IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion

Kai Ren, Qing Zheng, Swapnil Patil, Garth Gibson
Carnegie Mellon University
{kair, zhengq, swapnil.patil, garth}@cs.cmu.edu

Abstract—**The growing size of modern storage systems is expected to exceed billions of objects, making metadata scalability critical to overall performance. Many existing distributed file systems only focus on providing highly parallel fast access to file data, and lack a scalable metadata service. In this paper, we introduce a middleware design called IndexFS that adds support to existing file systems such as PVFS, Lustre, and HDFS for scalable high-performance operations on metadata and small files. IndexFS uses a table-based architecture that incrementally partitions the namespace on a per-directory basis, preserving server and disk locality for small directories. An optimized log-structured layout is used to store metadata and small files efficiently. We also propose two client-based storm-free caching techniques: bulk namespace insertion for creation intensive workloads such as N-N checkpointing; and stateless consistent metadata caching for hot spot mitigation. By combining these techniques, we have demonstrated IndexFS scaled to 128 metadata servers. Experiments show our out-of-core metadata throughput out-performing existing solutions such as PVFS, Lustre, and HDFS by 50% to two orders of magnitude.**

Keywords*—*Distributed file systems, file system metadata, stateless caching, bulk insertion, log-structured merge tree

I. Introduction

Lack of a highly scalable and parallel metadata service is becoming an important performance bottleneck for many distributed file systems in both the data intensive scalable computing (DISC) world [26] and the high performance computing (HPC) world [32], [40]. This is because most cluster file systems are optimized mainly for scaling the data path (i.e., providing high bandwidth parallel I/O to files that are gigabytes in size) and have limited metadata management scalability. They either use a single centralized metadata server, or a federation of metadata servers that statically partition the namespace (e.g., Hadoop Federated HDFS [26], PVFS [46], Panasas PanFS [62], and Lustre [34]).

Limited metadata scalability handicaps massively parallel applications that require concurrent and high-performance metadata operations. One such application, file-per-process (N-N) checkpointing, requires the metadata service to handle a huge number of file creates all at the beginning of the checkpoint [9]. Another example, storage management, produces a read-intensive metadata workload that typically scans the metadata of the entire file system to perform administrative tasks [28], [30]. Finally, even in the era of big data, most files in even the largest cluster file systems are small [19], [61], where median file size is often only hundreds of kilobytes. Scalable storage systems should expect the number of small files stored to exceed billions, a known challenge for many existing file systems [44].

We envision a scalable metadata service that provides distributed file system namespace support to meet the needs of parallel applications. Our design is driven by recent workload studies on traces from several academic and industry clusters [19], [48], [61]. We observed a heavy-tailed distribution in many structural characteristics of file system metadata such as file size, directory size, and directory depth. For example, many storage systems have median file size smaller than 64KB, even as the largest file size can be several terabytes. Nearly 90% of directories in these storage systems are small, having fewer than 128 directory entries. But a few of the largest directories can have more than a million entries. In terms of file system metadata operations, read-only operations on files and directories such as `open` for read, `stat` and `readdir` are the most popular operations [48].

Based on the workload analysis, we propose two mechanisms to load balance file system metadata access across servers. The first mechanism is to partition the namespace at the granularity of a directory subset and dynamically splits large directories to achieve load balance. The splitting of a large directory is based on GIGA+ [44], a technique that preserves disk locality of small directories for fast `readdir` and enables parallel access to large directories. The second mechanism is to maintain metadata caches on the client with minimal server state to reduce unnecessary requests to metadata servers during path resolution and permission validation. This prevents servers from tracking and synchronizing a large number of states.

Prior file systems store metadata out-of-core using 30-year old on-disk data structures (such as inode and directories) that suffer many disk seeks during metadata intensive workloads [36], [15], [47]. The need for a more efficient out-of-core representation of directory entries motivated us to develop a novel metadata layout using a column-based, log-structured approach [14], [42], [49]. All file system metadata (including directories and inode attributes) and the data for small files are packed into a few log files, along with fine-grained indexing to speed up lookup and scan operations. This organization facilitates high-speed metadata creates, lookups, and scans, even in a single-node local file system [47].

To demonstrate the feasibility of our approach, we implemented a prototype middleware service called IndexFS that incorporates the above namespace distribution and caching mechanisms as well as the on-disk metadata representation. Existing cluster file systems, such as PVFS, HDFS, Lustre, and PanFS, can benefit from IndexFS without requiring any modifications to the original system. We evaluated the prototype on multiple clusters consisting of up to 128 machines. Our

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00 ©2014 IEEE

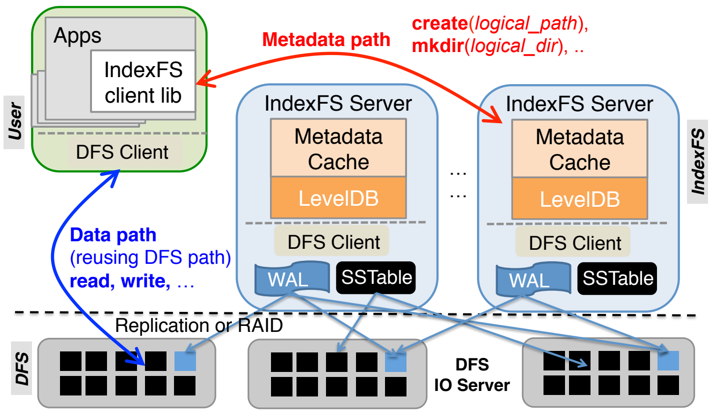


Fig. 1: *The IndexFS metadata system is middleware layered on top of an existing cluster file system deployment (such as PVFS or Lustre) to improve metadata and small file operation efficiency. It reuses the data path of the underlying file system and packs directory entries, file attributes and small file data into large immutable files (SSTables) that are stored in the underlying file system.*

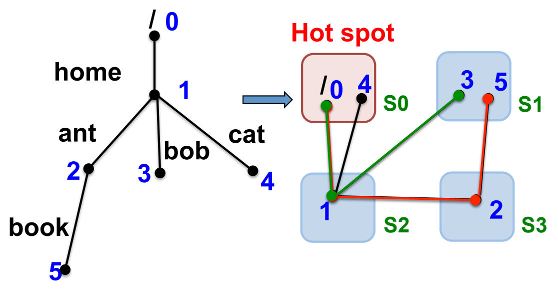


Fig. 2: *The figure shows how IndexFS distributes a file system directory tree evenly into four metadata servers. Path traversal makes some directories (e.g. root directory) more frequently accessed than others. Thus stateless directory caching is used to mitigate these hot spots.*

results show promising scalability and performance: IndexFS, layered on top of PVFS, HDFS, Lustre, and PanFS, can scale almost linearly to 128 metadata servers, performs 3000 to 10,000 operations per second per machine, and outperforms the underlying file system by 50% up to two orders of magnitude as the number of servers scales in various metadata intensive workloads.

II. Design and Implementation

IndexFS is middleware inserted into existing deployments of cluster file systems to improve metadata efficiency while maintaining high I/O bandwidth for data transfers. Figure 1 presents the overall architecture of IndexFS. The system uses a client-server architecture:

IndexFS Client: Applications interact with our middleware through a library directly linked into the application, through the FUSE user-level file system [2], or through a module in a common library, such as MPI-IO [17]. Client-side code redirects applications' file operations to the appropriate destination according to the type of operation. Metadata requests (e.g., `create` and `mkdir`), and data requests on small files with size less than 64KB (e.g., `read` and `write`), are handled by the metadata indexing module that sends these requests to the appropriate IndexFS server. For data operations on large files, client code redirects read requests directly to the underlying cluster file system to take full advantage of parallel I/O bandwidth. A newly created but growing file may be transparently reopened in the underlying file system by the client module. When a large file is reopened in the underlying file system for write, some of its attributes (e.g., file size and last access time) may change relative to IndexFS's per-open copy of the attributes. The IndexFS server will capture these changes on file close using the metadata path. IndexFS clients employ several caches to enhance performance for frequently accessed metadata such as directory entries, directory server mappings, and complete subtrees for (writeback) bulk-insertion. Details about these caches will be discussed in later sections.

IndexFS Server: IndexFS employs a layered architecture as shown in Figure 1. Each server owns and manages a non-overlapping portion of file system metadata, and packs metadata and small file data into large flat files stored in the underlying shared cluster file system. File system metadata is distributed across servers at the granularity of a subset of a directory's entries. Large directories are incrementally partitioned when their size exceeds a threshold. The module that packs metadata and small file data into large immutable sorted files (SSTables) uses a data structure called a log-structured merge (LSM) tree [42]. Since LSM trees convert random updates into sequential writes, they greatly improve performance for metadata creation intensive workloads. For durability, IndexFS relies on the underlying distributed file system to replicate or RAID encode the LSM tree's SSTable files and write-ahead logs. Details about fault tolerance techniques used in IndexFS is presented in Section II-E.

A. Dynamic Namespace Partitioning

IndexFS uses a dynamic namespace partitioning policy to distribute both directories and directory entries across all metadata servers. Unlike prior works that partition file system namespace based on a collection of directories that form a sub-tree [20], [60], our namespace partitioning works at the directory subset granularity. Figure 2 shows an example of distributing a file system tree to four IndexFS metadata servers. Each directory is assigned to an initial 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 (e.g., 90% of directories have fewer than 128 entries in many cluster file system instances [61]) since storing directory entries together preserves locality for scan operations such as `readdir`. The initial server assignment of a directory is done through random server selection. To reduce the variance in the number of directory entries stored in metadata servers, we also adapt the "power of two choices" load balancing technique [37] to the initial server assignment. This technique assigns each directory by probing two random servers and placing the directory on the server with fewer stored directory entries.

For the few directories that grow to a large number of entries, IndexFS uses the GIGA+ binary splitting technique to distribute directory entries over multiple servers [44]. Each directory entry is hashed to uniformly map it into a large hash-space that is range partitioned. GIGA+ incrementally splits a directory in proportion to its size: a directory starts small, on a single server that manages its entire hash-range. As the directory grows, GIGA+ 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 is using all metadata servers. This splitting stops after each server owns at least one partition of the distributed directory. IndexFS servers maintain and clients opportunistically cache a partition-to-server mapping to locate entries of distributed directories. These mappings are inconsistently cached at the clients to avoid cache consistency traffic; stale mappings are corrected by any server inappropriately accessed [43], [44].

B. Stateless Directory Caching

To implement POSIX file I/O semantics many metadata operations are required for each ancestor directory to perform pathname traversal and permission checking. This requires many RPC round trips if each check must find the appropriate IndexFS server for the directory entry subset that should contain this pathname component. IndexFS' GIGA+ removes almost all RPC round trips associated with finding the correct server by caching mappings of directory partition to server that tolerates inconsistency; stale mappings may send some RPCs to the wrong server, but that server can correct some or all of the client's stale map entries [44]. By using an inconsistent client cache, servers never need to determine which clients contain correct or stale mappings, eliminating the storms of cache updates or invalidation messages that occur in large scale systems with consistent caches and frequent write sharing [29], [51], [59].

Once the IndexFS servers are known, there is still a need for RPCs to test existence and permissions for each pathname component because GIGA+ caches only server locations. And this access pattern is not well balanced across metadata servers because pathname components near the top of the file namespace tree are accessed much more frequently than those lower in the tree (see Figure 2). IndexFS maintains a consistent client cache of pathname components and their permissions (but not their attributes) without incurring invalidation storms by assigning short term leases to each pathname component offered to a client and delaying any modification until the largest lease expires. This allows IndexFS servers to record only the largest lease expiration time with any pathname component in its memory and not per-client cache states. The server pins the entry in its memory and blocks updates until all leases have expired. This is a small amount of additional IndexFS server state (only one or two variables for each directory entry) and it does not cause invalidation storm.

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 outstanding leases to expire. Although this may incur higher latency for these mutation operations, client latency for non-mutation operations, memory and network resource consumptions are greatly reduced. This method assumes the clock on all machines are synchronized, which is commonly achievable in modern data centers [11], [16].

We investigate several policies for the lease duration for individual cached entries. The simplest is to use a fixed time interval (e.g., 200ms) for each lease. However, some directories, such as those at the top of the namespace tree, are frequently accessed and unlikely to be modified, so the lease duration for these directory entries benefits from being extended. Our non-fixed policies use two indicators to adjust the lease duration: one is the depth tree of the directory (e.g., $3sec/depth$), and the other is the recent read to write (mutation) ratio for the directory entry. This ratio is measured only for directory entries cached in the metadata server's memory. Because newly created/cached directory entries do not have an access history, we set the lease duration $L/depth$ where $L = 3s$ in our experiments. For directory entires that have history in the server's memory, we use a exponential weighted moving average (EWMA) to estimate the read and write ratio [4]. Suppose that r and w are the recent counts of read and write requests respectively, then the offered lease duration is $\frac{r}{w+r} \cdot L_r$, where $L_r = 1s$ in our experiments. This policy ensures that read-intensive directory entries will get longer lease duration than the write-intensive directory entries.

C. Log-Structured Metadata Storage Format

Our IndexFS metadata storage backend implements a LSM tree [42], using the LevelDB open source library [31], to pack metadata and small files into megabyte or larger chunks in the underlying cluster file system. LevelDB provides a simple key-value store interface that supports point queries and range queries. LevelDB accumulates the most recent changes inside an in-memory buffer and appends change to a write-ahead log for fault tolerance. When the total size of the changes to the in-memory buffer exceeds a threshold (e.g., 16 MB), these changed entries are sorted, indexed, and written to disk as an immutable file called an SSTable (sorted string table) [14]. These entries may then be candidates for LRU replacement in the in-memory buffer and reloaded later by searching SSTables on disk, until the first match occurs (the SSTables are searched most recent to oldest). The number of SSTables that are searched is reduced by maintaining the minimum and maximum key value and a Bloom filter [10] for each SSTable. However, over time, the cost of finding a LevelDB record that is not in memory increases. Compaction is the process of combining multiple overlapping range SSTables into a number of disjoint range SSTables by merge sort. Compaction is used to decrease the number of SSTables that might share any record, to increase the sequentiality of data stored in SSTables, and reclaim deleted or overwritten entries. We now discuss how IndexFS uses LevelDB to store metadata. We also describe the modifications we made to LevelDB to support directory splitting and bulk insertion.

Metadata Schema: Similar to our prior work on TableFS [47], IndexFS embeds inode attributes and small files with directory entries and stores them into a single LSM tree with an entry for each file and directory. The basic design of using an LSM tree to implement local file system operations is covered

key	*Parent directory ID, Hash(Name)*
value	*Name, Attributes, Mapping|File Data|File Link*

TABLE I: *The schema of keys and values used by IndexFS. Only the value of a directory contains the "mapping" data, which is used to locate the server of a directory partition.*

in TableFS [47], so here we only discuss the design details relevant to IndexFS. To translate the hierarchical structure of the file system namespace into key-value pairs, a 192-bit key is chosen to consist of the 64-bit inode number of a entry's parent directory and a 128-bit hash value of its filename string (final component of its pathname), as shown in Table I. The value of an entry contains the file's (unhashed) filename and inode attributes, such as inode number, ownership, access mode, file size, timestamps (*struct stat* in POSIX). For small files whose size is less than T (default is 64KB) the value field also embeds the file's data. For large files, the file data field in a file row of the table is replaced by a symbolic link pointing to the actual file object in the underlying distributed file system. The advantage of embedding small file data is to reduce random disk reads for lookup operations like `getattr` and `read` for small files, (i.e., when the users' working set cannot be fully cached in memory). However, this brings additional overhead during LSM Tree's compaction since embedding small files increases the data volume processed by each compaction.

Column-Style Table for Faster Insertion: Some applications, such as checkpointing, prefer fast insertion performance or fast pathname lookup rather than fast directory list performance. To better support such applications, IndexFS uses a second LSM table schema, called *column-style*, that speeds up the throughput of insertion, modification, and single-entry lookup. With this second smaller table for the most important operations, IndexFS can disable compaction of the full metadata table.

As shown in Figure 3, IndexFS's column-style schema adds a second index table sorted on the same key, stores only the final pathname component string, permissions and a pointer (to the most recent corresponding record in the full metadata table). Like a secondary index, this table is smaller than the full table, so it caches better and its compactions are less frequent. It can satisfy `lookup` and `readdir` operations, the most important non-mutation metadata accesses, without dereferencing the pointer. But it cannot statisfy `stat` and `read` without one more SSTable reference. IndexFS eliminates compaction in the full table (rarely, if ever, compacting the full table). Eliminating compaction speeds up insertion intensive workloads significantly (see Section III-D). Moreover, because the index table contains a pointer (log ID and offset in the appropriate log file), and because each mutation of a directory entry or its embedded data rewrites the entire row of the full table, there will only be one disk read if a non-mutation access is not satisfied in the index table, speeding up single file metadata accesses that miss in cache relative to the standard LevelDB multiple level search. The disadvantage of this approach is that the full table, as a collection of uncompacted log files, will not be in sorted order on disk, so scans that cannot be satisfied in the index table will be more expensive. Cleaning of no longer referenced rows in the full table and resorting by primary key (if needed at all) can be done by a background defragmentation service.

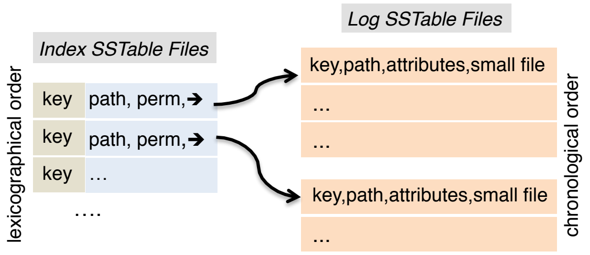


Fig. 3: *Column-style stores index and log tables separately. Index tables contain frequently accessed attributes for file lookups and a pointer to the location of full file metadata in the most recent log file. Index tables are compacted while log tables are not, reducing the total work for IndexFS.*

Partition Splitting and Migration: IndexFS uses a faster technique for splitting a directory partition than is used by GIGA+. The immutability of SSTables in LevelDB makes fast bulk insertion possible – a SSTable whose range does not overlap any part of a current LSM tree can be added to LevelDB (as another file at level 0) without its data being pushed through the write-ahead log, in-memory cache, or compaction process. To take advantage of this opportunity, we extended LevelDB to support a three-phase directory partition split operation:

- Phase 1: The server initiating the split locks the directory (range) and then performs a range scan on its LevelDB instance to find all entries in the hash-range that needs to be moved to another server. Instead of packing these into an RPC message, the results of this scan are written in SSTable format to a file in the underlying distributed file system.
- Phase 2: The split initiator sends the split receiver the path to the SSTable-format split file in small RPC message. Since this file is stored in shared storage, the split receiver directly inserts it as a symbolic link into its LevelDB tree structure without actually copying the file. The insertion of the file into the split receiver is the commit part of the split transaction.
- Phase 3: The final step is a clean-up phase: after the split receiver completes the bulk insert operation, it notifies the initiator, who deletes the migrated key-range from its LevelDB instance, unlocks the range, and begins responding to clients with a redirection for file in this range.

For the column-style storage schema, only index tables need to be extracted and bulk inserted at the split receiver. Data files, stored in the underlying shared distributed file systems, can be accessed by any metadata server. In our current implementation, there is a dedicated background thread that maintains a queue of splitting tasks to throttle directory splitting to only one split at a time. This is a simple way to reduce lock conflicts caused by multiple concurrent splits and migitate the variance in throughput and latency experienced by clients.

D. Metadata Bulk Insertion

Even with scalable metadata partitioning and efficient on-disk metadata representation, the IndexFS metadata server can only achieve about 10,000 file creates per second in our testbed cluster. This rate is dwarfed by the speed of non-server based systems such as the small file mode of the Parallel Log Structured Filesystem (PLFS [9]) which can achieve millions of file creates per second [57], [22]. Inspired by the metadata client caching and bulk insertion techniques we used for directory splitting, IndexFS implements write back caching at the client for creation of new directory subtrees. This technique may be viewed as an extension of Lustre's directory callbacks [51]. By using bulk insertion, IndexFS strives to match PLFS's create performance.

Since metadata in IndexFS is physically stored as SSTables, IndexFS clients can complete creation locally if the file is known to be new and later bulk insert all the file creation operations into IndexFS using a single SSTable insertion. This eliminates the one-RPC-per-file-create overhead in IndexFS allowing new files to be created much faster and enabling total throughput to scale linearly with the number of clients instead of the number of servers. To enable this technique, each IndexFS client is equipped with an embedded metadata storage backend library that can perform local metadata operations and spill SSTables to the underlying shared file system. As IndexFS servers are already capable of merging external SSTables, support at the server-side is straightforward.

Although client-side writeback caching of metadata can deliver ultra high throughput bulk insertion, global file system semantics may no longer be guaranteed without server-side coordination. For example, if the client-side creation code fails to ensure permissions, the IndexFS server can detect this as it first parses an SSTable bulk-inserted by a client. Although file system rules are ultimately enforced, error status for rejected creates will not be delivered back to the corresponding application code at the `open` call site, and could go undetected in error logs. Quota control for the (tiny fraction of) space used by metadata will be similarly impacted, while data writes to the underlying file system can still be growth limited normally.

IndexFS extends its lease-based cache consistent protocol to provide the expected global semantics. An IndexFS client wanting to use writeback caching and bulk insertion to speed up the creation of new subtrees issues a `mkdir` with a special flag "`LOCALIZE`", which causes an IndexFS server to create the directory and return it with a renewable write lease. During the write lease period, all files (or subdirectories) created inside such directories will be exclusively served and recorded by the client itself. Before the lease expires, the IndexFS client must return the corresponding subtree to the server, in the form of an SSTable, through the underlying cluster file system. After the lease expires, all bulk inserted directory entries will become visible to all other clients. While the best creation performance will be achieved if the IndexFS client renews its lease many times, it may not delay bulk insertion arbitrarily. If another client asks for access to the localized subtree, the IndexFS server will deny future write lease renewals so that the writing client needs to complete its remaining bulk inserts quickly. If multiple clients want to cooperatively localize file creates inside the same directory, IndexFS `mkdir` can use a "`SHARED_LOCALIZE`" flag, and conflicting bulk inserts will be resolved at the servers arbitrarily (but predictively) later. As bulk insertion cannot help data intensive workloads, IndexFS clients automatically "expire" leases once significant data writing is detected.

Inside a localized directory, an application is able to perform all metadata operations. For example, `rename` is supported locally but can only move files within the localized directory. Any operation not compatible with localized directories can be executed if the directory is bulk inserted to the server and its lease expired.

E. Fault Tolerance

IndexFS is designed as middleware layered on top of an underlying failure-tolerant and distributed file system. IndexFS's fault tolerance strategy is to push states into the underlying file system – large data into files, metadata into SSTables and recent changes into write-ahead logs (WAL). The IndexFS server processes are monitored by standby server processes that are prepared to replace failed server processes. Zookeeper, a quorum consensus replicated database, is used to store (as a lease) the location of each primary server [27]. Each IndexFS metadata server maintains a separate write-ahead log that records mutation operations such as file creates and renames. When a server crashes, its write-ahead log can be replayed by a standby server to recover consistent state.

Leases for client directory entry caching are not durable. A standby server restarting from logs blocks mutations for the largest possible timeout interval. The first lease for a localized directory should be logged in the write-ahead log so a standby server will be prepared for a client writing back its local changes as a bulk insert.

Some metadata operations, including directory splitting and rename operations, require a distributed transaction protocol These are implemented as a two-phase distributed transaction with failure protection from write-ahead logging in source and destination servers and eventual garbage collection of resource orphaned by failures. Directory renaming is more complicated than directory splitting because it requires multiple locks on the ancestor directories to prevent an orphaned loop [20]. Since this problem is beyond the scope of this paper, our current prototype has a limited-functionality `rename` operation that supports renaming only files and leaf directories.

IndexFS supports two modes of write-ahead logging: synchronous mode and asynchronous mode. The synchronous mode will group commit a number of metadata operations to disk to make them persistent. The asynchronous mode instead buffers log records in memory and flushes these records when a time (default 5 seconds) or size threshold (default 16KB) is exceeded. The asynchronous mode may lose data when a crash happens but provides much higher ingestion throughput than synchronous mode. Because most local file systems default to asynchronous mode, it is also our default in the experiments below.

III. Experimental Evaluation

The prototype of IndexFS is implemented in about 10,000 lines of C++ code using a modular design that is easily

	Kodiak	Susitna	LANL Smog
#Machines	128	5	32
HW year	2005	2012	2010
OS	Ubuntu 12.10	CentOS 6.3	Cray Linux
Kernel	3.6.6 x86_64	2.6.32 x86_64	
CPU	AMD Opteron 252, 2-core 2.6 GHz	AMD Opteron 6272, 64-core 2.1 GHz	AMD Opteron 6136, 16-core 2.4 GHz
Memory	8GB	128GB	32GB
Network	1GE NIC	40GE NIC	Torus 3D 4.7GB/s
Storage	Western Digital 1TB disk/node	PanFS 5-shelf 5 MDS,50 OSD	HW RAID array 8GB/s bandwdith
Tested FS	HDFS, PVFS	PanFS	Lustre

TABLE II: *Three clusters used for experiments.*

layered on existing cluster file systems such as HDFS [26], Lustre [34], PVFS [13], and PanFS [62]. Our current version implements the most common POSIX file system operations except `hardlink` and `xattr` operations. Some failure recovery mechanisms, such as replaying write-ahead logs, are not implemented yet.

All experiments are performed on one of three clusters. Table II describes the hardware and software configurations of the three clusters. The first cluster is a 128-node cluster taken from the 1000-node PRObE Kodiak cluster [25]. It is used to evaluate IndexFS's scaling performance and design trade-offs. In this cluster, IndexFS is layered on top of PVFS or HDFS, and its performance is compared against PVFS and HDFS. The second cluster (PRObE Susitna [25]) and the third cluster (LANL Smog [33]) are used to evaluate IndexFS's portability to PanFS and Lustre respectively. In all experiments, clients and servers are distributed over the same machines, every machine runs one or more client processes and a server process. The client uses an IndexFS library API, and the threshold for splitting a partition is always 2,000 entries. In asynchronous commit mode, the IndexFS server flushes its write ahead log every 5 seconds or every 16KB (similar to Linux local file systems like Ext4 and XFS [35], [56]). All tests were run for at least three times and the coefficient of variation of results is less than 2%.

A. Large Directory Scaling

This section shows how IndexFS scales to support large directories over multiple Kodiak servers. To understand its dynamic partitioning behavior, we start with a synthetic *mdtest* benchmark [3] to insert zero-byte files into a single shared directory [60], [44]. We generated a three-phase workload. The first phase is a concurrent create workload in which eight client processes on each node simultaneously create files in a common directory. The number of files created is proportional to the number of nodes: each node creates 1 million files, so 128 million files are created on 128 nodes. The second phase performs `stat` on random files in this large directory. Each client process performs 125,000 `stat` calls. The third phase deletes all files in this directory in a random order.

Figure 4 plots aggregated operation throughput, in file creates per second, averaged over the first phase of the benchmark as a function of the number of servers (1 server and 8

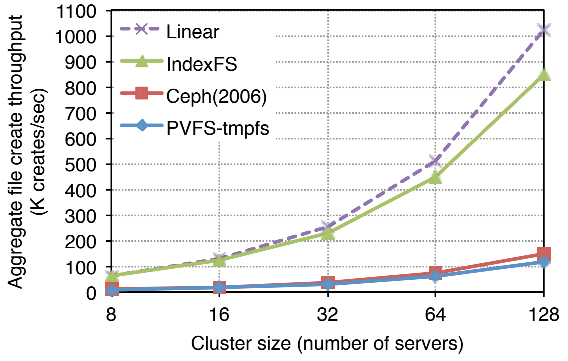


Fig. 4: *IndexFS on 128 servers deliver a peak throughput of roughly 842,000 file creates per second. The prototype RPC package (Thrift [1]) limits its linear scalability.*

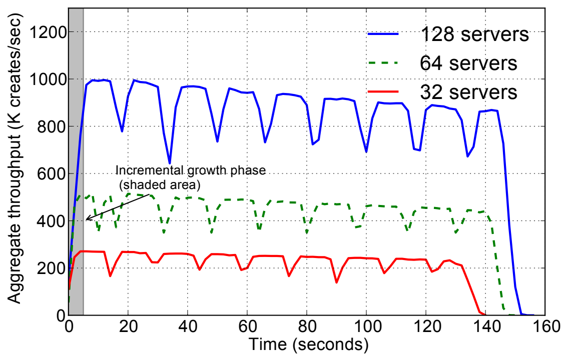


Fig. 5: *IndexFS achieves steady throughput after distributing one directory hash range to each available server. After scale-out, throughput variation is caused by the compaction process in LevelDB. Peak throughput degrades over time because the total size of the metadata table is growing so negative lookups do more disk accesses.*

client processes per node). IndexFS with SSTables and write-ahead logs stored in PVFS scales linearly up to 128 servers. IndexFS in this experiment uses only one LevelDB table to store metadata (without using column-style storage schema). With 128 servers, IndexFS can sustain a peak throughput of about 842,000 file creates per second, two orders of magnitude faster than current single server solutions.

Figure 4 also compares IndexFS with the scalability of Ceph and PVFS. PVFS is measured in the same Kodiak cluster, but since PVFS's metadata servers uses a transactional database (BerkeleyDB) for durability, which is stronger than IndexFS or Ceph, we let it store its records in a RAM disk to achieve better performance. When layered on top of Ext3 with hard disks, 128 PVFS servers only achieve one hundred creates per second. For Ceph, Figure 4 reuses numbers from the original paper [60] [1]. Still their experiments were performed on a cluster with a similar hardware and configuration. The reason that IndexFS outperforms other file systems is largely due to the use of log structured metadata layout.

[1]The directory splitting function in the latest version of Ceph is not stable. According to Ceph developers, the dynamic splitting function of current version of Ceph is often disabled when testing multiple metadata servers.

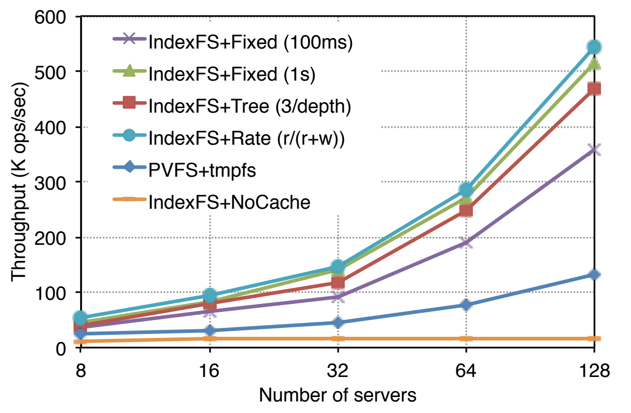


Fig. 6: *Average aggregate throughput of replaying 1 million operations per metadata server on different number of nodes using a one-day trace from a LinkedIn HDFS cluster.*

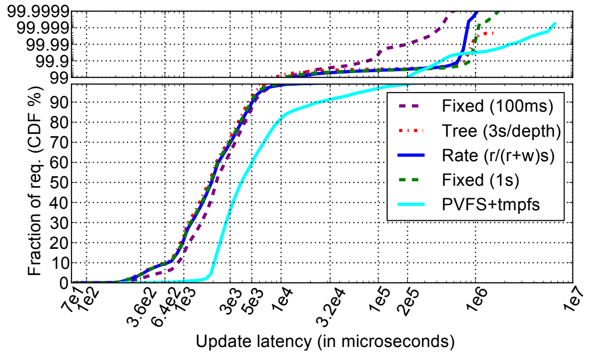
Figure 5 shows the instantaneous creation throughput during the concurrent create workload. IndexFS delivers peak performance after the directory has become large enough to be striped on all servers according to the GIGA+ splitting policy. During the steady state, throughput slowly drops as LevelDB builds a larger metadata store. This is because when there are more entries already existing in LevelDB, performing a negative lookup before each create has to search more SSTables on disk. The variation of the throughput during the steady state is caused by the compaction procedure in LevelDB. Section III-D wil discuss reducing compaction in IndexFS with column-style schema.

IndexFS also demonstrated scalable performance for the concurrent lookup workload, delivering a throughput of more than 1,161,000 file lookups per second for our 128 server configuration. Good lookup performance is expected because the first few lookups fetch the directory partitions from disk into the buffer cache and the disk is not used after that. Deletion throughput for 128 server nodes is about 930,000 operations per second.

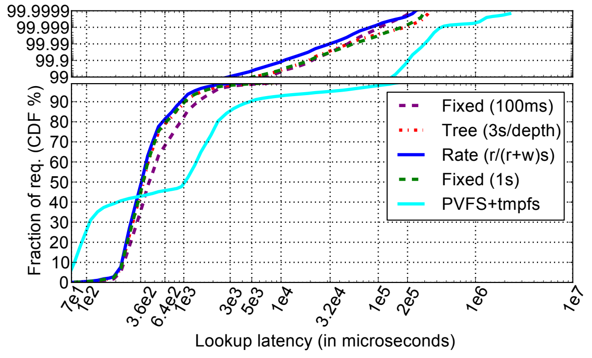
B. Metadata Client Caching

To evaluate IndexFS's client-side metadata caching, we replay a workload trace that records metadata operations issued to the namenode of a LinkedIn HDFS cluster covering an entire 24-hour period. An HDFS trace is used because it is the largest dynamic trace available to us. This LinkedIn HDFS cluster consisted of about 1000 machines, and its namenode during the trace accessed about 1.9 million directories and 11.4 million files. The trace captures 145 million metadata operations of which, 84% are lookup operations (e.g., `open` and `getattr`), 9% are create operations (including `create` and `mkdir`), and the rest (7%) are update operations (e.g., `chmod`, `delete` and `rename`). Because HDFS metadata operations do not use relative addressing, each will do full pathname translation, making this trace pessimistic for IndexFS and other POSIX-like file systems.

Based on this trace, we created a two-phase workload. The first phase is to re-create the file system namespace based on the pathnames referenced in the trace. Since this benchmark focuses on metadata operations, all created files have no data contents. The file system namespace is re-created by multiple clients in parallel in a depth-first order. In the second phase, the first 128 million metadata operations recorded in the original trace are replayed against the tested system. During the second phase, eight client processes are running on each node to replay the trace concurrently. The trace is divided into blocks of subsequent operations, in which each block consists of 200 metadata operations. These trace blocks are assigned to the replay clients in a round-robin, time-ordered fashion. The replay phase is a metadata read intensive workload that stresses load balancing and per-query metadata read performance of the tested systems. IndexFS in this section uses the LevelDB-only metadata schema.



(a) Update Latency



(b) Lookup Latency

Fig. 7: *Latency distribution of update operations (a) and lookup operations (b) under different caching policies* ($6.4e2$ *means* 6.4×10^2). *Rate-based policy offers the best average and* 99% *latency which yields higher aggregate throughput.*

Figure 6 shows the aggregated throughput of the tested system averaged over the replay phase at different cluster scales ranging from 8 servers to 128 servers. In this experiment, we compare IndexFS with three client cache policies for the duration of directory entry leases: fixed duration (100 milliseconds and 1 second), tree-based duration ($3/depth$ seconds), and rate-based hybrid duration ($\frac{r}{w+r}$ seconds). The duration of all rate-based leases and most tree-based lease is shorter than 1 second. We also compare against IndexFS without directory entry caching and PVFS on tmpfs.

From Figure 6, we can see that IndexFS performance does not scale without client-based directory entry caching because performance is bottlenecked by servers that hold hot directory entries. Equipped with client caches of directory entries, all tested systems scale better, and IndexFS with rate-based caching achieves the highest aggregate throughput of more than 514,000 operations per second; that is, about 4,016 operations per second per server.

The reason that the aggregate throughput of rate based caching is higher than the other policy is because it provides more accurate predictions for the lease duration. Since this workload is metadata read intensive, longer average lease duration can effectively reduce the number of unnecessary lookup RPCs between client and servers. So fixed duration caching with 1 second leases has higher average throughput than 100 millisecond leases. When increasing fixed duration lease to be 2 seconds and 4 seconds (not shown in the figure), the average throughput actually decreases because the latency delay of mutation operations now becomes more significant. On the contrary, the rate-based caching provides similiar average latency as 1 second fixed duration lease but has better control over the tail latency of mutation operations.

Figure 7 plots the latency distribution of lookup operations (e.g., `getattr`), and update operations (e.g., `chmod`) in the 128-node test. We can see that the rate-based case has the lowest median latencies and better 99th percentile latencies than all other policies. Its maximum write latency is higher than that of a fixed 100ms duration policy, because the rate based policy poorly predicts write frequencies of a few directory entries. PVFS has better 40th percentile lookup latency versus IndexFS because PVFS clients cache file attributes but IndexFS clients do not; they cache name and permissions only. For `getattr` operation, IndexFS clients need at least one RPC, while the PVFS client may directly find all attributes in its local cache.

C. Portability to Multiple File Systems

To demonstrate the portability of IndexFS, we run the *mdtest* benchmark and *checkpoint* benchmarks [41] when layering IndexFS on top of three cluster file systems including HDFS, Lustre and PanFS. The experiment on HDFS is conducted on the Kodiak cluster with 128 nodes, the experiment on PanFS is conducted on the smaller Susitna cluster with 5 nodes, and the experiment on Lustre is on a third cluster at Los Alamos National Laboratory (Smog). The three clusters have different configurations, so a comparison between systems is not valid. The setup of the *mdtest* benchmark is similar to the one described in Section III-A, and IndexFS uses the fixed 100ms duration metadata caching with LevelDB-only metadata schema and no client writeback caching.

Figure 8 shows the average per-server and aggregated throughput during the *mdtest* benchmarks when layering IndexFS on top of each of the three file systems, and is compared against the original underlying file systems. HDFS and Lustre only support one metadata server. PanFS supports a static partition of the namespace (each subtree at the root directory is a partition called a volume) over multiple metadata servers. Thus we compare IndexFS to native PanFS by creating 1

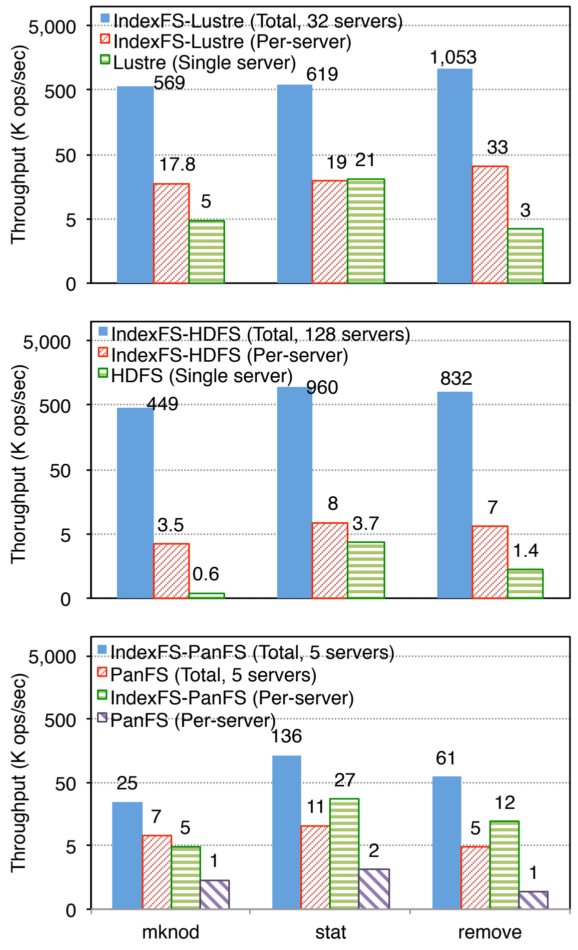


Fig. 8: *Per-server and aggregated throughput during mdtest with IndexFS layered on top of Lustre (on Smog), HDFS (on Kodiak), and PanFS (on Susitna) on a log scale. HDFS and Lustre have only one metadata server.*

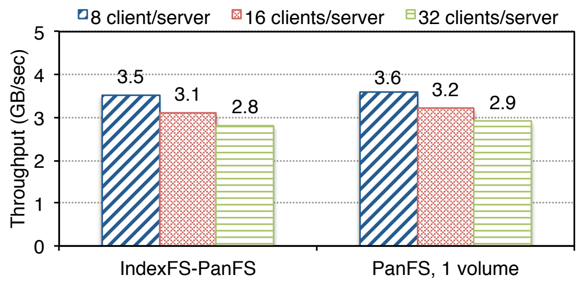


Fig. 9: *The aggregate write throughput for the N-N checkpointing workload. Each machine generates 640 GB of data.*

million files in 5 different directories (volumes) owned by 5 independent metadata servers.

For all three configurations and all metadata operations, IndexFS has made substantial performance improvement over the underlying distributed file systems by reusing their scalable client accessible data paths for LSM storage of metadata. The lookup throughput of IndexFS on top of PanFS is extremely fast because IndexFS packs metadata into file objects stored

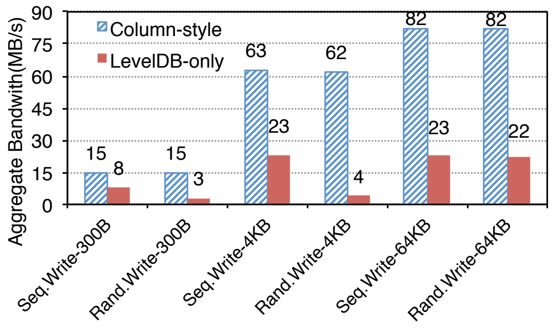


Fig. 10: *Average benchmark bandwidth when inserting 3 million entries with different sizes into the column-style storage schema and the LevelDB-only on a single Kodiak server.*

	random read after sequential write	**random read after random write**
Column-style	350 op/s	139 op/s
LevelDB-only	219 op/s	136 op/s
	read hot after sequential write	**read hot after random write**
Column-style	154K op/s	8K op/s
LevelDB-only	142K op/s	80K op/s

TABLE III: *Average throughput when reading 5 million 320B entries from the column-style schema and original LevelDB-only on a single Kodiak server.*

in PanFS, and PanFS has more aggressive data caching than HDFS. Compared to native Lustre, IndexFS's use of LSM tree improves file creation and deletion. However, for *stat*, it achieves only similar per-server performance because Lustre's clients also cache attributes of files created in the first phase.

We use Los Alamos National Lab's filesystem checkpoint benchmark [41] on Susitna and PanFS storage to test the overhead of our middleware approach on the data-path bandwidth for large file reads and writes. In the checkpoint benchmark, N processes independently write a single checkpoint file each in the same directory; this is called "N-N checkpointing". All processes are synchronized using a barrier before and after writing the checkpoint file. In this test, we also vary the number of client processes per node from 8 to 32 clients. Each client process will generate a total of $640GB/\#clients$ amount of checkpoint data to the underlying file system. The size of the per-call data buffer is set to be 16KB. For IndexFS, the checkpoint files generated in the test will first store 64KB in the metadata table, and then migrate this 64KB and the rest of the file to the underlying distributed file system. Figure 9 shows the average throughput during the write phase in the N-N checkpoint workload. IndexFS's write throughput is comparable to the native PanFS, with an overhead of at most 3%. Reading these checkpoint files through IndexFS has a similar small performance overhead.

D. Metadata Storage Backend

In this section, we demonstrate the trade-offs between the two metadata storage formats used in IndexFS: the two table column-style storage schema and the one table LevelDB only schema. We use a two-phase key-value workload that inserts and reads 3 million entries containing 20-byte keys and variable length values. The first phase of the workload inserts 3 million entries into an empty table in either a sequential or random key order. The second phase of the workload reads all the entries or only the first 1% of entries in a uniformly random order. This micro-benchmark was run on a single metadata server node on Kodiak. To ensure that we are testing out-of-RAM performance, we limit the machine's available memory to 300MB so the entire data set does not fit in memory.

Insertion Throughput: The column-style schema sustains an average insert rate of 56,000 320-byte key-value pairs per second for sequential insertion order, and 52,000 pairs per second for random insertion order. Figure 10 shows the insertion bandwidth for different value sizes (disks are fully saturated in all cases). Column-style is about two to four times faster than LevelDB-only in all cases. Its insertion performance is insensitive to the key order because most of its work is to append key-value pairs into the data file. By only merge-sorting the much smaller index, column-style incurs fewer compactions than the LevelDB-only format, significantly reducing hidden disk traffic.

Read Throughput: Table III shows the average read throughput in the second phase of the workload (with 320-byte key-value pairs). The column-style schema is about 60% faster than LevelDB-only for random read after sequential writes, but the former is about 10 times slower in the *read hot after random write* case. This is because the read pattern does not match the write pattern in the data files, and unlike LevelDB-only schema, column-style does not sort entries stored in data files. In this workload, LevelDB-only caches key-value pairs more effectively than column-style. Therefore, column-style is suitable for write critical workloads that are not read intensive or have read patterns that match the write patterns. For example, distributed checkpointing, snapshot and backup workloads are all suitable for column-sytle storage schema.

E. Bulk Insertion and Factor Analysis

This experiment investigates four optimizations contributing to the bulk insertion performance. We break down the performance difference between the base server-side execution and the client-side bulk insertion, using the following configurations:

- **IndexFS** is the base server-executed operation with synchronous write-ahead logging in the server;
- **+async** enables asynchronous write-ahead logging (4KB buffer) in the server, increasing the number of recent operation vulnerable to server failure; this is almost the configuration used in the experiments of Section III parts A through C, which flushes the write-ahead log every 5 seconds or 16KB.
- **+bulk** enables client-side bulk insertion to avoid RPC overhead with asynchronous client side write ahead logging;
- **+column-style** enables column-style storage schema in client-side when the client builds SSTables;

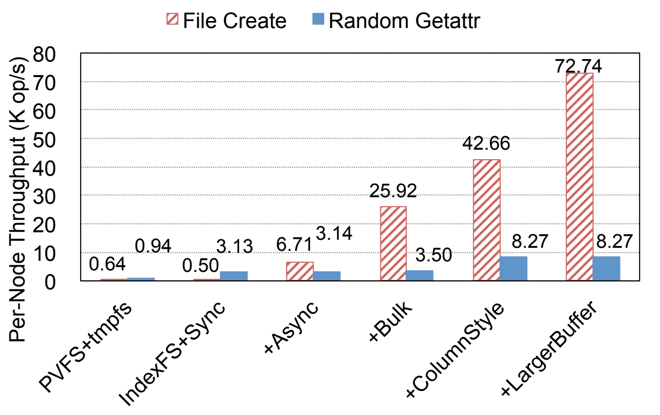


Fig. 11: *Contribution of optimizations to bulk insertion performance on top of PVFS. Optimizations are cumulative.*

- **+larger buffer** uses a larger buffer (64KB) for write-ahead logging, increasing the number of recent operations vulnerable to server failures.

All experiments are run with 8 machines in the Kodiak cluster, each hosting 16 client processes and 1 IndexFS server process, a load high enough to benefit from group commits. The workload we use is the *mdtest* benchmark used in Section III-A. We compare the performance of native PVFS (using tmpfs) with IndexFS layered on top of PVFS (using Ext3 on a disk). Figure 11 shows the performance results. In general, combining all optimizations improves file creation performance by 113x compared to original PVFS mounted on tmpfs. Asynchronous write-ahead logging can bring 13x improvement to file creation by buffering 4KB of updates before writing. Bulk insertion avoids overheads incurred by per-operation RPC to the server and compactions in the server. This brings another 3x improvement. Using a column-style storage schema in the client helps with both file creation and lookup performance since the memory index caches well. The improvement to file creation speed provided by enlarging the write-head log buffer increases sub-linearly because it does not reduce the disk traffic caused by building and writing SSTables.

IV. Related Work

This paper proposes a layered cluster file system to optimize metadata service and to distribute both namespace and large directories. In this section, we discuss prior work related to metadata services in modern cluster file systems and optimized techniques for high-performance metadata.

Namespace Distribution PanFS [62] uses a coarse-grained namespace distribution by assigning a subtree (called a volume) to each metadata server (called a director blade). PVFS [46] is more fine-grained: it spreads different directories, even those in the same sub-tree, on different metadata servers. Ceph [60] dynamically distributes collections of directories based on server load. The distributed directory service [20] in Farsite [6] uses tree-structured file identifiers for each file. It partitions the metadata based on the prefix of file identifiers, which simplifies the implementation of rename operations. Giraffa [5] builds its metadata service on top of a distributed key value store, HBase [24]. It uses full pathnames as the key for file metadata by default, and relies on HBase to achieve load balancing on directory entries, which suffers the hot directory entries problem IndexFS fixes. Lustre [34] mostly uses one special machine for all metadata, and is developing a distributed metadata implementation. IBM GPFS [50] is a symmetric client-as-server file system which distributes mutation of metadata on shared network storage provided the workload on each client does not generally share the same directories.

Metadata Caching For many previous distributed file systems, including PanFS, Lustre, GPFS, and Ceph, clients employ a name space cache and attribute cache for *lookup* and *stat* operations to speed up path traversal. Most distributed file systems use cache coherent protocols in which parallel jobs in large systems suffer cache invalidation storms, causing PanFS and Lustre to disable caching dynamically. PVFS, like IndexFS, uses fixed-duration timeout (100 ms) on all cached entries, but PVFS metadata servers do not block mutation of a leased cache entry. Lustre offers two modes of metadata caching depending on different metadata access patterns [51]. One is a writeback metadata caching that allows clients to access a subtree locally via a journal on the client's disk. This mode is similar to bulk insertion used in IndexFS, but IndexFS clients replicate the metadata in the underlying distributed file system instead of the client's local disk enabling failover to a remote metadata server. Another mode offered by Lustre and PanFS is to execute all metadata operations on the server side without any client cache during highly concurrent accesses. Farsite [20] employs field-level leases and a mechanism called a disjunctive lease to reduce false sharing of metadata across clients and mitigate metadata hotspots. This mechanism is complementary to our approach. However, it maintains more state about the owner of the lease at the server in order to later invalidate the lease.

Large Directories Support A few cluster file systems have added support for distributing large directories, but most spread out the large namespace without partitioning any directory. A beta release of OrangeFS, a commercially supported PVFS distribution, uses a simplified version of GIGA+ to distribute large directories on several metadata servers [38]. Ceph uses an adaptive partitioning technique for distributing its metadata and directories on multiple metadata servers [60]. IBM GPFS uses extensible hashing to distribute directories on different disks on a shared disk subsystem and allows any client to lock blocks of disk storage [50]. Shared directory inserts by multiple clients are very slow in GPFS because of lock contention, and it only delivers high read-only directory read performance when directory blocks are cached on all readers [44].

Metadata On-Disk Layout A novel aspect of this paper is the use of log-structured, indexed metadata representation for faster metadata performance. Several recent efforts have focused on improving external indexing data-structures, such as bLSM trees [52], stratified B-trees [58], fractal trees [8], and VT-trees [53]. bLSM trees schedule compaction to bound the variance of latencies on insertion operations. VT-trees [53] exploit the sequentiality in the workload by adding another indirection to avoid merge sorting all aged SSTables during compaction. Stratified B-trees provides a compact on-disk representation to support snapshots. TokuFS [21], similar to TableFS [47], stores both file system metadata and data blocks into a fractal tree which utilizes additional on-disk indices called the fractal cascading index. The improvements from bLSM and TokuFS are orthogonal to metadata layout used in IndexFS, and could be integrated into our system.

Small Files Access Optimization It has long been recognized that small files can be packed into the block pointer space in inodes [39]. C-FFS [23] takes packing further and clusters small files, inodes and their parent directory's entries in the same disk readahead unit, the track. A variation on clustering for efficient prefetching is replication of inode fields in directory entries, as is done in NTFS[18]. Previous work [12] proposed several techniques to improve small-file access in PVFS. For example, stuffing file content within an inode, coalescing metadata commits and prefetching small file data during `stat` speed up for small file workloads. These techniques have been adopted in our implementation of IndexFS. Facebook's Haystack [7] uses a log-structured approach and holds the entire metadata index in memory to serve workloads with bounded tail latency.

Bulk Loading Optimization Considerable work has been done to add bulk loading capability to new shared nothing key value databases. PNUTS [54] has bulk insertion of range-partitioned tables. It attempts to optimize data movement between machines and reduce transfer time by adding a planning phase to gather statistics and automatically tune the system for future incoming workloads. The distributed key-value database Voldemort [55] like IndexFS, partitions bulk-loaded data into index files and data files. However, it utilizes offline MapReduce jobs to construct the indices before bulk loading. Other databases such as HBase [24] use a similar approach to bulk load data. The paper on benchmark suites YCSB++ [45] reports that if range partitioning is not known as a priori, some databases may incur expensive re-balancing and merging overhead after bulk insertion is logically complete.

V. Conclusion

Many cluster file systems lack a general-purpose scalable metadata service that distributes both namespace and directories. This paper presents an approach that allows *existing* file systems to deliver scalable and parallel metadata performance. The key idea is to re-use a cluster file system's scalable data path to provide concurrent fast access on the metadata path. Our experimental prototype, IndexFS, has demonstrated a fifty percent to two orders of magnitude improvement in the metadata performance over several cluster file systems including PVFS, HDFS, Lustre, and Panasas's PanFS.

This paper makes three contributions. First, it demonstrates an efficient combination of scale-out indexing technique with a scale-up metadata representation to enhance the scalability and performance of metadata service. Second, it proposes the use of client caching with minimal server state to enhance load balancing and insertion performance for creation intensive workloads. Finally, IndexFS uses a portable design that works with existing file system deployment without any configuration changes (but possibly with different fault tolerance assumptions) to the file system or the systems software on compute nodes.

Acknowledgment

We thank Chuck Cranor, Mitch Franzos, Aaron Caldwell, Alfred Torrez and Sage Weil for helping us run and understand experiments on PanFS, PVFS, HDFS, Lustre and Ceph. We thank Brett Kettering, Andy Pavlo and Wolfgang Richter for helpful discussions and comments on this paper. We especially thank Los Alamos National Laboratory for running our software on one of their HPC clusters (Smog), Panasas for providing a storage cluster and LinkedIn for giving us a trace of its HDFS metadata workload. 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 (IRHPIT), 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 also thank the member companies of the PDL Consortium (Actifio, APC, EMC, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, Intel, NEC, NetApp, Oracle, Panasas, Samsung, Seagate and Western Digital)

References

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

[2] FUSE. http://fuse.sourceforge.net/.

[3] mdtest: HPC benchmark for metadata performance. http://sourceforge.net/projects/mdtest/.

[4] Wikipedia: Exponential Moving Weighted Average. http://en.wikipedia.org/wiki/Moving_average.

[5] Giraffa: A distributed highly available file system. https://code.google.com/a/apache-extras.org/p/giraffa/, 2013.

[6] 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.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th symposium on operating systems design and implementation (OSDI)*, 2010.

[8] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the 19th annual ACM symposium on parallel algorithms and architectures (SPAA)*, 2007.

[9] 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.

[10] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7*, 1970.

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

[12] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2009.

[13] 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.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: a distributed storage system for structured data. In *Proceedings of the 7th symposium on operating systems design and implementation (OSDI)*, 2006.

[15] D. Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 1979.

[16] 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.

[17] P. Corbett and et al. Overview of the mpi-io parallel i/o interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[18] H. Custer. Inside the windows NT file system. *Microsoft Press*, 1994.

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

[20] 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.

[21] J. Esmet, M. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. *Proceedings of the 4th USENIX conference on Hot topics in storage and file systems (HotStorage)*, 2012.

[22] S. Faibish, J. Bent, J. Zhang, A. Torres, B. Kettering, G. Grider, and D. Bonnie. Improving small file performance with PLFS containers. Technical Report LA-UR-14-26385, Los Alamos National Laboratory, 2014.

[23] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX annual technical conference (ATC)*, 1997.

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

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

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

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference (ATC)*, volume 8, page 9, 2010.

[28] S. N. Jones, C. R. Strong, A. Parker-Wood, A. Holloway, and D. D. Long. Easing the burdens of HPC file management. In *Proceedings of the 6th workshop on parallel data storage (PDSW)*, 2011.

[29] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[30] A. Leung, I. Adams, and E. L. Miller. Magellan: A searchable metadata architecture for large-scale file systems. Technical Report UCSC-SSRC-09-07, University of California, Santa Cruz, 2009.

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

[32] B. Loewe, L. Ward, J. Nunez, J. Bent, E. Salmon, and G. Grider. High end computing revitalization task force. In *Inter agency working group (HECIWG) file systems and I/O research guidance workshop*, 2006.

[33] C. Lueninghoener, D. Grunau, T. Harrington, K. Kelly, and Q. Snead. Bringing up Cielo: experiences with a Cray XE6 system. In *Proceedings of the 25th international conference on Large Installation System Administration (LISA)*, 2011.

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

[35] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new EXT4 filesystem: current status and future plans. In *Ottawa Linux symposium*, 2007.

[36] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[37] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[38] M. Moore, D. Bonnie, B. Ligon, M. Marshall, W. Ligon, N. Mills, E. Quarles, S. Sampson, S. Yang, and B. Wilson. OrangeFS: Advancing PVFS. *FAST Poster Session*, 2011.

[39] S. J. Mullender and A. S. Tanenbaum. Immediate files. *SoftwarePractice and Experience*, 1984.

[40] H. Newman. HPCS Mission Partner File I/O Scenarios, Revision 3. http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf, 2008.

[41] J. Nunez and J. Bent. LANL MPI-IO Test. http://institutes.lanl.gov/data/software/, 2008.

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

[43] S. Patil and G. Gibson. GIGA+: scalable directories for shared file systems. In *Proceedings of the 2nd workshop on parallel data storage (PDSW)*, 2007.

[44] 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 stroage technologies (FAST)*, 2011.

[45] 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*, 2011.

[46] PVFS2. Parallel Virtual File System, Version 2. http://www.pvfs2.org.

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

[48] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads. *Proceedings of very large data bases (VLDB)*, 2013.

[49] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM symposium on operating systems principles (SOSP)*, 1991.

[50] 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.

[51] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, 2003.

[52] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proceedings of the ACM SIGMOD international conference on management of data*, 2012.

[53] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-Trees. In *Proccedings of the 11th conference on file and storage technologies (FAST)*, 2013.

[54] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the 2008 ACM SIGMOD international conference on management of data*, 2008.

[55] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project Voldemort. In *Proceedings of the 10th USENIX conference on file and storage technologies (FAST)*, 2012.

[56] A. Sweeney. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, 1996.

[57] A. Torres and D. Bonnie. Small file aggregation with PLFS. http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-22024, 2013.

[58] A. Twigg, A. Byde, G. Milos, T. Moreton, J. Wilkes, and T. Wilkie. Stratified B-trees and versioning dictionaries. *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HotStorage)*, 2011.

[59] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur, and R. Ross. Discretionary caching for I/O on clusters. In *Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2003.

[60] 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.

[61] 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.

[62] 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.