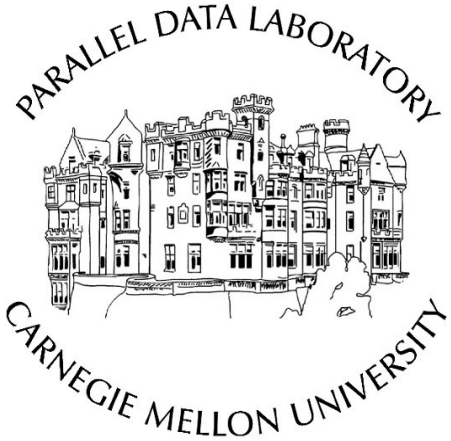


DeltaFS: A Scalable No-Ground-Truth Filesystem For Massively-Parallel Computing

Qing Zheng, Chuck Cranor, Greg Ganger, Garth Gibson, George Amvrosiadis
Brad Settlemyer (LANL†), Gary Grider (LANL†)
†Los Alamos National Laboratory

CMU-PDL-21-101
July 2021

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

Abstract

High-Performance Computing (HPC) is synonymous to massive concurrency. But it can be challenging on a large computing platform for a parallel filesystem's control plane to utilize CPU cores when every process's metadata mutation is globally synchronized and serialized against every other process's mutation. We present DeltaFS, a new paradigm for distributed filesystem metadata. DeltaFS allows jobs to self-commit their namespace changes to logs, avoiding the cost of global synchronization. Followup jobs selectively merge logs produced by previous jobs as needed, a principle we term No Ground Truth which allows more efficient data sharing. By following this principle, DeltaFS leans on the parallelism found when utilizing resources at the nodes where job processes run, improving metadata operation throughput up to 98×*, a number rising as job processes increase. DeltaFS enables efficient inter-job communication, reducing overall workflow runtime by significantly improving client metadata operation latency, and resource usage up to* 52.4×*.*

Acknowledgements: We would like to thank the members and companies of the PDL Consortium for their interest, insights, feedback, and support.

Keywords: Distributed filesystem metadata, massively-parallel computing, data storage

1 Introduction

It is easy to slow down a C program — just define every variable of the program as `_Atomic` and rerun the program. Atomic variables are globally synchronized across all compute cores. Every load on a compute node reads from memory the latest store, except that doing so is not always necessary and can significantly slow down a modern program [31, 36, 1]. In C, variables are not atomic by default. Applications explicitly request it when needed.

Unfortunately, while a parallel C application running on a modern HPC platform is able to request memory atomicity on an as-needed basis, its persistent state — stored as files and accessed through a shared underlying parallel filesystem [55, 67, 56] — remains globally synchronized at all times even on the world's largest computers: every process sees every other process's latest file metadata mutations all the time regardless of whether these processes communicate and despite possibly huge performance penalties. Alas, today's parallel filesystems are old-fashioned and unilaterally fully coordinate metadata for all accesses across all nodes. This results in scalability issues and performance bottlenecks that reduce the effectiveness of HPC parallelism.

We propose DeltaFS, a new way of providing distributed filesystem metadata on modern parallel computing platforms. DeltaFS re-imagines the roles filesystems play in delivering performance and consistency to applications. First, today's filesystem clients tend to synchronize too frequently with their servers for metadata reads and writes. We show deep relaxation of filesystem namespace synchronization and serialization through client logging and subsequent merging of filesystem namespace changes on an as-needed basis. Second, today's filesystems map all application jobs to a single filesystem namespace. Our work enables jobs to self-manage their synchronization scopes to avoid false sharing and to minimize per-job filesystem namespace footprint to improve performance. Finally, modern filesystems achieve scaling primarily by dynamic namespace partitioning over multiple dedicated metadata servers [65, 67, 50, 71]. Filesystem metadata performance is a function of, and is fundamentally limited by, the amount of compute resources that are dedicated to servers. We show that dynamic instantiation of filesystem metadata processing functions on client nodes enables highly agile scaling of filesystem metadata performance beyond a fixed set of dedicated servers.

The core of DeltaFS is a transformation of today's globally synchronized filesystem metadata to per-job metadata log records that can be dynamically merged to form new filesystem namespace views when requested by a followup job. To achieve this, DeltaFS defines an efficient log-structured filesystem metadata format that an application job process can use to log its namespace changes as a result of its execution. DeltaFS does not require all client changes to be merged back to a server-managed global tree for a consistent view of the filesystem. Instead, a job selectively merges logs produced by previous jobs for sequential data sharing. Unrelated application jobs never have to communicate.

With DeltaFS we envision a full reduction of today's parallel filesystems to scalable object stores. On top of these stores, applications independently instantiate services for per-job filesystem namespace management. There is no longer a global namespace. Instead, applications communicate only when they need to and communication is done primarily through sharing and publishing immutable log records stored in a shared underlying object store for minimum synchronization. Meanwhile, there no longer needs to be any dedicated metadata servers. With DeltaFS, jobs dynamically utilize their compute nodes for metadata processing enabling them to overcome limitations and bottlenecks seen in today's parallel filesystem metadata designs. We call this new way of managing distributed filesystem metadata *No Ground Truth*, as it requires no global synchronization. Unrelated jobs are no longer forced to see each other's files and pay for each other's metadata updates. Despite not having a global filesystem namespace, jobs using DeltaFS can still perform inter-job communication through it. In fact, decoupling and parallelizing metadata accesses in DeltaFS vastly reduces a metadata-intensive workflow's inter-job communication latency compared to today's filesystems.

DeltaFS is designed for massively-parallel computing jobs [5] and scales to exascale computing platforms. Our work builds upon DeltaFS [76], but intensively extends it to include a more complete design, implementation, and comparison. Our experiments show that DeltaFS improves metadata performance by using the parallelism that can be found when utilizing resources at the nodes where job processes run. This parallelism is unlocked due to DeltaFS's no-ground-truth property that allows jobs to selectively merge logs produced by previous jobs *as needed*. We show up to 98× faster metadata operation throughput compared to the current state-of-the-art, a number that rises as job processes increase. DeltaFS further enables efficient inter-job communication, vastly reducing overall workflow runtime by significantly reducing the latency of client filesystem operations and the CPU time clients are blocked on such operations by up to 52.4×.

The rest of this paper is structured as follows. Section 2 shows the motivation and rationale behind our work. Section 3-7 detail our design. We report experiment results in Section 8, related work in Section 9, and conclude in Section 10.

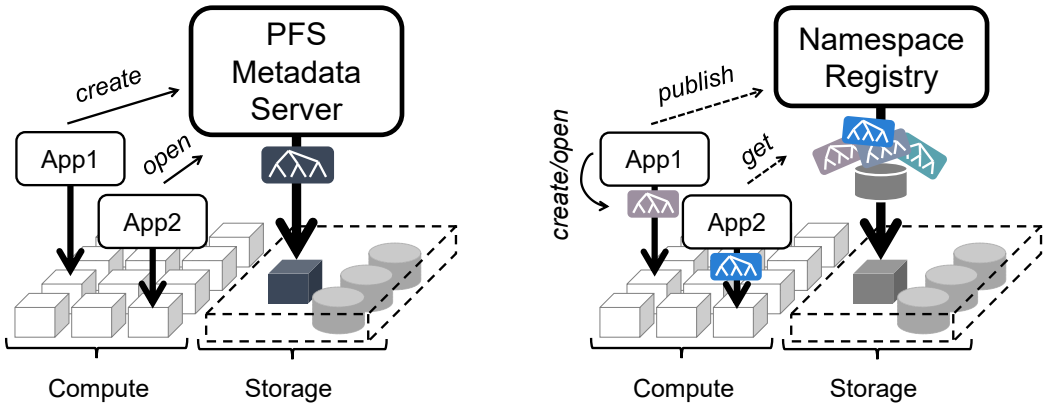
2 Motivation

Three factors motivate our work: (1) the high cost of global synchronization for strong consistency in today's massively-parallel computing environments, (2) the inadequacy of the current state-of-the-art for scalable parallel metadata performance, and (3) the promise of a relaxed "no ground truth" parallel filesystem for today's non-interactive parallel computing workloads.

Global Synchronization Filesystems are the main way applications interact with persistent data. While a local filesystem manages the files of a single node, distributed parallel filesystems such as Lustre [56], GPFS [55], PVFS [14], and the Panasas filesystem [67] manage the files of today's largest supercomputers [63]. By stripping data across a large pool of object storage devices, parallel filesystems have long enabled fast concurrent access to file data [16, 21]. However, in terms of metadata management, a strategy not too different from early network filesystems is used: all client metadata mutations are synchronously processed; they are first sent to a server, checked and serialized by it, and then appended to the server's write-ahead log for eventual merging into the filesystem's on-disk metadata representation managed by the server [42, 26, 54].

While not necessarily the best way to handle file metadata on a large supercomputer, an important reason modern parallel filesystems continue to use this old strategy is that it enables distributed application processes to communicate as if they were on a local machine thanks to constant global synchronization [51]. Unfortunately, early network filesystems were not developed with today's massively-parallel computing environments in mind. While the early CM-5 computer at LANL — the fastest machine of its time — had only 1024 CPU cores, the fastest computer today has as many as 7 million CPU cores [25, 41]. As the best way to utilize a modern supercomputer is to keep all of its compute cores busy, global synchronization in modern parallel filesystems has become a growing source of performance bottlenecks in today's leading computing systems, increasingly nullifying the very parallelism that these systems enable in the first place. This needs to be changed.

Inadequacy of the Current State-of-the-Art To attain high metadata performance, modern scalable parallel metadata services use dynamic namespace partitioning over multiple metadata servers [65, 67, 50, 71]. In these filesystems, each metadata server manages a partition of the filesystem's namespace. The overall metadata performance is a function of the number and compute power of these servers. However, dynamic namespace partitioning does not remove global synchronization; it partitions it. Meanwhile, even these scalable filesystems can require a significant number of dedicated metadata servers to achieve high performance. Worse, as not all applications use the filesystem the same way, the amount of compute resources devoted



(a) Today's Parallel Filesystems

(b) Our Vision

Figure 1: Motivation for no ground truth. Rather than synchronously communicating with a slow global filesystem namespace, jobs communicate instead by publishing and sharing filesystem namespace snapshots on an as-needed basis through a public registry for high performance.

to the filesystem can be difficult to determine beforehand. This leads to performance bottlenecks when the demand of the application is too high compared with the estimated amount, and a waste of resources when otherwise.

In addition to dynamic namespace partitioning, another way to improve performance is to employ an efficient log-structured metadata format that hides the processing delay associated with today's parallel filesystem metadata servers [50, 39]. Such a system is able to quickly absorb a large amount of client changes without immediately optimizing them for fast reads; a separate set of server threads do so asynchronously in the background so that the client need not be blocked. However, in cases in which the background process cannot keep up with the foreground insertion, the time required for these background operations will have to be amortized immediately. When the server compute resources are insufficient for the said workload, a client still experiences delays.

Recent work has also showed ultra fast file creation speed through deep client logging [8, 50, 74]. Yet client logging alone does not address read performance and does not support inter-job communication.

From One-Size-Fits-All to No Ground Truth As we keep increasing the capacity and parallelism of our computers, an emerging reality we need to confront is that we are fast approaching a point in time when there will be no one-size-fits-all parallel metadata systems [60, 2]. While the high cost of global synchronization will continue to be necessary in cases where the applications use the filesystem to communicate, it is also important to realize that many of today's parallel applications are no longer a group of laboratory scientists sharing their text files. Instead, they are for the most part non-interactive batch jobs that do not necessarily benefit from many of the semantic obligations that early network filesystems carried in their computing environments [42, 26, 54].

A modern HPC application is first submitted to a job scheduler queue [63]. When scheduled, it reads from the parallel filesystem, writes to the parallel filesystem, and then ends. Looking at the job, the input it reads is likely ready and static at the time the job is submitted. The output it generates is probably not examined, except possibly by the job owner trying to figure out how the job is progressing, until after the job is done [5, 38, 8]. This is effectively sequential data sharing. We argue that a parallel filesystem could serve this through simple publication and sharing of filesystem namespace snapshots without requiring any

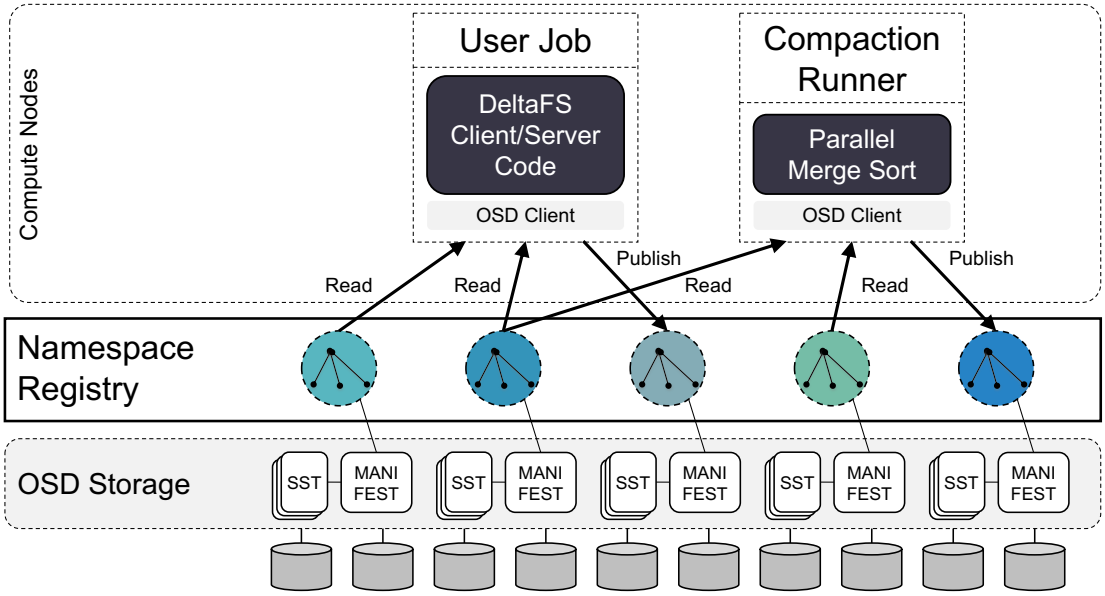


Figure 2: Architecture of DeltaFS. An DeltaFS cluster consists of per-job DeltaFS client/server instances and dynamically instantiated compaction runners on compute nodes reading, merging, and publishing filesystem namespace snapshots to a public registry that maps snapshot names to snapshot data stored in a shared underlying object store.

---

global synchronization as we show in Figure 1. To achieve this, we imagine running a public namespace registry to which all jobs can publish their namespaces as snapshots. When a job starts, it selects a subset of these snapshots as input and ends by publishing a new snapshot comprising all of the job's output. This new snapshot can then be used by an interested followup job to serve as its input, achieving efficient inter-job data propagation.

Namespace snapshots can be compact, and easy to generate given a log-structured filesystem metadata format: each snapshot is simply pointers to a set of filesystem namespace change logs [48, 50, 75]. In addition, with jobs each referencing a snapshot to start, there need not even be a global filesystem namespace. When a job needs to access data from multiple input snapshots, it simply merges all of these snapshots to form a new, flattened representation for fast metadata read performance. Better, the job can start its own server processes on its own compute nodes to perform the merge and then to serve these reads, achieving scalable read performance not restricted by the resources dedicated by the cluster administrators and better utilizing the massive parallelism enabled by today's computing platforms. Plus, unrelated jobs never have to communicate: they simply work on different snapshots. Out of this relaxed, scalable, log-structured, no global namespace, no ground truth, parallel filesystem metadata principle, we have designed DeltaFS as we now discuss.

3 System Overview

DeltaFS is a service that can be dynamically instantiated as threads or as standalone processes running on compute nodes within a job to provide scalable parallel metadata access private to the job on top of a shared underlying object store [66, 29, 9]. In addition to jobs, the building blocks of the DeltaFS filesystem include also public *Namespace Registries* and dynamically instantiated parallel *Compaction Runners* as Figure 2 illustrates.

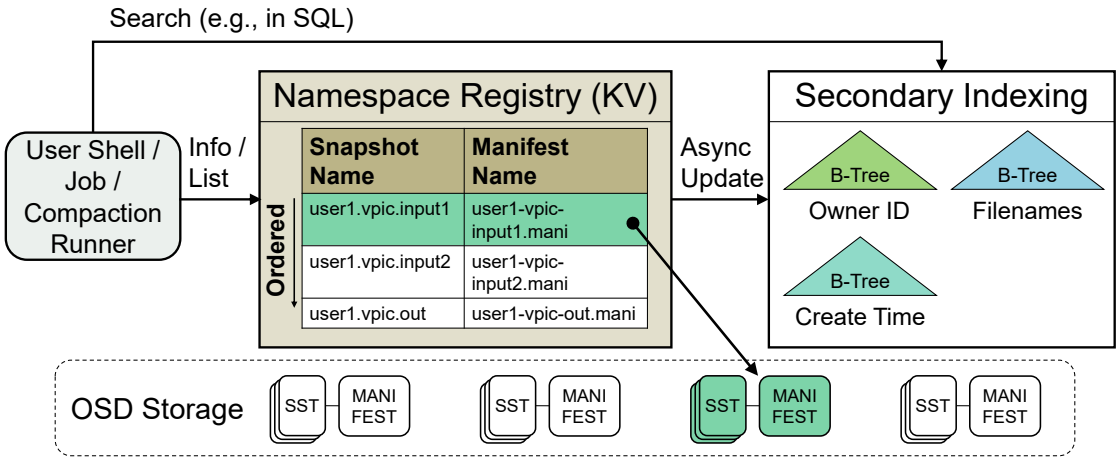


Figure 3: Locating snapshots in DeltaFS. A job, a compaction runner, or an interactive user uses namespace registries to locate snapshots according to their names. One or more secondary indexes may be built to allow for richer queries.

Jobs Jobs are parallel programs or scripts that we submit to run on compute nodes [63]. In DeltaFS, jobs are managers of their own filesystem metadata — each job self-defines its filesystem namespace at job bootstrapping (by looking up and potentially merging namespace snapshots published by previous jobs), allocates compute cores to serve the namespace, and may release its namespace as a public snapshot for sequential data sharing at the end of the job. This approach contrasts with today's parallel filesystems in which a single global filesystem namespace is provided to all jobs and dedicated metadata servers are deployed to maintain it.

To access and manage metadata, a job instantiates DeltaFS client and server instances in the job's processes. A job may have many processes (e.g., a parallel simulation). Each of these processes can act as a client, and additionally as a server — the DeltaFS library code linked into them is capable of being both a client and a server. Clients communicate with servers using RPCs [10, 58]. Addresses of the servers are sent to clients through a bootstrapping mechanism [23, 53] at the beginning of each job. When these addresses need to be known by code other than the job (e.g., the owning user's job monitoring program), they may be published at an external coordination service [27, 12, 34] for subsequent queries.

When a set of related jobs form a workflow and are scheduled to run consecutively (e.g., a simulation directly followed by post processing and then data analytics), it is possible to use a single DeltaFS filesystem instance to serve the entire workflow to improve performance — no need to repeatedly acquire and release filesystem namespace snapshots within a workflow and repeatedly restart from an empty filesystem metadata cache. To achieve this, the workflow manager (can be as simple as a job script) spawns DeltaFS servers as standalone processes (not embedded in a job process) on compute nodes. These standalone servers can then outlive each individual job and be reused by them for efficient metadata access. The workflow manager de-allocates the servers when the last job ends concluding the workflow.

Within a job, application code interacts with DeltaFS by making DeltaFS library calls. DeltaFS handles all metadata operations (e.g., `mkdir` and `create`). Data operations (e.g., `read` and `write`) are redirected to the underlying object storage for scalable processing [21, 65, 50]. A newly created but growing file may be transparently stripped across multiple data objects for parallel data operations within a single file [16, 14]. When a file is opened for writing, some of its attributes (e.g., file size and last access time) may change relative to DeltaFS's per-open copy of the attributes. DeltaFS captures these changes on file close using its metadata path.

To attain high metadata performance, DeltaFS aggressively partitions a namespace to achieve fast reads, uses client logging to quickly absorb bursts of writes, and packs metadata into large log objects (SSTables) stored in the shared underlying object store for efficient storage accesses as we explain more in later sections.

Namespace Registries Registries are keepers of all published DeltaFS filesystem namespace snapshots. A registry can be thought of as a Key-Value (KV) table mapping snapshot names (K) to pointers (V) to the snapshots' manifest objects stored in the shared underlying object store, as we show in Figure 2 and 3. As Section 5 will explain in more detail, DeltaFS namespace snapshots are made up of packed metadata mutation logs that are stored on storage as SSTables [22]. The manifest is a special object that is inserted into each snapshot to serve as its root index. It contains, among others, the names of all member logs (SSTables) of a snapshot and the key range of each of these logs. The read path code uses this information to locate snapshot data and to speed up queries against it.

In DeltaFS, unrelated jobs never have to communicate. Related jobs may communicate using DeltaFS to achieve efficient sequential data sharing. This is done by a preceding job first publishing its namespace as a snapshot and then by a followup job looking the snapshot up at a later point in time. To publish a namespace as a snapshot, a job (at the end of its run) flushes its remaining in-memory state (MemTables) to storage, writes the manifest, and then sends the object name of the manifest and the name of the snapshot to the registry for publication. To read back a snapshot, a followup job sends the name of the snapshot to the registry in exchange for the name of the snapshot's manifest object. The job then reads the manifest object and uses it to inform queries into the snapshot.

Namespace snapshots are named by jobs the same way as files are named by applications in today's filesystems — jobs present names; DeltaFS checks uniqueness. Similarly, just as today's applications must know the name of a file in order to operate on it, an DeltaFS job must know the name of a snapshot in order to look it up at a registry. To obtain filenames, it is possible for today's applications to list files under a given parent filesystem directory. DeltaFS retains the same capability by allowing jobs to list snapshots according to a job-specified prefix string (snapshot names are indexed as ordered strings). In addition to simple snapshot listing, there can also be a secondary indexing tier where snapshots are indexed by attributes other than their names (e.g., owner ID, create time, and even filenames within a snapshot) as Figure 3 shows. Modern database techniques could do this and allow for rich SQL queries [37, 13, 6, 40, 33]. Meanwhile, we also imagine running an Internet-style search engine where users can search snapshots as if they were searching the web (e.g., "the latest App X's input deck").

Registries run on dedicated server nodes in a computing cluster. A cluster may be paired with one or more registries. Each registry then manages a partition of the snapshots' key space [15, 35]. While running registries using dedicate resources limits performance to the machines dedicated, registries do not sit on the critical path for filesystem metadata operations so their performance is less critical to the overall metadata performance of DeltaFS even for metadata intensive workloads. In practice, we expect registries to be as busy as today's job scheduler queues for write operations (i.e., snapshot publications) [45, 3]. To provide low-latency, interactive read access (i.e., snapshot queries) to users invoking DeltaFS commands (e.g., `DeltaFS-snap-list` and `DeltaFS-snap-info`) on login nodes, DeltaFS registries can be hardened through established techniques such as bigger memory, replication, and an increase in registry count.

Compaction Runners Compaction runners are parallel log compaction code dynamically launched on compute nodes to merge and re-partition the metadata mutation logs (SSTables) generated by one or more previous jobs to form a flattened, read-optimized view of a filesystem namespace for efficient queries by a followup job. The ability to run compaction over a large number of compute cores on an as-needed basis is an important way DeltaFS differs from today's parallel filesystems, in which global filesystem metadata is maintained by a dedicated server process on behalf of all jobs, leaving the server often unable to keep

up with the clients under metadata-intensive workloads [2, 8]. To run parallel compaction, a user submits a special DeltaFS program (`DeltaFS-compaction-runner`) to the job scheduler queue and waits for it to be launched on compute nodes, as we discuss in Section 7.

4 No Ground Truth

DeltaFS does not provide a global filesystem namespace to all application jobs. Instead, it records the metadata mutations each job generates as immutable logs in a shared underlying object store and allows subsequent jobs to use these logs as "facts" to compose their own filesystem namespaces without requiring a single global ordering for all logs and without requiring all logs to be merged. Enabling jobs to choose what they see prevents unnecessary synchronization on top of a large computing cluster. A smaller filesystem metadata footprint per job further improves performance.

A Log-Structured Filesystem In DeltaFS, filesystem metadata information is persisted as logs. A metadata write operation (e.g., `mkdir` and `chmod`) applies changes by writing new log entries to storage. A metadata read operation (e.g., `lstat`) recalls information by searching and reading related log entries from storage. The DeltaFS library code linked into each job process knows the format of the log. Logs written by one job can be understood by all jobs, making inter-job data propagation possible.

Without assuming a global filesystem namespace, an DeltaFS job starts by defining a base filesystem namespace. In the simplest case, a job starts with an empty base and ends with a log recording all filesystem metadata mutations that the job has performed on the base. When a job needs to see the data output of a previous job, it uses the log produced by the previous job to instantiate its base namespace, absorbing all files and directories created by that job into its own filesystem namespace view. As the job later executes, it records all of its changes as new log entries and can elect to publish them at the end of the job. When published, these log entries can then be used by a subsequent job for instantiating its base namespace, achieving efficient sequential data sharing.

We call the log entries that a job generates, across all its processes, a *change set*. We call the final filesystem namespace view with which a job ends a *snapshot*. Thus each DeltaFS job can be thought of as a big log append operation: it appends a change set onto a previous snapshot producing a new snapshot, as Figure 4 shows. We call the change set a job appends in producing a new snapshot the *root change set* of the snapshot, with the manifest of it representing both the root change set itself and the entire snapshot it encompasses.

Multi-Inheritance & Name Resolution When instantiating a base, it is possible for a job to use multiple input snapshots. To ensure consistency within a job, a job specifies a priority ordering for all its input snapshots such that records from a higher priority snapshot take precedence. Figure 5 shows an example where jobs `A`, `B`, `C`, and `D` each take 0, 1, or more preceding jobs' snapshots as input (by referencing their root change sets), append a new change set onto it, and conclude by producing a new snapshot (namely snapshot `A`, `B`, `C`, and `D`). While both job `B` and `C` have created a "`/p/y`" in their respective namespaces, D sees the `/p/y` created in `B` rather than that in `C` due to B having a higher priority than C in D.

Custom client filesystem namespace views are also available from systems such as UnionFS [70] and OverlayFS [44]. DeltaFS differs from them in that it allows complex client namespace views to be efficiently materialized for fast reads through parallel compaction (§7) that can be dynamically invoked on compute nodes on an as-needed basis. Meanwhile, DeltaFS's log-structured metadata format (§5) enables efficient recording of client metadata mutations without being limited by copy-on-write and other overlay filesystem techniques. Finally, as a parallel filesystem, DeltaFS is able to spread workloads to distributed job processes

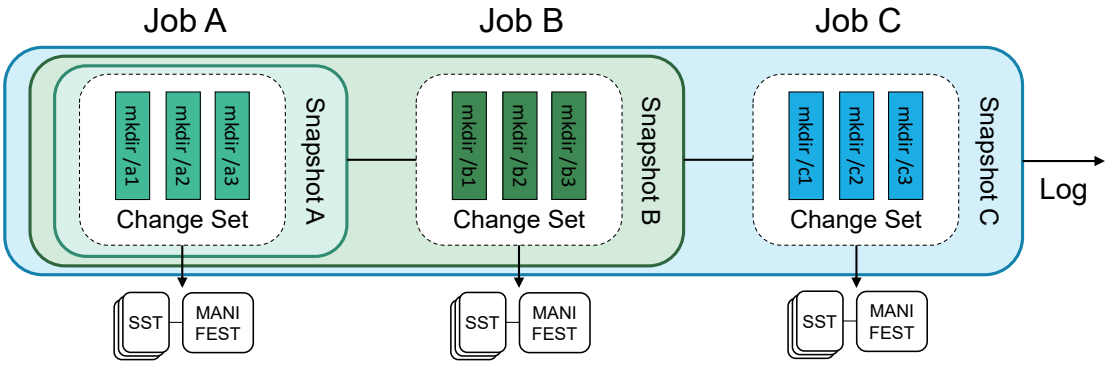


Figure 4: Job execution in DeltaFS. Each job generates a change set, extending a previous namespace snapshot and producing a new snapshot.

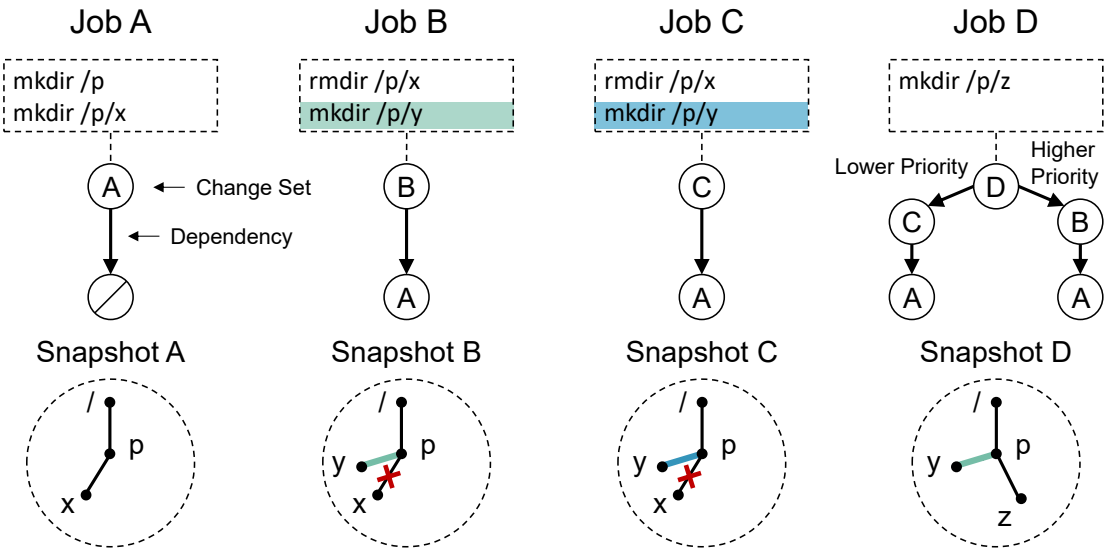


Figure 5: Priority-based name resolution in DeltaFS. Job D sees the /p/y created in B rather than that in C due to snapshot B having a higher priority than C.

to achieve scaling (§6) while a local overlay filesystem is fundamentally limited by the capacity of the local machine to attain high performance.

5 Per-Job Log Management

An DeltaFS job executes metadata operations by recording them as logs on storage. To attain high performance, DeltaFS uses a log format in which each filesystem metadata mutation is recorded as a KV pair in a table constructed with a Log-Structured Merge (LSM) Tree [43]. Tree data is persisted as named SSTables [22] indexed by a manifest object in a per-job change set. Background log compaction improves log storage for fast reads and handles garbage collection.

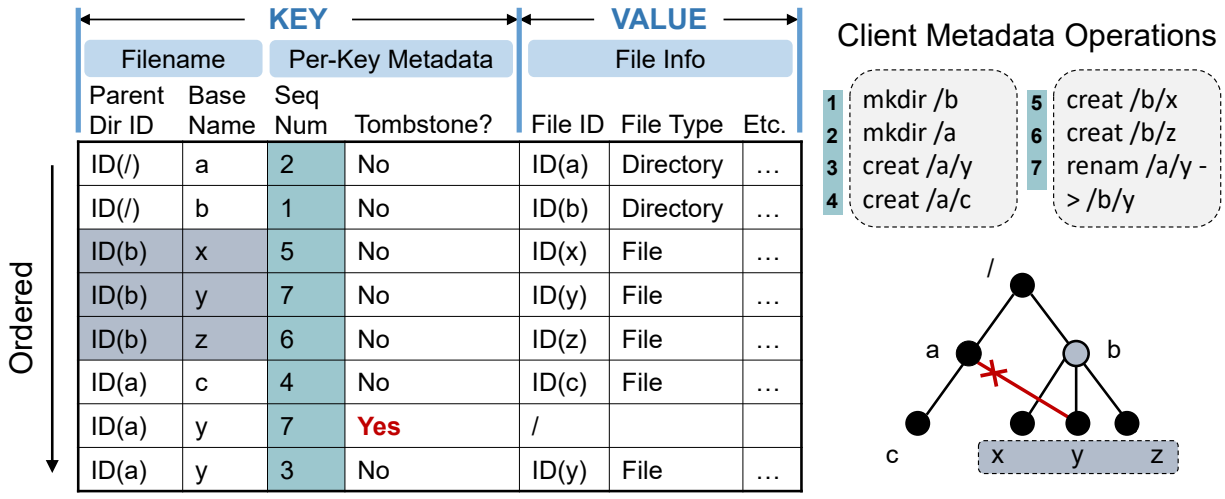


Figure 6: DeltaFS's table-based metadata log format. Each executed filesystem metadata mutation has an associated row in the table. Keys are ordered, enabling fast reads and scans.

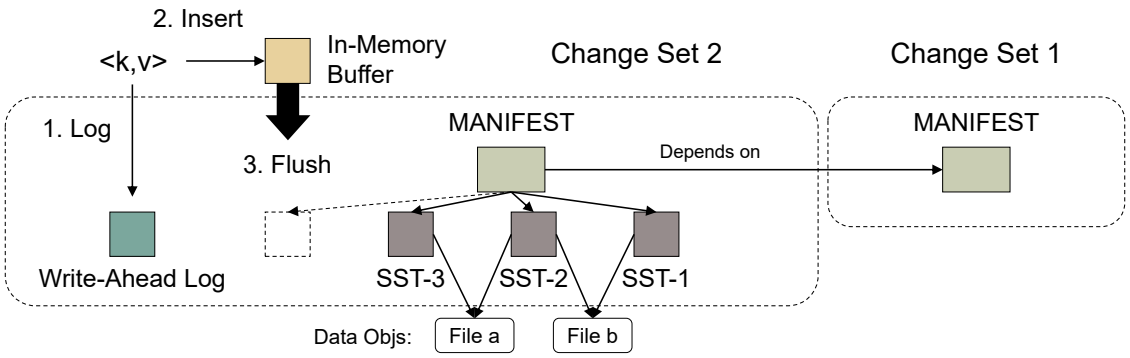


Figure 7: On-storage representation of an DeltaFS LSM-Tree in a per-job change set consisting of a manifest object, a write-ahead log, and a set of SSTables with references to separately stored data objects for large files.

Log Format DeltaFS logs a KV pair for each filesystem metadata mutation executed. The key stores the name of the file involved in a mutation. The value stores the metadata information (i.e., the inode) of the file after the change. A special tombstone bit is recorded in each key to indicate whether a logged mutation is a delete. Additionally, each key is associated with a sequence number; keys with higher sequence numbers supersede lower-numbered keys allowing newly logged changes to override older changes. All keys are inserted into a per-job table constructed with an DeltaFS-modified LevelDB realization of an LSM-Tree for high performance [43].

As Figure 6 illustrates, we use parent directory IDs and the base names of files to represent filenames. Using parent directory IDs (instead of their full pathnames) as key prefixes prevents potentially massive key updates when a user renames a directory [11, 71]. The metadata information we store for each file includes file ID, file type, file permissions for hierarchical access control, and file data for small files [50]. DeltaFS keys are ordered, allowing for efficient filesystem metadata lookups and directory scans [48, 39].

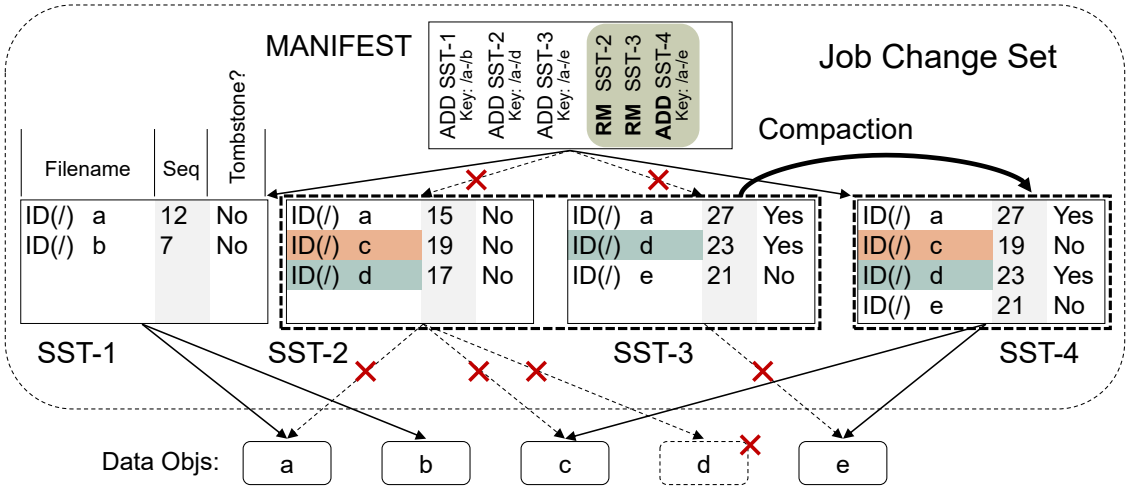


Figure 8: Example of an LSM-Tree compaction within a job change set. SSTable 2 and 3 are merged forming SSTable 4 reducing the number of SSTable lookups needed for key /c and garbage collecting data object d on storage.

On-Storage Log Management DeltaFS implements an LSM-Tree [43] — through reusing and modifying open source LevelDB code to support DeltaFS semantics — to manage logged filesystem metadata mutations in a per-job change set. As Figure 7 shows, when initializing an LSM-Tree for a change set, a job first creates a manifest object in the underlying object store to record information of the change set. This includes the name of the change set and the names of all its dependencies defined as the root change sets of all the job's input snapshots. Next, an in-memory buffer space is allocated in the job's process to buffer incoming metadata changes formatted as KV pairs as the job runs. Whenever the in-memory write buffer is full, all KV pairs in the buffer will be sorted and written to storage as an SSTable [22]. The name of the SSTable, as well as the key range of the table, is then recorded at the manifest for use by subsequent queries. To prevent data loss, a write-ahead log is created in the underlying object store for failure recovery of the job process's in-memory write buffer. A KV pair is first recorded at the write-ahead log before it is inserted into the in-memory buffer. The manifest, the write-ahead log, and the SSTables listed in the manifest constitute the entire state of a change set.

Change sets can be deleted when they are no longer needed. A user deletes a change set by invoking a special DeltaFS program (`DeltaFS-changeset-delete`) using the name of the change set as argument. To prevent deleting a change set while others may still depend on it, DeltaFS has each change set hold a reference to itself and to each of its dependencies. When a user deletes a change set, its reference to itself is removed. All member objects of the set — the manifest, the write-ahead log, and all of its SSTables — will be deleted when no other change set has a reference to it. When a change set is deleted, all its references to others will be removed enabling these change sets to be deleted too. A utility program (`DeltaFS-changeset-clean`) is provided that a user can periodically run to delete change sets that are no longer referenced. When a change set is deleted, DeltaFS deregisters its corresponding snapshot from the registry. For large files, their data objects are referenced counted as well. A data object is deleted when all the SSTables and write-ahead logs referencing it are deleted.

Log Compaction As a job runs, DeltaFS writes SSTables to persist changes whenever its in-memory write buffer is full. Entries in these SSTables may then be candidates for LRU replacement in the job's memory and get reloaded later by querying SSTables on storage until the first match occurs. SSTables are queried

backwards from the most recent to the oldest. Over time, the cost of finding a record that is not in the cache increases as the number of SSTables increases. To improve read performance, DeltaFS runs compaction to merge sort overlapping SSTables and decrease the number of SSTables that might share a key. As SSTables are merged, new SSTables are generated and old deleted. Data objects no longer referenced by any SSTables can then be deleted, achieving garbage collection.

Figure 8 shows an example where SSTable 2 and 3 are compacted to form SSTable 4. Before the compaction, looking up key /c would require searching 2 SSTables. SSTable 3 is searched first as its key range [/a-/e] overlaps key /c. Since SSTable 3 does not have the key, SSTable 2 is searched next finding the key. With compaction merging SSTable 3 and 2 into 4, key /c can now be found with a single SSTable lookup which improves read performance. When tables are merged, records sharing a same filename prefix are merged such that only the one with the highest sequence number is copied into the new table. The rest are discarded. After tables are merged, information on the newly constructed table is logged at manifest with old tables dissociated. These tables are then deleted from the underlying storage along with their references to the data objects for large files, enabling them to be garbage collected too.

6 Dynamic Service Instantiation

An DeltaFS filesystem consists of no dedicated metadata servers. Instead, a job dynamically instantiates DeltaFS client and server instances in the job's processes to provide parallel filesystem metadata access private to the job. DeltaFS aggressively partitions a namespace across a job's servers to achieve scalable read performance, and uses client logging to quickly absorb bursts of writes.

Per-Job Metadata Processing Within a job, DeltaFS metadata operations are processed by DeltaFS clients sending RPCs to servers. A server is capable of executing both read and write operations. Write operations are executed by logging the resulting metadata mutation on storage using an LSM-Tree making up the job's change set. Read operations are executed first by queries into the job's own change set, and then by querying any change set that the job declares as a dependency in a priority ordering that is, too, defined by the job (§4). A server performs log compaction asynchronously in the background as the job executes (§5).

When a job instantiates multiple DeltaFS servers, each server then manages a partition of the job's private filesystem namespace view. DeltaFS uses a namespace partitioning scheme derived from GIGA+ [46, 50] in which each newly created directory is randomly assigned to a server and gets gradually partitioned to more servers as it grows. Per-server metadata mutations are logged such as they each form a separate LSM-Tree in the job's change set. Each LSM-Tree represents a partition, and is indexed by a dedicated manifest object. The manifest object of the 0th partition additionally serves as the manifest of the entire change set, and is referenced by registries in their mapping tables (§3).

Client Logging Synchronization between DeltaFS clients and servers within a parallel job ensures that files created by one job process are immediately visible to all processes of the job. For workloads (e.g., N-N checkpointing) where files are opened for per-process writing [8, 9], DeltaFS allows a client to defer even job-wide synchronization and to directly log metadata mutations in a per-client LSM-Tree for ultra high metadata write performance. A client could perform background log compaction against its private LSM-Tree to improve its read performance. However, when files created by a client are known to be write-only and are not opened for read until after the job completes, an DeltaFS client can elect to further defer its log compaction and utilizes a subsequent parallel log compaction program to merge and re-partition all of the job clients' LSM-Trees in a single large batch enabling efficient job-wide read access, which we will discuss in Section 7.

Namespace Curation While a job logs its mutations in per-process logs for ultra high write performance, read requests unrelated to these mutations can still be served through the job's per-process DeltaFS servers according to the partitioning of the job's namespace. When the job's compute cores are insufficient for the workload at hand, a job could allocate a separate, larger set of compute nodes to run DeltaFS servers, utilizing the compute cores and the memory on those nodes to scale reads and to achieve low-latency access to the job's metadata on storage. These separately allocated DeltaFS servers may be reused by multiple jobs within a workflow, a project, or even a campaign [38], with the campaign manager requesting a persistent allocation of a set of compute nodes for running DeltaFS servers for an extended period of time. We call these read-only, job-specific, potentially long-running DeltaFS servers namespace curators. Critically, the amount of compute resources available to these curator processes is not decided by the cluster administrators — it is decided by the owners of the jobs, projects, or campaigns that are running, for high performance.

7 Cross-Job Parallel Log Compaction

All log-structured filesystems require compaction to achieve good read performance [52, 48, 49]. While today's parallel filesystem design limits compaction activities to only dedicated metadata servers, DeltaFS allows a user to dynamically launch compaction on compute nodes utilizing a potentially large amount of compute cores to minimize delays, and to perform compaction only on the job change sets that are known to be read by a followup job to optimize per-compaction metadata footprint.

While per-job log compaction is done by DeltaFS servers embedded in the job's processes as the job runs, a user launches separate parallel compaction runners for cross-job log compaction to merge and re-partition multiple job change sets on an as-needed basis. Typically, a user launches parallel compaction when a complex job change set hierarchy needs to be flattened for efficient queries (§4), a large set of per-client logged SSTables within a job change set needs to be merge-sorted for fast reads (§6), or a previously flattened, re-partitioned change set contains too few partitions for sufficient load balance across the job processes of a followup job.

We use parallel merge sort to perform cross-job compaction, with each compaction process acting as a mapper for a subset of input SSTables, and simultaneously as a reducer responsible for a partition of the target change set. Figure 9 shows an example where snapshot `C` — made up of a Directed Acyclic Graph (DAG) of change set `A`, `B`, and `C` with `C` being the root change set — is parallel compacted to form snapshot `D`. Note that change sets `A`, `B`, and `C` were originally partitioned by the jobs that generated them: jobs `A`, `B`, and `C` had 1, 1, and 2 server partitions, resulting in their change sets to be partitioned accordingly. Before parallel compaction, reading a key from snapshot `C` requires searching potentially 1 partition of change set `C`, 1 partition of change set `B`, and then 1 partition of change set `A`. After flattening, each key lookup requires searching only a single change set D and only 1 partition of it. In addition, change set `D` is expanded to have 8 parallel partitions — a followup job with 8 job processes can assign a partition to each of its per-process DeltaFS server instances, fully load balancing its reads.

8 Experiments

We implemented a prototype of DeltaFS in C++. A modular design was used such that DeltaFS can be layered on top of different object storage backends such as Ceph RADOS [66], PVFS [14], HDFS [57], and other generic POSIX parallel filesystems [55, 67, 56].

Our experiments evaluate the performance of DeltaFS both in terms of a single application job (§8.1) and multiple jobs sharing a single computing cluster (§8.2). We test cases in which jobs are related and use the filesystem for sequential data sharing, and cases in which jobs are unrelated and do not read each other's files. We compare DeltaFS with current state-of-the-art approaches: IndexFS [50] for scalable parallel

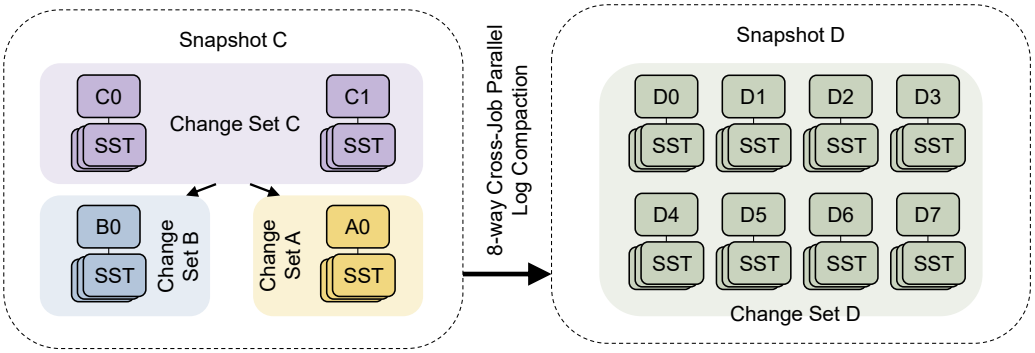


Figure 9: Example of one cross-job parallel log compaction in DeltaFS.

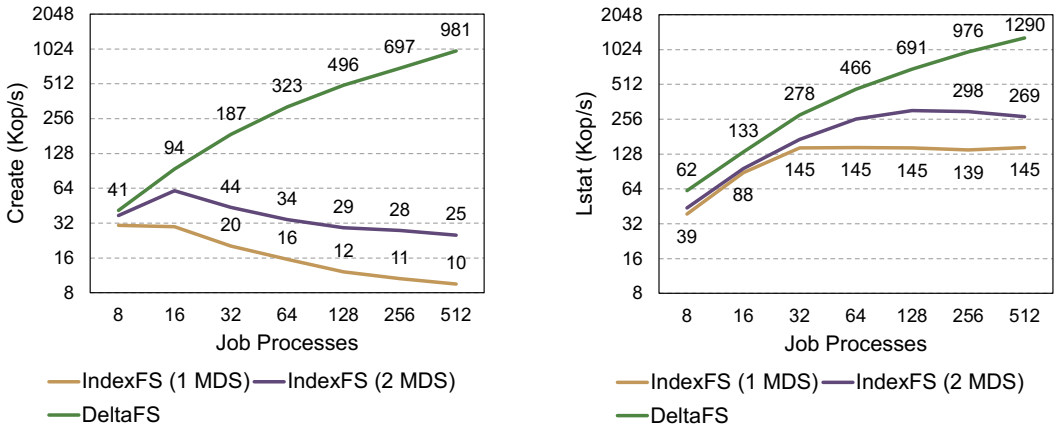
metadata performance, and PLFS [8] for ultra fast client-based metadata logging. We also compare against a special mode in which IndexFS allows clients to log metadata mutations for later bulk insertion. We run `mdtest` [30] to generate filesystem metadata operations. All our experiments store file metadata in a shared underlying object store implemented with Ceph RADOS on top of 8 dedicated Ceph OSD nodes along with 1 Ceph Manager and 1 Ceph Monitoring node. Each Ceph OSD features one 1GbE connection for foreground communication between Ceph clients and OSD servers and another 1GbE connection for background communication among Ceph OSDs, Managers, and Monitors.

8.1 Single-Job Performance

Today's parallel filesystems use dedicated metadata servers. Their performance is limited by the compute resources that a cluster admin assigns to the filesystem. With DeltaFS we show that parallel filesystems scale better without dedicated metadata servers. DeltaFS enables jobs to self-instantiate their metadata services on compute nodes, decoupling them from the decisions made by cluster admins and enabling scaling beyond a fixed set of server machines. To demonstrate this, our first experiment compares DeltaFS with IndexFS [50], a state-of-the-art approach for scalable parallel filesystem metadata performance using dynamic namespace partitioning.

Dynamic Namespace Partitioning IndexFS is a scalable parallel filesystem whose metadata is partitioned for load balancing across multiple dedicated metadata servers [50, 46]. To compare with it, our DeltaFS code implements the same namespace partitioning strategy as used in IndexFS. We dedicate 1 server node to run an DeltaFS server process to emulate an IndexFS filesystem with 1 dedicated metadata server and then 2 nodes (with 1 DeltaFS server process per node) to emulate an IndexFS with 2 dedicated metadata servers. In the latter case, the filesystem's namespace is partitioned and each dedicated DeltaFS server process manages one of the two partitions.

We use PRObE's Susitna cluster to run tests. Each Susitna compute node has four 16-core AMD Opteron 6272 2.1GHz CPUs, 128 GB memory, one 40GbE NIC, and one 1GbE NIC. A total of 10 nodes are used: 8 as client nodes (512 CPU cores), 2 as dedicated metadata servers. We use the 40GbE for filesystem operations and the 1GbE for accessing the shared underlying RADOS storage. Each test consists of running a parallel `mdtest` job that inserts empty files into a pool of parent directories and then queries the files it just created using the `stat` command. All runs start with an empty filesystem namespace. Files are created and `stat`'ed in random order in the namespace. Each job process creates and `stat`s 200K files. Our smallest run used 8 job processes and created 1.6M files. Our largest run consisted of 512 job processes



(a) Metadata Writes

(b) Metadata Reads

Figure 10: Results of parallel `mdtest` runs against IndexFS and DeltaFS.

and created 102.4M files. For IndexFS runs, all filesystem metadata operations are processed by up to 2 dedicated metadata servers. For DeltaFS runs, filesystem metadata operations are handled by the DeltaFS servers that have been instantiated dynamically by the job in its processes. Each job process launches 1 DeltaFS server instance, and manages a partition of the job's filesystem namespace.

Figure 10a shows the file insertion performance. IndexFS runs are limited by their dedicated metadata servers to deliver high performance. Their performance reduces as job size grows due to increased log compaction overhead at the server(s) as more files are inserted into the filesystem. Adding more dedicated metadata servers to IndexFS would alleviate this bottleneck, but a large number of servers might have to be dedicated permanently. DeltaFS decouples per-job metadata performance from dedicated resources. **By distributing work across all available compute nodes within a job, DeltaFS shows scalable performance that increases as job size grows.** At 512 job processes, DeltaFS is up to 98.1x faster than IndexFS, thanks both to having more CPU cores for clients' RPC requests and to less log compaction overhead due to namespace partitioning. Figure 10b shows file `stat` command performance. Similar to file creates, DeltaFS shows scalable performance that is not limited by a dedicated set of machines. At 512 job processes, DeltaFS is up to 8.9x faster.

Client Logging Ultra fast filesystem metadata insertion performance has been recently demonstrated through client side logging for metadata-intensive workloads such as N-N checkpointing in which newly created files are not immediately opened for read [8, 50]. Client logging allows for fast writes, but does not necessarily address the performance of the reads following the writes. Our second experiment shows that, by combining client logging with client-funded parallel log compaction (§7), DeltaFS is able to achieve not only fast writes, but also fast reads, more completely addressing the metadata bottlenecks seen by today's extreme workloads.

We compare DeltaFS against IndexFS and PLFS. PLFS was developed for concurrently writing a single file (e.g., N-1 checkpoints) [8]. It defers global synchronization of writes by logging the writes of each process instead of processing them immediately. In addition to file data mutations, PLFS-like techniques have also been used to record filesystem namespace mutations [7]. The IndexFS scalable parallel filesystem includes an extension that allows a set of client processes to write-lock a newly created directory and instead of synchronously integrating every filesystem metadata mutation beneath this directory, they each simply

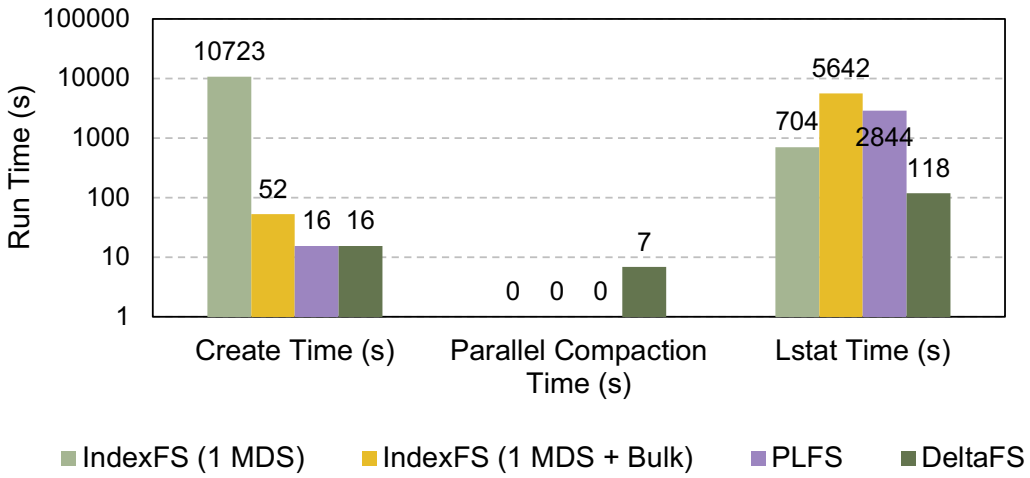


Figure 11: Comparison of client logging and subsequent read performance among IndexFS, PLFS, and DeltaFS.

log operations to be applied later in a bulk insertion [50]. We implemented both PLFS and the IndexFS's bulk insertion extension in our DeltaFS code, taking advantage of DeltaFS's log-structured metadata representation and the presence of a shared underlying object store.

We run the same test as we did in our first experiment but this time with client-side logging enabled. We focus on the configuration in which 512 job processes are launched creating 102.4M files. For IndexFS bulk insertion runs, all clients log their file creates in per-client SSTables and inform the server of their SSTables at the end of the write phase. Subsequent read operations are processed by the server, as in the original IndexFS runs. For PLFS runs, per-client SSTables logged at the write phase are directly opened by clients at the read phase. All clients open all other clients' SSTables for random reads. Finally, for DeltaFS runs, client logged SSTables at the write phase are merged and re-partitioned through a 512-way parallel log compaction process invoked by DeltaFS in the job's processes before moving to the read phase, allowing for fast reads.

Figure 11 shows the results in the form of the time it takes for each run to finish the write, the parallel log compaction (DeltaFS only), and the read phases. For reference, we included in the figure the original IndexFS (1 MDS) results from our previous experiment. By not synchronously integrating every file create operation to a dedicated server, client logging significantly improves a job's write performance for all of IndexFS, PLFS, and DeltaFS. Bulk-inserting IndexFS takes a little longer to finish due to having to report to the server per-client SSTables at the end of the write phase.

On the read side, even with bulk insertion, IndexFS performance is limited due to having only a single dedicated metadata server for reads. Worse, server-based background log (SSTable) compaction — which asynchronously optimizes SSTables to a read-optimized representation — is now deferred to the read phase after clients bulk inserting their tables, resulting in a much slower read phase overall. PLFS spreads reads across all of the job's processes. However, each read is likely to have to query an excessive number of SSTables due to their overlapping key ranges in the absence of any log compaction operations. **DeltaFS leans on the parallelism available by leveraging the job's processes to complete log compaction faster, speeding up subsequent reads.** Specifically, DeltaFS queries files 5.9×, 47.8×, 24.1× faster than IndexFS, IndexFS bulk insertion, and PLFS (Figure 11). It took DeltaFS longer to finish the reads (118s) than it did in Figure 10b (80s) due to having to start from a cold metadata cache following client logging and parallel log compaction. Nevertheless, through decoupling and parallelizing all of the write, read, and log compaction

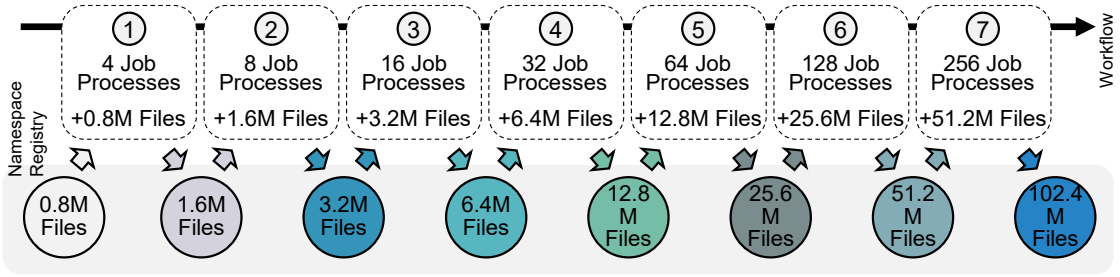


Figure 12: Illustration of our test workflow. Increasing file count calls for increasing parallelism to minimize delays.

operations, DeltaFS managed a much shorter overall run time (write + read + compaction) compared with IndexFS, IndexFS bulk insertion, and PLFS.

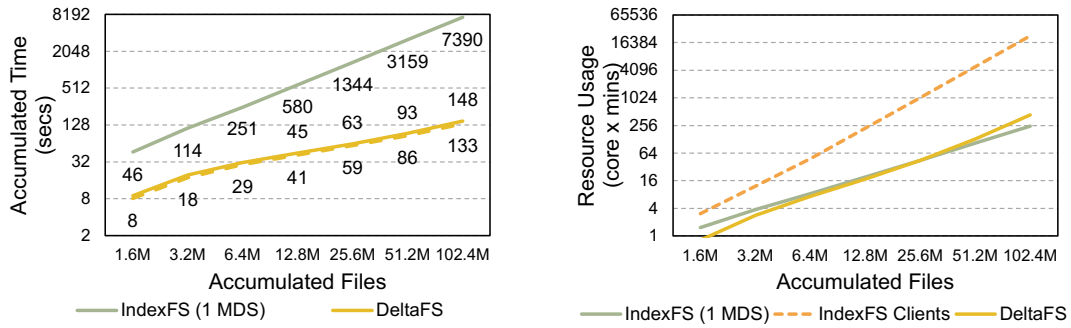
8.2 Multi-Job Performance

Enabling jobs to self-fund their metadata read, write, and compaction operations allows DeltaFS to vastly improve per-job metadata performance. In this section, we show that the same DeltaFS techniques can be applied to multi-job scenarios as effectively as they are for single jobs even in cases in which related jobs use the filesystem for sequential data sharing in the absence of a global filesystem namespace.

No Ground Truth Freedom from global serialization comes at the cost of the additional need for jobs to explicitly merge and compact related namespace snapshots before they can access them efficiently. It is difficult to identify typical workflow patterns [69], so to measure this cost we devised a synthetic 7-stage workflow shown in Figure 12. Each workflow stage takes a previous filesystem namespace snapshot as input, inserts a certain number of files into it doubling the amount files in the namespace, and ends by publishing it as a new snapshot. The workflow starts with a snapshot consisting of 0.8M files. It ends with 7 new snapshots with the last one consisting of 102.4M files. We compare running the workflow using IndexFS (1 MDS) with running it using DeltaFS. For IndexFS runs, all files are directly inserted into the global IndexFS namespace, with the namespace starting with 0.8M preexisting files and ending with 102.4M files.

We use PRObE's Narwhal cluster to run tests. Each Narwhal compute node has 4 Dual-Core AMD Opteron 2210 1.8GHz CPUs, 16 GB memory, and two 1GbE NICs. We use one NIC for filesystem operations and the other for accessing the shared underlying RADOS. Up to 128 nodes were allocated to run workflow jobs. We use `mdtest` to create files. Each workflow stage runs an increasing amount of `mdtest` job processes. The first workflow stage consists of 4 `mdtest` job processes inserting 0.8M files. The last stage consists of 256 processes inserting 51.2M files. For IndexFS runs, all job processes synchronize with a dedicated IndexFS metadata server to create files. For DeltaFS runs, file creates are first logged as per-client SSTables. A parallel log compaction program follows each `mdtest` job. It merges both the per-client SSTables generated by the job and the SSTables belonging to the original input snapshot to form a combined, read-optimized namespace view which is then published as a new snapshot. This new snapshot is then logically equivalent to the IndexFS's global filesystem namespace at the moment in that both contain all files that have been created so far and that both are read-optimized (IndexFS server runs log compaction in the background). Meanwhile, the extra processing time and compute resources that a job pays to run parallel log compaction to generate this namespace captures the cost of no ground truth in DeltaFS.

Figure 13a shows the accumulative time it takes for each filesystem to finish the workflow stages. Due to a lack of a global filesystem namespace and a dedicated metadata server to perform background



(a) Run Time

(b) Resource Usage

Figure 13: Results of running a 7-stage workflow against IndexFS and DeltaFS.

log compaction, DeltaFS jobs must self-merge and compact logs to achieve sequential data sharing and for fast reads. Nevertheless, by spreading the work across all job processes, DeltaFS still runs 50x faster than IndexFS, which are limited by a dedicated metadata server to achieve high performance. The dotted line in the figure depicts an DeltaFS run where jobs in the workflow are configured to be unrelated; they each start with an empty input and end by publishing a snapshot made up only of its own files. Due to not having to merge logs from previous snapshots, DeltaFS finished faster, although the difference is small due to the efficiency of parallel compaction which increases with the job size. The difference between the two DeltaFS runs demonstrates the cost of sequential data sharing in DeltaFS.

Figure 13b shows the accumulative resource usage (in the form of CPU cores × mins) each workflow takes to process all of the filesystem metadata operations. For IndexFS, this is the usage of its dedicated metadata server. For DeltaFS, this aggregates all of its job processes' resource usage. DeltaFS used around 2× more compute resources than IndexFS due to aggressive deep client logging such that all metadata changes of a job are first logged and then subsequently merged in their entirety through a parallel log compaction program causing lots of I/O operations and job compute core usage. **Even though IndexFS spent fewer total CPU core-minutes at the server, it effectively wasted a massive amount of client-side compute resources by having processes blocked on filesystem operations** (plotted in Figure 13b as a dotted line), since they cannot make progress until those operations complete. Specifically, DeltaFS uses 52.9× fewer CPU core-minutes than IndexFS in total, of which 98.9% were core-minutes spent on the clients alone. This result is due to IndexFS limiting metadata processing to dedicated server machines.

9 Related Work

Large-scale parallel filesystems have long served as an important storage infrastructure in modern computing data centers [63]. While the conventional wisdom is to put both namespace servers and file storage into a single layered system [14, 55, 56], DeltaFS envisions them to be loosely related but separate systems. This allows metadata to be accessed from a provisional service spawned by each application on-demand. Data is placed on a set of unrelated traditional "forever-running" service silos that can be upgraded or extended independently [53].

Prior on-compute filesystem research such as FusionFS [73] has focused on forming a scalable storage tier using distributed compute node resources.

When local storage is available, DeltaFS can be configured to store data on compute nodes to achieve the same effect. 1. reduce inteference 2. perjob communication

Modern burst buffer software such as Cray's DataWarp [24] provides data stage in and out services. DeltaFS's namespace publication mechanism works differently in that it does not require all changes to be merged back to a single, global namespace at job completion and instead enables jobs to communicate only on an as-needed basis.

Novel burst-buffer-based filesystems such as BurstFS [64] and GekkoFS [62] provide applications with an ephemeral namespace that has the same life cycle as the job. While DeltaFS is able to provide the same semantics as these filesystems, DeltaFS additionally allows namespaces to outlive their jobs as immutable snapshots in a public registry for inter-job communication. Related snapshots can be efficiently merged as needed through parallel log compaction on compute nodes.

Quickly absorbing a large amount of small files has become an important capability for modern parallel filesystems. PLFS [8, 7] used an append-to-end format for high metadata write throughput, but at the expense of reads. To achieve more balanced reads and writes, recent filesystems have leveraged more advanced data structures for filesystem metadata management. Examples of these efforts include TABLEFS [48], IndexFS [50], and XtreemFS [28, 59] which used LSM-Trees [43], BetrFS which used Fractal Trees [32, 19], and LocoFS [39] which used a combination of hash tables and B-Trees for file and directory management separately.

Many filesystems partition their namespaces for dynamic load balancing across their servers. Farsite [18] used a strategy in which namespace is partitioned according to a novel tree structured file identifying mechanism that minimizes data movement when directories are renamed. GIGA+ [46], IndexFS [50], and skyFS [72] achieved load balancing by aggressively splitting a directory as it grows. Ceph [65] partitioned its namespace on a per-subtree basis for improved locality. Finally, ShardFS [71], LocoFS [39], and MarFS [29] decoupled file partitioning from directory management for improved access performance on files.

It is not new to have a cluster of computers collectively share a filesystem on a distributed data store without requiring a dedicated metadata manager [34, 4, 61]. In these shared environments, each filesystem client runs an embedded metadata manager. This manager serves both the client and other clients sharing the same filesystem in a local area network. All metadata manager understand the filesystem's on-storage data format, and can dynamically assume responsibility for any files or directories in the filesystem when accessed. To achieve synchronization, distributed locking is used to control access to the shared filesystem and to client data and metadata caches.

Today, such a filesystem metadata approach is mostly seen in Storage Area Network (SAN) filesystems [47, 20, 68], including the GPFS filesystem [55] widely used in HPC environments. Scalability is often an issue due to the large amount of synchronization needed to access the filesystem [17]. To mitigate this problem, real world deployments typically dedicate a small set of nodes to run filesystem clients with embedded metadata managers. These clients then act as filesystem servers, exporting the filesystem to a larger cluster of filesystem users on job-running compute nodes without metadata managers. Notwithstanding many benefits, such deployment approaches largely defeat the goal of having no dedicated metadata managers, and fail to utilize compute node resources to operate on filesystem metadata.

10 Conclusion

It has been a tradition that, every once in a while, we stop and reassess whether we need to build our next filesystems differently. A key previous effort was made by the NASD project [21], which decoupled filesystem data communication from metadata management and leveraged object storage devices for scalable data access. Similar bold ideas that reassess component communication are needed to advance parallel filesystem performance if we are to keep with up the rapidly increasing scale of today's massively-parallel computing environments.

DeltaFS is based on the premise that at exascale and beyond, synchronization of anything global should

be avoided. Conventional parallel filesystems, with fully synchronous and consistent namespaces, mandate synchronization with every file create and other metadata operations. This has to stop. Moreover, the idea of dedicating a single filesystem metadata service to meet the needs of all applications running on a single computing environment, is archaic and inflexible. This too must stop. DeltaFS shifts away from constant global synchronization and dedicated filesystem metadata servers, towards the notion of viewing the filesystem as a service instantiated at each process of a running job, leveraging client resources to scale its performance along with the job size. Synchronization is limited to an as-needed basis that is determined by the needs of followup jobs, through an efficient, log-structured format that lends itself to deep metadata writeback buffering and merging.

Our evaluation of DeltaFS suggests that its aggressive approach to handling filesystem metadata may be the way forward in order to unlock scalable parallel metadata performance that is unattainable with today's monolithic, one-size-fits-all storage solutions.

References

[1] S. V. Adve and K. Gharachorloo. "Shared memory consistency models: a tutorial". In: *Computer* 29.12 (Dec. 1996), pp. 66–76. DOI: 10.1109/2.546611.

[2] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. "Parallel I/O and the Metadata Wall". In: *PDSW*. 2011, pp. 13–18. DOI: 10.1145/2159352.2159356.

[3] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben. "On the diversity of cluster workloads and its impact on research results". In: *USENIX ATC*. July 2018.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. "Serverless Network File Systems". In: *ACM Trans. Comput. Syst.* 14.1 (Feb. 1996), pp. 41–79. DOI: 10.1145/225535.225537.

[5] *APEX Workflows*. https://www.nersc.gov/assets/apex-workflows-v2.pdf. Mar. 2016.

[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *SIGMOD*. 2015, pp. 1383–1394. DOI: 10.1145/2723372.2742797.

[7] J. Bent. "PLFS: Software-Defined Storage for HPC". In: *High Performance Parallel I/O*. Ed. by Prabhat and Koziol. Chapman & Hall/CRC Computational Science. CRC Press, 2015. Chap. 14, pp. 169–176.

[8] 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: *SC*. 2009, 21:1–21:12. DOI: 10.1145/1654059.1654081.

[9] J. Bent, B. Settlemyer, and G. Grider. "Serving Data to the Lunatic Fringe: The Evolution of HPC Storage". In: *USENIX ;login:* 41.2 (June 2016).

[10] A. D. Birrell and B. J. Nelson. "Implementing Remote Procedure Calls". In: *SIGOPS Oper. Syst. Rev.* 17.5 (Oct. 1983). DOI: 10.1145/773379.806609.

[11] S. A. Brandt, E. L. Miller, D. D. E. Long, and Lan Xue. "Efficient Metadata Management in Large Distributed Storage Systems". In: *MSST*. 2003, pp. 290–298. DOI: 10.1109/MASS.2003.1194865.

[12] M. Burrows. "The Chubby Lock Service for Loosely-coupled Distributed Systems". In: *OSDI*. 2006, pp. 335–350.

[13] J. Camacho-Rodrıguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, S. Shelukhin, P. Jayachandran, S. Seth, D. Jaiswal, S. Bouguerra, N. Bangarwa, S. Hariappan, A. Agarwal, J. Dere, D. Dai, T. Nair, N. Dembla, G. Vijayaraghavan, and G. Hagleitner. "Apache Hive: From MapReduce to Enterprise-Grade Big Data Warehousing". In: *SIGMOD*. 2019, pp. 1773–1786. DOI: 10.1145/3299869.3314045.

[14] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. "PVFS: A Parallel File System for Linux Clusters". In: *ALS*. 2000.

[15] 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: *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. DOI: 10.1145/1365815.1365816.

[16] P. F. Corbett and D. G. Feitelson. "The Vesta Parallel File System". In: *ACM Trans. Comput. Syst.* 14.3 (Aug. 1996), pp. 225–264. DOI: 10.1145/233557.233558.

[17] M. Devarakonda, A. Mohindra, J. Simoneaux, and W. H. Tetzlaff. "Evaluation of Design Alternative for a Cluster File System". In: *USENIX ATC*. 1995.

[18] J. R. Douceur and J. Howell. "Distributed Directory Service in the Farsite File System". In: *OSDI*. 2006, pp. 321–334.

[19] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. "The TokuFS Streaming File System". In: *HotStorage*. 2012.

[20] M. Fasheh. "OCFS2: The Oracle Clustered File System, Version 2". In: *Proceedings of the 2006 Linux Symposium*. 2006, pp. 289–302.

[21] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. "A Cost-effective, High-bandwidth Storage Architecture". In: *ASPLOS*. 1998, pp. 92–103. DOI: 10.1145/291069.291029.

[22] Google. *LevelDB*. https://github.com/google/leveldb/. 2013.

[23] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014.

[24] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright. "Architecture and Design of Cray Datawarp". In: *CUG*. https://cug.org/proceedings/cug2016_proceedings/includes/files/pap105s2-file1.pdf. 2016.

[25] W. D. Hillis and L. W. Tucker. "The CM-5 Connection Machine: A Scalable Supercomputer". In: *Commun. ACM* 36.11 (Nov. 1993), pp. 31–40. DOI: 10.1145/163359.163361.

[26] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. "Scale and Performance in a Distributed File System". In: *ACM Trans. Comput. Syst.* 6.1 (Feb. 1988), pp. 51–81. DOI: 10.1145/35037.35059.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *USENIX ATC*. 2010.

[28] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. "XtreemFS - A Case for Object-Based Storage in Grid Data Management". In: *2007 VLDB Workshop on Data Management in Grids (DMG)*. 2007.

[29] J. Inman, W. Vining, G. Ransom, and G. Grider. "MarFS, a Near-POSIX Interface to Cloud Objects". In: *USENIX ;login:* 42.1 (Jan. 2017).

[30] *IOR/mdtest*. https://github.com/hpc/ior. 2020.

[31] *ISO/IEC 9899:2018 Information technology — Programming languages — C*. https://www.iso.org/standard/74528.html. June 2018.

[32] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. "BetrFS: A Right-Optimized Write-Optimized File System". In: *FAST*. 2015, pp. 301–315.

[33] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. "Impala: A Modern, Open-Source SQL Engine for Hadoop". In: *CIDR*. 2015.

[34] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. "VAXcluster: A Closely-Coupled Distributed System". In: *ACM Trans. Comput. Syst.* 4.2 (May 1986), pp. 130–146. DOI: 10.1145/214419.214421.

[35] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922.

[36] Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9 (Sept. 1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[37] LANL. *Grand Unified File-Index*. https://github.com/mar-file-system/GUFI. 2018.

[38] LANL, NERSC, SNL. *Crossroads Workflows*. https://www.lanl.gov/projects/crossroads/__internal/__blocks/xroads-workflows.pdf. July 2018.

[39] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li. "LocoFS: A Loosely-coupled Metadata Service for Distributed File Systems". In: *SC*. 2017, 4:1–4:12. DOI: 10.1145/3126908.3126928.

[40] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 330–339. DOI: 10.14778/1920841.1920886.

[41] D. Monroe. "Fugaku Takes the Lead". In: *Commun. ACM* 64.1 (Dec. 2020), pp. 16–18. DOI: 10.1145/3433954. URL: https://doi.org/10.1145/3433954.

[42] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. "Caching in the Sprite Network File System". In: *ACM Trans. Comput. Syst.* 6.1 (Feb. 1988), pp. 134–154. DOI: 10.1145/35037.42183.

[43] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The Log-structured Merge-tree (LSM-tree)". In: *Acta Inf.* 33.4 (June 1996), pp. 351–385. DOI: 10.1007/s002360050048.

[44] *OverlayFS*. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. 2014.

[45] T. Patel, Z. Liu, R. Kettimuthu, P. Rich, W. Allcock, and D. Tiwari. "Job Characteristics on Large-Scale Systems: Long-Term Analysis, Quantification, and Implications". In: *SC*. Nov. 2020, pp. 1–17. DOI: 10.1109/SC41405.2020.00088.

[46] S. Patil and G. Gibson. "Scale and Concurrency of GIGA+: File System Directories with Millions of Files". In: *FAST*. 2011.

[47] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. "A 64-bit, shared disk file system for Linux". In: *IEEE MASS*. 1999, pp. 22–41. DOI: 10.1109/MASS.1999.829973.

[48] K. Ren and G. Gibson. "TABLEFS: Enhancing Metadata Efficiency in the Local File System". In: *USENIX ATC*. 2013, pp. 145–156.

[49] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. “SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data”. In: *Proc. VLDB Endow.* 10.13 (Sept. 2017), pp. 2037–2048. DOI: 10.14778/3151106.3151108.

[50] K. Ren, Q. Zheng, S. Patil, and G. Gibson. “IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion”. In: *SC*. 2014, pp. 237–248. DOI: 10.1109/SC.2014.25.

[51] D. M. Ritchie and K. Thompson. “The UNIX Time-Sharing System”. In: *Commun. ACM* 17.7 (July 1974), pp. 365–375. DOI: 10.1145/361011.361061.

[52] M. Rosenblum and J. K. Ousterhout. “The Design and Implementation of a Log-structured File System”. In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52. DOI: 10.1145/146941.146943.

[53] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, and Q. Zheng. “Mochi: Composing Data Services for High-Performance Computing Environments”. In: *Journal of Computer Science and Technology* 35.1, 121 (2020), p. 121. DOI: 10.1007/s11390-020-9802-0.

[54] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. “Design and Implementation of the Sun Network Filesystem”. In: *Proceedings of the USENIX 1985 Summer Conference*. 1985.

[55] F. B. Schmuck and R. L. Haskin. “GPFS: A Shared-Disk File System for Large Computing Clusters”. In: *FAST*. 2002, pp. 231–244.

[56] P. Schwan. “Lustre: Building a File System for 1000-Node Clusters”. In: *Proceedings of the 2003 Linux Symposium*. 2003, pp. 380–386.

[57] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. “The Hadoop Distributed File System”. In: *MSST*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[58] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. “Mercury: Enabling remote procedure call for high-performance computing”. In: *IEEE CLUSTER*. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702617.

[59] J. Stender, B. Kolbeck, M. Högqvist, and F. Hupfeld. “BabuDB: Fast and Efficient File System Metadata Storage”. In: *2010 International Workshop on Storage Network Architecture and Parallel I/Os*. 2010, pp. 51–58. DOI: 10.1109/SNAPI.2010.14.

[60] M. Stonebraker and U. Cetintemel. “"One Size Fits All": An Idea Whose Time Has Come and Gone”. In: *ICDE*. 2005, pp. 2–11. DOI: 10.1109/ICDE.2005.1.

[61] C. A. Thekkath, T. Mann, and E. K. Lee. “Frangipani: A Scalable Distributed File System”. In: *SOSP*. 1997, pp. 224–237. DOI: 10.1145/268998.266694.

[62] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann. “Gekkofs — a temporary burst buffer file system for HPC applications”. In: *Journal of Computer Science and Technology* 35.1 (2020), pp. 72–91.

[63] J. S. Vetter. *Contemporary High Performance Computing: From Petascale toward Exascale*. Vol. 1-2-3. CRC Press, 2019.

[64] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. “An Ephemeral Burst-buffer File System for Scientific Applications”. In: *SC*. 2016, 69:1–69:12. DOI: 10.1109/SC.2016.68.

[65] 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: *OSDI*. 2006, pp. 307–320.

[66] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters". In: *PDSW*. 2007, pp. 35–44. DOI: 10.1145/1374596.1374606.

[67] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. "Scalable Performance of the Panasas Parallel File System". In: *FAST*. 2008, 2:1–2:17.

[68] S. Whitehouse. "The GFS2 Filesystem". In: *Proceedings of the 2007 Linux Symposium*. 2007, pp. 253–260.

[69] *Workflows Community Summit: Bringing the Scientific Workflows Community Together*. https://arxiv.org/abs/2103.09181. 2021. DOI: 10.5281/zenodo.4606958.

[70] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. "Versatility and Unix Semantics in Namespace Unification". In: *ACM Trans. Storage* 2.1 (Feb. 2006), pp. 74–105. DOI: 10.1145/1138041.1138045.

[71] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson. "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems". In: *SoCC*. 2015, pp. 236–249. DOI: 10.1145/2806777.2806844.

[72] J. Xing, J. Xiong, N. Sun, and J. Ma. "Adaptive and Scalable Metadata Management to Support a Trillion Files". In: *SC*. 2009, 26:1–26:11. DOI: 10.1145/1654059.1654086.

[73] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. "FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems". In: *IEEE BigData*. 2014, pp. 61–70. DOI: 10.1109/BigData.2014.7004214.

[74] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo. "Scaling Embedded In-situ Indexing with DeltaFS". In: *SC*. 2018, 3:1–3:15. DOI: 10.1109/SC.2018.00006.

[75] Q. Zheng, K. Ren, and G. Gibson. "BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers". In: *PDSW*. 2014, pp. 1–6. DOI: 10.1109/PDSW.2014.7.

[76] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider. "DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers". In: *PDSW*. 2015, pp. 1–6. DOI: 10.1145/2834976.2834984.