations in LinkedIn, Yahoo!, and CMU's OpenCloud traces.*

In this paper, we focus on scaling out metadata management by distributing metadata across many servers. We assume that clients and servers are always connected, which represents the scenario inside a datacenter.

2.2 Metadata Workload Analysis

We analyze file system traces obtained from previously published work [29, 45] or collected from our colleagues in industry. These storage systems include desktop file systems from Microsoft engineers [29], commercial HPC customer installations from Panasas [45], and several Hadoop clusters (from Yahoo!, Facebook, LinkedIn, and CMU). The size of storage systems in these traces ranges from one machine to thousands of machines.

File System Namespace Structure File system namespaces exhibit a heavy-tailed distribution of directory sizes. Typically, a file system consists of many small directories and relatively few large directories. About 60 out of 70 file systems in our traces have nearly 90% of directories with size fewer than 128 entries. On the other hand, large directories continue to grow with the increasing size of the storage system. Many parallel computing applications (e.g. map-reduce, checkpoint) require concurrent access to large directories [6, 17]. File size has a similar distribution: about 64% of traced file systems have median file size smaller than 64KB, while the largest file size can be several terabytes. We found that the average file system tree depth does not increase much as the system grows: most file systems have 90% of their directories with a depth of 16 or less. Thus, a scalable metadata service should support a large amount of small objects, concurrent access to large directories, and can perhaps expect a short pathname depth.

Skewed Distribution of Operations We also studied the dynamic statistics of four Hadoop clusters: LinkedIn, Yahoo! Research, Yahoo! Production and CMU's Hadoop cluster OpenCloud. The number of individual file system metadata operations are counted over the entire traces. Figure 2 shows the fraction of each metadata operation type. In all traces one or two operations dominate: *open* is the most prevalent metadata operation, and the next is *readdir*. This skewed distribution of file operation types motivates us to explore design choices that trade the performance of some metadata operations for that of others.

2.3 Previous Work

Many popular distributed file systems in clouds such as HDFS [25] and the first generation of Google file system [21] store file metadata on a single server with limited scalability. Although the Lustre 1.x [28] parallel file system supports multiple metadata servers, only one of them can be active. Federated HDFS, Lustre 2.x and PanFS [25, 46] further scale metadata management by using an approach where individual non-scalable namespace servers are responsible for an administrator selected subtree of the overall namespace. Such solutions lack the flexibility to automatically balance load across multiple metadata servers.

A few distributed file systems have added support to achieve a fully distributed file system namespace. PVFS [11] and Ceph [44] dynamically distribute the file system namespace at the granularity of a directory. Ceph uses an adaptive partitioning technique to split large directories, though users are advised to disable this feature when using multiple metadata servers [13].

The distributed directory service [18] in Farsite [3] uses a tree-structured file identifier for each file. It partitions the metadata based on the prefix of file identifiers, which simplifies the implementation of rename operations. Farsite employs field-level leases and disjunctive leases to reduce false sharing of metadata across clients and mitigate metadata hot spots. However, unlike IndexFS, servers maintain state about the owner of each lease in order to later invalidate the lease and is not designed for highly concurrent mutations.

Both Frangipani [42] and GPFS [38] are built on shared disks. Frangipani uses distributed coarse-grained locks on entire files. GPFS uses token-based distributed fine-grained locking on ranges in files for high throughput and strong consistency among clients. GPFS uses whole directory locks to maintain consistency of inodes within a single directory, which inhibits scalability for file creations in one directory.

CalvinFS [43] leverages a shared-nothing distributed database system for metadata management with consistent WAN replication. File metadata is partitioned by hash of the full pathname. To optimize for single-file operations (read and write), all permissions from the root are stored with each entry. A directory's files and subdirectories are stored as its value for *readdir*. This makes file creation a distributed transaction and hence does not scale well. Other metadata operations such as directory rename and permission changes need to recursively modify all entries under the affected subtree. Unlike CalvinFS, Giraffa maintains locality for entries within one directory for *readdir* and relaxes some POSIX semantics for single-file operations.

H-store [41] is a modern main memory databases that uses a sophisticated concurrency control strategy. H-store classifies transactions into multiple classes where single-sited or sterile transactions are executed without locks. For non-sterile and non-single-sited transactions, H-store uses an optimistic method to execute distributed transactions

and dynamically escalates to more sophisticated strategies when the percentage of aborted transactions exceeds certain threshold. ShardFS follows a similar design methodology.

ShardFS also shares properties with the distributed B-tree in [5] on top of Sinfonia [4]. The B-tree uses optimistic concurrency control and commits with minitransactions provided by Sinfonia. It lazily replicates inner nodes at clients and eagerly replicates version table for load balance and optimization for single server transaction, which is similar to ShardFS's replication of directory lookup state. ShardFS optimizes specific distributed metadata mutations even further to reduce the number of RPCs.

Object storage systems such as CohoData [15], Nutanix and Microsoft Azure [10] target different problems than metadata performance. They focus on providing high data throughput with little or no hierarchy in the namespace of objects. The address mapping for objects is usually stored in systems such as Zookeeper [26] or Cassandra [12].

3. Comparison of System Designs

As indicated in the introduction, pathname lookup is an important factor restricting the scaling of distributed metadata management. Traditional file systems scale pathname resolution by coherently caching namespace information (structure, names, permissions) in each client under the protection of a (leased) lock. Our representative traditional file system, IndexFS, implements coherent client caching, simplifying server error handling logic by blocking all mutations to a directory until all leases have expired (e.g. there is no immediate revocation). Caching state under coherent leases is a replication strategy with replication costs proportional to the number of clients and to the size of the working set for the cache (number of directories frequently consulted in pathname lookup). Since the number of clients is rarely less than the number of servers, fully replicating the same information in each server can, in principle, be a less costly way to avoid multiple RPCs and locking congestion across multiple metadata servers. ShardFS is an example system using full replication of directory lookup states. An alternative approach taken by Giraffa is to relax file system access control semantics and store the full pathname for each entry to reduce the number of lookups.

In the following sections, we explore the viability of server replication of namespace information, and discuss the performance implication brought by these three design choices.

3.1 Dynamically partitioned namespace (IndexFS)

In IndexFS, the namespace is partitioned dynamically at the directory subset granularity. As shown in Figure 3, each directory is assigned to an initial, randomly selected metadata server when it is created. The directory entries of all files in that directory are initially stored in the same server. This works well for small directories since storing directory entries together can preserve locality for scan operations such as *readdir*.

For the few directories that grow to a very large number of entries, IndexFS uses the GIGA+ binary splitting technique to distribute directory entries over multiple servers [32]. Each directory entry is hashed into a large hash-space that is range partitioned. A directory gets split incrementally in proportion to its size: it starts small on a single server that manages its entire hash-range. As the directory grows, IndexFS splits the hash-range into halves and assigns the second half of the hash-range to another metadata server. As these hash-ranges gain more directory entries, they can be further split until the directory expands to use all metadata servers. Once the distributed directory is utilizing and well balanced on all servers, further splitting is inhibited.

Metadata Caching POSIX semantics require many metadata operations to perform pathname traversal and permission checking across each ancestor directory. This access pattern is not well balanced across metadata servers because pathname components near the top of the namespace tree are accessed much more frequently than those lower in the tree. IndexFS maintains a consistent cache of pathname components and their permissions (called the directory entry cache). IndexFS avoids invalidation storms by assigning short term leases to each pathname component accessed and delaying any modification until the largest lease expires. This allows IndexFS servers to only record the largest lease expiration time with any pathname component in its memory, pinning the entry in its memory and blocking updates to the entry until all leases have expired.

Any operation that wants to modify the server's copy of a pathname component, which is a directory entry in the IndexFS server, blocks operations that want to extend a lease (or returns a non-cacheable copy of the pathname component information) and waits for the lease to expire. While this may incur high latency for these mutation operations, client latency for non-mutation operations, memory and network resource consumptions is greatly reduced. This method assumes the clocks on all machines are synchronized, which is achieved in modern data centers [9, 14].

Implementation IndexFS is distributed metadata middleware, written in C++, that can be layered on top of existing file systems such as HDFS to scale their metadata performance [37]. IndexFS uses the Thrift [1] RPC library for network communication. Each stores its metadata into LevelDB [27] which implements a write-optimized B-tree: a Log-structured Merged tree (LSM tree) [31]. LevelDB stores the leaves of its LSM tree in HDFS where they are replicated to tolerate failures. Our previous work on TableFS [35] showed that storing metadata in LevelDB can achieve better file system performance than existing local file systems.

Fault Tolerance IndexFS's primary fault tolerance strategy is to push state (metadata files and write-ahead logs for recent changes) into the underlying object storage e.g.

HDFS, that provides replication. IndexFS server processes are all monitored by standby server processes prepared to replace those failed. Each IndexFS metadata server maintains a separate write-ahead log that records mutation operations such as file creation and rename. Leases for client caching are not made durable; a standby server restarting from logs simply waits for the largest possible timeout interval before granting access.

For operations requiring a distributed transaction protocol such as directory splitting and rename operations, two phase distributed transactions are used in IndexFS. Server failure is protected by write-ahead logging in source and destination servers and eventual garbage collection of resource orphaned by failures. Currently IndexFS only implements a simplified version of the *rename* operation that supports renaming files and leaf directories.

3.2 Table partitioned namespace (Giraffa)

As shown in Figure 4, Giraffa stores file system metadata inside a distributed table, HBase [20], which provides single-row transaction guarantees. Each file in a Giraffa namespace is stored as one row inside a table in HBase. In order to maintain the hierarchical namespace of the file system, Giraffa embeds its file system tree structure inside the row keys of all file system objects. The default strategy is to use a full pathname prefixed with the depth (of this file system object) as the key in the namespace tree. This ensures that all entries within the same directory share the same row key prefix, which provides locality to implement *readdir* efficiently. Giraffa translates metadata operations into a set of key-value operations on HBase, and reuses the load balancing techniques and persistence guarantees provided by HBase [20], built on HDFS [25].

Implementation Details The Giraffa metadata server is implemented as a "coprocessor" embedded in each HBase region server [20], working as a "stored procedure" in a relational database [34]. The current implementation of Giraffa relies on the underlying HBase to dynamically partition and distribute the metadata table across all its region servers to achieve load balance. By default, HBase horizontally partitions its table according to a fixed region size. Since HBase is unaware of any semantic meaning of stored table contents, it will not deliberately partition a large directory or cluster small directories as IndexFS does. The split threshold for a HBase region is as large as 10GB, which is much larger than the split threshold for directories in IndexFS. During our experiments, we found Giraffa can easily suffer a skewed distribution in its lexicographic key space, in part due to its default schema for generating row keys. In order to mitigate this problem, we modified Giraffa's code by prefixing a hash string calculated from the parent path to the original row key, In addition, we also pre-split the namespace table in HBase at the beginning of each experiment. HBase allows users to pre-split tables to help better balance the system during the initial workload, provided that the key distribution is known

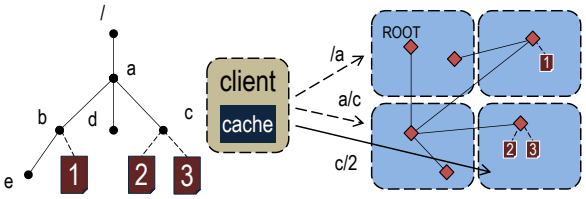


Figure 3: *IndexFS distributes its namespace on a per-directory partition basis. Path resolution is conducted at clients with a consistent lease-based directory lookup cache. Directories are not partitioned until they are large and partitioning is done by repeated binary splitting.*

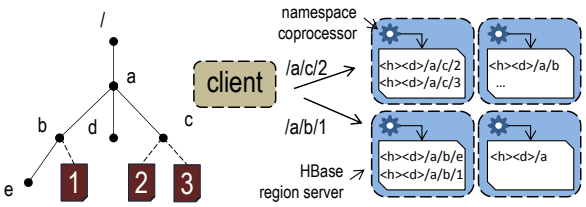


Figure 4: *Giraffa stores its metadata in HBase, which partitions its table as a B-Tree. Each file or directory is mapped to an unique row at one HBase region server. The current implementation of Giraffa does not have hierarchical permission checking so no pathname resolution is performed.*

beforehand. In our experiments, this trick allows Giraffa to immediately distribute the key space to all region servers as soon as the system starts up, achieving static load balance without the overhead of incremental data migration.

Relaxed Operation Semantics Giraffa relaxes semantics of several metadata operations. For access control, Giraffa does not check the permission information of every ancestor directory when accessing a file. This reduces the number of client-sever communications and improves performance. To support a POSIX access control model, it could either adopt a metadata caching technique used by IndexFS, or use a schema like Lazy Hybrid (LH) [8] or CalvinFS [43] that replicates a directory's permission bits to all files nested beneath it. The row key schema used by Giraffa also affects directory rename operation. Since the directory name is part of its children's row keys, rename requires read-modify-write on all of its children files. Because of the difficulty of supporting atomic rename for directories, Giraffa currently only supports rename of files in the same directory.

3.3 Replicated directories w/sharded files (ShardFS)

ShardFS scales distributed file system metadata performance by fully replicating directory lookup state across servers, insuring each file operation on a server is single-site [41] so it need not obtain locks from multiple servers. This replication strategy slows down mutations that affect this information (changes to directory names, directory access permissions, or namespace parent-child structure of pairs of directories)

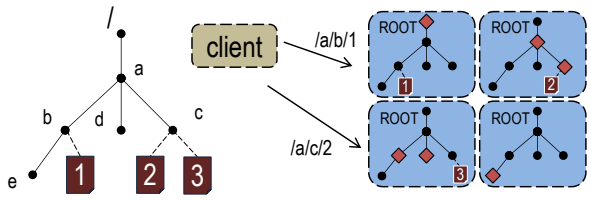


Figure 5: *ShardFS replicates directory lookup state to all metadata servers so every server can perform path resolution locally. File metadata (boxes) and non-replicated directory metadata (diamonds) is stored at exactly one (primary) server determined by a hash function on the full pathname.*

in order to speed up and load balance accesses to the objects reached by a successful pathname lookup as shown in Figure 5. Every metadata server contains a complete replica of this namespace information, so a single RPC to the appropriate (single-site) metadata server will be sufficient to complete pathname resolution, similar to [5]. Unlike namespace tree structures, file metadata is stored only in one metadata server. They are distributed by a sharding function using pathname as input. The metadata server that a pathname shards to is defined as the primary metadata server for this pathname. When a pathname represents a file, its metadata is stored only on the primary metadata server.

Because it fully replicates directory lookup state, ShardFS is based on pessimistic multi-server locking for all metadata operations. However, operations accessing metadata only on one metadata server can execute with one RPC to the primary metadata server, internally serializable and transactional for metadata operations. Almost all operations on file metadata or non-replicated directory metadata (e.g. timestamps) are single RPC operations. The scaling benefit that these operations get from namespace replication is the primary motivation for this technique.

File System Specific Optimizations ShardFS strives to reduce the latency for single RPC operations on files by not blocking them on locks taken by a replicated state mutation. Specifically, using analysis similar to what was used for H-store [41], ShardFS uses optimistic concurrency control for single RPC operations and will fall back to retry with two phase locking when optimistic verification fails. With the limited number and semantics of file system metadata operations, ShardFS does not compare the entire read and write sets for concurrent operations. Instead, file system operation semantics are used to detect optimistic verification failures, causing single site transactions to abort and retry with pessimistic locking.

The file system operations that are not single RPC are:

1. increasing permitted operations on replicated state: create a directory (*mkdir*) or increase permitted operations (*chmod* +mode, or *chgrp* +group)
2. decreasing permitted operations on replicated state: remove a directory (*rmdir*) or decrease permitted operations (*chmod*-mode, or *chgrp* -group)
3. increasing and decreasing permitted operations on replicated state: changes that both allow and deny operations (rename, mode changes that add and delete permissions, changes to ownership).

For the first two, monotonic changes in the set of operations that will be permitted, we design our distributed transactions so that concurrent single RPC operations that might detect inconsistent replicated state will all fail, allowing the calling client to recognize potential optimistic concurrency control failure and retry with pessimistic locking.

For the third class of operations, non-monotonic changes in the set of operations that will be permitted, we design our distributed transactions to be maximally defensive—they serialize all operations (even optimistic single RPC operation) with conflicting scope by taking locks at every server. As slow as these operations can be, they are quite rare in our traces and do not significantly impact performance results.

For distributed transactions requesting monotonic changes in permitted operations, ShardFS first serializes distributed transactions with conflicting scope. This could be done by taking locks on any deterministically selected single metadata server, but we instead use a separate lock server because it will also serve as the redo log server for clients that fail during a distributed transaction. This ensures that among all concurrent transactions with conflicting scope there is either one non-monotonic distributed transaction running serially or one monotonic distributed transaction running concurrently with multiple single RPC transactions.

For a conflicting concurrent single RPC transaction with target object t, a monotonic distributed transaction either changes object t or conflicts on the pathname to object t. Fortunately pathname lookup only depends on each pathname component's existence and permission check, so concurrent single RPC operations only learn that this operation is permitted or they fail and fall back to pessimistic concurrency control. Concurrent single RPC operations with success on their pathname lookups will not observe changes made by conflicting concurrent distributed transactions, placing constraints on the commit point for the distributed transactions.

There are two cases to consider: monotonically increasing permitted operations and monotonically decreasing permitted operations. The procedures to implement these two types of distributed transactions are presented in Procedure 1 and 2. In our analysis of these cases we separately consider conflicts involving replicated state on its primary server and conflicts involving non-primary replicated state.

For type 1, distributed transactions requesting monotonically increasing permitted operations on object t, concurrent single RPC operations may be affected by object t on their pathname or their target. In the latter case the concurrent

Procedure 1 type 1 distributed transaction *op*

I1: Log and take a lock on lock server
I2: Execute *op* on primary server
I3: If I2 succeeds [commit point], execute *op* (which must now succeed) in parallel on all non-primary (replica) servers
I4: Log and unlock

Procedure 2 type 2 distributed transaction *op*

D1: Log and take a lock on the lock server
D2: In parallel execute *op* on all non-primary servers
D3: If every *op* in D2 succeeds, execute *op* on primary
D4: If every *op* in D2&D3 succeeds, continue [commit point]; otherwise rollback *op* on non-primary servers
D5: Log and unlock

single RPC is running on the primary server and is serialized with step I2 in Procedure 1. In the former case, the concurrent single RPC may see an inconsistent version of t on their pathname lookup during step I3. Since the distributed transaction is only increasing permitted operations, this inconsistency could at worst force the conflicting single RPC operation to fail and retry taking locks at every server.

For type 2, distributed transactions requesting monotonically decreasing operations on object t, concurrent single RPC operations whose target is also object t, will be serialized on t's primary server and will not see inconsistent state. However, if the concurrent single RPC transaction sees t on its pathname, it could see changes to t before the commit point in step D4. Since the concurrent distributed transaction is only decreasing permitted operations, concurrent single RPC operations cannot pass pathname lookup unless that would have been true before the distributed transaction started. Any single RPC transaction seeing a temporary value for t during pathname lookup sees only reduced permissions, possibly causing it to fail and retry taking locks at every server.

In summary, any concurrent single RPC operation that might observe inconsistent state because of replicated pathname lookup will experience failure, then retry with pessimistic concurrency control.

Implementation ShardFS is implemented as a new HDFS client library written in JAVA on top of standalone IndexFS servers [35, 37] with modifications to support server side pathname resolution. Sharing the same IndexFS code base provides a fair comparison between these two approaches. The namespace replication with optimized directory lookup state mutation operations is implemented on the client side. We use a simple lock server to resolve races among all directory lookup state mutation operations and to store redo logs. Locking a pathname is equivalent to acquiring a write lock on the last component of the path and read locks on all ancestors. We also implemented a HDFS client library with HDFS's Namenode as its metadata server. However, since the performance of a single Namenode turns out to be inferior to a single IndexFS server, we only show ShardFS on top of IndexFS servers in the evaluation section.

Fault Tolerance Replication in ShardFS is only for performance. ShardFS relies on high availability of its underlying metadata servers. IndexFS tolerates failures by replicating data in its underlying storage (HDFS in our experiments) as described in section 3.1. As described before, the lock server also keeps logs of outstanding directory metadata mutation operations to tolerate client failures. When a client fails, later lock acquisition on the same path will trigger a recovery process for the failed operation. To tolerate lock server failure, we can run a backup lock server. The active lock server only grants a lock after the request has been replicated to the backup lock server.

3.4 Comparison Summary

Table 1 summarizes the design differences among the three metadata systems. To analyze the performance implication of these designs under various workloads, we discuss a few aspects of the three systems:

RPC amplification of metadata operations RPC amplification is defined as the the number of RPCs sent for an operation. One major source of RPC amplification comes from path resolution. POSIX semantics require accessing each ancestor directory on the path to check permissions. The path components may be stored on different servers in a distributed namespace, which require multiple RPCs to fetch. ShardFS replicates directory metadata to every metadata server for server local pathname resolution, complicating the protocol of directory metadata mutation operations. In IndexFS, both clients and servers maintain a consistent cache of path attributes to reduce RPCs, which can be limited by cache effectiveness. Giraffa abandons the access control model of POSIX by only checking permission bits of the final component. By using the full pathname as part of the primary key, most single file metadata operations in Giraffa require only one RPC.

Another source of RPC amplification is the partitioning and replication of metadata. For directory metadata mutation operations, the ShardFS client contacts all metadata servers to execute distributed transactions, adding to RPC amplification. For IndexFS, attribute modification operations such as *chmod* and *chown* require only one RPC but may need to wait for lease expiry. Moreover, *mkdir* and *splitting* involve two servers to perform a distributed transaction. Also *rmdir* is a distributed transaction that checks each partition to see if the directory is empty. The RPC amplification for *readdir* is proportional to the number of partitions in the directory.

Metadata operation latencies Both Giraffa and ShardFS try to keep the latency of file metadata operations low as one round-trip RPC. In this case, latency is mostly affected by server load. For most directory metadata mutations in ShardFS, multiple RPCs are issued to all metadata servers in parallel. Thus its latency is sensitive to the slowest RPC.

	Replicated directories with sharded files	**Dynamically partitioned namespace**	**Table partitioned namespace**
Example system	ShardFS	IndexFS	Giraffa
Metadata distribution	Replicated directory lookup state; sharded files	Partitioned into directory subsets	Partitioned by HBase
Metadata addressing	hash(pathname)	parent directory's inode num + hash(filename)	hash(path prefix) + depth + pathname
File operation (*stat*, *chmod*, *chown*, etc.)	1 RPC for all operations	Many RPCs for path traversal depending on cache locality	1 RPC for *stat* and chmod, 2 RPCs for mknod
Directory metadata mutations (*mkdir*, *chmod*, etc.)	Optimized monotonic distributed transactions	1 RPC that waits for lease expiration	2 RPCs, similar to file operations
Concurrency control	Optimistic single RPC ops retry w/pessimistic locking	Locking at directory partition level	Serialized by each tablet server
Client caching	Only cache configuration info	Cache path prefix ACLs and directory partition location	Cache the location of tablets
Load balancing	Load balance file access by hashing	Dynamically assign directory; split large directory by size	Pre-split and dynamically split tablets by size

Table 1: Summary of design choices made by three metadata services.

For IndexFS, the latency of metadata operations is affected by the hit ratio of the client's directory entry cache. Directory metadata mutations such as *chmod* and *chown* are also sensitive to the lease duration in IndexFS. Giraffa will have problems similar to IndexFS when it plans to support POSIX semantics for access control.

Consistency model for metadata operations All three metadata systems guarantee serializability for file metadata operations. The accuracy of access and modification time stamps for directories are relaxed for better performance in all three systems. *readdir* behavior under concurrency is not well defined in POSIX, which makes it flexible for system designers. All three systems provide an isolation level called "read committed" [7]. A *readdir* will always reflect a view of the directory at least as new as the beginning of the operation. It reflects all mutations committed prior to the operation, but may or may not reflect any mutation committed after the issue of the operation. This allows the systems to implement *readdir* as multiple independent requests to each directory partition without locking the entire directory.

Load balance across metadata servers An important scalability factor is the balance of load across metadata servers. Overall performance is limited by the slowest server. IndexFS uses a dynamic growth policy such that a newly created directory starts with one server and is dynamically split as it grows. While this policy maintains locality for small directories, it may experience higher load variance when a directory is close to splitting. Servers containing the top of the namespace may get higher load due to client cache misses or renewal. Giraffa will experience similar problems as it also splits the tablet according to the size. Since the split is handled by HBase, a directory in Giraffa is not necessarily clustered nor well balanced across severs. ShardFS, on the contrast, maintains both capacity and load balance by sharding files and replicating directory metadata at all times. It has a more balanced load across servers, but its directory metadata mutation is slower. Currently none of the systems implement strategies for the case where a few popular files dominate a workload [19].

Scalability Both IndexFS and Giraffa scale in throughput as more servers are added to the system for most workloads. In contrast, ShardFS's directory metadata mutation operations gets slower with more servers due to replication. In the extreme, if the workload only contains directory metadata mutation operation, ShardFS with more servers gets slower. However, when the ratio between directory metadata mutation operations and file metadata operations drops with more servers (see section 4.3 on weak scaling metadata workload), performance scales in ShardFS.

High availability Currently all three systems rely on the underlying storage system for information reliability. In future work we could replicate incoming operations to backup servers and use auxiliary service such as Zookeeper for leader selection and fast failover to provide higher availability. The quorum leases project [30] offers a technique to lower read latency in highly available Paxos-style systems.

4. Evaluation

In this section, we report experiments done on IndexFS, ShardFS, and Giraffa. To run these experiments, we have used up to 256 machines partitioned into up to 128 server nodes and up to 128 client nodes. Running client and server processes on separate machines facilitates better resource isolation and simpler performance measurement. All ma-

CPU	2x AMD Opteron 252 (2.6GHz)
Memory	8GB
NIC	1Gbps Ethernet
Storage	1TB 7200 RPM HDD
OS	64-bit Ubuntu 12.04.5 with Linux 3.2
Software	Hadoop Release 2.4.1, Giraffa Release 0.0.1

Table 2: PRObE Kodiak testbed configuration.

chines are part of the NSF PRObE Kodiak cluster [22] described in Table 2.

All systems were built on top of HDFS, which served as an underlying object storage infrastructure holding three copies of file data and file system metadata logs. Giraffa additionally depended on HBase (version 0.94) to manage its file system metadata. For IndexFS, the threshold for directory splitting (Section 3.1) was set to 2,000 entries in all tests. In addition, all systems were configured to asynchronously commit (sync) metadata log entries to the underlying storage (HDFS) every 5 seconds. All experiments were repeated at least three times and showed insignificant variance in observed performance. The benchmark framework that we used is a modified version of YCSB++ [33, 36] that supports multi-client coordination.

4.1 Microbenchmarks

In this section, we report on a set of microbenchmarks run on the three targeted file systems to study the tradeoffs among their designs. To evaluate the impact of different namespace distribution techniques, we prepared three distinct file system images representing different namespace tree structures.

#1. Balanced Tree: In a balanced tree, each internal directory has 10 sub-directories, and each leaf directory has 1,280 children files. The height of the tree is 5. In total there are 111 K directories and 128 M files.

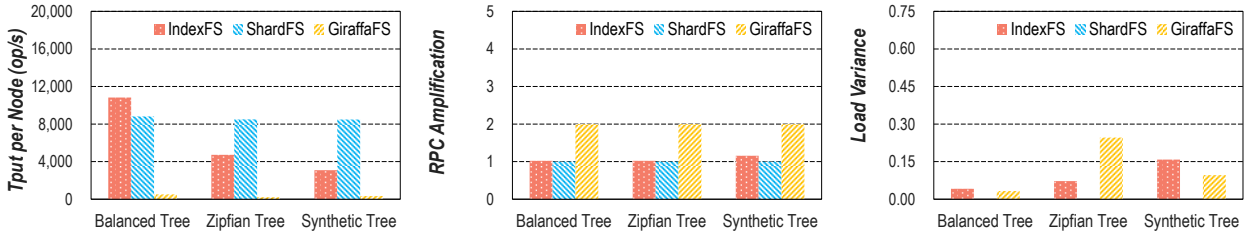
#2. Zipfian Tree: Zipfian trees share the same internal tree structure as balanced trees, except that the sizes of its leaf directories are randomly generated by a Zipfian distribution with an exponent parameter of 1.8. There are 111 K directories and approximately 128 M files inside the tree.

#3. Synthetic Tree: This tree is generated by a workload generator named Mimesis [2]. Mimesis models existing HDFS namespace images based on several statistical characteristics such as the distribution of the number of files per directory and the number of sub-directories per directory. For this tree, we used Mimesis to model and scale a HDFS trace extracted from a Yahoo! cluster. The original Yahoo! namespace had 757 K directories and 49 M files. Mimesis expanded this file system image to contain 1.9 M directories and 128 M files with the same statistical characteristics as the original Yahoo! namespace.

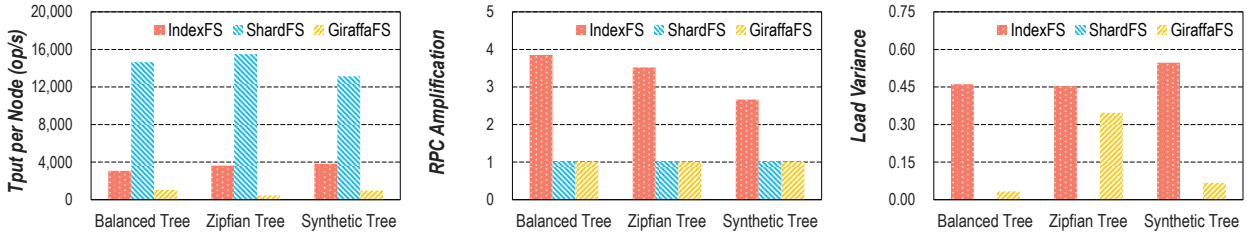
To microbenchmark the metadata performance of each namespace tree, we designed a simple three-phase workload. The first phase creates all internal directories. The second phase populates the namespace with empty files. During the third phase, each client performs *stat* on files randomly selected from the namespace. To model different access patterns, files to be accessed are chosen either uniformly or following a Zipfian distribution with an exponent parameter of 1.8. In all cases, the number of client threads is selected to saturate the target file system at its maximal throughput (an external tuning knob for YCSB++). We measured three different metrics: *average throughput per server*, *RPC amplification*, and *load variance*. The RPC amplification is reported as the total number of RPCs over the total number of application-level file system operations. The load variance is measured as the coefficient of variation of the number of RPC requests received by each metadata server. All microbenchmark experiments were run with 128 machines with 64 configured as servers and 64 as clients.

Figure 6 shows experimental results for the file creation and query phases. In general, Giraffa is much slower than IndexFS and ShardFS. According to profiling, we believe that the main reason lies in less optimized code in Giraffa's implementation, such as inefficient memory copies, overhead of communicating with HBase and the use of global locks. As a result, in this section, we mainly focus on RPC amplification and load variance when comparing with Giraffa.

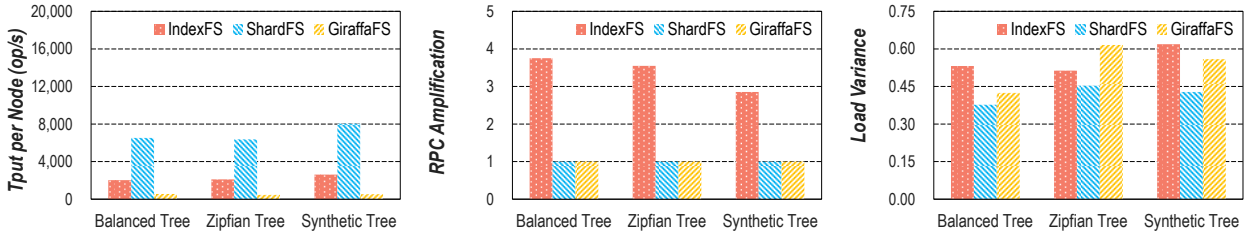
In the file creation phase, IndexFS achieves its highest throughput in the balanced tree workload, because the other two workloads have a few very large directories. Very large directories gives rise to a set of hot servers performing background activities to spread those large directories to multiple servers and balance the system for future operations. IndexFS also shows higher load variance in the later two workloads. This is because populating files for the Zipfian and synthetic trees produces namespace lookup requests that are imbalanced by nature. Fortunately, as files are created with depth-first order preserving path locality, the RPC amplification of IndexFS during the entire file creation phase is relatively low compared to that observed in the query phase. Unlike IndexFS, the performance of ShardFS tends to be stable for all three workloads. In fact, with files uniformly distributed across all of its servers, ShardFS can often achieve good load balance for file creates regardless of the actual file system tree structure. Different from both IndexFS and ShardFS, a large directory in Giraffa can easily be held entirely by a single HBase region server and become a performance bottleneck. In addition to this vulnerability to load imbalance, Giraffa also shows higher RPC amplification. As Giraffa has to check the existence of the parent directory when creating a new file, there is an additional RPC for almost all file creation operations, because parent directories are likely to be distributed to a remote region server according to Giraffa's current namespace partitioning strategy. In fact, according to POSIX semantics, Giraffa should have consulted all ancestor directories before it can ever create a new file. Unfortunately, if enforced, this would lead to even more severe RPC overhead.



(a) Creation Phase: Throughput, RPC amplification, and Load variance



(b) Query Phase (with uniformly distributed file stats): Throughput, RPC amplification, and Load variance



(c) Query Phase (with zipfian distributed file stats): Throughput, RPC amplification, and Load variance

Figure 6: *Performance comparison for creating and stating zero-byte files with 64 server machines and 64 client machines.*

Lesson#1: ShardFS shards each directory to all servers, maximizing the insertion throughput for every directory. IndexFS incrementally partitions directories as they grow, which can slow down the system when directories are small. Giraffa rarely splits individual directories and can suffer high load variance in practice.

For the query phase with uniform file selection, ShardFS shows excellent load balance, lower RPC amplification, and higher throughput than the other two. This is because ShardFS always distributes files evenly to all of its metadata servers and each metadata server can perform pathname lookups locally without contacting peer servers. Unlike ShardFS, IndexFS's performance is largely limited by its RPC amplification, which can in part be attributed to this random read access pattern. Random requests make IndexFS's client-side lookup cache relatively useless, causing more cache misses and forcing IndexFS clients to frequently fetch directory lookup state from servers. However, since IndexFS is able to dynamically split large directories, it doesn't get bottlenecked on large directories even with a skewed namespace such as the Zipfian and synthetic trees. In fact, IndexFS performs better in these namespaces as its client-side lookup cache becomes more effective. This leads to lower RPC amplification and higher throughput, albeit higher load variance. For Giraffa, due to its inability to split large directories, its load variance is largely determined by the shape of the namespace tree. For example, it shows higher load variance with the Zipfian tree workload. In its favor, since Giraffa does not actually perform pathname resolution like IndexFS and ShardFS, the RPC amplification for Giraffa is always one for all file stat operations. Finally, all three file systems show lower performance and higher load variance when files are selected following the Zipfian distribution.

Lesson#2: IndexFS's file *stat* performance is mainly a function of its cache effectiveness. ShardFS is able to deliver deterministic fast file *stat* performance by replicating directory lookup state. Giraffa often suffers load imbalance even without performing pathname resolution.

4.2 Trace Replay and Scaling Experiment

In this section, we compare the performance of IndexFS and ShardFS using workloads constructed from a real-world trace. We also performed experiments on Giraffa but decided not to show these results because the throughput is much lower than IndexFS and ShardFS. For this macrobenchmark, we used an one-day trace collected from a LinkedIn cluster and generated a two-phase trace-replay workload. The namespace in this trace consists of 1.9 M directories and 11.4 M files. We set all files to be empty since we only fo-

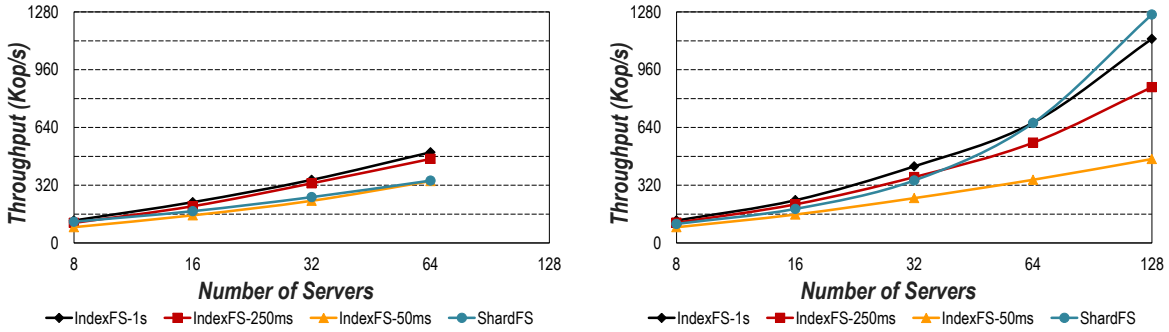


Figure 7: *Aggregated throughput replaying an one-day trace collected from LinkedIn versus number of machines using either strong scaling (a) (Section 4.2) or (b) weak scaling (Section 4.3). Figure 7(a) is on the left and Figure 7(b) is on the right.*

cus on metadata scalability in this paper. The first phase of the workload recreates all the files and directories referenced in the trace. In the second phase, each client independently replays a set of non-overlapping portions of the trace, which has been pre-divided into a series of sub-traces each containing 200 consecutive operations.

We used this trace-replay workload to test the strong scaling performance of both file systems. By strong scaling, we mean that the workload remains the same as we increase the size of the system. For IndexFS, in addition to the default client cache expiry of 1 second, we also tested two shorter cache expiration times: 250ms and 50ms. This allows us to demonstrate the performance of IndexFS under requirements for shorter directory mutation latencies.

Figure 7(a) shows the aggregated throughput during the replay phase when scaling the number of servers and a matching number of clients. In general, both systems demonstrate sub-linear scalability. As the number of clients increases, each client gets fewer operations to replay. For IndexFS, this inevitably reduces cache effectiveness and introduces higher RPC amplification. Since all clients will have to refresh directory lookup state for entries at the top of the tree when the cache is cold, metadata servers storing these directory entries will eventually become a performance bottleneck, effecting the scalability of IndexFS. On the other hand, ShardFS suffers from the increasing overhead of directory lookup state mutation operations as system resources grow. Though each metadata server gets fewer file operations in a bigger system, replication causes the number of directory lookup state mutations to remain the same. Besides, directory lookup state mutations in ShardFS can experience more blocking than IndexFS because there are more state updates in each ShardFS server, leading to more LevelDB compactions. With strong scaling, directory lookup state mutation operations can dominate ShardFS servers' load and strongly inhibit its scalability.

Figure 8(a) and 8(c) show the latency distribution for *stat* (read) and *chmod* (mutation) on directories respectively. Because tail latency is mainly dominated by major compactions, we choose to show 50th, 70th and 99th percentile latency. For *stat*, since ShardFS completes the request in a single RPC, its latency is better than that in IndexFS even at the 99th percentile. IndexFS with a shorter expiry time sees longer latency for both 50th and 70th percentile latency due to more cache misses and renewals. For *chmod* on directories, because ShardFS always waits for the slowest *chmod* to finish, the latency is much higher than that in IndexFS. For IndexFS, the 70th percentile latencies are comparable among all IndexFS configurations (and are much smaller than the lease expiration time) because directory lookup state not cached by any IndexFS clients can simply be modified without waiting for leases to expire. The 99th percentile tail latency, however, does reflect the actual lease expiration setting used in IndexFS.

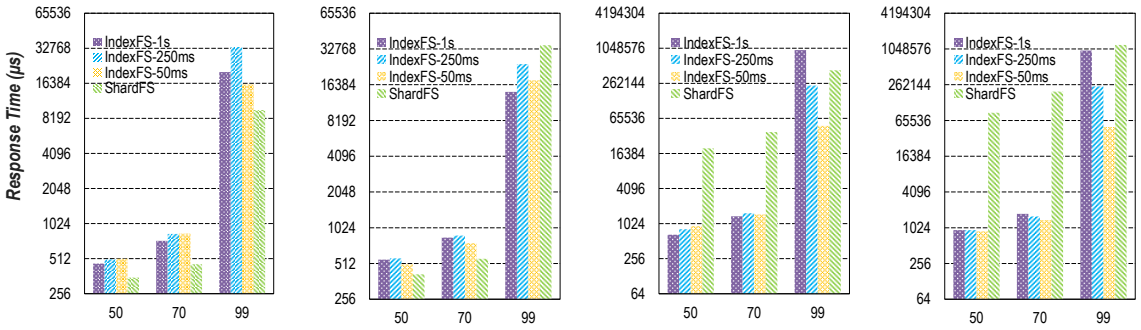
Lesson#3: Pathname lookup is a key bottleneck limiting file system metadata scalability. IndexFS suffers from load imbalance caused by cache renewals at top of the namespace tree. ShardFS is throttled on the write path with increasing overhead of replication.

4.3 Weak Scaling Workload Experiment

Because of the unavoidable non-scalability of strong scaling experiments, we define weak scaling for metadata workloads. Strong scalability measures speedup with an increasing amount of resources under a fixed amount of work, demonstrated with experiments inthe previous section (4.2).

Weak Scaling Metadata Workload Conjecture: Large parallel jobs may divide their work into a fixed number of files per core, but their change to the broader namespace (that is, directory creation) is likely to be fixed per job. If a large cluster is used by a weak scaling workload, it will do scalably more work in the same number of jobs (often a single scalable job).

We define weak scalability for metadata workloads as a linear increase in job throughput when file operation rate scales with increasing resources but mutations on directory lookup state remains constant. As Amdahl's law predicts, strong scalability is rarely achievable; weak scaling workloads therefore are a better way to measure if an application and the file systems it uses are scalable.



(a) *Stat* in Strong Scaling (b) *Stat* in Weak Scaling (c) *Chmod* in Strong Scaling (d) *Chmod* in Weak Scaling

Figure 8: *50th, 70th, and 99th percentile latency for stat operations, and chmod operations on directories.*

We enhanced our benchmark tool with support for weak scaling workloads. We first create file variants with names formed by appending a suffix. This allows us to scale the number of files in each directory as the system size grows. File operations are also generated against these files to mimic the effect of more files as the system scales. Directories and their operations remain the same with increasing scale, but both the number of files and file operations increases linearly with the size of the system. For each experiment, we adjust the number of YCSB threads in each client to achieve the best performance.

Figure 7(b) shows a weak scaling version of the experiment in the last section. All systems show the same performance as in Figure 7(a) with 8 servers because this is the same workload. With more servers, all systems scale better than with a strong scaling workload. IndexFS-50ms starts to flatten out as it reaches 128 servers. IndexFS-1s performs 17% worse than ShardFS at 128 servers even though it has 12% better performance at 8 servers, a relative shift down by 25%. With a weak scaling workload, the number of requests sent to each ShardFS metadata server is expected to be the same. The load is balanced across all metadata servers when there are no extremely popular files,so ShardFS scales well with a weak scaling workload. On the contrary, IndexFS experiences higher RPC amplification and thus results in higher load variance as the system grows, due to more client requests for the directories at the top of the namespace.

Figure 8(b) and 8(d) show the latency distribution for *stat* and *chmod* on directories operations with weak scaling workload at 64 servers. Without extra RPC requests for *stat* operations, ShardFS completes 70th percentile operations faster than IndexFS. ShardFS sees higher *stat* latencies in the tail than in strong scaling workload because *stat* operations can be affected by rare but very long compactions. Since directory mutations are replicated on each metadata server generating more changes in the state on each server, compactions are triggered more frequently in the ShardFS metadata servers, and compete with request execution for CPU resources. This can be alleviated by optimizing individual ShardFS metadata servers to separate file and directory metadata, reducing the amount of rewriting data in compactions. For *chmod* on directories, most operations in IndexFS finish much before the timeout. Because IndexFS tracks the last time any client cached any entry in a directory, IndexFS can complete a directory mutation operation on cold directories without waiting for an expiry time. In ShardFS, clients not only need to acquire locks for the operation, but also need to wait for the operation to complete on all metadata servers machines. A slow *chmod* slows the whole operation and results in higher latency in ShardFS.

Lesson#4: ShardFS is able to drive more scalability when the frequency of metadata updates is fixed and independent of the size of the system. Latencies, however, are increased proportionally to throughput increase.

5. Conclusion

Many cloud storage systems lack a general-purpose scalable metadata service that distributes both namespace and directories. This paper identifies key design trade-offs that need to be made for delivering scalable and parallel metadata performance. A novel system design called ShardFS is proposed specifically for weak scaling metadata workloads. ShardFS replicates directory lookup state for higher throughput and better load balance. Our experiments systematically compare the metadata operation performance of ShardFS with alternative designs under a variety of common cloud storage workloads. We found that an IndexFS-like solution highly relies on cache locality to reduce pathname lookup overhead and achieve load balancing. Although it can provide better performance for strong scaling metadata workloads and certain metadata mutation operations like *chmod* on directories. Giraffa sacrifices file system semantics for reducing network round trips. Because of its layering approach, the underlying table software may not provide ideal load balancing for metadata workload, especially concurrent accesses to large directories. ShardFS with server replication can scale more linearly and provide lower 50th and 70th percentile read response time than systems with client caching for weak scaling and read intensive metadata workloads.

Acknowledgments

This research was supported in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE, www.nmc-probe.org), the DOE and Los Alamos National Laboratory, under contract number DE-AC52-06NA25396 subcontracts 161465 and 153593 (IRH-PIT), the Qatar National Research Foundation, under award 09-1116-1-172, a Symantec research labs graduate fellowship, and Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC). We thank the DoD and Sterling Foster for support and problem formulation. We thank Robert Chansler, Chenjie Yu, LinkedIn and Yahoo! for giving us a trace of its HDFS metadata workload. We thank Sage Weil and Plamen Jeliazkov for helping us run and understand experiments on Ceph and Giraffa. We thank Sanjay Radia and Eric Baldeschwieler for helping us to get traces. We thank the members and companies of the PDL Consortium: Actifio, Avago, EMC, Facebook, Google, Hewlett-Packard, Hitachi, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Seagate, Symantec and Western Digital for their interest, insights, feedback, and support. We thank Dahlia Malkhi for shepherding our paper.

References

[1] Apache thrift. `http://thrift.apache.org`.

[2] C. L. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. H. Campbell. Metadata traces and workload models for evaluating big storage systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC)*. IEEE Computer Society, 2012.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th symposium on operating systems design and implementation (OSDI)*, 2002.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, volume 41, pages 159–174. ACM, 2007.

[5] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1(1):598–609, Aug. 2008. ISSN 2150-8097. . URL `http://dx.doi.org/10.14778/1453856.1453922`.

[6] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the conference on high performance computing networking, storage and analysis (SC)*, 2009.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, volume 24, 1995.

[8] S. A. Brandt, E. L. Miller, D. D. Long, and L. Xue. Efficient metadata management in large distributed storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*. IEEE Computer Society, 2003.

[9] J. Burbank, D. Mills, and W. Kasch. Network time protocol version 4: Protocol and algorithms specification. *Network*, 2010.

[10] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011.

[11] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[12] cassandra. Apache cassandra database. `http://cassandra.apache.org/`.

[13] Ceph. 'benchmark ceph mds' in user mailing list. `http://ceph-users.ceph.narkive.com/cTkbQ2C3/benchmarking-ceph-mds`.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed Database. In *Proceedings of the 10th USENIX conference on operating systems design and implementation (OSDI)*, 2012.

[15] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–31, 2014.

[16] S. Dayal. Characterizing HEC storage systems at rest. *Carnegie Mellon University PDL Technique Report CMU-PDL-08-109*, 2008.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, Berkeley, CA, USA, 2004.

[18] J. R. Douceur and J. Howell. Distributed directory service in the farsite file system. In *Proceedings of the 7th symposium on operating systems design and implementation (OSDI)*, 2006.

[19] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, page 23. ACM, 2011.

[20] L. George. Hbase: The definitive guide. In *O'Reilly Media*, 2011.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the 19th ACM symposium on operating systems (SOSP)*, 2003.

[22] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems

research. *USENIX; login*, 38(3), 2013.

[23] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *ACM SIGPLAN Notices*, volume 33. ACM, 1998.

[24] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of hdfs under hbase: a facebook messages case study. In *Proceedings of the 12th USENIX conference on file and storage technologies (FAST)*, pages 199–212, 2014.

[25] HDFS. Hadoop file system. `http://hadoop.apache.org/`.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[27] LevelDB. A fast and lightweight key/value database library. `http://code.google.com/p/leveldb/`.

[28] Lustre. Lustre file system. `http://www.lustre.org/`.

[29] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on file and storage technologies (FAST)*, 2011.

[30] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 1–13. ACM, 2014.

[31] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree. *Acta Informatica*, 33(4):351–385, 1996.

[32] S. Patil and G. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX conference on file and storage technologies (FAST)*, 2011.

[33] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[34] R. Ramakrishnan and J. Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[35] K. Ren and G. Gibson. TableFS: Enhancing metadata efficiency in the local file system. *USENIX annual technical conference (ATC)*, 2013.

[36] K. Ren, J. López, and G. Gibson. Otus: resource attribution in data-intensive clusters. In *Proceedings of the second international workshop on MapReduce and its applications (MapReduce)*. ACM, 2011.

[37] K. Ren, Q. Zheng, S. Patil, and G. Gibson. Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the conference on high performance computing networking, storage and analysis (SC)*, 2014.

[38] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX conference on file and storage technologies (FAST)*, 2002.

[39] ShardFS. Shardfs code release. `http://www.pdl.cmu.edu/ShardFS/index.shtml`.

[40] K. V. Shvachko. Hdfs scalability: The limits to growth. *USENIX ;login*, 35(2):6–16, 2010.

[41] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, 2007.

[42] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 224–237. ACM, 1997.

[43] A. Thomson and D. J. Abadi. CalvinFS: consistent wan replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[44] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on operating systems design and implementation (OSDI)*, 2006.

[45] B. Welch and G. Noer. Optimizing a hybrid ssd/hdd hpc storage system based on file size distributions. *Proceedings of 29th IEEE conference on massive data storage (MSST)*, 2013.

[46] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX conference on file and stroage technologies (FAST)*, 2008.