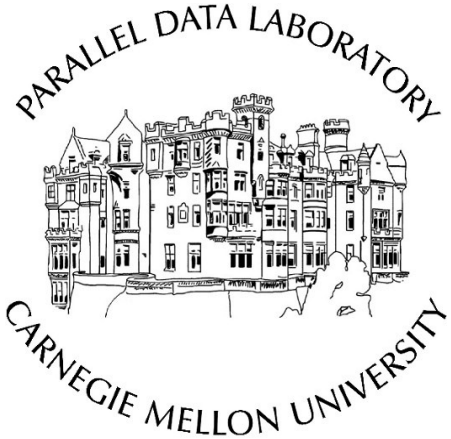


Scaling File System Metadata Performance With Stateless Caching and Bulk Insertion

Kai Ren, Qing Zheng, Swapnil Patil and Garth Gibson
(kair, zhengq, svp, garth}@cs.cmu.edu)

CMU-PDL-14-103
May 2014

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The growing size of modern storage systems is expected to achieve and exceed billions of objects, making metadata scalability critical to overall performance. Many existing parallel and cluster file systems only focus on providing highly parallel access to file data, but 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 and Hadoop HDFS for scalable high-performance operations on metadata and small files. IndexFS uses a tabular-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 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 successfully scaled IndexFS to 128 metadata servers for various metadata workloads. Experiments demonstrate that our out-of-core metadata throughput out-performs PVFS by 50% *to an order of magnitude.*

Acknowledgements: This research is supported in part by The Gordon and Betty Moore Foundation, the University of Washington eScience Institute, NSF under award, SCI-0430781 and CCF-1019104, Qatar National Research Foundation 09-1116-1-172, DOE/Los Alamos National Laboratory, under contract number DE-AC52-06NA25396/161465-1, by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by gifts from Yahoo!, APC, EMC, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, IBM, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware. We thank the member companies of the PDL Consortium for their interest, insights, feedback, and support.

Keywords: Parallel File System, Metadata, Storm-free Caching, Log-Structured Merge Tree, Bulk Insertion

1 Introduction

Lack of a highly scalable and parallel metadata service is the Achilles' heel for many distributed file systems in both the Data Intensive Scalable Computing (DISC) world [22] and the High Performance Computing (HPC) world [32, 27]. 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 namespace (e.g. Hadoop Federated HDFS [22], PVFS [38], Panasas PanFS [54] and Lustre [28]).

The lack of 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 at the same time [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 [23, 25]. Finally, even in the era of big data, most files in even the largest cluster file systems are small [17, 53], where median file size is often only hundreds of kilobytes. Scalable storage systems should expect the numbers of small files stored to exceed billions, a known challenge for many existing cluster file systems [36].

We envision a scalable metadata service that provides distributed file system namespace support to meet the needs of parallel applications. Our design of the metadata service is led by recent workload studies on the traces from several academic and industry clusters [17, 40, 53]. We find 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 very 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 [40].

This workload analysis leads us to 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 large directory is based on our previous work on GIGA+ [36]. This preserves disk locality of small directories for fast `readdir()` and enhances parallel access to large directories. The second mechanism is to maintain metadata caching at the client side with minimal server state to reduce unnecessary requests to metadata servers during path resolution and permission validation.

Previous file systems (such as Google File System [21] and HDFS) use in-memory metadata services that limit the number of objects supported by the file system. The need for a new representation of directory entries encouraged us to develop a out-of-core metadata representation using a column-based, log-structured approach [14, 34, 41]. All file system metadata (including directories and i-node attributes) and the data for small files are packed into a few log files. Fine-grained indexing is used to speed up lookup and scan operations. This organization facilitates high-speed metadata creates, lookups, and scans, even in a single computer local disk configuration [39].

To demonstrate the feasibility of our approach, we implemented a prototype middleware service called IndexFS that incorporates the above namespace distribution mechanism and on-disk metadata representation. Existing cluster file systems such as PVFS, HDFS, and PanFS can benefit from IndexFS without requiring any modifications to the original system. We evaluated the prototype on a cluster consisting of 128 machines. Our results show promising scalability and performance: IndexFS, layered on top of PVFS, HDFS and PanFS, can scale almost linearly to 128 metadata servers, performs 3000 to 10,000 operations per second per machine, and outperforms PVFS by 50% to an order of magnitude in various metadata intensive workloads.

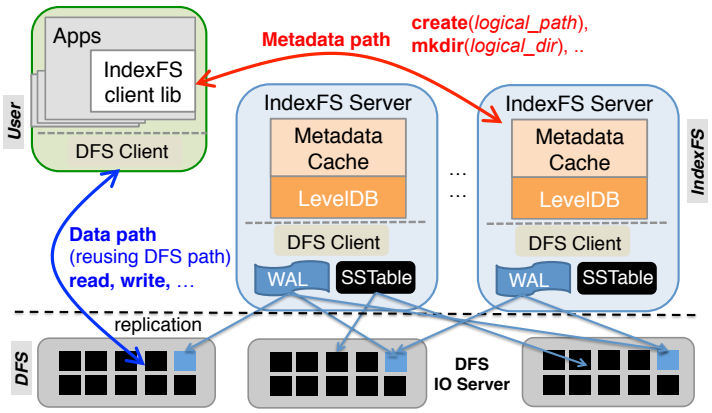


Figure 1: *The IndexFS metadata system is middleware layered on top of an existing cluster file system deployment (such as PVFS) to improve metadata and small file operation efficiency. It reuses the data path of the underlying file system and packs directory entries, file attributes and small file data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

2 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 [16]. The client-side code redirects applications' file operations to the appropriate destination according to the types of operations. All metadata requests (e.g. `create()` and `mkdir()`), and data requests on small files (e.g. `read()` and `write()`), are handled by the metadata indexing module that sends these requests to the appropriate IndexFS server. For all data operations on large size files, the client code redirects read requests directly to the underlying cluster file system to take full advantage of data I/O bandwidth. A newly created, but growing file may be transparently reopened in the underlying file system by the client module. While 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 on the metadata path. IndexFS clients employ caches to enhance performance for frequently accessed metadata such as directory entry, directory server mapping, and subtree for bulk-insertion. Details about these caches will be discussed in later sections.

IndexFS Server: IndexFS employs a layered architecture derived from TableFS [39] and similar to BigTable [14]. Each server 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 distributed 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 using an algorithm called GIGA+ [36] when their size exceeds a threshold. The module that packs metadata and small file data into large immutable sorted files uses a data structure called a log-structured merge (LSM) tree [34]. 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 LSM Tree's data files and write-ahead logs. More details about fault tolerance techniques used in IndexFS is presented in Section 2.5.

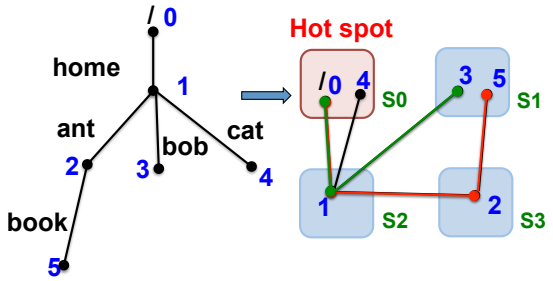


Figure 2: *The figure shows how IndexFS distributes a file system directory tree evenly into four metadata servers. Path traversal makes some directories (e.g. root directory) more frequently accessed than others. Thus state-less directory caching is used to mitigate these hot spots.*

2.1 Dynamic Namespace Partitioning

IndexFS uses a dynamic namespace partitioning policy to distribute both directories and directory entries across all metadata servers. Unlike previous works [18, 52] that partition file system namespace based on a collection of directories that form a sub-tree, 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 file system directories have fewer than 128 entries in many cluster file system instances [53]) since storing directory entries together can preserve locality for scan operations such as `readdir()`. The initial server assignment of a directory can be done through random server selection. To further reduce the variance in the number of directory entries stored in metadata servers, we also adapt the power of two choices load balancing technique [30] to the initial server assignment. The power of two choices 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 very large number of entries, IndexFS uses the GIGA+ binary splitting technique to distribute directory entries over multiple servers [36]. 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 expands to use all metadata servers. This splitting stops after the distributed directory is well balanced on all servers. IndexFS servers and clients maintain 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 [35, 36].

2.2 Stateless Directory Caching

To implement POSIX file I/O semantics many metadata operations are required across each ancestor directory to perform pathname traversal and permission checking. This is expensive in RPC round trips if each check must find the appropriate IndexFS server for the directory entry subset that should contain this pathname component, if it exists. 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 clients stale map entries [36]. By using an inconsistent client cache, servers never need to determine or remember

which clients contain correct or stale mappings, eliminating the storms of cache update or invalidation messages that occur in large scale systems with consistent caches and frequent write sharing [24, 43, 51]. Round trip RPCs to test existence and permissions for each pathname component are often endured in scalable storage systems to avoid invalidation storms, but 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 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 only record a largest lease expiration time with any pathname component in its memory, pinning the entry in its memory and blocking 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. [1]

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 are greatly reduced. This method assumes the clock on all machines are synchronized, which is achieved in modern data centers [11, 15].

We investigated 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 policies use two indicators to adjust the lease duration: one is the depth tree of the directory, 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. 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 in [4]) to estimate the read and write ratio. 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.

2.3 Log-Structured Metadata Storage Format

Our IndexFS metadata storage backend implements a modified version of log-structured merge (LSM) tree [34] (based on an open source library called LevelDB [26]) 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, supporting point queries and range queries. LevelDB, by default, 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 need to be searched is reduced by maintaining the minimum and maximum key value and a Bloom filter[10] on each. However, over time, the cost of finding a LevelDB record 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

[1] A typical optimization to speed up invalidations that are not part of a storm would be to record up to N client IDs with each lease and send explicit invalidations provided the number of leases does not exceed N.

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

Table 1: *The schema of keys and values used by IndexFS.*

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 in TableFS [39], IndexFS embeds i-node attributes and small files with directory entries and stores them into one LSM tree with an entry for each file and directory. The design of using an LSM tree to implement local file system operations is covered in TableFS [39], 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 i-node 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 1. The value of an entry contains the file's full path name and i-node attributes, such as i-node number, ownership, access mode, file size, timestamps (*struct stat* in POSIX). For smaller files whose size is less than T (defaulting to 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 to compaction processes since embedding increases the data volume processed by each compaction.

Column-Style Table for Faster Insertion: Some applications such as check-pointing prefer fast insertion performance or fast pathname lookup rather than fast directory list performance. To adapt for such applications, IndexFS supports a second table schema called *column-style* that speeds up the throughput of insertion, modification, and single entry lookup.

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 corresponding record in the full 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 also satisfy `lookup()` and `readdir()` accesses, the most important non-mutation metadata accesses, without reference to the full table. With these read-only accesses satisfied in the smaller, more cacheable table, IndexFS eliminates the compaction overhead in the full table by treating it as a set of logs and rarely, if ever, compacting the full table. Eliminating compaction speeds up insertion intensive workloads significantly (see section 3.3). Moreover, because the index table contains a pointer (log ID and offset in this immutable 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 unreferenced rows in the full table and resorting by primary key (if needed at all) can be done by a background defragmentation service. This service would write new logs and add replacement entries into the index table before deleting the original logs.

Partition Splitting and Migration: IndexFS uses a faster and safer technique for splitting a directory partition than is used by GIGA+. The immutability of SSTables in LevelDB makes a fast bulk insert possible – an 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

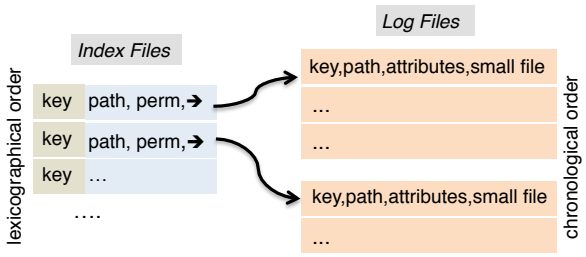


Figure 3: *Column-style stores index and log tables separately. Index tables are kept in LevelDB which contains frequently accessed attributes for file lookups and the pointer to the location of full file metadata in the log file.*

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 column-style storage schema, only index tables need to be extracted and bulk inserted into 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 to reduce the performance impact on concurrent metadata operations.

2.4 Metadata Bulk Insertion

Unfortunately, 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. 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) which is report to achieve a million file creates per second per server [49]. Inspired by the metadata client caching and bulk insertion techniques we used for directory splitting, IndexFS implments write back caching at the client for creation of currently non-existent directory subtrees. This technique may be viewed as an extension of Lustre's directory callbacks [43]. By using bulk insertion, IndexFS matches PLFS's create performance and achieves better lookup 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 file 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, which 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 to support efficient 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 that permissions are enforced, 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 and rejected creates will not be delivered back to the corresponding application code at the `open()` call site and will more likely remain somewhat undetected in error logs. Quota control for the space used by metadata will be similarly impacted, while data writes directly to the underlying file system can still be appropriately growth limited.

IndexFS extends its lease-based cache consistent protocol to provide expected global semantics. IndexFS client wanting to use 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 lease. During lease period, all files (or subdirectories) created inside such directories will be exclusively served and recorded by the client itself with high throughput. 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. When another client is waiting for access to the localzied subtree, the IndexFS server may deny a lease renewal so that the client needs to complete its remaining bulk inserts quickly. If multiple clients want to localize file creates inside the same directory, IndexFS `mkdir()` can take a "`SHARED_LOCALIZE`" flag, and conflicting bulk inserts will be resolved at the servers later. As bulk insertion cannot help data intensive workloads, IndexFS clients automatically "expire" leases once such an IO pattern is detected.

Inside a localized directory, an application is able to perform all metadata operations, not just `mknod()`. `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 and its lease expired.

2.5 Fault Tolerance

IndexFS is designed as a middleware layered on top of an underlying failure-tolerant, distributed and parallel file system. IndexFS's primary fault tolerance strategy, then, is to push states into the underlying file system – large data in files, metadata in SSTables and recent changes in write-ahead logs. IndexFS server processes are all monitored by standby server processes prepared to replace failed server processes. Each IndexFS metadata server maintains a separate write-ahead log that records mutation operations such as file creation and rename. All logged operations are associated with a global logical time stamp. When a server crashes, its write-ahead log can be replayed by a stand-by server to recover consistent state.

Leases for client caching are not made durable. A standby server restarting from logs simply waits for the largest possible timeout interval. The first lease for a localized directory should be logged so a standby server will recognize a client writing back its local changes.

Some metadata operations require a distributed transaction protocol, including directory splitting and rename operations. These are implemented as 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 [18]. Since this problem is beyond the scope of this paper, we instead implemented a simplified version of the `rename()` operation that only supports renaming files and leaf directories.

	Cluster 1	Cluster 2
#Machines	128	5
OS	Ubuntu 12.10	CentOS 6.3
Kernel	3.6.6 x86_64	2.6.32 x86_64
CPU	AMD Opteron 252 Dual Core	AMD Opteron 6272 64 Cores
Memory	8GB	128GB
Network	1GE NIC	40GE NIC
Storage System	Western Digital hard disk 1TB per machine	PanFS 5 Shelves (5 MDS + 50 OSD)

Table 2: *Two clusters used for experiments.*

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 or size threshold is exceeded. The asynchronous mode may lose more data than synchronous mode when a crash happens. However, the asynchronous mode provides much higher ingestion throughput.

3 Experimental Evaluation

Our experimental evaluation answers the following questions: (1) How well does IndexFS scale and load balance? (2) What are the trade-offs of IndexFS's selection of the underlying metadata storage formats? (3) How does bulk insertion improve file creation speed? and (4) How portable is IndexFS to use on other cluster storage systems and configurations?

The prototype of IndexFS is implemented in about 10,000 lines of C++ code using a modular design, which can be easily layered on multiple distributed file systems such as HDFS [22], PVFS [13] and PanFS [54]. 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 two clusters. Table 2 describes the hardware and software configurations of the two clusters. The first cluster is a 128-machine cluster with old hardware. It is used to evaluate IndexFS's scaling performance and design trade-offs. In this cluster, IndexFS is layered on top of PVFS and HDFS, and its performance is compared against PVFS and Ceph [52]. The second cluster is a 5-machine cluster with state-of-the-art hardware, which is also connected to a 5-shelf PanFS storage cluster. This is used to evaluate IndexFS's portability to a proprietary federation parallel file system (PanFS). In all experiments, clients and servers are distributed over the same machines. The client uses a library, 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 [29, 48]).

3.1 Large Directory Scaling

This section shows how IndexFS scales large directories over multiple 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 [52, 36]. We generated a three-phase workload. The first phase is a concurrent create

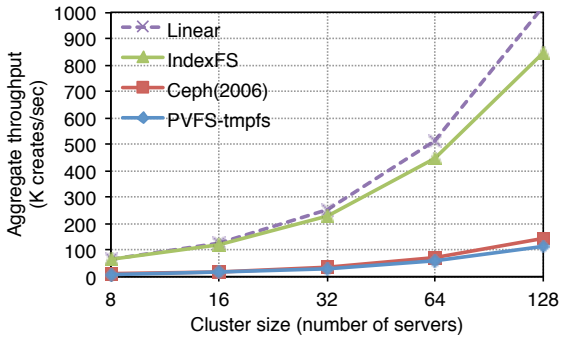


Figure 4: *IndexFS on 128 servers deliver a peak throughput of roughly 842,000 file creates per second. The RPC module (Thrift [1]) limits its linear scalability.*

workload in which eight client processes per server node simultaneously create files in a common directory. The number of files created is proportional to the number of servers: each server 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 (on a log-scale X-axis). IndexFS with SSTables and logs stored in PVFS scales linearly up to 128 servers. IndexFS uses only LevelDB 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. Figure 4 also compares IndexFS with the scalability of Ceph and PVFS. For PVFS, the number is measured in the same cluster. But since PVFS's metadata servers uses a transactional database (Berkeley DB) for durability, which is stronger than IndexFS or Ceph, we let it store its records in a RAM disk to achieve better performance When layering PVFS on top of Ext3 with hard disks, it only achieves one hundred creates per second. For Ceph, [2] Figure 4 reuses numbers from the original paper [52]. The experiment was performed on a cluster with a similar configuration, where the notable difference is their use of dual-core 2.4GHz CPUs. The reason that IndexFS outperforms other file systems is largely due to the use of log structured metadata layout.

Figure 5 shows the instantaneous creation throughput during the concurrent create workload. IndexFS delivers peak performance after the directory has been large enough to use 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.

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 index is not mutating and load is well-distributed among all servers. 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 nodes is about 830,000 operations per second.

[2] The directory splitting function in the latest version of Ceph is not stable. Ceph can only run a single metadata server in our cluster. We are currently working with Ceph developer team to fix the problem before publishing the final paper.

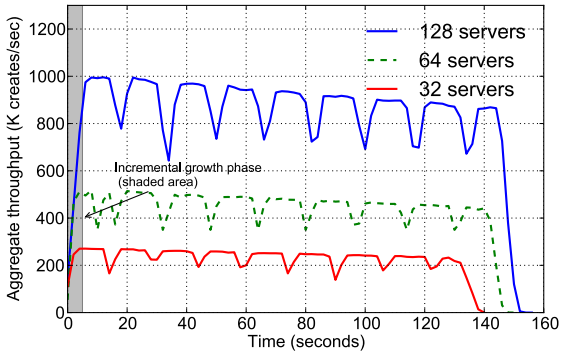


Figure 5: *IndexFS achieves steady throughput after distributing one directory hash range to each available server. After scale-out, throughput variation is caused by the compaction process in LevelDB. Peak throughput degrades over time because the total size of the metadata table is growing so negative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

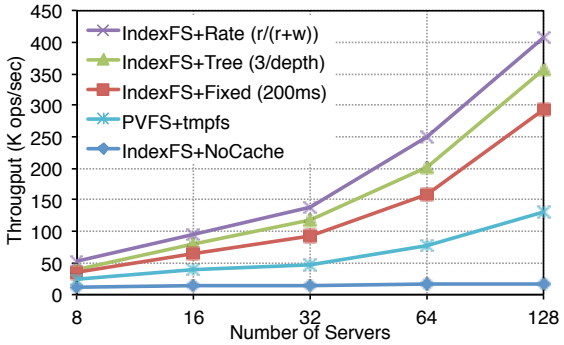
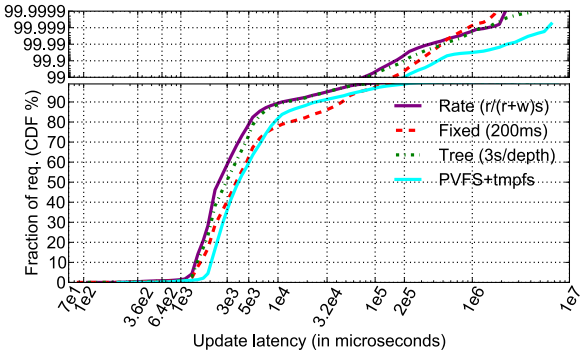


Figure 6: *Average aggregate throughput of replaying 1 Million operations per metadata server on different number of nodes using a one-day trace from LinkedIn HDFS Cluster.*

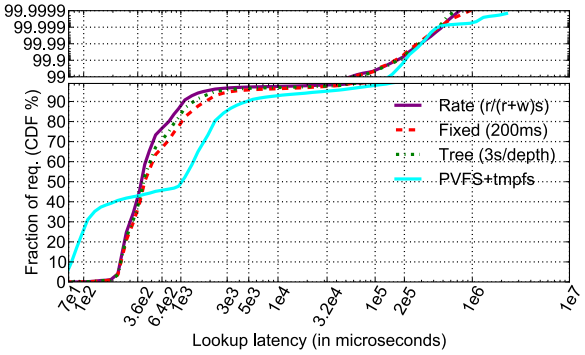
3.2 Metadata Client Caching

To evaluate IndexFS's client-side metadata caching, we replay a log trace that records metadata operations issued to the namenode of a LinkedIn HDFS cluster covering an entire day. An HDFS trace is used because it is the largest dynamic trace available to use. This LinkedIn HDFS Cluster consisted of about 1000 machines, and its namenode log accessed about 1.9 million directories and 11.4 million files, and consists of 145 million metadata operations. Among these metadata operations, 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 log 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 read performance of the tested systems.



(a) Update Latency



(b) Lookup Latency

Figure 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 lease: fixed duration (200 ms), tree-based duration ($3/depth$ sec), and rate-based hybrid duration ($\frac{r}{w+r}$ sec). We also compare them 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 407,600 operations per second; that is, about 3,185 operations per second per server.

The reason that the aggregate throughput of rate based caching is higher than other policy is because it provides more accurate predictions for the lease duration. Since this workload is read intensive, rate based caching provides longer average lease duration than other policy that can effectively reduce the number of unnecessary lookup RPCs between client and servers. To compare different policy's impact on latency, the latency distribution of lookup operations (e.g. `getattr()`), and update operations (e.g. `chmod()`) in the 128-node test is plotted in Figure 7. We can see that the rate-based case has the lowest average and 99 percentile latencies among all policies. However, its maximum write latency is higher than that of a fixed 200ms duration policy. This is because the rate based policy fails to predict write frequencies of a few directory entries. PVFS has better 40 percentile lookup latency versus IndexFS because PVFS clients cache

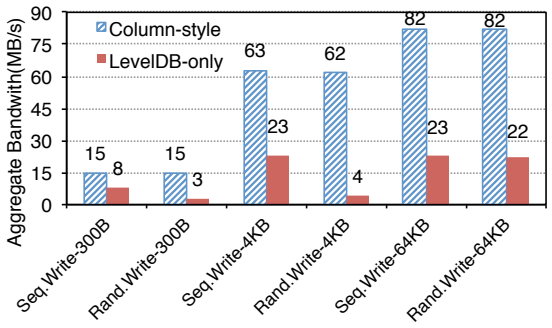


Figure 8: *Average disk throughput when inserting 3 million entries with different sizes into the column-style storage schema and the original LevelDB-only on a single server.*

file attributes but IndexFS clients do not; they cache name and permissions only. For `getattr()` operation, IndexFS client needs at least one RPC, while the PVFS client may directly find all attributes in its local cache.

3.3 Metadata Storage Backend

To demonstrate the trade-offs between the two metadata storage formats used in IndexFS, we have run several key-value workloads to compare column-style with LevelDB only format. By default, we use a two-phase workload that inserts and reads 3 million entries containing 20-byte keys and variable length values. The first phase of the workload is to insert 3 million entries into an empty table in either a sequential or random key order. The second phase of the workload is to read all the entries or only the first 1% of entries in a uniformly random order. This micro-benchmark was run on one single metadata server node. To ensure 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. All tests were run for three times and the coefficient of variation is less than 1%.

Insertion Throughput: column-style can sustain an average insert rate of 56,596 320B key-value pairs per second for sequential insertion order, and 52,192 pairs per second for random insertion order. Figure 8 shows the insertion throughput for different value sizes in terms of aggregate disk bandwidth. Column-style is about 2 to 4 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 index, column-style incurs fewer compactions than the LevelDB-only format, which reduces a lot of disk traffic.

Read Throughput: Table 3 shows the average read throughput in the second phase for the workload with 320B key-value pairs. Column-style is about 60% faster than LevelDB-only for random read after sequential writes, but 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 column-style does not sort entries stored in data files as does LevelDB-only. 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.

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

3.4 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 logging (4KB buffer) in the server, increasing the number of recent operation vulnerable to server failure;
- **+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;
- **+large buffer** uses a larger buffer (64KB) for write-ahead logging, increasing the number of recent operation vulnerable to server failure.

All experiments are run with 64 machines in the first cluster each hosting 2 clients and 1 server process. The workload we use is the *mdtest* benchmark used in Section 3.1. We compare the performance of IndexFS with PVFS, where IndexFS is layered on top of PVFS. To test IndexFS in synchronous mode, 16 clients per server are used, a load high enough to benefit from group committing. And PVFS is mounted on Ext3 under IndexFS, and on tmpfs when we compare to it, to bias against IndexFS. Figure 9 shows the performance results. In general, combining all optimizations improves file creation performance by 113x compared to original PVFS mounted on tmpfs, and improves file lookup performance by 8x. Asynchronous write-ahead logging can bring 13x improvement to file creation by buffering 4KB of updates before writing. Bulk insertion avoids overheads per-operation RPC to the server and compaction in the server, which 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.

3.5 Portability to Multiple File Systems

To demonstrate the portability of IndexFS, we run *mdtest* benchmark and *checkpoint* benchmark when layering IndexFS on top of other distributed file systems such as HDFS [22] and PanFS [54]. The experiment on HDFS is conducted on the cluster with 128 machines, and the experiment one PanFS is conducted on

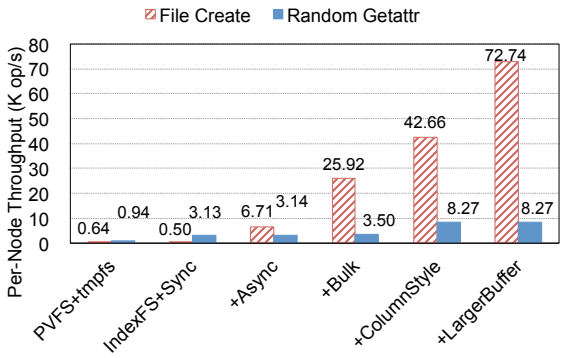


Figure 9: *Contribution of optimizations to bulk insertion performance on top of PVFS. Optimizations are cumulative.*

the smaller (5-node) cluster. The machines of the smaller cluster have 32x more cores and 40x network bandwidth, so a HDFS versus PanFS comparison is not valid.

Figure 10 shows the average per-server throughput during *mdtest* benchmarks when layering IndexFS on top of HDFS and PanFS. The setup of *mdtest* benchmark is similar to the one described in Section 3.1. 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 also compare IndexFS to native PanFS when creating 1 million files in 5 different directories (volumes) owned by 5 independent metadata servers. For all three tested metadata operations, IndexFS has made substantial performance improvement over the underlying distributed file systems by reusing their scalable client accessible data paths. The lookup throughput of IndexFS on top of PanFS is extremely fast because IndexFS packs metadata into file objects stored in PanFS and PanFS has more aggressive data caching than HDFS.

We use Los Alamos National Lab's filesystem checkpoint benchmark [33] to test the bandwidth overhead of our middleware approach upon the data path for large file reads and writes. In the checkpoint benchmark, a set of processes independently write a single checkpoint file each in the same directory. 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 test machine from 8 to 32 clients. Each client process will generate a total of $640GB/\#clients$ 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 metadata server, and then migrate this 64KB and the rest of the file to the underlying distributed file system. Figure 11 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.

4 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 [54] uses a coarse-grained namespace distribution by assigning a subtree (called a volume) to each metadata server (called a direct blade). PVFS [38] is more fine-grained: it spreads different directories, even those in the same sub-tree, on different metadata servers. Ceph [52] dynamically distributes the file system namespace based on server loads on collections of directories. The

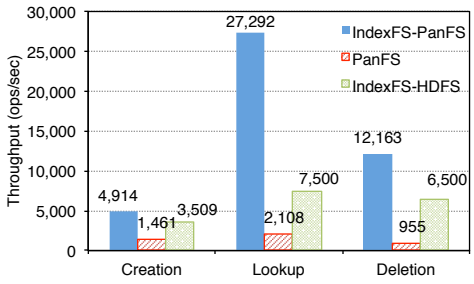


Figure 10: *Per-server throughput during mdtest benchmark with IndexFS layered on top of HDFS and PanFS one each machine (128 for HDFS and 5 for PanFS). Running 8 clients per machine is able to saturate IndexFS and PanFS.*

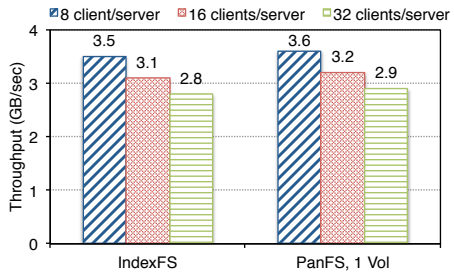


Figure 11: *The aggregate write throughput for the N-N checkpointing workload. Each machine (5) generates 640 GB of data.*

distributed directory service [18] in Farsite [6] uses a 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 [20]. 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 [28] mostly uses one special machine for all metadata, and is developing a distributed metadata implementation. IBM GPFS [42] is a symmetric client-as-server file system which distributes mutation of metadata on shared network storage provided the workload on each client is different and does not share the same directories.

Metadata Caching For most traditional file systems, including PanFS, Lustre, GPFS, and Ceph, clients employ a name space cache and attribute cache for *lookup* and *getattr* operations to speed up path traversal. Most distributed file systems use cache coherent protocols with 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 duration entry. Lustre offers two modes of metadata caching depending on different metadata access patterns [43]. 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 sometimes execute all metadata operations on the server side without any client cache, especially during highly concurrent accesses. Farsite [18] employs the 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 states 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 direc-

tories, but most spread out the directories of a large namespace. A beta release of OrangeFS, a commercially supported PVFS distribution, uses a simplified version of GIGA+ to distribute large directories on several metadata servers [31]. Ceph uses an adaptive partitioning technique for distributing its metadata and directories on multiple metadata servers [52]. 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 [42]. 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 [36].

Metadata On-Disk Layout A novel aspect of this paper is the use of log-structured, indexed single-node metadata representation for faster metadata performance. Several recent efforts have focused on improving external indexing data-structures, such as bLSM trees [44], stratified B-trees [50], fractal trees [8], and VT-trees [45]. bLSM trees schedule compaction to bound the variance of latencies on insertion operations. VT-trees [45] 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 [19], similar to TableFS [39], 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 Previous work [12] proposed several techniques to improve small-file access in PVFS. For example, stuffing file content within inode, coalescing metadata commits and prefetching small file data during $getattr()$ 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 [46] 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 [47] 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 [20] use a similar approach to bulk load data. The paper on benchmark suites YCSB++ [37] discovers that if range partitioning is not known as a priori, some databases may incur expensive re-balancing and merging overhead after bulk insertion.

5 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 access on the metadata path. Our experimental prototype IndexFS has demonstrated a 50% to an order of magnitude improvement in the metadata performance over several cluster file systems including PVFS, HDFS, and Panasas's PanFS.

This paper makes three contributions. The first contribution is an efficient combination of scale-out indexing technique with a scale-up metadata representation to enhance the scalability and performance of metadata service. The second contribution is the novel use of client caching and buffering with minimal sever state to enhance load balancing and insertion performance for creation intensive workloads. The third contribution is the portable design that can use any 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.

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] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. 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] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter 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] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the 19th annual ACM symposium on parallel algorithms and architectures (SPAA)*, 2007.

[9] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan 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.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7*, 1970.

[11] Jack Burbank, David Mills, and William Kasch. Network time protocol version 4: Protocol and algorithms specification. *Network*, 2010.

[12] Philip Carns, Samuel Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2009.

[13] Philip H. Carns, Walter B. Ligon III, Robert B Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert 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] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed Database. In *Proceedings of the 10th USENIX conference on operating systems design and implementation (OSDI)*, 2012.

[16] Peter 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.

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

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

[19] John Esmet, Michael Bender, Martin Farach-Colton, and Bradley C Kuszmaul. The TokuFS streaming file system. *Proceedings of the 4th USENIX conference on Hot topics in storage and file systems (HotStorage)*, 2012.

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

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

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

[23] Stephanie N Jones, Christina R Strong, Aleatha Parker-Wood, Alexandra Holloway, and Darrell DE Long. Easing the burdens of hpc file management. In *Proceedings of the 6th workshop on parallel data storage (PDSW)*, 2011.

[24] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/o performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

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

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

[27] Bill Loewe, Lee Ward, James Nunez, John Bent, Ellen Salmon, and Gary Grider. High End Computing Revitalization Task Force (HECRTF), . In *Inter agency working group (HECIWG) file systems and I/O research guidance workshop*, 2006.

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

[29] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new EXT4 filesystem: current status and future plans. In *Ottawa Linux symposium*, 2007.

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

[31] Micheal Moore, David Bonnie, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles, Sam Sampson, Shuangyang Yang, and Boyd Wilson. OrangeFS: Advancing PVFS. *FAST Poster Session*, 2011.

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

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

[34] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree. *Acta Informatica*, 33(4):351–385, 1996.

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

[36] Swapnil Patil and Garth 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.

[37] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, 2011.

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

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

[40] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads. *Proceedings of very large data bases (VLDB)*, 2013.

[41] Mendel Rosenblum and John 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.

[42] Frank Schmuck and Roger 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.

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

[44] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proceedings of the ACM sigmod international conference on management of data*, 2012.

[45] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with VT-Trees. In *Proccedings of the 11th conference on file and storage technologies (FAST)*, 2013.

[46] Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the 2008 ACM SIGMOD international conference on management of data*, 2008.

[47] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project Voldemort. In *Proceedings of the 10th USENIX conference on file and storage technologies (FAST)*, 2012.

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

[49] Aaron Torres and David Bonnie. Small File Aggregation with PLFS. http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-22024, 2013.

[50] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified b-trees and versioning dictionaries. *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HotStorage)*, 2011.

[51] Murali Vilayannur, Anand Sivasubramaniam, Mahmut Kandemir, Rajeev Thakur, and Robert Ross. Discretionary caching for i/o on clusters. In *Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2003.

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

[53] Brent Welch and Geoffrey 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.

[54] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX conference on file and stroage technologies (FAST)*.