Distributed Metadata and Streaming Data Indexing as Scalable Filesystem Services

Qing Zheng

CMU-CS-21-103

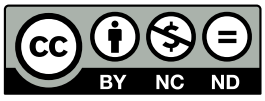
February 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Garth Gibson (Co-Chair)
George Amvrosiadis (Co-Chair)
Gregory Ganger
Bradley Settlemyer (Los Alamos National Laboratory)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was sponsored by the Los Alamos National Lab's Institute for Reliable High Performance Information Technology (DE-AC52-06NA25396/394903), the U.S. Dept of Energy's Advanced Scientific Computing Research (DE-SC0015234), the U.S. National Science Foundation's Parallel Reconfigurable Observational Environment (1042543), the New Mexico Consortium, and the Parallel Data Lab at Carnegie Mellon University. *The views and the conclusions are those of the author and do not necessarily represent the opinions of the sponsoring organizations, agencies, or U.S. Government.*



Keywords: High-performance computing, Distributed storage, File system metadata, Data indexing

Abstract

As people build larger and more powerful supercomputers, the sheer size of future machines will bring unprecedented levels of concurrency. For applications that write one file per process, increased concurrency will cause more files to be accessed simultaneously and this requires the metadata information of these files to be managed more efficiently. An important factor preventing existing HPC filesystems from being able to more efficiently absorb filesystem metadata mutations is the continued use of a single, globally consistent filesystem namespace to serve all applications running on a single computing environment. Having a shared filesystem namespace accessible from anywhere in a computing environment has many welcome benefits, but it increases each application process's communication with the filesystem's metadata servers for ordering concurrent filesystem metadata changes. This is especially the case when all the metadata synchronization and serialization work is coordinated by a small, fixed set of filesystem metadata servers as we see in many HPC platforms today. Since scientific applications are typically non-interactive batch programs, the first theme of this thesis is about taking advantage of knowledge about the system and scientific applications to drastically reduce, and in extreme cases, remove unnecessary filesystem metadata synchronization and serialization, enabling HPC applications to better enjoy the increasing level of concurrency in future HPC platforms.

While overcoming filesystem metadata bottlenecks during simulation I/O is important, achieving efficient analysis of large-scale simulation output is another important enabler for fast scientific discovery. With future machines, simulation output will only become larger and more detailed than it is today. To prevent analysis queries from experiencing excessive I/O delays, the simulation's output must be carefully reorganized for efficient retrieval. Data reorganization is necessary because simulation output is not always written in the optimal order for analysis queries. Data reorganization can be prohibitively time-consuming when its process requires data to be readback from storage in large volumes. The second theme of this thesis is about leveraging idle CPU cycles on the compute nodes of an application to perform data reorganization and indexing, enabling data to be transformed to a read-optimized format without undergoing expensive readbacks.

Acknowledgments

Getting a PhD is not easy, especially for a man who is not good at it. For this, I am grateful to my advisors, Garth Gibson and George Amvrosiadis, who supported and guided me throughout the years and were always there for me when I screwed up and needed their help. On top of that, Garth showed me how to efficiently communicate with people of different backgrounds and how to think and act as a researcher. Looking back, it was truly a pleasure and honor working with him. George took good care of me and my research during my final years and I am deeply thankful that he pushed me to finish my thesis writing early so that I didn't further delay my graduation. It was my great fortune to have both Garth and George as my advisors. I could not have asked for a better one. Brad Settlemyer invited me to a LANL internship in 2015 and has stood by me ever since supporting me and my research. I am grateful that he always had time to discuss research with me and that he gave me the freedom to pursue my own research agenda and ensured that I had all the resources that I needed to achieve my goals. This man is simply my hero. Greg Ganger and Gary Grider supports my research behind the scenes. They don't talk much, but comments tend to be critical when they do. Chuck Cranor is a hardcore computer scientist and a disciplined coder. He always manages to get things done and it was tremendous fun working with him. Joan Digney is the lead artist in the team. She did me numerous favors and I hope one day I could possess some of her great skills. Phil Carns, Rob Ross, and Jerome Soumagne provided lots of technical guidance on my research under the collaboration among ANL, LANL, HDF5, and CMU. John Bent kept being a good guy to chat with for research ideas during my internship at LANL. Deb Cavlovich handles all the administrative work at the department side. I owe her a great deal as she always promptly responses to my various requests. No doubt she is the best in class. As PDL's grandma, Karen Lindenfelser helped me a ton during those colorful PDL event days. Many thanks to Jinliang Wei and Chen Jin for being such a good friend. Many thanks to Kai Ren and Lin Xiao for their help and advice during the early stages of my research. Many thanks to Lynn Strauss for making the administrative process as smooth as possible at the NMC side. And many thanks to Catherine Copetas for walking me through and showing me how to make a good thesis title page during my final thesis submission. Finally, I am grateful to my family for their unconditional love and support.

Contents

List of Figures

1 Introduction

For decades high performance computing (HPC) applications have been technology leaders in utilization of massive concurrency, in no small part because they have not made all their memory store operations to be immediately visible to the next load operations invoked anywhere in the computing cluster. Instead, HPC applications run on distributed memory and use software components that provide only the consistency and synchronization needed for the task at hand. Unfortunately, while HPC applications communicate their in-memory state on an as-needed basis, their persistent state, stored as files and accessed through a shared underlying parallel filesystem service, is still globally synchronized even for the world's largest computing clusters — files created by one application process thread are immediately visible to any other process thread in a computing cluster. Alas, sequentially ordered, fully consistent for all accesses across all nodes, with dedicated servers in charge of all the heavy lifting, today's parallel filesystems are old-fashioned system services invented for single-core machines 50 years ago but forced to scale as rapidly as today's parallel computing machines. This is too difficult, and leads to performance bottlenecks that increasingly defeat massive parallelism.

To change this, this dissertation re-imagines the roles filesystems play in delivering performance and services to applications and presents new ways of providing filesystem and data processing capabilities on modern HPC platforms. First, today's filesystem clients synchronize too much with their servers for metadata reads and writes. We show deep relaxation of filesystem metadata synchronization and serialization through client logging and selective merging of filesystem metadata changes on an as-needed basis. Second, modern filesystems achieve scaling primarily by dynamic namespace partitioning over multiple dedicated metadata servers. Filesystem metadata performance is a function of, and thus fundamentally limited by, the amount of compute resources that are dedicated to servers. We show dynamic instantiation of filesystem metadata control and processing functions over client compute cores, enabling highly agile scaling of filesystem metadata performance beyond a fixed set of dedicated servers. Finally, an important reason applications write data to storage is that they

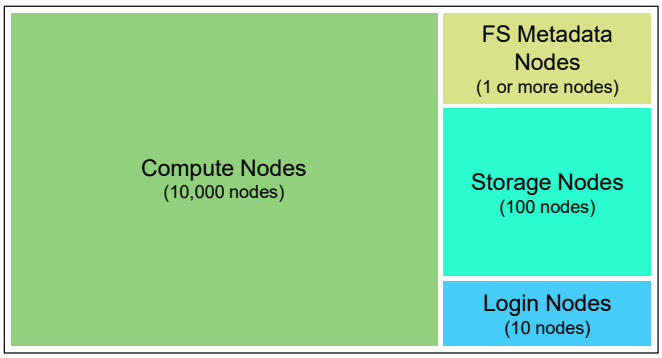


Figure 1.1: Anatomy of a typical computing cluster in HPC environments made up of compute nodes, storage nodes, filesystem (FS) metadata nodes, and login nodes. Scientific jobs run on compute nodes. A parallel filesystem provides shared underlying storage for all jobs. It uses the computing cluster's storage nodes for scalable I/O and its dedicated metadata nodes for filesystem metadata management. Scientists access a computing cluster through its login nodes, where scientists compile code, submit jobs, and manage their files. The best performance is achieved when work is spread across all job compute nodes and there are as few synchronization and inter-process communication activities as possible both across jobs and within a job.

can later query the data for insights. Efficient query performance has largely relied on post data processing, which reads back the data written by the application and re-writes it in a format that is optimized for the queries. We show scalable streaming data processing with rich filesystem directory types that transforms data to a read-optimized format as an application writes it to storage, preventing the potentially massive data movement and rewriting caused by today's post data processing steps.

The end result: a drastic reduction of modern parallel filesystems to plain object stores, and on top of it applications independently instantiate per-application client services for scalable filesystem and data management. There need not be a global filesystem namespace. Instead, applications communicate only when they need to and communication is done primarily through immutable log objects stored in the shared underlying object store for maximal concurrency. At the same time, filesystem metadata processing need not be limited to dedicated servers and query acceleration need not be deferred to post processing. Abundant compute resources available on job-running compute nodes are dynamically utilized for scaling storage services overcoming limitations and bottlenecks at dedicated servers. Streaming processing data as it is written using these abundant resources further allows queries to be accelerated without sacrificing write performance.

To better understand why big alterations of today's parallel filesystems are needed, in the rest of this chapter we look at the components that make up today's HPC platforms and how HPC applications run on these platforms. We then explain why conventional parallel filesystem designs and data processing techniques are unable to keep up with the growing

scale of today's scientific computation. Finally, we list contributions of this dissertation and present its outline.

1.1 Today's HPC Computers

Key Information

1. Modern HPC computers separate compute from storage, forming a disaggregated cluster architecture unlike many big data or cloud data centers which tend to co-locate compute and storage in the same node for cost-effective high bandwidth;

2. Today a distributed parallel filesystem service manages the storage of an HPC computer, providing a global, strongly consistent filesystem namespace that is shared by all applications in a computing cluster;

3. The best way to utilize an HPC computer is to keep all of its compute cores busy, which makes reducing global synchronization and increasing I/O efficiency on compute nodes important.

From the early Cray machines [1] in the 1970s to modern supercomputers, HPC systems are long known for their remarkable performance. Today, the world's fastest are capable of hundreds of thousands of teraflops [2]. These large computers are used for a variety of computationally intensive tasks and play an important role in modern scientific discovery [3, 4].

To deliver high performance, modern HPC machines are built as massively parallel computing clusters [5]. A cluster consists primarily of compute nodes, with dedicated storage forming a disaggregated compute and storage architecture [6] as we show in Figure 1.1. Scientific applications run on compute nodes. A distributed parallel filesystem service [7–9] manages the storage. It provides a global filesystem namespace[1] through which applications can store and communicate information as named data files.

1: Here, global refers to the fact that the namespace is shared by all applications and can be accessed from any compute node in a computing cluster.

Files stored in the parallel filesystem remain in existence until their explicit destruction by a user command. On top of that, a data retention policy governs the maximum lifespan of data in the filesystem [10]. Once this maximum is reached, data is typically moved to a longer-term storage space for nearline or offline access [11, 12]. In this dissertation, we focus rather on managing files for online access.

One reason HPC separates compute from storage is that this can better isolates the failure domain of compute from that of storage [10, 13]. As a result, application jobs running on compute nodes are able to use checkpoint/restart [14, 15] to achieve fault tolerance while the underlying storage uses RAID [16] to recover from disk failures [17].

Decoupling compute from storage also improves system manageability. This is done by allowing storage nodes to have a separate network and to run a different operating system than that of those compute nodes. As a result, site administrators are able to launch maintenance operations on storage nodes without impacting user jobs [18]. At the same time, it is much simpler for multiple computing clusters to share storage, as the example we show in Figure 1.2.

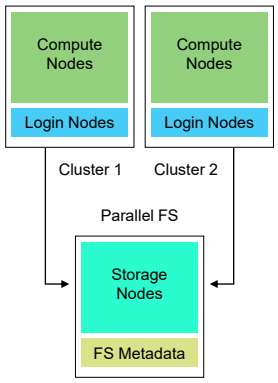


Figure 1.2: An example where two computing clusters share a parallel filesystem.

Today, the parallel filesystem that manages the storage consists of three key components: a) filesystem clients that run on compute nodes, b) object storage servers that run on storage nodes, and c) filesystem metadata servers that run on dedicated filesystem metadata server nodes [19–21]. A filesystem manages a collection of files. These files are organized into directories [22, 23], forming a hierarchical tree-like structure as Figure 1.3 shows that is known as the filesystem's namespace. Each file created in the filesystem has a record stored in a metadata server for information of the file. A filesystem may use one or more metadata servers. When multiple metadata servers are used, each metadata server typically manages a partition of the filesystem's namespace [24–27].

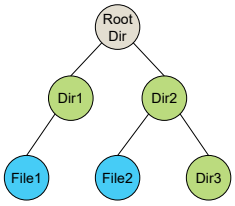


Figure 1.3: An example filesystem namespace consisting of 4 directories (including 1 special root directory) and 2 regular files. When multiple metadata servers serve a filesystem, they typically each manage a partition of the filesystem's namespace.

Unlike file metadata, file data is stored separately as one or more data objects in the filesystem's object storage servers [8, 28, 29]. To read or write a file, a filesystem client first communicates with a metadata server to obtain information on the file's data objects. The client then goes to the right object storage servers for data of the file. Modern parallel filesystems tend to be strongly consistent in metadata operations. Files created by one client in the filesystem are immediately visible to all clients in the computing cluster.

With today's HPC computers being massively parallel computing clusters, success in HPC depends heavily on utilization of massive concurrency. The best performance is achieved when work is spread across all available compute cores with as little global synchronization and as few I/O delays on compute nodes as possible. Unfortunately, by mandating a global filesystem namespace and dedicated servers to provide all services, existing HPC storage and parallel filesystem designs are preventing applications from

fully achieving this goal. This dissertation sets out to address this problem. We explain more in Section 1.3 and Section 1.4.

In addition to the compute, storage, and filesystem metadata server nodes, a small number of login nodes are provided per cluster for scientists to access the computing cluster from a remote network. Scientists use these nodes to manage code, write scripts, and submit jobs as Section 1.2 now discusses.

1.2 Today's HPC Applications

Key Information
1. HPC consists primarily of non-interactive batch applications. They form a fixed number of workflow types;
2. HPC workflows tend to have much relaxed consistency requirements than an interactive application. However, it is for those interactive applications that today's distributed filesystem metadata is optimized;
3. Knowledge about HPC systems and their workflows can be leveraged to reduce filesystem metadata synchronization and serialization and to enable more efficient use of client compute resources for higher, more scalable performance.

While it may be taken for granted that modern parallel filesystems were designed, from day one, to be optimal for the scientific applications that we run on HPC platforms, this was not entirely true. Parallel filesystems are defined by their concurrent access to file data [8, 28, 29]. But for metadata, modern parallel filesystems remain a serial service: all metadata operations are globally serialized [7, 9], and then written to a single write-ahead log. Global serialization results in a strongly-consistent filesystem namespace. It ensures that all clients see a single filesystem history and that updates made by one client are immediately visible to all clients. However, global serialization was not developed with today's massively-parallel computing environments in mind. Instead, it was largely a legacy from the early filesystems that served single core machines 50 years ago when the most important application for the filesystem was to enable a small group of computer scientists to interactively edit their text files in a shared programming environment [23]. Modern HPC platforms feature far more

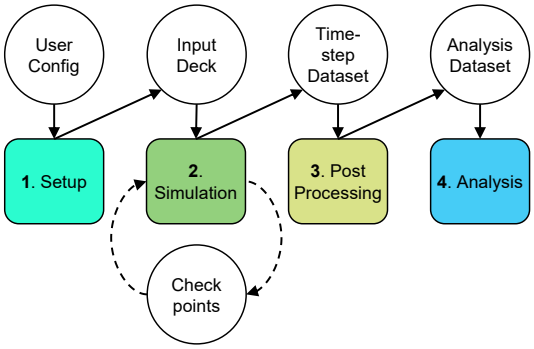


Figure 1.4: Illustration of a typical scientific simulation workflow. It starts with setting up the simultation's input deck according to user configuration. This is then followed by the actual simulation which reads the input deck for initialization and then runs in timesteps resulting in a series of timestep dumps. Before analysis, these timestep dumps are down-sampled and post-processed to form a separate analysis dataset. Scientists then run queries against this analysis dataset to obtain insights. Large-scale simulations may require the use of an entire cluster machine. For fault tolerance, simulations periodically write checkpoints of their in-memory state for failure recovery.

than 1 cores and has a significantly larger number of concurrent users. At the same time, modern scientific applications use filesystems much differently than human editing and sharing their text files. In this section, we show how scientific applications run on modern HPC platforms. In the next section, we show why radical changes of today's parallel filesystems are needed for future larger scale scientific computation and data processing.

Scientific applications are typically run by scientists as part of a larger workflow [13]. Before being able to run workflows, a scientist must be associated with an allocation, which grants the scientist permissions to access a computing cluster and the cluster's shared underlying filesystem.

To submit a workflow to run in a computing cluster, a scientist is expected to prepare a batch script specifying the programs and the amount of compute resources needed to run the workflow. The scientist then submits the batch script to a job queue. A workload manager monitors the queue and is responsible for launching the job when resources become available.

Multiple independent workflows may run in a computing cluster simultaneously. In such cases, each workflow typically owns a disjoint set of compute nodes in the computing cluster.

Based on their execution patterns, we categorize today's scientific workflows into: (1) simulations, (2) high-throughput computing workflows, and (3) uncertainty quantification workflows. Understanding how these workflows operate helps us reassess, from a modern scientific application's perspective, what is important and what is not for a parallel filesystem, and provides

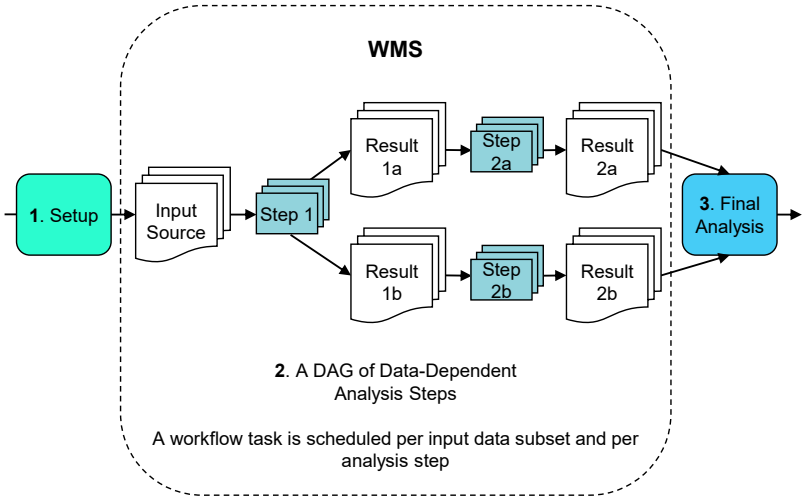


Figure 1.5: Illustration of an example high-throughput computing (HTC) workflow characterized by running as many as tasks as possible in parallel according to a directed acyclic analysis graph under the coordination of Workflow Management Software (WMS). Such a workflow typically starts with setting up an input data source and then initializing a workflow manager for scheduling tasks within the workflow. Input data for the workflow typically comes from an external scientific experiment or instrument which periodically sends data (an image, a chemical molecule, a genome segment) for HTC. Each data subset is processed through a DAG of analysis steps. A task is scheduled whenever a step needs to be invoked on a subset of data. A final analysis stage aggregates results from all data subsets and generates the final analysis output.

us with insights for more scalable filesystem designs in the context of high-performance scientific computing. We start with simulations.

Simulation Workflows are an important form of scientific computing. They are marked by using up to an entire computing cluster to run a single, long-running simulation through a parallel program. Typically, a multi-dimensional mesh is simulated. Each simulation process manages a disjoint mesh space. Simulations run in timesteps. Every few timesteps the simulation stops and the current simulation state is saved to storage. A simulation ends with a series of timestep dumps, which are then queried by the scientists for insights. To ensure high query performance, real-world simulations are often followed by a post-processing step that transforms the simulation timestep data to a read-optimized format for fast queries. Figure 1.4 shows the overall workflow process. For fault tolerance, simulations periodically write checkpoints of their in-memory state for failure recovery. **The job of a parallel filesystem is to** quickly absorb the timestep and the checkpoint data — potentially in the form of a massive amount of files — that the simulation generates during a run and to facilitate fast transformation of simulation timestep data for efficient post-simulation data analysis (post-analysis).

High-Throughput Computing (HTC) Workflows are another important form of scientific computing. As we show in Figure 1.5, they are marked by running a large number of small tasks in parallel to apply

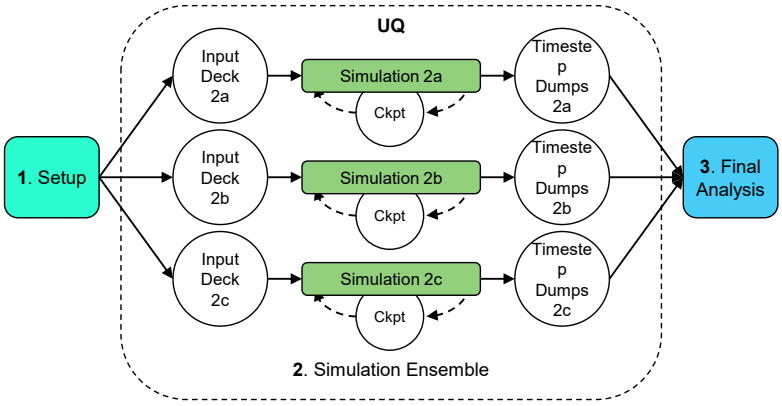


Figure 1.6: Illustration of an example Uncertainty Quantification (UQ) workflow characterized by running an ensemble of N simulations ($N = 3$ in this example) with each simulation using a slightly different input deck. Simulations run in the same way as we showed in Figure 1.4 within an ensemble. An ensemble management tool is used for simulation instantiation and failure recovery. Multiple member simulations may run as a single parallel program or as separate programs depending on UQ software implementation and resource availability. A final analysis stage collects results from all member simulations and performs the final analysis computation.

a Directed Acyclic Graph (DAG) of analysis steps to a certain input data source. This input data source is typically an external scientific experiment or instrument which periodically generates data (an image, a chemical molecule, a genome segment) for HTC. Typically, a workflow task is scheduled whenever an analysis step needs to be invoked on a data subset. A final analysis stage concludes the workflow by aggregating results from all data subsets and computing the final output. Unlike simulation workflows that use checkpoints, HTC achieves fault tolerance typically by rerunning a task when it fails. A workflow manager is launched at the beginning of a workflow for task scheduling and error handling. **The job of a parallel filesystem is to** facilitate massively parallel data processing by minimizing the synchronization and serialization among workflow tasks and to provide fast data access within each task.

Uncertainty Quantification (UQ) can be viewed as a combination of HTC and simulation workflows. As Figure 1.6 shows, UQ workflows are marked by running an ensemble of simulations in parallel for parametric analysis. A final analysis stage ends a workflow. It combines results from all ensemble members and produces the final analysis output. Depending on UQ software implementation and resource availability, multiple UQ simulations may run as a single parallel program or as separate programs. **The job of a parallel filesystem is to** enable fast data capture for each member simulation and to facilitate fast transformation of simulation timestep data for efficient post ensemble data analysis.

Based on knowledge of how today's HPC workflows run, we consider the following workflow properties to be important when rethinking parallel

filesystems for modern scientific computing as we now discuss.

Disjoint Access. Different workflows tend to access disjoint sets of data and do not usually communicate with each other. This indicates the possibility of using physically separated filesystems to serve different workflows to minimize interference while maximizing performance.

Sequential Sharing. Within a workflow, data tends to be shared sequentially across different workflow steps. That is, typically, one workflow step writes a dataset, closes all the files, and ends. Then, another followup workflow step starts, opens all the files, and reads the dataset. This indicates the possibility of decreasing the frequency of filesystem synchronization and serialization. For example, a filesystem may defer applying filesystem namespace changes in the form of deep client logging until these changes are accessed by a followup reader program which then causes all these changes to be bulk applied.

Mostly Self-Coordinated. Within a workflow step, the processes tend to know each other and do not usually require the filesystem to achieve synchronization. For example, the processes of a parallel simulation job may be programmed to mechanically generate unique filenames beneath a parent directory and therefore do not require the filesystem to synchronously act as an arbitrator for settling competing file creates. This indicates the possibility of delaying filesystem integrity checks for less synchronization and improved parallelism while not risking losing a potentially large amount of application work.

Known Readers. Scientific workflows tend to conclude with an analysis step that runs queries against the data generated by previous steps. In many scenarios, while these queries may be launched in an ad-hoc manner, they tend to access data in a pattern that is known to the early steps of a workflow. This indicates the possibility of combining data capturing and post-processing into a single step in early stages of a workflow so that analyzing data no longer first requires reading back data for post-processing. As an example, a filesystem may be configured to dynamically transform data to a read-optimized format utilizing the compute power at the client side when a workflow step writes data for subsequent analysis.

Having showed the components that make up today's HPC platforms, reviewed how modern scientific workflows run on these platforms, and

discussed what parallel filesystems could look like from a modern scientific application's point of view, in the next section we explain why radical changes of today's parallel filesystems are needed and how modern parallel filesystems and modern HPC storage can be re-imagined for future larger scale scientific applications and computing environments.

1.3 Drivers for Changes

The need for immense and rapidly increasing scale in scientific computation drives the need for rapidly increasing scale in storage for scientific data processing. Three factors motivate our re-invention of modern parallel filesystems and modern HPC storage for more scalable performance: (1) the high cost of global serialization for strong consistency in modern HPC environments, (2) the incompetence of the current state-of-the-art for scalable query acceleration on small data, and (3) the severe performance bottleneck caused by today's parallel filesystem servers.

Global Serialization. Scientific applications achieve the best performance when they spend most of their compute time performing computation. However, communication in storage systems is preventing applications from achieving this goal. Modern parallel filesystems such as Lustre [30] and GPFS [7] ensure strong consistency in filesystem namespace updates. They do so by using distributed locking [31] to maintain global synchronization at all times. That is, even though modern HPC platforms are frequently made up of hundreds of thousands of processor cores [32], every filesystem namespace mutation invoked by any process in a computing cluster will be globally synchronized and serialized with respect to all namespace mutations invoked by any other process in that cluster. This is too costly, and often unnecessary [33, 34]. Modern database research has declared this to be unscalable without relaxing transactional properties and requiring applications to understand and limit their transactions to the physical partitioning of a database [35]. As we further increase the size of our computers [36–38], it is imperative for parallel filesystems to be able to take advantage of knowledge from scientific applications to decrease the frequency and scope of metadata synchronization and enable performance that would otherwise be impossible due to global serialization.

Limitation of the Current State-of-the-Art. Analysis of large-scale simulation output is a core element of scientific inquiry. Over the years, advances in storage and network hardware and systems software have been instrumental in mitigating the effect of I/O bottlenecks in HPC environments. Still, many scientific applications that deal with small data or small files are limited by the ability of both the hardware and the software to handle such workloads efficiently. This problem will be exacerbated with exascale computers, which will allow scientific applications to run simulations at even larger scales and finer granularity levels. Worse, scientific data is typically persisted out of order, creating the need to budget time and resources for a costly, massive sorting operation in the form of post processing to speed up the queries at the final data analysis stage of a scientific workflow [13, 39–41]. As the gap between compute and I/O will further widen in future HPC environments [10, 42, 43], the cost of reading back data — especially from a slow storage tier — for sorting will only increase. It is therefore crucial to be able to speed up queries in new ways that minimize readback and that better help scientists leverage the massive amounts of small information they generate to derive insights.

Filesystem Server Bottlenecks. Modern parallel filesystems are made up of dedicated storage servers that handle data operations and dedicated metadata servers for filesystem metadata processing [8, 19]. Unfortunately, today these dedicated servers have become increasingly a performance bottleneck. First, due to limited compute and memory resources on dedicated parallel filesystem storage servers, complex storage software stacks providing data compression, indexing, and analytics have been increasingly unable to utilize all of the underlying device performance on emerging HPC systems. At the same time, dedicated parallel filesystem metadata servers have been increasingly unable to promptly absorb the bursty metadata load created by today's metadata-intensive scientific applications [42, 44–46]. It is possible to hide the processing delay associated with these dedicated servers through asynchronous processing [47–49]. However, in cases where the data needs to be immediately available for queries or when the data must be stable on disk prior to application completion, the time required for these operations will have to be amortized immediately. When the server CPU cycles and memory resources are insufficient for the demand of the incoming workload, it is then critical for parallel filesystems to find a way to leverage

the processing power and bandwidth of other parts of the system to make up the difference, more effectively shielding applications from the bottleneck and resource limitations of their dedicated servers.

When reimagining parallel filesystems and HPC storage in general, the need for a deep relaxation of global serialization for parallel filesystem metadata management, for new ways of accelerating small data queries, and for a more drastic decoupling of scientific application from the underlying filesystem server bottlenecks can be combined. First, parallel filesystems can be provided as a lightweight storage service that is dynamically instantiated on the compute nodes of a scientific workflow, taking advantage of the workflow's own initiative to flexibly assign compute and bandwidth resources for high-performance filesystem metadata processing and query acceleration while fully utilizing the underlying storage media to prevent bottlenecks. At the same time, scientific workflows can leverage a more lightweight filesystem to more efficiently orchestrate their data and metadata activities, streaming packing and cataloging their data for fast subsequent inquiry with minimum readback while performing only the filesystem namespace synchronization and serialization that are needed for the task at hand.

1.4 Summary of Work and Contributions

Thesis Statement
Parallel filesystems can keep up with the rapidly increasing scale of modern scientific computation when they (a) decrease the frequency of metadata synchronization according to application knowledge, (b) stop integrating all namespace changes into a single, globally shared filesystem namespace, and (c) re-invent themselves as non-dedicated client services that are dynamically instantiated on job compute nodes for scalable streaming data processing and filesystem namespace management on top of a shared underlying object store.

To defend this statement, this thesis presents the following pieces of work.

We show that LSM-Trees can be leveraged to enable efficient logging and deferring of client namespace changes for scalable filesystem metadata write performance. Prior work has showed the effectiveness of LSM-Trees in parallel filesystem metadata management at the servers [26, 50, 51].

In this thesis, we extend the use of LSM-Trees to client-side metadata processing for fast metadata writeback caching and bulk insertion. Clients execute metadata operations and log the resulting namespace mutations in LSM-Tree table files stored in shared underlying storage for writeback caching. We show that these client logged mutations can be efficiently integrated with one big log append operation — that we call bulk insertion — at the server for scalable metadata write performance. Bulk insertion requires all clients and servers to agree on the format of their tables and there is a way to verify delayed changes and a way to resolve conflicts. Verification of delayed changes may be unneeded if changes are delayed with locks and filesystem integrity is checked synchronously (Chapter 2). However, the best performance is achieved when changes are delayed without locks (Chapter 3) and the filesystem no longer requires all changes to be integrated into a single, globally shared filesystem namespace — applications merge changes only when they need to (Chapter 4).

We show that the relaxed consistency requirements of modern scientific workflows can be used to defer metadata synchronization for increased parallelism. Modern parallel filesystems are designed for a worst-case scenario in which applications rely on a strongly consistent filesystem to achieve synchronization. However, this is often overkill for HPC applications, which tend to seek synchronization out side of the filesystem and may not need to observe each other's namespace updates immediately. In this thesis, we show that it is possible for parallel filesystems to take advantage of this property by not immediately integrating every filesystem metadata mutation as soon as a client executes it. Instead, they defer it through deep client metadata writeback caching and bulk insertion as informed by application programs. While we see significant performance gains when deferred metadata synchronization is restricted to locked empty subtrees (Chapter 2), this thesis explores bolder designs in which metadata synchronization is deferred through logging against coarse-grained public filesystem snapshots for maximum performance and parallelism. To efficiently merge delayed changes, we discuss using client-constructed correctness proofs to speed up the verification process during client metadata bulk insertion (Chapter 3). However, the ultimate reduction of parallel filesystem metadata synchronization comes from a complete relaxation of global serialization, in which client logged mutations are only on-demand merged at the beginning of a scientific pipeline with unrelated pipelines never having to communicate — their metadata synchronization is deferred forever (Chapter 4).

We show that non-dedicated client compute resources can be utilized for scalable absorption of bursty filesystem metadata workloads. Modern parallel filesystem metadata performance requires dynamically partitioning the filesystem namespace over multiple filesystem metadata servers to achieve scalability [26, 27]. But even with a scalable namespace partitioning scheme, potentially a significant number of dedicated metadata servers may be needed in order for the system to be ready for an envisioned peak metadata demand. Performing such planning and calculation of in modern HPC environments is becoming increasingly inconvenient. In this thesis, we explore the use of non-dedicated client compute resources to scale parallel filesystem metadata performance by dynamically offloading filesystem metadata processing functions to clients. The simplest form of such offloading is deep client metadata writeback caching, in which a filesystem client process obtains capability to execute filesystem metadata operations and to format changes as metadata mutation logs (Chapter 2 and Chapter 3). When delayed client changes must be merged, it is possible to extend the use of client compute power to the pre-computation of correctness proofs for fast server verification of delayed client changes and bottleneck prevention at dedicated metadata servers (Chapter 3). Nevertheless, the potential of client offloading is best observed when the whole parallel filesystem is reduced to a simple object store that serves the filesystem's data plane, with the rest of the filesystem — the filesystem's metadata plane — entirely provided as client services dynamically instantiated on application compute nodes and providing only the namespace synchronization and serialization that are needed for the task at hand — such a design completely decouples application metadata performance from the performance of the underlying storage system in handling metadata-intensive workloads (Chapter 4).

Finally, we show that knowledge of scientific workflows can be taken advantage of to process data early in a scientific pipeline for fast post-hoc data analysis without requiring post data processing. Scientific applications perform data analysis by writing data to storage and then running queries against the data [13]. As data is not necessarily written in a format that is optimized for the queries, a post-processing step is often inserted after the writes, transforming data to a read-optimized format before analysis takes place. Post-processing can significantly improve query performance, but the need to read back a potentially massive amount of data from storage is rendering it increasingly prohibitive as data size grows. While the high cost of post-processing is not new to the HPC community, in this thesis we show a new way of addressing this drawback — Indexed Massive Directories (IMDs) — in which the storage writeback buffers and

the CPU cycles available on the compute nodes of the writer application are harvested for streaming data processing, which dynamically transforms data to a read-optimized format as the writer application writes it to storage: no post-processing is then needed as long as the data access pattern of the queries is known at data write time (Chapter 5). IMDs demonstrate a new way of providing data acceleration capabilities in modern HPC environments. Unlike emerging storage designs in which dedicated compute resources near data at rest are leveraged to speed up data analysis, in IMDs the acceleration takes place at the source of a data pipeline in a streaming manner and available compute resources on the main computing platform of an HPC cluster are harvested to perform the data acceleration computation.

The invention of IMDs completes **our repicturing of the future HPC storage landscape**: a plain object store serving as shared underlying storage and on top of it dynamically instantiated client services providing rich filesystem and data acceleration functions taking advantage of application knowledge and scalable client compute resources to unlock performance that is unattainable with today's monolithic storage solutions.

This thesis makes the following contributions.

- A new form of parallel filesystem client metadata writeback caching in which appendable LSM-Tree data structures are leveraged for efficient logging and deferring of client namespace changes beneath a locked empty subtree (Chapter 2);
- A client-funded[2] parallel filesystem metadata design, BatchFS, in which application compute resources are leveraged for massively parallel filesystem metadata processing on top of immutable filesystem snapshots with client namespace changes optimistically logged for deferred verification upon job completion (Chapter 3);
- The use of client-constructed correctness proofs for speeding up server verification of delayed client namespace changes (Chapter 3);
- The use of dynamically instantiated VMs as auxiliary metadata servers for scalable handling of delayed client changes preventing bottlenecks at dedicated metadata servers (Chapter 3);
- A no-ground-truth parallel filesystem metadata design, DeltaFS, in

2: In the form of compute resources.

which application filesystem namespaces are dynamically composed on an as-needed basis with unrelated applications never having to communicate and being forced to undergo unnecessary global serialization (Chapter 4);

- The reduction of parallel filesystem metadata to only client middleware that is dynamically instantiated on top of a shared underlying object store for scalable performance beyond a fixed set of dedicated server machines and a full decoupling of application metadata performance from the performance of the underlying storage platform in handling metadata-intensive workloads (Chapter 4);

- The use of filesystem snapshot registries and Internet-style search engines as a new paradigm for sequential data sharing among related scientific applications on HPC platforms (Chapter 4);

- A novel query acceleration mechanism, Indexed Massive Directories (IMDs), in which available compute and storage writeback buffer resources on job compute nodes are harvested for stream processing, dynamically reorganizing data to a query-optimized format as an application writes it to storage (Chapter 5);

- The use of aggressive data partitioning and filtering as a write-once[3], readback-free alternative to popular data indexing mechanisms such as LSM-Trees for sequential-write, random-reads problems (Chapter 5);

3: No write amplification.

1.5 Thesis Outline

The rest of this thesis is structured as follows.

In **Chapter 2**, we review our prior work, IndexFS, and propose a new client metadata writeback caching and bulk insertion mechanism that takes advantage of the log-structured metadata representation of IndexFS to enable efficient logging and deferring of client namespace changes. Our work on IndexFS extends the benefits of a log-structured filesystem to client-side metadata processing and demonstrates the importance of decreasing the frequency of metadata synchronization and serialization in parallel filesystems for scalable performance.

In **Chapter 3**, we study a more aggressive form of logging and deferring — BatchFS — in which application clients obtain capabilities to locally execute filesystem metadata operations and to log changes in shared storage by attaching to a filesystem snapshot at program startup and later bulk inserting all their changes at program completion for deferred verification and integration. While BatchFS strives to handle delayed changes in a scalable and efficient manner, its development exemplifies the inevitable cost of maintaining even an asynchronously updated global filesystem namespace in a large computing cluster. This leads us to develop a parallel filesystem that simply does not use a global filesystem namespace and is completely free of global serialization.

In **Chapter 4**, we present a no ground truth parallel filesystem design — DeltaFS — that does not require global serialization and does not use any dedicated metadata servers. Instead, a parallel job instantiates a filesystem namespace service in client middleware that operates on only scalable object storage and communicates with other jobs by sharing or publishing namespace snapshots. DeltaFS demonstrates a new paradigm of providing filesystem namespace services on shared storage. While being an extreme design, DeltaFS maximizes the use of application knowledge and scalable client compute resources to attain high performance and best exposes the inherent concurrency in modern HPC environments.

In **Chapter 5**, we move our attention from parallel filesystem metadata management to the data path where scientific workflows write data for later analysis. We propose Indexed Massive Directories (IMDs) as a new way for scientists to speed up their post-hoc data analysis queries. IMDs take advantage of the available compute resources of a parallel scientific workflow stage and dynamically reorganize data for fast subsequent reads as data is written to storage. By making previously intractable problems possible [52], our work on IMDs is another evidence of the effectiveness of using application knowledge and scalable client compute resources to attain high performance in modern HPC environments. The combined creation of IndexFS [26], BatchFS [33], DeltaFS [34], and IMDs [53–56] constitutes the full picture of our re-invention of scalable HPC storage.

With these, we conclude and discuss future work in Chapter 6.

Client Metadata Bulk Insertion | 2

The evolution of distributed filesystem metadata has been a process of decoupling. In this thesis, our journey of decoupling starts with extending today's client metadata writeback caching mechanisms to take advantage of the log-structured filesystem metadata representation and bulk insertion capabilities in emerging filesystem metadata designs. Our approach, named Client Metadata Bulk Insertion, enables a set of lock-holding filesystem clients to securely log namespace changes in shared storage and then bulk apply all their changes in one big log append step. Client Metadata Bulk Insertion drastically decreases the frequency of filesystem metadata synchronization and serialization, enabling a filesystem to quickly absorb a large amount filesystem metadata writes with minimum application communication. At the same time, the development of this technique lays the groundwork for more aggressive levels of logging and deferring for parallel filesystem metadata management which we explore in later chapters of this thesis.

The rest of this chapter is structured as follows. In Section 2.1, we discuss the salient features of our prior work, IndexFS, upon which our client metadata bulk insertion technique is built. In Section 2.2, we present the design of our technique. Section 2.3 reports experimental results. We show related work in Section 2.4 and summarize in Section 2.5.

2.1 Background: The IndexFS Filesystem

IndexFS [26] is a scalable parallel filesystem whose metadata is partitioned for load balancing across multiple servers. Our client metadata bulk insertion technique is based upon the IndexFS design to take advantage of its log-structured filesystem metadata representation and its metadata bulk insertion capability. To manage metadata information, IndexFS uses a log format derived from TableFS [51] in which each file or directory has an associated row in a table constructed with the LevelDB [57] realization of a Log-Structured Merge (LSM) Tree [58]. A modified LevelDB is used to

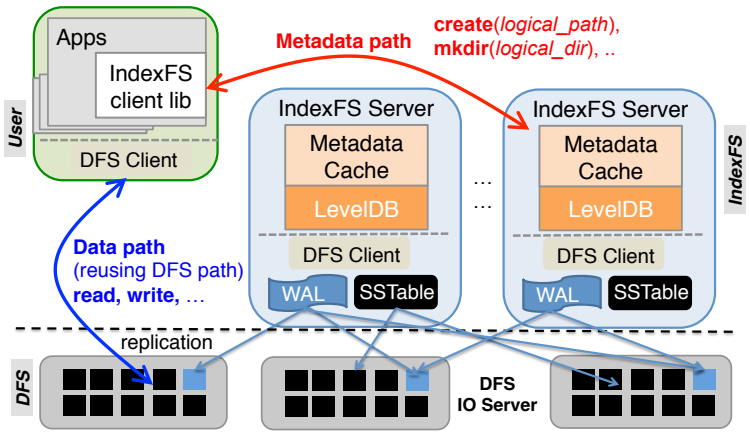


Figure 2.1: IndexFS is middleware layered on top of an existing cluster filesystem deployment to improve metadata and small file operation efficiency. It reuses the data path of the underlying filesystem and packs directory entries, file attributes and small file data into large immutable files that are stored in the underlying filesystem.

enable direct insertion of native LevelDB table files to a LevelDB instance. In this section, we present an overview of this scalable filesystem.

A Server Middleware Design. IndexFS is middleware inserted into existing deployments of parallel filesystems to improve the metadata efficiency of the base filesystem while maintaining high I/O bandwidth for data transfers. Figure 2.1 depicts the overall design of the filesystem. IndexFS uses a client-server architecture.

IndexFS Clients. Applications interact with IndexFS through a library directly linked into the application, through the FUSE user-level filesystem, or through a module in a common library such as MPI-IO [59]. The client-side code redirects application 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 filesystem to take full advantage of data I/O bandwidth. A newly created, but growing file may be transparently reopened in the underlying filesystem by the client module. While a large file is reopened in the underlying filesystem 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 Servers. IndexFS employs a layered architecture derived from TableFS [51] and similar to BigTable [60]. Each server manages a non-overlapping portion of filesystem metadata, and packs metadata and small file data into large flat files stored in the underlying shared distributed filesystem.

Filesystem 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+ [61] 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 LSM-Tree [58]. 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 filesystem to replicate LSM-Tree's data files and write-ahead logs to achieve fault tolerance.

Dynamic Namespace Partitioning. IndexFS uses a dynamic namespace partitioning policy to distribute both directories and directory entries across all metadata servers. Unlike previous works [24, 25] that partition file system namespace based on a collection of directories that form a subtree, our namespace partitioning works at the directory subset granularity. Figure 2.2 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 [62]) 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 [63] 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. To reduce the number of probes, the metadata server can cache the number of directories store on each server, and update these numbers less

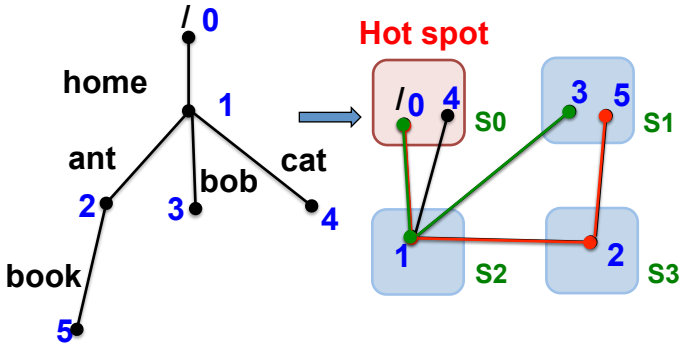


Figure 2.2: The figure shows how IndexFS distributes a file system directory tree evenly into four metadata servers. Path traversal makes some directories (e.g. root directory) more frequently accessed than others. Thus state-less directory caching is used to mitigate these hot spots.

frequently.

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 [61]. 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.

Log-Structured Metadata Representation. The IndexFS metadata storage backend implements a modified version of LSM-tree [58] 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) [60]. 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

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

Table 2.1: The schema of keys and values used by IndexFS.

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[64] 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 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.

Table Schema. Similar to our prior work in TableFS [51], 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 [51], 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 2.1. 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.

The log file stores all file and directory attributes such as i-node number, ownership, access mode, file size, timestamps (`struct stat` in Linux). For small files with size less than T (defaulting to 64KB), the file's data is also appended into the log file. For large files, only the symbolic link that points

to the actual file object in the underlying file system is kept in the log file.

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 2.3, 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. 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. To effectively integrate the dynamic namespace distribution mechanism with our on-disk metadata representation, we have modified LevelDB to support splitting and migrating directory partitions as required by GIGA+.

The file system metadata, including GIGA+ directory partitions and their directory entries are stored in LevelDB as a set of immutable files (SSTable

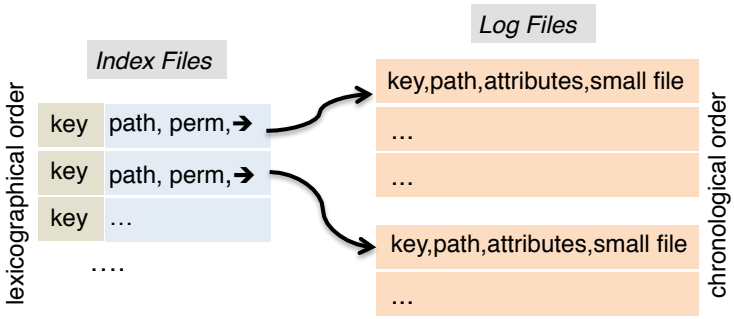


Figure 2.3: Column-style stores index and log tables separately. Index tables are kept in LevelDB which contains frequently accessed attributes for file lookups and the pointer to the location of full file metadata in the log file.

files) in a server specific directory in the underlying distributed file system. Each metadata indexing server process splits a large partition P on into itself and another hash partition P' which is managed by a different server; this split involves migrating approximately half the entries from old partition P to the new partition P' on another server. During splitting, the partition in migration is locked against client for simplification. We explored several ways to perform this cross-server partition split.

A straightforward solution would be to perform a range scan on partition P, and remove about half the entries (that will be migrated to the new partition P') from P. All removed entries would then be batched together and sent in a large RPC message to the server that will manage partition P'. The split receiver would insert each key in the batch into its own LevelDB instance. While simplicity of this solution makes it attractive, it is slow in practice and vulnerable to failures during splitting.

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 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.

2.2 Bulk Insertion Protocol

Motivation. 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 [44]. Inspired by the metadata client caching and bulk insertion techniques we used for directory splitting, IndexFS implements write back caching at the client for creation of currently non-existent directory subtrees. This technique may be viewed as an extension of Lustre's directory callbacks [30]. 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.

Protocol Design. 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.

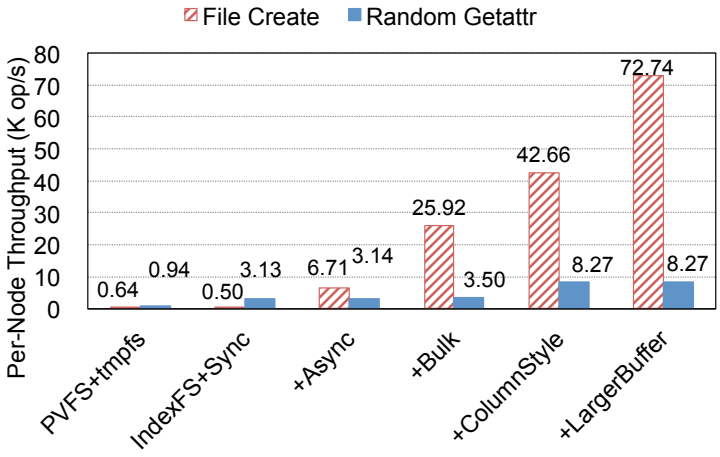


Figure 2.4: Contribution of optimizations to bulk insertion performance on top of PVFS. Optimizations are cumulative.

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.

IndexFS bulk insertion clients choose fault tolerance levels according to the needs of the task at hand. To reach the highest throughput, write ahead log can be disabled and all client-side SSTables can be stored in RAM. However, all local metadata will get lost if that application crashes. To get maximum durabilty, on the other hand, both write ahead log and SSTables should be persisted in the underlying cluster file system, which maintains replications for each single object.

2.3 Evaluation

Our 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 Baseline** 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. 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 2.4 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.

2.4 Related Work

In this section, prior work related to metadata services in modern distributed filesystems and optimization techniques for high-performance metadata is discussed.

Namespace Distribution. The Panasas filesystem [8] uses a coarse-grained namespace distribution by assigning a subtree (called a volume) to each

metadata server (called a direct blade). PVFS[29] is more fine-grained: it spreads different directories, even those in the same subtree, on different metadata servers. The Ceph FS [24] dynamically distributes the filesystem namespace based on server loads on collections of directories. The distributed directory service [25] in the Farsite filesystem [65] 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. The Giraffa filesystem [66] builds its metadata service on top of a distributed key value store, HBase [67]. 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 that IndexFS fixes. Lustre [30] mostly uses one special machine for all metadata, and is developing a distributed metadata implementation. The IBM GPFS filesystem [7] is a symmetric client-as-server filesystem 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 filesystems, including Panasas [8], Lustre [9], GPFS [7], and Ceph [24], clients employ a name space cache and attribute cache for *lookup* and *getattr* operations to speed up path traversal. Most distributed filesystems use cache coherent protocols with which parallel jobs in large systems suffer cache invalidation storms, causing Panasas 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 [30]. 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 filesystem instead of the client's local disk enabling failover to a remote metadata server. Another mode offered by Lustre and Panasas is to sometimes execute all metadata operations on the server side without any client cache, especially during highly concurrent accesses. Farsite [25] 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 filesystems have added support for distributing large directories, 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+ [61] to distribute large directories on several metadata servers [68]. Ceph uses an adaptive partitioning technique for distributing its metadata and directories on multiple metadata servers [24]. 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 [7]. 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 [61].

Metadata On-Disk Layout. A novel aspect of IndexFS 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 [69], Stratified B-trees [70], Fractal Trees [70], and VT-trees [71]. bLSM trees schedule compaction to bound latency variance on insertion operations. VT-trees [71] 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 [72], similar to TableFS [51], stores both filesystem metadata and data blocks into a fractal tree which utilizes additional on-disk indices called the fractal cascading index. Spyglass [73] partitions namespace metadata into smaller KD-trees that fit in RAM so that namespace metadata can be searched in multiple dimensions. However, their implementation avoids tree compaction, which is often required; comparing its performance against other write-optimized index is unclear.

Small Files Access Optimization. Previous work [74] 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 [75] 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 [76] 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 [77] 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 [67] use a similar approach to bulk load data.

2.5 Summary

While dynamic namespace partitioning enables IndexFS to scale beyond a single metadata server, the client metadata bulk insertion technique discussed in this chapter further boosts IndexFS's metadata performance by another order of magnitude. Our work on extending IndexFS shows the importance of decreasing the frequency of filesystem metadata synchronization and serialization for parallel filesystem metadata management. Our work also demonstrates the effectiveness of utilization of a log-structured metadata representation for new ways of client metadata writeback caching that are more efficient compared with the current state-of-the-art. The idea of deep metadata logging and deferring is further extended in later chapters of this thesis to enable more scalable parallel filesystem metadata designs, namely the client-funded filesystem metadata design of BatchFS and the no-ground-truth filesystem metadata design of DeltaFS, as we discuss in Chapter 3 and Chapter 4.

Relaxed Consistency and Client-Funded Filesystem Metadata

3

Modern parallel filesystems are designed for a worst-case scenario in which applications rely on a strongly consistent filesystem to achieve synchronization. However, this is often overkill for HPC applications, which tend to seek synchronization out side of the filesystem and may not need to observe each other's namespace updates immediately. In this chapter, we describe a client-funded filesystem metadata design, BatchFS, that optimistically assigns metadata processing capabilities to filesystem clients if they are willing to take advantage of a relaxed consistency model to reduce the frequency of global serialization and increase parallelism.

In BatchFS, a filesystem client obtains capability to handle its own filesystem metadata operations by entering into a special batch mode. Each batch client sees a snapshot of a filesystem. All filesystem metadata mutations executed by the client are optimistically logged for later verification and bulk insertion. To handle deferred verification in a scalable and secure manner, BatchFS clients construct proofs of the correctness of their mutations. Dynamically instantiated server VM processes validate these proofs and apply client mutations in one bulk insertion.

BatchFS extends on the client metadata bulk insertion technique we presented in Chapter 2. While our previous technique restricted the logging and deferring of namespace changes to newly created subtrees and locked out non-cooperating processes from seeing the newly created subtrees, BatchFS undertakes a bolder idea in which namespace synchronization is deferred without locks.

BatchFS provides important insights on new ways of providing filesystem metadata processing capabilities on modern HPC platforms. On the one hand, BatchFS demonstrates the effectiveness of utilizing application knowledge and non-dedicated client compute resources to scale filesystem metadata performance. On the other hand, BatchFS shows the inevitable cost of maintaining one globally serialized filesystem namespace in a large computing cluster. These insights encouraged us to develop deeper levels of decoupling that more completely free applications from the burden of

global filesystem metadata synchronization and serialization, which we show in Chapter 4.

The rest of this chapter is structured as follows. In Section 3.1, we show the motivation behind BatchFS. In Section 3.2, we present the BatchFS design. Section 3.3 reports experimental results. We show related work in Section 3.4 and summarize in Section 3.5.

3.1 Motivation

Filesystems define the interface between applications and the underlying storage provider. Unlike local filesystems that sit directly upon a block device, most parallel filesystems have adopted a layered architecture characterized by a dedicated metadata service layer backed by an object storage infrastructure serving as the persistence layer for both file data and filesystem metadata [19, 24]. With namespace management decoupled from the data path, filesystem clients are able to stream data directly through individual object storage servers in a scale-out manner, allowing these backend servers to better utilize available hardware resources and maximize I/O bandwidth. Unfortunately, since filesystem clients are required to perform namespace lookups and undergo semantic checks before they can access file contents through the data path, metadata intensive workloads can still bottleneck at the filesystem metadata layer, which is typically designed as a centralized metadata server for ease of implementation [20, 21].

To alleviate this bottleneck, modern filesystems have been rebuilding their control planes with multiple metadata servers [7, 8, 29] featuring techniques ranging from static namespace sharding or federation, to dynamic namespace partitioning. In addition, some of these filesystems are able to perform directory splitting under different heuristics, representing different design trade-offs and facilitating fine-grained parallelism on their metadata paths [26, 27, 61]. Notwithstanding a multi-server architecture, client-side metadata throughput can often be throttled by lock contention on common directories, server-side transaction ordering, and RPC overhead, let alone other common performance issues such as load imbalance and skewed access patterns. As exascale data centers are starting to emerge, it can become even more challenging to achieve a scale-out filesystem for metadata-hungry applications.

A filesystem control plane consisting of multiple dedicated metadata servers is limited by the maximum metadata performance that this number of machines is able to deliver[1]. In order to better accommodate metadata-heavy massive-scale parallel HPC workloads, we propose a client-driven filesystem metadata architecture that allows applications to handle their own metadata operations locally mostly without server intervention. Unlike existing filesystems that dedicate metadata server processes and machines to coordinate every metadata request in a centralized way, our filesystem can avoid inefficient RPC overheads and safeguards applications from unnecessary resource contention at the server side, effectively allowing the system to scale beyond a fixed sized control plane.

1: Metadata servers are generally allocated entire machines because their in-memory state will consume all available resources.

In addition, many filesystems are designed for a worst-case scenario where applications rely on a strongly consistent filesystem to synchronize with each other. However, this is often overkill for batch applications, which are usually carefully programmed to perform tasks cooperatively with little external coordination. Our client-driven filesystem design exploits this opportunity and uses a relaxed consistency model to manage batch application metadata.

We envision a better filesystem interface that can adapt to coordinated metadata access by providing options for applications to batch metadata operations and delay semantic checks, allowing cooperating clients to pay for synchronization only when it is indeed necessary. Applications obtain capabilities to locally process namespace operations by linking to a user-level library filesystem, which enjoys direct access to filesystem metadata represented as a set of log-structured table-based data structures[2] stored in a shared underlying persistence layer [26, 51, 72, 78, 79]. Each job operates upon a filesystem snapshot and self-manages its namespace. By reusing server-side logic, applications are able to encode metadata mutations directly into on-disk metadata representations (as well as write-ahead log entries). When synchronization is eventually needed, a client process can submit its modified filesystem image to the global master image in order to merge updates. Part of this design, known as metadata bulk insertion, along with the efficient underlying on-disk metadata representation, has been implemented in our previous work, IndexFS [26].

2: These data structures collectively represent a set of metadata mutation logs that are carefully grouped by common ACL settings in order to ensure high-level data privacy.

To secure global namespace semantics in a decentralized filesystem metadata layer, optimistic server-side namespace verification is used to establish a total order for all legitimate metadata operations bulk inserted by batch applications[3] In order for this to work in a scalable way, we introduce auxil-

3: Applications anticipating or experiencing substantial interference from other non-cooperating processes can fall back to synchronous (but potentially slow) access on dedicated metadata servers.

iary metadata servers, which are temporary daemon processes running on client resources with root privileges. Guarded by trusted virtual machines, these worker servers are responsible for verifying untrusted client-side filesystem mutations and committing them into the global master image. In addition, we envision an efficient mechanism for clients to pre-compute proofs of the correctness of their filesystem mutations. With these proofs, server-side namespace verification may be effectively simplified as a process of proof validation[4].

4: We focus on high-level namespace integrity, which is stronger than the low-level data integrity that is secured by the underlying storage infrastructure.

By reorganizing the filesystem control plane into a set of decoupled and client-based metadata processing endpoints, we envision a highly scalable filesystem metadata architecture, BatchFS, which extends our previous work, IndexFS [26]. IndexFS features a log-structured on-disk metadata representation and a simplified implementation of metadata bulk insertion. BatchFS makes the following contributions: 1) a re-designed filesystem control plane harnessing client resources to achieve scalability, 2) deferred namespace synchronization enabled by efficient filesystem snapshots and fast metadata bulk insertion, 3) optimistic server-side metadata verification enabled by client-generated proofs, and 4) weak filesystem semantics targeting batch (or cooperating) applications.

3.2 System Design

In general, interaction among a set of processes comes in two forms: 1) interaction among the processes of an integrated job or framework, and 2) interaction between unrelated jobs or systems, which, unfortunately, specifically use the file system as a synchronization platform. We refer to file system clients that demonstrate the former kind of interaction as batch clients, and the latter as interactive clients or non-cooperating clients.

Assumptions. In designing BatchFS, we have been guided by common metadata access patterns and execution environments shared by batch applications in HPC data centers.

- Batch workloads usually only access a pre-constructed set of files for input, and normally generate their output files in a way that deterministically avoids name conflicts.

- Batch workloads are characterized by a natural preference for high throughput as opposed to low latency. Few place stringent response time requirements on individual metadata requests, provided overall throughput is high enough.

- Files generated by batch jobs rarely get accessed until after these batch jobs have completed execution. There is a special case, however, for job owners monitoring output files in order to enable online user steering and fast detection of wasted resources; we will deal with this case separately.

- Batch jobs normally protect themselves against failures by stop-and-copy checkpointing, backing up their in-memory state into an underlying cluster file system before moving to the next stage. In this paper, we focus on one-file-per-process (N-N) checkpointing because one-file-per-job (N-1) checkpointing can be transformed into N-N checkpointing using a user-space translation layer such as PLFS [44].

- Most HPC data centers are equipped with dedicated storage infrastructures hosting large-scale cluster file systems, which are optimized for maximum data bandwidth but often lack a scalable metadata path [42].

Interfaces. BatchFS targets the standard POSIX API. Applications invoke BatchFS routines with BatchFS's client-side user library or indirectly through FUSE or MPI-IO [59]. Like its predecessor, IndexFS [26], BatchFS handles namespace related operations but redirects all data requests to a specific underlying storage infrastructure, which holds both file contents and namespace metadata images generated by BatchFS. Different from IndexFS, BatchFS allows its clients to dynamically select between different namespace synchronization modes. This, along with its new consistency model, is discussed further later in this section.

Architecture. A BatchFS cluster is organized as a metadata control plane layered on top of a scalable storage infrastructure serving as the underlying data plane. Rather than only having a fixed set of dedicated metadata servers like the original IndexFS, BatchFS's unique control plane features three different types of metadata processing engines: Primary Metadata Servers, Auxiliary Metadata Servers, and Private Metadata Servers, illustrated in

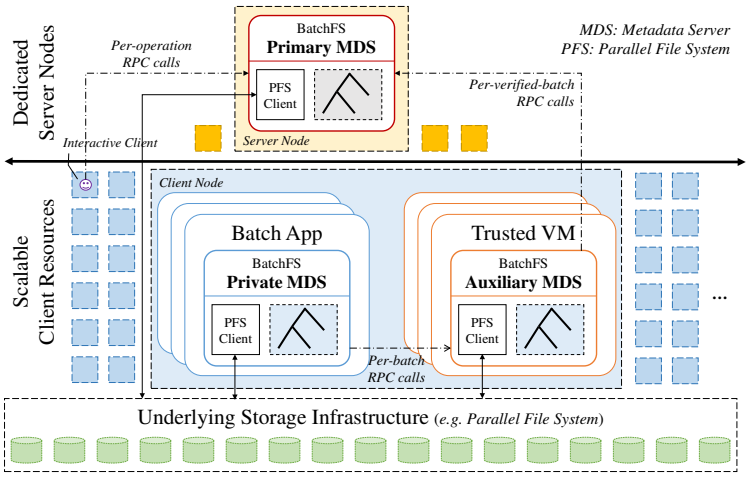


Figure 3.1: BatchFS is designed as file system metadata middleware layered on top of an existing cluster file system or an object storage platform exposing a flat namespace, which allows BatchFS to reuse the data path offered by these underlying storage substrates already optimized and tuned for maximum bandwidth. BatchFS features a client-driven metadata architecture that can shift server computation to client machines to achieve highly agile scalability.

Figure 3.1.

Primary metadata servers are dedicated servers running on dedicated server nodes. These non-scaling[5] servers collectively manage the master image of the file system, with each server hosting a non-overlapping set of directory partitions [26, 61]. Interactive clients communicate with primary metadata servers to update the namespace synchronously, obtaining a latest view of the global namespace, albeit without scalable performance.

5: While some file systems, like IndexFS, can add additional dedicated servers, this is a slow, physical provisioning action done by an administrator. We consider this non-scalable because it is so slow and unlikely to be triggered by the needs of specific jobs.

Both auxiliary and private metadata servers are designed to be temporary metadata processing entities that run on-demand using client resources. We call these Client-Funded Metadata Servers. BatchFS library code linked into each application constitutes a private metadata server, which can be viewed as a full-fledged but untrusted metadata server associated with a file system snapshot and subject to access control provided synchronously by the underlying scalable data plane. These embedded metadata servers enable batch clients to access and modify their private namespaces locally without contacting any primary metadata servers, until they want their local mutations to be visible to non-cooperating jobs.

Auxiliary metadata servers are trusted daemon processes responsible for merging client-side namespace mutations into the global master image. Outsourcing this potentially heavy computation to auxiliary metadata servers shields primary metadata servers from becoming a performance bottleneck, and leads to better utilization of the underlying system.

Metadata Representation. In IndexFS, the global file system namespace is represented by a set of large directory entry tables with embedded inodes and possibly embedded file contents as well. Each file is mapped to a unique row inside the table. When flushed to disk, each table will be transformed into a set of log-structured table-based data structures formatted as `SSTables` (Sorted String Table). SSTables [57] are immutable data containers that are partially ordered so that newer table entries always supersede older ones. SSTable serves as the physical format for metadata migration and bulk insertion [26, 51]. In addition, a tree of SSTables form a file system snapshot. BatchFS extends IndexFS and inherits its efficient metadata processing engine to support high-performance metadata mutation, migration, and bulk insertion.

As a new feature, BatchFS allows batch clients to request snapshots (collections of SSTables) of the current file system image, which they can use to establish a private copy of the snapshot namespace. SSTables no longer referenced in any file system snapshot can be safely discarded. In addition, files marked deleted in all SSTables can be purged from the underlying storage infrastructure without leaving null pointers.

To generate a snapshot for a requesting client, a primary metadata server performs an in-memory cache flush and sends the client a manifest that lists all SSTables comprising the current file system image. To prohibit a malicious client from accessing restricted data by scanning an entire file system snapshot inappropriately, BatchFS generates separate SSTables for each user-group combination so that every file referred in an SSTable has the same permission bits[6]. This way, high-level access control can be directly enforced by the underlying storage infrastructure. Note that this is designed for environments with simple ACL practices, which we expect in most HPC data centers. Similarly, user quota control is also synchronously enforced by the underlying storage infrastructure. For ease of implementation, quota management may only apply to file data, as the size of metadata is almost always dwarfed by the size of data in the file system as a whole.

6: This means a client with permission to read a directory must also be allowed to read the attributes (and embedded file data, if any) of the files under that directory.

Consistency Model. BatchFS features a relaxed consistency model targeting HPC applications. BatchFS clients associated with different snapshots of the file system can observe distinct namespaces. Interactive clients communicating with non-scaling primary metadata servers are kept in sync with the master image of the file system, which reflects all committed operations executed at the global namespace. Any individual metadata operation

accepted by a primary metadata server is immediately committed, with its consequence immediately becoming visible to all subsequent file system calls and snapshots. On the other hand, batch workloads associated with local private metadata servers are logically disconnected from primary metadata servers. In fact, each batch client operates within its own file system image that originates from a specific snapshot of the master image. Metadata operations accepted by a private metadata server are immediately executed in the private namespace albeit remain uncommitted and invisible to other parallel file system clients, until an auxiliary server validates and publishes these changes.

Initially, BatchFS clients act as interactive clients, transmitting every metadata operation to a primary metadata server for global execution. Batch applications wishing to exploit BatchFS's fast asynchronous metadata interface explicitly call BatchFS to establish their own private namespaces, converting themselves to `batch` execution mode (possibly for only a subtree of the overall namespace). BatchFS clients can switch back to the original `synchronous` execution mode whenever server coordination is needed. This entails closing the current private namespace, flushing changes to the underlying storage infrastructure, starting a new auxiliary metadata server, and asking it to merge the corresponding namespace mutations into the global file system image. We expect most batch applications to stay in batch execution mode until they complete a checkpoint or terminate.

Namespace Verification. To protect namespace integrity in the presence of untrusted clients, changes to a client namespace must be verified by an auxiliary metadata server before being inserted into the global namespace[7]. In BatchFS, any client namespace re-producible from a legal sequence of file system operations is considered a legitimate candidate for namespace merging. Unfortunately, in practice, it could be hard for an auxiliary metadata server to guess the valid operation sequence that happens to yield a given client namespace. To resolve this problem, BatchFS requires every client to submit, in addition to the resulting namespace changes, a proof of the correctness of its namespace mutations. As such, an auxiliary metadata server should be able to efficiently verify a client namespace by validating its associated proof.

7: This verification and commit process can be delayed (potentially forever) until the first non-cooperating access, or delayed verification may be processed in the background in order to reduce (eventual) access latency.

A proof can be easy to construct, but unnecessarily large, if it is provided directly as the original sequence of operations executed at the client side. With such a proof, an auxiliary metadata server can perform checks simply

by re-executing those operations and comparing results. To reduce the proof size, clients can reorder and merge commutative and associative file system operations. This leads to reduced verification computation, allowing client mutations to be checked faster. In the extreme, we envision a logic-based namespace certification process where clients provide formal proofs that their namespace changes respect the file system's semantics. An auxiliary metadata server could validate these proofs without re-executing any file system operations, without using cryptography, and without consulting any external trusted entities. Once the validation succeeds, the associated namespace can be safely trusted and accepted. We refer to this as self-provable metadata representation, inspired by similar techniques in other contexts [80, 81].

Optimistic Concurrency Control. To reconcile conflicts between global file system semantics and asynchronous namespace management, BatchFS employs optimistic concurrency control [82] commonly seen in database systems to order metadata operations. That is, clients do not do two-phase locking; they optimistically assume there will be no conflict. However, instead of using the generic notion of read/write set to verify its transactions, BatchFS applies file system semantics to verify namespace integrity and consistency. For example, a file creation can be reordered with a `chmod` if it is compatible with both the before and after permissions [83]. This resists a batch of file system operations from being unnecessarily rejected. In addition, different from a traditional database, BatchFS allows a batch of client-side namespace mutations to fail partially during the verification phase and does not necessarily roll back the whole transaction. This avoids a minor infraction from destroying a potentially large amount of work. Rejected namespace mutation notifications are available to users in external logs and associated files may be retained with mechanically modified names by BatchFS for users to resolve conflicts later.

Adaptation of Job Monitoring Programs. BatchFS's design is in part motivated by the compatibility of a relaxed consistency model with batch applications. However, weak consistency could introduce problems for job monitoring utilities, which normally rely on strongly consistent file systems to achieve live user feedback, such as progress indication and data preview, because they appear to be non-cooperating processes.

In practice, live feedback, progress indication in particular, can be estimated

via a proxy, such as the size of output files generated by a batch job, which can in turn simply be a stale number as opposed to an up-to-date value [84]. Based on this observation, BatchFS includes a separate metadata interface targeted at monitoring utilities. Designed with weak consistency in mind, this special interface provides applications with "metadata snapshots" (snapshots of file size for example) that do not necessarily represent a latest value but are designed to return newer versions under a sequence of repeated file system calls. With this interface, progress monitoring can be efficiently implemented without forcing batch jobs to perform unnecessary synchronization, thus ensuring high performance.

3.3 Evaluation

In this section, we report experiments done on our previous work, IndexFS [26], to show the promise of BatchFS's file system metadata architecture. As BatchFS's predecessor, IndexFS allows its clients to complete file creation operations locally if these files are known to be new, and will later bulk insert all these cached operations into the global namespace using a single SSTable insertion. This feature, a simple metadata bulk insertion, is extended by BatchFS to provide additional features such as ACL-specific file system snapshots, self-provable metadata representation, client-funded auxiliary metadata servers, and optimistic concurrency control. As a result, performance measurements of IndexFS showed in this section can be viewed as an optimistic projection of future BatchFS performance.

To show the efficiency of IndexFS's metadata bulk insertion, a consistent client-side metadata write-back cache, we ran our experiments with this cache mechanism either enabled or disabled. More specifically, we measured performance on an IndexFS cluster where IndexFS was layered upon HDFS [21]. It consisted of 8 HDFS data nodes and 1 HDFS metadata node (an HDFS name node) and was configured with four different settings, as is illustrated in Figure 3.2.

To model metadata-intensive applications, we used a synthetic micro-benchmark tool, mdtest [85], to insert zero-byte files into multiple newly created directories. We generated a two-phase workload. In the first phase, N clients each create one private directory and then insert files into that directory. In the second phase, each of these clients performs `getattr` in a random order on the files created under its own directory. In total, M

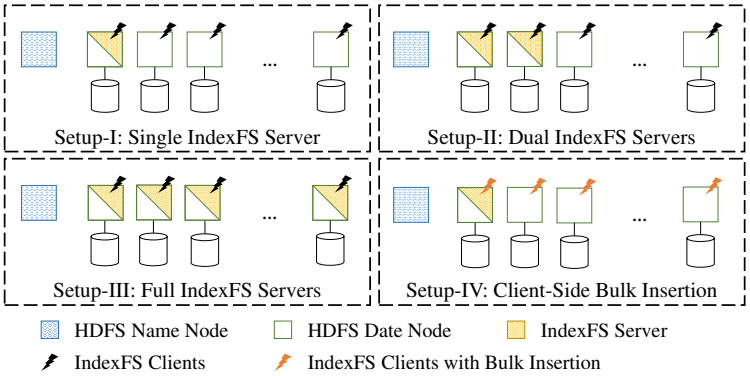


Figure 3.2: IndexFS with four different setups to model different amount of server resources. All our machines are from the NSF PRObE Kodiak cluster and are configured with dual 2.6 GHz AMD Opteron processors, 8 GB of memory, two 1 TB 7200 rpm SATA disks, and a 1000 Mbps Ethernet NIC, with each running 64-bit Ubuntu 12.04 upon Linux 3.2.16. Note that this configuration was set up for ease of testing, real clusters often use dedicated hardware for the storage infrastructure and their file system servers.

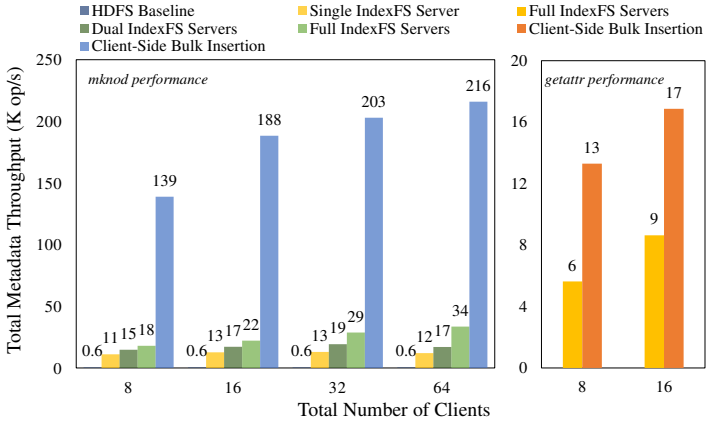


Figure 3.3: With bulk insertion, BatchFS's predecessor, IndexFS, is able to outperform HDFS by 360x and IndexFS without bulk insertion by as much as 18x. Bulk insertion also results in better metadata read performance, but metadata footprint is too large for 100% cache hits and `getattrs` may have to wait on HDFS data nodes.

million files will be created and stat'd, where M is the total number of IndexFS servers or bulk insertion clients. The total number of clients, N, varies from 8 to 64.

Our experimental results are illustrated in Figure 3.3. In general, with the presence of a client-side metadata write-back cache, clients are able to drive an 8x to 360x increase in file creation throughput, in addition to better metadata read performance. This can in part be attributed to the faster metadata path, the more aggressive usage of client resources, as well as a local in-memory metadata cache provided by their embedded metadata processing modules. Moreover, compared with synchronous metadata processing that must be coordinated by a set of centralized metadata servers, allowing clients to self-manage their private namespaces indeed leads to better utilization of the underlying hardware resources, since the same system is delivering much higher throughput when bulk insertion is activated.

3.4 Related Work

Serverless filesystems [7, 86] are often characterized by a set of symmetric file servers that are each capable of serving the whole filesystem namespace. This architecture is used by BatchFS primary and client-funded auxiliary metadata servers to achieve both high scalability and high efficiency. To improve write performance, PLFS [44] devised a library-based user-level index structure capable of shaping I/O access patterns and aggregating small files, which allows it to avoid performance bottlenecks at the underlying cluster filesystem. PLFS inspired our approach to improving speed by decoupling sharing where possible and showed us where read performance would be of concern. BatchFS's client-side metadata write-back cache can also be used to buffer small files. In addition, BatchFS's log-structured table-based on-disk metadata representation is general purpose and integrated with the filesystem.

RPC is an important but performance limiting component in distributed filesystems. Mercury [87] is an efficient RPC subsystem optimized for HPC data centers. It uses the traditional TCP/IP based network channel to send control messages but redirects bulk data to dedicated RDMA channels for fast transfer. BatchFS features a similar optimization mechanism that separates the filesystem control plane and data plane. In addition, BatchFS uses SSTables stored in the shared underlying storage infrastructure as a virtual communication channel to enable efficient metadata access and bulk insertion.

Modern HPC filesystems often use dedicated I/O nodes for integrated data buffering and I/O forwarding [88, 89]. These techniques are orthogonal and complementary to BatchFS, which could utilize them for its private, auxiliary, or primary metadata servers.

BatchFS employs optimistic concurrency control protocol [82] commonly seen in data-intensive and database systems to increase concurrency. However, BatchFS does not enforce transaction atomicity as it allows a batch of metadata operations to fail partially during verification without rollback. In addition, BatchFS, like other weakly consistent systems [90–92], allows its applications to later merge conflicts and retry specific operations if the initial verification was rejected by the server.

Optimistic concurrency control and conditional operations have been used

on file data in order to enable richer filesystem primitives for application coordination [93–95]. BatchFS is different from these techniques in that it focuses on metadata concurrency and targets batch applications that do not use the filesystem as a synchronization mechanism.

In order to secure auxiliary metadata servers, BatchFS relies on VMs to isolate these servers from adversary attacks. This can also be achieved through remote attestation, which is available in today's software distributions [96, 97] as well as commodity security co-processors such as industry standard Trusted Platform Modules. Finally, BatchFS's self-provable metadata representation is designed upon the observation that validating an answer is much simpler than solving the original problem [80, 81].

3.5 Summary

Metadata scalability is limited in today's HPC parallel filesystems because (1) it usually employs an RPC per operation; (2) its semantics require the status results of one operation to be known before the next is submitted; (3) its authorization enforcement requires dedicated (fixed number of) server machines; and (4) its durable representation of metadata is usually updated with random seeks for every mutation. With techniques for deferring and batching metadata operations, for reconciling concurrent weakly consistent mutations with optimistic concurrency control, for log-structured and indexed on-disk representation, for trusting code running in a user-contributed virtual machine, and for weakened semantics in high performance filesystems, BatchFS proposes to allow the cooperating processes of a single job to pre-execute all of their own metadata operations on a snapshot of the filesystem namespace, presenting all resulting mutations back to the central filesystem in as few as a single batch, and providing a "proof" of the correctness and authorization of these mutations for optimistic concurrency control and selective reconciliation as late as possible to maximize job independence and throughput. Moreover, BatchFS allows user jobs to allocate virtual machines running filesystem server code to scale metadata throughput with resources that local administrators would not want to dedicate to filesystem servers. To show the promise of this vision for scalable metadata services, BatchFS reports on a less expensive implementation of client-embedded metadata writeback caching, which shows 8x to 360x throughput improvement over today's common implementations based on a single or a few dedicated metadata servers.

A No Ground Truth Filesystem | 4

It is an idea universally acknowledged that a distributed parallel filesystem serving an exascale supercomputer must be in need of a large number of dedicated metadata servers for high metadata performance. In this chapter, we show that this need not be the case. Quite the opposite. We present a no ground truth parallel filesystem metadata design, DeltaFS, and argue that parallel filesystems scale better without dedicated metadata servers.

Dedicated metadata servers are inconvenient. A filesystem with dedicated metadata servers is limited by the maximum performance that that sum of machines are able to deliver. In addition, a cluster designer needs to determine the amount of compute resources to be dedicated to the filesystem's metadata beforehand and, to ensure high performance, a significant amount of resources may be needed.

DeltaFS uses no dedicated metadata servers. Instead, a parallel job instantiates a filesystem namespace service in client middleware that operates on only scalable object storage and communicates with other jobs by sharing or publishing namespace snapshots. DeltaFS scales better because it dynamically distributes filesystem metadata processing across job compute nodes rather than restricting it to dedicated metadata servers. Moreover, DeltaFS does not use the idea of a global namespace, so unrelated jobs never have to communicate.

4.1 Motivation

Three factors motivate our work: the limitations of today's distributed filesystem metadata techniques, the undue cost of today's filesystem metadata semantics, and the inflexibility of today's resource provisioning strategies for distributed filesystems.

Distributed filesystems today are known for their scalable access to file data [7, 9, 19, 29]. Scalable filesystem metadata access, on the other hand,

depends on dynamic namespace partitioning which spreads the processing of metadata across multiple filesystem metadata servers [8, 24–27, 61]. While being a crucial technique, dynamic namespace partitioning does not prevent globally serialized namespaces. Worse, in order for the filesystem to be ready for an envisioned peak metadata demand, dynamic namespace partitioning may require a large number of servers to be dedicated. This reduces the overall resource utilization of a computing cluster. It is possible to improve resource usage by utilizing a more efficient data structure to manage the metadata information of files as fewer servers will then be needed [26, 51, 72, 78, 79]. But even these efficient filesystem designs will not stop global serialization, and applications continue to experience bottlenecks.

To address the bottlenecks caused by global serialization and to avoid it for new ways of providing filesystem namespace services on distributed computing environments, we need to examine the filesystem metadata semantics that underlie today's distributed filesystems. First, distributed filesystems today use one filesystem namespace to store all files. Having one namespace simplifies data sharing. But for applications that do not communicate [13], storing their files in one namespace is largely unnecessary. In addition, modern filesystems tend to serialize namespace changes at the moment they occur. But in cases where files created by an application are left untouched until a later reader program that reads these files in a batch [10, 98], immediate serialization can be overkill. To avoid limiting application performance for features that are not necessarily helpful, in this paper we show a new paradigm for distributed filesystem metadata where filesystem namespaces are explicitly composed from namespace mutation logs written by previous application runs. This enables applications to control the scope and timing of data sharing and allows costly synchronization and serialization activities to be deferred until they are needed by the job at hand.

Explicitly composing namespaces for applications causes delays due to log compaction that an application must experience before it can access a namespace efficiently. To minimize such delays, having a high peak filesystem metadata performance that can quickly absorb these compaction operations becomes critical. Unfortunately, modern distributed filesystem metadata performance is coupled with dedicated resources. The peak metadata performance of a filesystem is limited by the total amount of computing power of its dedicated metadata servers [8, 24]. To enable higher levels of peak metadata performance, our solution is to have applications

independently allocate compute resources for metadata processing. This distributes resource provisioning decisions and the control and processing of metadata over individual applications in a computing cluster. Applications independently determine the amount of resources to be devoted to the filesystem and instantiate services to process metadata operations. In extreme cases, as much as an entire cluster can be utilized to scale filesystem metadata performance.

We propose DeltaFS, a new paradigm for managing distributed filesystem namespace information on shared storage. Rather than providing a single shared filesystem namespace service for all applications in a computing cluster, DeltaFS uses application knowledge to avoid unnecessary global serialization and to simply filesystem resource provisioning.

A DeltaFS computing cluster features no dedicated metadata servers. Application jobs independently instantiate filesystem namespace services on client nodes, and record namespace changes (such as the creation and deletion of files) as immutable namespace mutation logs stored in a shared underlying storage system. We refer to these shared logs as *delta*s. Deltas are log-structured. They can be chained together and merged to form namespace views that reflect changes made through a set of previous application runs. DeltaFS users use this mechanism to orchestrate data sharing for their applications, preventing unnecessary synchronization and serialization.

4.2 A Serverless Architecture

A DeltaFS computing cluster to have no filesystems as we know today [10, 12]. Instead, a shared scalable object store serves as a cluster's underlying storage [99]. Applications use only client middleware software to initialize and access their filesystem namespaces, and independently allocate computing resources for metadata processing. There are no dedicated filesystem metadata servers. Both file data and filesystem metadata are directly stored as compact log objects in the underlying object store. Data is not discarded after an application run and can be made accessible to followup jobs and other cluster users.

Pitfalls of Dedicated Servers. The current state-of-the-art distributed filesystem metadata is defined by client processes communicating with a set of dedicated filesystem metadata servers [8, 24]. Each filesystem metadata server manages a partition of the filesystem's namespace. The overall metadata performance of the filesystem can be viewed as a function of the number of servers and the amount of compute resources each server has.

The canonical way to improve filesystem metadata performance is to add more nodes to run as dedicated filesystem metadata servers. This approach works best when performance scales linearly with the number of servers dedicated. While scalable designs exist to enable dynamic namespace partitioning over large numbers of servers [8, 24–27, 61], we expect this approach to be increasingly inconvenient as future clusters enable higher levels of scale and performance.

First, as cluster size grows, the metadata demand of the cluster grows with it. Worse, with emerging computing clusters combining multiple compute and network technologies to meet performance, reliability, and cost goals, the new types of applications they enable can be far more metadata intensive than the ones we see today. To match the performance of these applications, potentially a significant portion of a computing cluster will have to be devoted to the filesystem to become dedicated metadata servers, which leaves fewer compute resources available for running user jobs. This reduces the effective capacity of the cluster.

Second, relying on dedicated resources to serve filesystem metadata requires knowing the right amount of resources to dedicate. But because not all applications of a cluster are metadata-intensive and metadata-intensive ones may not use the filesystem all the time, the amount of resources to be devoted to the filesystem can be hard to determine beforehand. This leads to bottlenecks when the actual filesystem metadata demand of the cluster is too high compared with the estimation, and a waste of resources when otherwise.

DeltaFS allows for more flexible resource provisioning for distributed filesystem metadata. This is because that in DeltaFS metadata is a service that is dynamically instantiated by applications on compute nodes, so the resource for metadata can be configured at a finer granularity and adjusted on a per-application basis. A metadata-intensive application may employ as much as an entire computing cluster to process metadata operations,

whereas a metadata-light application can allocate as few as a single CPU core to do so.

Formative Examples and Their Problems. The opposite of dedicating servers is to be completely serverless. In a serverless filesystem, there is no dedicated file managers [31, 86, 100, 101]. Instead, a cluster of client nodes collectively manage a filesystem namespace. They do so by storing the data and the metadata of the files in a shared underlying storage system [102]. Each node understands the filesystem's on-storage data structures and can dynamically assume responsibility for any file in the filesystem when the file is accessed by the node. To achieve synchronization, all nodes use locks [31, 103–105] to reach consensus on the current owners of the files in the filesystem. The current owner flushes all dirty information to the shared underlying storage before a new owner takes over. This approach contrasts with filesystems such as Ceph [24] and the Panasas Filesystem [8] where dedicated metadata managers each own a specific partition of the filesystem and all its associated data structures on storage.

Today, serverless designs are often found in Storage Area Network (SAN) filesystems [7, 106–109] serving enterprise applications. Scalability is frequently an issue due to the large amount of communication needed to achieve synchronization. To mitigate this problem, real world deployments typically limit cluster size to a small number (usually less than 10). For larger scale deployments, a small set of client nodes are often specialized as dedicated file managers. These dedicated managers then export the filesystem to the rest of the cluster which no longer act as managers of the filesystem [110, 111]. Notwithstanding many benefits, such deployment approaches largely defeat the goal of being serverless, and filesystem performance reverses back to being coupled with the amount of resources dedicated.

Our work can be viewed as a revival of this classic shared-storage approach for filesystem metadata. To more scalably distribute filesystem metadata computing capabilities across clients, our design decouples clients from each other. A client metadata manager no longer serves all jobs under a single cluster-wide filesystem namespace. Instead, jobs each launch their own metadata managers. Different jobs work on different per-job namespaces. Cross-job communication is done through shared storage with jobs independently reading and merging immutable namespace data published by previous jobs — no need to perform global synchronization

among all job processes all the time.

Decreasing Synchronization Frequency. Frequent synchronization increases interprocess communication, which limits the levels of throughput (op/s) that can be achieved from a given set of compute resources. One way to decrease the frequency of synchronization and serialization in distributed filesystems is client-side write buffering. Clients locally apply changes to the filesystem. A record is put on a lock manager for each file with outstanding changes buffered at one or more clients.

4.3 Filesystem Format

DeltaFS provides scalable filesystem metadata services by leveraging application knowledge and resources. The philosophy of our approach can be summed up with the phrase — no false sharing, no free lunch. On the one hand, applications self-define their filesystem namespaces to decouple from each other. On the other hand, applications are expected to devote their own compute node resources to progress filesystem metadata operations and to perform necessary work (*e.g.*, log compaction) to speedup their followup metadata accesses. There is no longer a dedicated service maintaining a shared namespace for everyone causing unnecessary performance and scalability bottlenecks.

Our design consists of two parts: A) a format that the filesystem uses to record metadata information on shared storage, and B) a set of software entities that run in client middleware to provide filesystem metadata services. In this section, we describe the filesystem format and its associated data structures on storage.

A Log-Structured Approach for Filesystem Metadata. DeltaFS is log-structured. Filesystem metadata information is persisted as logs. A metadata write operation applies changes by writing new log entries to storage (no in-place update, even for deletes, which are marked by tombstones). A metadata read operation recalls information by searching and reading related log entries from storage. Each log entry has a key, identifying the file or the directory being recorded in the entry.

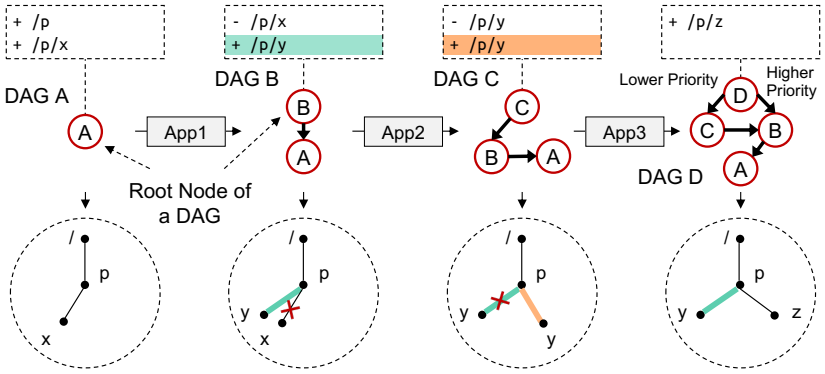


Figure 4.1: Each application can be viewed as executing a big append. It takes one or more DAGs of previously logged filesystem metadata changes as input, appends new changes on top of it, and produces a new DAG of changes as output. In this figure, App1 takes DAG A as input, producing DAG B. App2 takes DAG B as input, producing DAG C. App3 takes both DAG B and C as input, producing DAG D. App3 gives DAG B a higher priority than C, so it sees the /p/y created in B (Cyan) rather than the one created in C (Orange).

In its simplest form an DeltaFS filesystem is an assemble of filesystem metadata logs. Logs are stored as named objects in a shared underlying object store accessible to all applications on a computing platform. Applications operate directly upon logs. They read logs to recover filesystem metadata information previously recorded on storage, and write new logs to append new information. Logs are named by applications. The underlying object store ensures that log names are unique.

DeltaFS applications start by defining a base filesystem namespace. In the simplest case, an application starts with an empty base and exits with a log recording all filesystem metadata changes that the application makes to the base. In cases where an application's execution requires reading the filenames written by a set of preceding applications (sequential data sharing), the application uses the logs produced by those preceding applications to instantiate its base namespace, obtaining the metadata information of all files and directories recorded in the logs. The application then executes filesystem metadata operations on top of the base, creating new names or having existing ones (including those inherited from the logs) modified or deleted. The application writes log entries to record each such change. Upon termination, the application releases all its log entries. These log entries comprise all metadata changes that the application makes to the base. These logged changes (referred to as a change set), along with the base (which essentially is filesystem metadata changes logged previously), can in turn be used by subsequent applications to instantiate their own base namespaces.

When building a base namespace atop two or more sets of previously logged changes, it is possible for names recorded in different change sets to conflict. To resolve this problem, an application defines a priority ordering for all change sets on which it depends such that names recorded in a

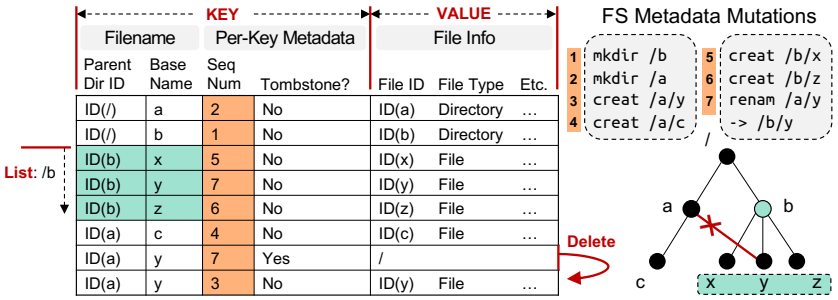


Figure 4.2: Illustration of representing filesystem metadata as a sequence of logged changes formatted as KV pairs. Each KV pair formats a change. It stores the metadata information (file ID, type, permissions, etc.) of a file after a change. Changes are logged as an application executes metadata mutation operations. Each mutation is logged as one or more changes. Each change is keyed on the name of the file being changed. Logged changes are sorted by key enabling fast directory lookups and scans. Changes are time ordered by their sequence numbers. A tombstone bit shows if a change is a delete.

higher priority set take precedence.

The execution of an DeltaFS application can be viewed as a big append operation: it appends a set of logged changes to one or more DAGs of previously logged changes, forming a new DAG of changes. As Figure 4.1 illustrates, each DAG has a root. Each node in a DAG denotes a change set (corresponding to an application execution). Each directed edge denotes a dependency between two sets of logged changes (corresponding to a sequential data sharing relationship between two applications). Changes at the top of a DAG override changes at the bottom. While it may be tempting to deem DeltaFS similar to filesystems like UnionFS [112] and OverlayFS [113], there are a few key distinctions. UnionFS-like filesystems try to provide a unified namespace view over heterogeneous filesystem formats. They are limited by direct updates, fan-out executions, and copy-on-write overlay techniques. DeltaFS enables efficient composing and even merging of namespaces atop a single filesystem format. Plus, it uses a log-structured filesystem metadata representation to achieve high performance.

To recall a filename from a DAG, the root node of the DAG is searched first. If a non-tombstone record is found, the record is immediately returned. If a tombstone record is found, the filename is immediately considered non-existent. Otherwise, the sub-DAGs beneath the root will be searched, starting from the sub-DAG with the highest priority. The filename will eventually be deemed non-existent when no non-tombstone record can be found in any of the sub-DAGs.

Log Format. As Figure 4.2 illustrates, we use key-value (KV) pairs to format the metadata changes we write to storage. Each KV pair formats a change. We store the name of the file being changed in key and the metadata

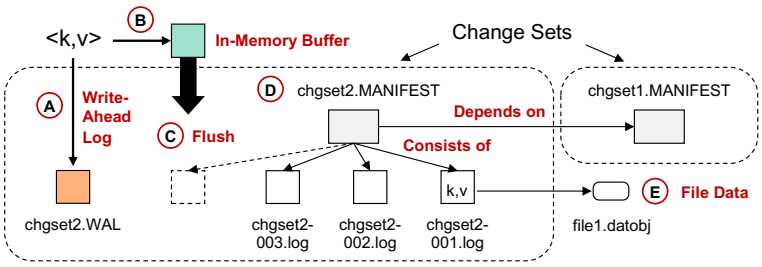


Figure 4.3: Illustration of writing a filesystem metadata change set recording the metadata changes an application makes during its execution. A) Metadata changes made by an application are first formatted as KV pairs and written to a write-ahead log (chgset2.WAL) for fault tolerance. B) The changes are then inserted into an in-memory buffer space for fast sorting when the buffer is full. C) Sorted memory buffers are flushed to storage as KV tables stored as metadata log objects (chgset2-xxx.log) in an underlying object store. D) A manifest object (chgset2.MANIFEST) is created to remember all metadata log objects created as part of the set and all input change sets of the application as the dependencies of the set. E) Data for large files is stored at separate data objects referenced by the metadata log objects in the underlying object store.

information of the file *after* the change in value. A special tombstone bit is recorded in key to indicate whether a logged change is a delete. Additionally, each change is assigned a sequence number. Changes with higher sequence numbers supersede changes with lower numbers.

We use parent directory IDs and the base names of files to represent filenames. Storing directory IDs (instead of their full pathnames) in keys prevents potentially massive key updates when an entire directory tree is renamed [27, 114]. To further improve performance, we keep keys indexed and sorted on storage. Sorting speeds up directory lookups and improves the locality needed for efficient directory scans [26].

The metadata information we store for each file includes file ID, file type, file data for small files, and file permissions for hierarchical access control. For small files, storing data with metadata improves overall data locality and reduces the total number of seeks to the remote storage for reading or writing a file [26]. For large files, we store file data as plain objects in a shared underlying object store. Per-file data may be spread across multiple data objects in order to provide scalable performance to individual files [28]. A file layout is put in the metadata record of each large file so that a reader or a writer knows how to access a file.

Log Storage and Reference Counting. We use change sets to group the metadata changes we put on storage. Each change set is a recording of all metadata changes that an application makes during an execution. An application takes one or more change sets as input and produces a new

change set as output. To start writing a change set, an application creates a manifest object in the underlying object store and records in it the names of all the input change sets of the application as the dependencies of the set. Next, an in-memory buffer space is allocated in application memory to buffer incoming metadata changes formatted as KV pairs as the application executes its program. Whenever the in-memory write buffer is full, it is sorted in memory and appended to storage as a sorted KV table. A metadata log object is created in the underlying object store to store the contents of the table (one log object per table). The name of the log object, as well as the key range of the table, is then logged in the manifest object so that a subsequent reader knows which object to open to recall the contents of the table. To prevent data loss, a write-ahead log is created in the underlying object store for failure recovery of the in-memory write buffer. Metadata changes are written to the in-memory buffer only *after* they have been logged in the write-ahead log. The manifest object, the write-ahead log, the metadata log objects listed in the manifest, and all the data objects storing file data for large files constitute the entire state of a change set, as we illustrate in Figure 4.3.

Change sets can be deleted when they are no longer needed. A user deletes a change set by invoking a special filesystem program using the name of the change set as argument. To prevent deleting a change set while other change sets may still depend on it, we have each change set hold a reference to itself and to each of its dependencies. When a user deletes a change set using a special filesystem program, the set's reference to itself is removed. All member objects (the manifest, the write-ahead log, and all metadata log objects) of the set will be deleted if no other change set has a reference to it. When a change set is deleted, all its references to other change sets will be removed enabling these change sets to be garbage collected too.

We perform reference counting on data objects as well. Data objects store data for large files. Two reasons make reference counting them important. First, the metadata information of a file may be updated across multiple change sets causing all of these change sets to have dependencies on the data of the file. Second, it is possible for an application to delete a file in one change set while other applications working on other change sets may still need the file to be alive. Data objects are referenced by metadata log objects and write-ahead logs. A metadata log object or a write-ahead log holds a reference to a data object when it has file data stored in it. A data object is deleted when all the metadata log objects and the write-ahead logs referencing it have been deleted. We use a shim layer running on top of the

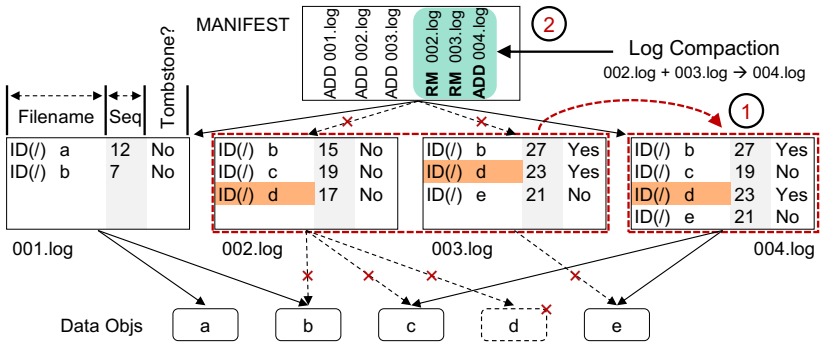


Figure 4.4: Illustration of log compaction within a single change set. In this example, Table 2 and 3 are merged into a new table, Table 4. When merging tables, keys with low sequence numbers are retired by their high sequence number counterparts. For example, Key d in Table 2 is retired by its counterpart in Table 3. After compaction, a record with updated member log information is inserted into the manifest object of the change set, committing the result of the compaction. All metadata log objects and data objects no longer referenced are then deleted from the underlying object store such as Obj d. Reading Key a required 3 table lookups before compaction: Table 3, 2, and 1. After compaction, only 2 lookups are required: Table 4 and 1.

underlying object store to provide scalable reference counting services for both data objects and change sets.

Log Compaction. We run log compaction to clean and reorganize the logs within a single change set. Each log compaction operation can be viewed as a N-way merge sort. It takes N individually sorted tables as input and replaces them with a single sorted table as output. Log compaction serves two purposes: garbage collection (by finishing data removal deferred by logged tombstones) and read optimization (by reducing the total number of table lookups needed to search a filename).

Figure 4.4 shows an example of log compaction. Before logs are compacted, there are a total of 3 tables in the change set and looking up a key could take up to 3 table lookups. After log compaction, the set consists of just 2 tables and looking up a key takes no more than 2 table lookups. When merging tables, keys sharing a same filename prefix are merged such that only the one with the highest sequence number is copied into the new table. Old metadata log entries no longer taking effect are discarded. After tables are merged, the newly constructed table is registered at the manifest object of the change set and the old tables being merged are deregistered. Next, the metadata log objects storing the old tables are deleted from the underlying object store and their references to data objects are removed. Finally, all data objects whose reference count falls to zero after compaction are deleted and storage space used by these data objects is reclaimed.

4.4 Experiments

We run experiments to demonstrate the effectiveness of the DeltaFS no ground truth parallel filesystem metadata design. All our experiments are performed on the CMU Narwhal computing cluster. Each Narwhal node has 4 CPU cores, 16GB of memory, and 70GB of HDD storage. A total of 140 nodes are used. Among them 128 are used as filesystem clients, 2 as dedicated filesystem metadata servers when the run requires it, and the remaining 10 as storage nodes providing shared underlying storage for both the filesystem and the filesystem clients. We use Ceph RADOS to implement our underlying storage [24, 99, 115]. Our 10 storage nodes consist of 8 Ceph OSD servers, 1 Ceph manager, and 1 Ceph monitor. Two 1GbE networks are used. One for the foreground communication between a filesystem client and a filesystem server or among filesystem clients. The other for the background communication between the filesystem (either clients or servers) and the underlying storage.

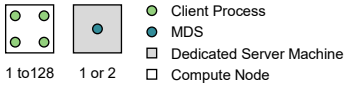


Figure 4.5: Our baseline runs launch metadata servers on dedicated server machines. This is how the current state-of-the-art parallel filesystems such as Lustre and GPFS are deployed in production.

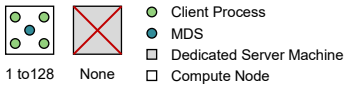
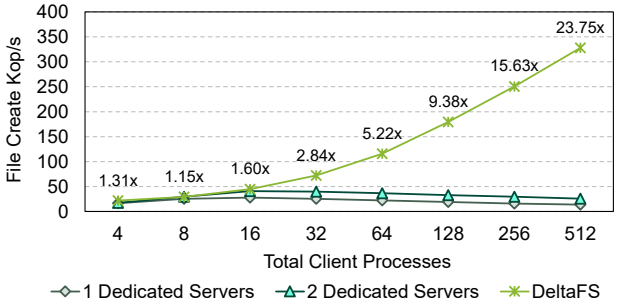


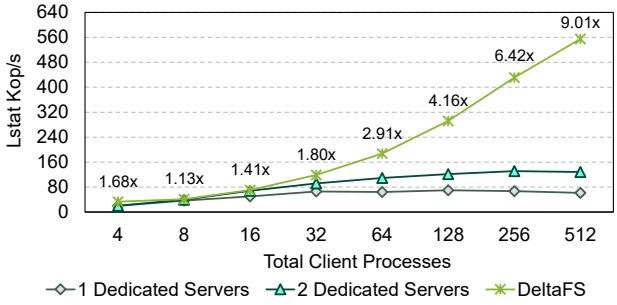
Figure 4.6: DeltaFS does not use dedicated metadata servers. Instead, jobs dynamically instantiate filesystem namespace services on client machines. In our experiments, we view each test as one single job and launch 1 metadata server instance per client node used. These metadata servers serve all clients of a test. Each server is responsible for a partition of the test job's filesystem namespace.

Advantage of Being Free from Dedicated Servers. Our first set of experiments compare the performance of filesystems that use dedicated metadata servers (baseline) and filesystems that do not use dedicated metadata servers and instead distribute metadata processing across client machines (DeltaFS). We use mdtest to drive our tests. Each our test consists of clients inserting empty files into a pool of pre-created parent directories and then stat'ing the files they just created. All runs start with an empty filesystem (except for those pre-created parent directories). Each client creates and stats 200K files. Both the creation and stating are done in a random order with regard to filenames. For baseline runs, we use 1 or 2 dedicated metadata servers to serve clients using 1 or 2 dedicated server nodes, as Figure 4.5 shows. For DeltaFS runs, we run 1 metadata server instance for each client compute node used, as Figure 4.6 shows. Our smallest run used 1 client node, 4 client processes, and created 800K (0.8M) files. Our largest run used 128 client nodes, 512 client processes, and created a total of 102.4M files.

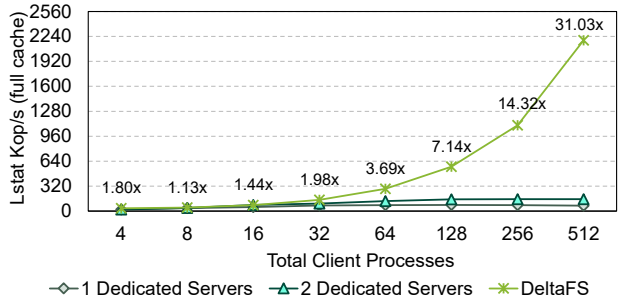
Figure 4.7a compares the insertion performance between baseline runs and DeltaFS runs. The canonical way to improve parallel filesystem metadata performance is to add more dedicated metadata servers. This works. Our baseline runs with 2 dedicated servers are about 2x faster than the runs with 1 dedicated server. Unfortunately, filesystems that use dedicated metadata



(a) Comparison of write (`fcreate`) performance between baseline runs and DeltaFS runs using up to 128 client nodes and 512 client processes (4 client processes per client node). Baseline runs are limited by their dedicated servers to deliver high insertion performance. DeltaFS runs spawn 1 metadata server per client node (up to 128), show scalable write performance as the amount of available client resources increases, and are up to 23.75x faster. The speedup is less than 128x because of CPU saturation at the client nodes and bottlenecks at the shared underlying storage.



(b) Comparison of read (`lstat`) performance between baseline runs and DeltaFS runs. Similar to what Figure 4.7a shows, baseline runs are limited by their dedicated servers to deliver high performance whereas DeltaFS runs show scalable performance and are up to 9.01x faster. The speedup is less than that in Figure 4.7a due to bottlenecks at the shared underlying storage when filesystem metadata is read from it.



(c) Comparison of read (`lstat`) performance when metadata information is fully cached at server memory. DeltaFS runs use up to 128 servers and are up to 31.03x faster compared with the baseline. The speedup is less than 128x due to CPU saturation at the client nodes.

Figure 4.7: Experiment results comparing read (`lstat`) and write (`fcreate`) performance of filesystems that use dedicated metadata servers and filesystems that do not use dedicated metadata servers and instead distribute metadata processing over client machines. For runs with dedicated metadata servers (baseline runs), up to 2 dedicated server nodes and up to 128 client machines are used. For runs without dedicated metadata servers (DeltaFS runs), up to the same amount of client nodes are used and DeltaFS spawns 1 server instance per client node and on that very client node. Up to 128 client-colocated metadata servers are used. All runs execute the same filesystem server code and are configured with the same amount of total metadata cache. DeltaFS runs may be able to use more writeback buffers and more LSM-Tree compaction memories due to having more server instances.

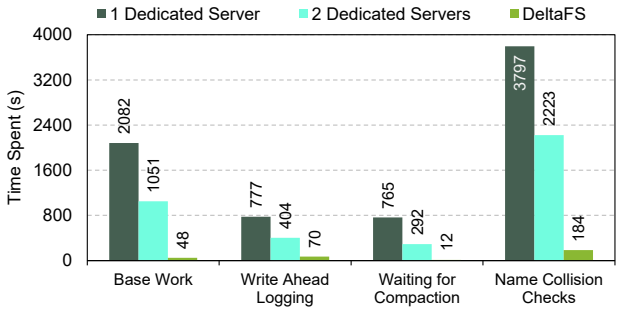
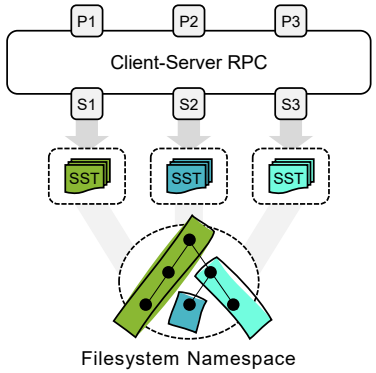


Figure 4.8: Breakdown of cost in the form of time spent in seconds for creating a total of 102.4 million files using 128 client nodes (512 client processes) and 1 dedicated (baseline), 2 dedicated (baseline), or 128 non-dedicated (DeltaFS) metadata servers. DeltaFS is 43.38x faster in base work, 11.10x faster in write-ahead logging, 63.75x less costly in being blocked by background compaction activities, and 20.64x faster in name collision checks for an aggregate speedup of 23.75x as we report in Figure 4.7a.

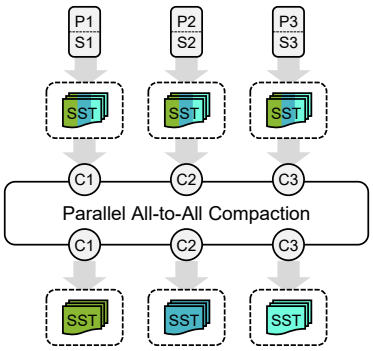
servers are ultimately limited by their dedicated servers to deliver high performance. Once the dedicated servers are saturated, performance stops increasing and to enable higher performance more servers will need to be added. DeltaFS decouples filesystem metadata performance from the use of dedicated server machines. By distributing work across all available client nodes, DeltaFS runs show scalable metadata performance that increases as the amount of available client resources increase. At 512 client processes, DeltaFS is 23.75x faster than the baseline with 1 dedicated metadata server.

To better understand the 23.75x speedup of DeltaFS at 512 client processes, Figure 4.8 breaks down the cost of file creates at the server side. We divide the cost into 4 categories: (1) performing name collision checks, (2) waiting for background compactions, (3) write-ahead logging, and (4) the rest of the server-side work for creating a file (base work). To measure the cost of one category, we do two runs — one with the cost and one without — and then measure the difference in time. Cost is measured accumulatively. We start with runs without collision checks, background compaction, or write-ahead logging. We then gradually add write-ahead logging, background compaction, and collision checks.

By distributing work across 128 client-colocated metadata servers, DeltaFS is up to 43.38x faster in performing the base work. The speedup is less than 128x because of CPU saturation at the client side. At the same time, the most significant cost of file creates lies in performing name collision checks for which DeltaFS is up to 20.64x faster than baseline runs. The reason that this speedup is less than that for the base work is due to bottlenecks in the sharing underlying storage when performing collision checks requires reading metadata information from it. DeltaFS gains speedup from less background compaction throttling (due to having a smaller metadata footprint per metadata server) and more concurrent write-ahead logging



(a) Early Integration in a parallel DeltaFS job. Job namespace is dynamically partitioned across job metadata servers (S1, S2, and S3). Each server is responsible for one namespace partition. Namespace updates executed by job processes (P1, P2, and P3) are immediately sent to the corresponding servers for integration both for strong consistency and for eager SSTable compaction for fast reads.



(b) Deferred Integration in DeltaFS. Per-process namespace updates are directly logged through process-local metadata servers as SSTables. SSTable merging, partitioning, and compaction are delayed until post-processing and are carried out in the form of a job-wide parallel all-to-all compaction program where each compaction process (C1, C2, and C3) acts both as an SSTable reader and as an SSTable writer.

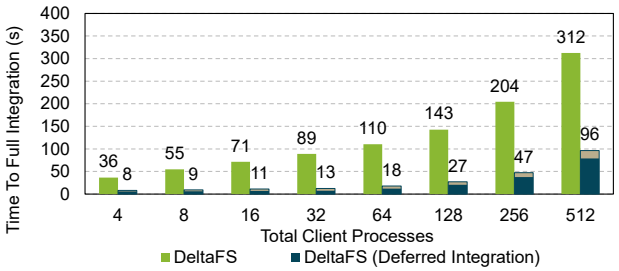
Figure 4.9: Illustration of early integration and deferred integration in DeltaFS. Whereas early integration ensures strong consistency and immediately optimizes metadata storage for fast reads, deferred integration delays merging and read optimization until post-processing which performs both in a large batch.

(due to having more servers) too, though their effect is less significant in our test runs.

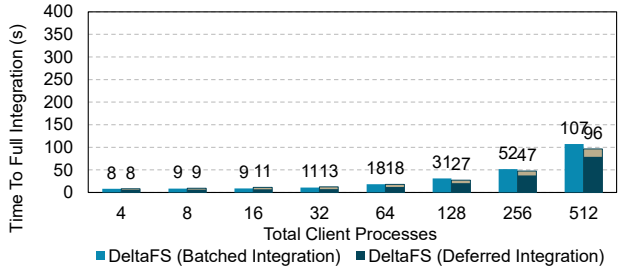
Figure 4.7b compares the metadata read performance between baseline runs and DeltaFS runs. Similar to metadata writes, DeltaFS shows scalable performance and is not limited by dedicated servers to deliver high performance. At 512 client processes, DeltaFS is up to 9.01x faster than baseline runs. This speedup is less than the 23.75x that we see in Figure 4.7a because of bottlenecks at the shared underlying storage.

Figure 4.7c shows performance of metadata reads when all metadata information is cached in memory. Without being blocked by storage reads, full-cache DeltaFS is up to 31.03x faster than full-cache baseline runs. Due to CPU saturation at the client nodes, the speedup is still less than 128x.

Advantage of Deferred Integration. DeltaFS distributes work across client machines to achieve scalable performance. Filesystem namespace updates are immediately sent to client-colocated metadata servers for early integration. Early integration has two advantages. First, it ensures strong consistency. Filesystem namespace updates executed by one client process are immediately visible to all other processes of a parallel job. At the same time, early integration allows the underlying metadata storage



(a) Comparison of early integration and deferred integration in DeltaFS. Early integration used up to 512 client processes and 128 client-colocated metadata servers. In deferred integration each client process doubles as a metadata server for sequential metadata logging and there are no standalone metadata servers. A followup compaction program merges and rewrites all namespace data for fast reads (time shown separately in figure). Deferred integration yields better performance due to aggressive batching and a more streamlined inter-process communication process during post-processing. The reason that the gap between early integration and deferred integration appears to be reducing is due to bottlenecks at the shared underlying storage and the fact that deferred integration runs are overall more efficient so that their performance is more sensitive to the increasing cost of name collision checks and filesystem write-ahead logging as job size increases.



(b) Comparison of early integration and deferred integration with early integration runs now spawn the same amount of metadata servers as clients (up to 512 client-colocated metadata servers) and use batching when making RPC calls to servers (96 file creates per RPC instead of 1). Due to more efficient client-server communication, batched early integration shows comparable performance with deferred integration. Deferred integration exhibits reduced total cost for large job sizes and has the advantage of being able to skip merging and compaction altogether when fast metadata read performance is not immediate required following job completion.

Figure 4.10: Experiment results comparing early metadata integration and deferred integration in DeltaFS. With early integration, per-job client metadata updates are immediately merged at job metadata servers and eagerly optimized for fast reads. With deferred integration, by contrast, there are no standalone job metadata servers. Instead, per-job metadata updates are directly logged at per-process log files during job execution and later bulk merged and parallel compacted for fast reads.

representation (SSTables) to be immediately optimized for fast subsequent reads. This is done by dynamically partitioning a parallel job's filesystem namespace across all available metadata servers of the job, immediately sending client filesystem namespace updates to job servers according to this partitioning, and then by having servers constantly run background compactions to optimize SSTables for fast reads, as we show in Figure 4.9a.

For scientific workflows that do not require filesystem namespace updates to be immediately visible job-wide and do not require these updates' underlying storage representation to be immediately optimized for fast

reads, it is possible to defer integration by having clients directly log filesystem namespace updates in shared storage through an embedded local metadata server and then by running a job-wide parallel compaction program to merge and rewrite these metadata logs, partitioning and reorganizing them for fast reads. We call this deferred integration, as Figure 4.9b depicts.

Our second set of experiments compares the performance of early integration and deferred integration in DeltaFS. We focus on time to full integration. For early integration, this measures the time for all client filesystem metadata operations to be processed at the server end. For deferred integration, this measures the time for all client filesystem metadata operations to be processed and logged at the client end and then for a followup parallel post-processing program to finish merging and compacting all logged namespace updates.

We use our DeltaFS results in Figure 4.7a to show the performance of early integration. To evaluate the performance of deferred integration, we perform the same tests but with a symmetric DeltaFS deployment where each application process acts both as a filesystem client and as a filesystem metadata server as Figure 4.11 illustrates. These per-process metadata servers directly process the local client's metadata operations and log namespace updates in shared storage. Processing metadata operations includes performing write-ahead logging for fault tolerance and per-process name collision checks for filesystem integrity but excludes compaction. A followup parallel compaction program merges and rewrites all these updates, priming the namespace data for fast future reads.

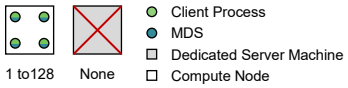


Figure 4.11: Illustration of a symmetric DeltaFS deployment. Each job process acts both as a filesystem client and as a metadata server managing a partition of the job's filesystem namespace. The job no longer spawns standalone metadata servers as Figure 4.6 shows.

Figure 4.10a shows the results. While distributing work across all available client machines enables DeltaFS to be up to 23.75x faster than baseline runs as we see in Figure 4.7a, deferred integration enables DeltaFS to be even faster in absorbing metadata write operations. While deferred integration requires per-job filesystem namespace updates to be first logged in a write-optimized format and then read back and reorganized for fast metadata reads, the reason deferred integration still yields better performance is two-fold. First, deferred integration enables metadata information to be partitioned and exchanged over the network in large batches which are more efficient than doing it with per-file operations. At the same time, deferring all filesystem-wide inter-process communication activities to a separate post-processing program enables a more streamlined process that is not limited by filesystem write-ahead logging and collision checks to

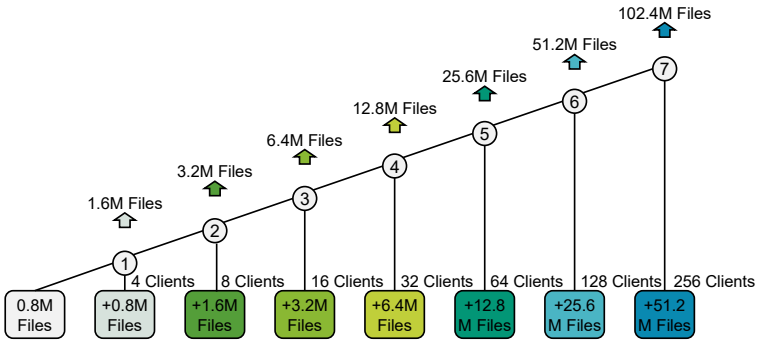
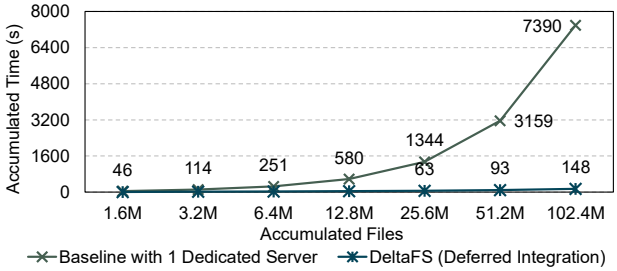


Figure 4.12: Test workflow for measuring the cost of a lack of a global filesystem namespace in DeltaFS. Each our test starts with a filesystem namespace containing 0.8M files. Then at each workflow stage, a total of n files are inserted into the filesystem raising the total file count to $2n$. Files are inserted through a parallel client application. Each client process creates 200K files. Our first stage consists of 4 client processes inserting 800K (0.8M) files ($n = 0.8M$). We do a total of 7 workflow stages. The last stage consists of 256 client processes inserting 51.2M files ($n = 51.2M$) concluding the test with a total of 102.4M files in the filesystem.

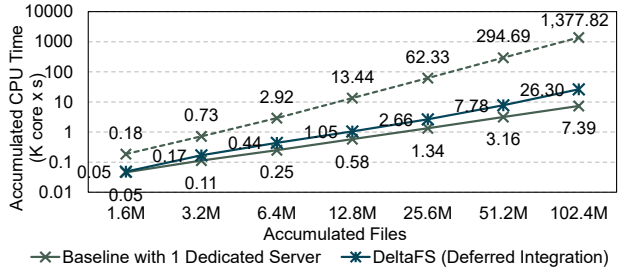
deliver high performance. The reason that in our results the gap between early integration and deferred integration appears to be reducing as job size increases is due to bottlenecks at the shared underlying storage and the fact that deferred integration runs are overall more efficient so that their performance is more sensitive to the increasing overhead of name collision checks and filesystem write-ahead logging (both require namespace data to be read and written against storage in relatively small I/O sizes) as per-job file count and file activities increase. Our largest run used 128 client nodes, 512 client processes, and created 102.4M files. The cost of reading back namespace data for post processing increases as job size increases too, but its effect is less significant to the overall performance due to the ability to use large I/O sizes when performing these operations.

Figure 4.10b further compares the performance of early integration and deferred integration with early integration runs now spawn the same amount of metadata servers as clients (up to 512 client-colocated servers) and use batching when making RPC calls to servers (96 file creates per RPC instead of 1). With a smaller metadata footprint per server and RPC batching, the cost of early integration reduces and now becomes comparable with deferred integration. On the other hand, deferred integration still wins and has the advantage of being able to further defer metadata merging and compaction when fast metadata read performance is not immediately needed following job completion.

Cost of No Global Namespace. A key property of DeltaFS is no global namespace. While no global namespace permits freedom from global serialization, the cost of it is the added need for jobs to merge and compact related namespace data before they can access their namespaces efficiently.



(a) Accumulative time it takes for a test run to complete each workflow stage. Workflow stages are marked by the accumulative number of files that have been inserted into the filesystem. For baseline runs, 1 dedicated metadata server is used and all files are inserted into a global namespace. For DeltaFS runs, up to 512 client machines are used and each workflow stage consists of executing two application programs. One is the main application that creates files and logs namespace updates in SSTable files stored in the shared underlying storage. The other is a parallel compaction program that merges the SSTables from both the first program and the previous workflow stage to form a combined namespace consisting of all created files. While DeltaFS performs more work, it shows significantly less latency due to aggressive utilization of client resources.



(b) Accumulative amount of compute resources that it takes for a test run complete each workflow stage. Workflow stages are marked by the accumulative number of files that have been inserted into the filesystem. Resource usage, shown logarithmically in figure, is measured in the form of CPU seconds times cores. For baseline runs, two types of resource usage are reported. One for the resource usage at the server end (shown as solid line). The other for the usage at the client end (shown as dotted line). DeltaFS does not use dedicated metadata servers. We report its resource usage at the client end. DeltaFS consumed more compute resources for filesystem metadata processing due to repeated work in merging workflow namespace data. While baseline runs spent less total resources at the server end, they wasted a massive amount of client CPU cycles for doing nothing.

Figure 4.13: Experiment results measuring the cost of no global namespaces in a DeltaFS filesystem compared with the current state-of-the-art which defines a global namespace and uses dedicated metadata servers. By distributing work across client machines, DeltaFS shows significantly lower latency in absorbing bursty filesystem metadata operations. The cost of no global namespace is the increased work that DeltaFS causes for merging namespace data potentially repeatedly. Even so, DeltaFS shows better overall resource utilization compared with the current state-of-the-art.

Our last set of experiments measures this cost through running a synthetic workflow as Figure 4.12 shows. Our workflow starts with a filesystem containing 0.8M files. There are a total of 7 workflow stages. Each workflow stage inserts new files into the filesystem and doubles the amount of files in the filesystem. At the end of the last stage, there will be 102.4M files in the filesystem. We use a parallel client application to create files. Each client process creates 200K files. Our first stage consists of 4 processes inserting 800K (0.8M) files. The last stage consists of 256 processes inserting a total of 51.2M files.

We focus on the accumulative time and resource usage for a test run to finish each workflow stage. We compare the performance of a run with a global namespace (baseline) and a run without one (DeltaFS). For baseline runs, all files are directly inserted into a dedicated metadata server providing a global filesystem namespace. For DeltaFS runs, file inserts are first recorded at per-process log files (SSTables) and then merged to form "global" filesystem namespaces as Figure 4.14 depicts. That is, each DeltaFS workflow stage consists of running two application programs. The first program is the parallel file-creating driver application with embedded DeltaFS metadata servers logging file creates as per-process SSTables stored in the shared underlying storage. The second program is a parallel compaction program that merges both the per-process SSTables generated by the first program and the set of SSTables produced by the previous workflow stage to form a combined filesystem namespace logically equivalent to the global filesystem namespace that is defined in the baseline runs. The need to run a parallel merge and compaction program at the end of each workflow stage represents the cost of no global namespaces with DeltaFS.

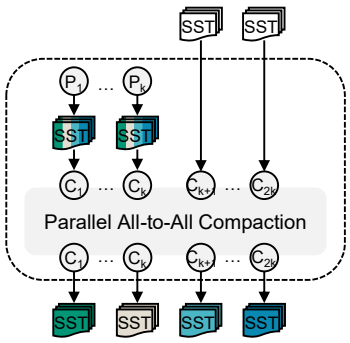


Figure 4.14: Illustration of a DeltaFS workflow stage. Each DeltaFS stage consists of running two application programs. The first program is a parallel file-creating driver application (P_1-P_k) with embedded DeltaFS metadata servers logging file creates as per-process SSTables stored in the shared underlying storage. The second program is a parallel compaction program (C_1-C_{2k}) that merges both the per-process SSTables generated by the first program and the set of SSTables produced by the previous workflow stage to form a combined filesystem namespace containing all files that have ever been created so far since workflow inception — logically equivalent to a global filesystem namespace. There are a total of 7 workflow stages. Each stage is 2x the size of its predecessor. The first stage consists of 4 file create processes and 8 compaction processes ($k = 4$). The last stage consists of 256 file create processes and 512 compaction processes ($k = 256$).

Figure 4.13a shows the results in terms of the time it takes for a test run to finish each workflow stage. While DeltaFS requires jobs to explicitly merge and compact namespace data for fast reads, its utilization of client resources for scalable metadata processing still enables it to show significantly less processing delays (49.93x faster) compared with baseline runs, which are limited by a dedicated metadata server to deliver high metadata performance. The cost of no global namespace is the increased work that DeltaFS causes for merging namespace data repeatedly — each workflow stage remerges the namespace data from previous stages. As Figure 4.13b shows, DeltaFS used up to 3.56x more compute resources than baseline runs. Meanwhile, even though the baseline runs spent less total resources at the server end, they effectively wasted a massive amount of client CPU cycles by blocking them for doing nothing. DeltaFS shows better overall resource utilization by not limiting metadata processing to only dedicated server machines.

4.5 Summary

At exascale, metadata is no longer a trivial step that adds only a tiny latency before data operations. In LANL's Trinity cluster [32], it takes 256s for

every CPU core to create a file in the global Lustre namespace[1], but only 600s for the entire 2PB of memory to be dumped from compute nodes to Trinity's on-platform burst-buffer storage nodes. Traditional file systems are unlikely to scale to exascale because: 1) centralized metadata requires either expensive hardware to scale-up or a large number of dedicated machines to scale-out; 2) imposing a single namespace forces applications to frequently synchronize with each other mostly unnecessarily; 3) ensuring metadata integrity and strong consistency over a global namespace demands the use of a dedicated (and easily bottlenecked) coordinator to enforce system invariants; and 4) classic on-disk metadata representation lacks efficient support for fast metadata insertion, migration, redistribution, and aggregation.

1: Assuming a 5000 file creates/sec for a typical Lustre configuration.

Through a serverless design, DeltaFS does not need the dedicated server machines found in conventional parallel filesystems. Applications start from immutable snapshots and self-manage their namespace data using their own compute resources. DeltaFS's LSM-Tree based metadata representation is optimized for writes, efficient to share and merge, and can be appropriately compacted to optimize later retrieval. At the same time, DeltaFS advocates the use of scalable object stores to provide shared underlying storage, and to assist with security enforcement, garbage collection, and administrative data purging. Our experiments show that DeltaFS shows orders of magnitude lower latency in finishing a metadata-intensive workflow step compared with the current state-of-the-art. Meanwhile, the cost of no global namespace and no dedicated server is the increased work that DeltaFS causes for merging namespace data potentially repeatedly. Such cost may be reduced through utilization of metadata curators both within and across scientific workflows.

Indexed Massive Directories | 5

Scientific applications perform data analysis by writing data to storage and then running queries against the data. It can be bad experience for scientists when they either have to wait for a large amount of data to be pre-transformed to a read-optimized format before they can start the queries, or they must risk having each of their queries fetch an excessive amount of data from storage while only a tiny percent of it is actually needed for the results. In this chapter, we show a scalable streaming data processing mechanism — Indexed Massive Directories (IMDs) — that liberates scientists from this dilemma.

IMDs dynamically transform data to a read-optimized format as a parallel writer application writes it to storage. Scientists enjoy fast queries while not having to experience long post-processing (after writes) or pre-processing (before queries) delays. More crucially, our work shows that it is possible to leverage only idle CPU cycles available on the compute nodes of the writer application to perform all these operations — no dedicated compute resources are needed. IMDs demonstrate a new way of providing data acceleration capabilities in modern HPC environments. Unlike emerging storage designs in which dedicated compute resources near data at rest are leveraged to speed up data analysis, in IMDs the acceleration takes place at inception of a data pipeline and available compute resources on the main computing platform of an HPC cluster are harvested to perform the data acceleration computation. The unique way IMDs operate complements emerging designs.

The rest of this chapter is structured as follows. Section 5.1 describes the background and the motivation behind this work. Section 5.2 presents a high-level design of IMDs. Section 5.3 discusses challenges and a series of techniques for tackling these challenges. Section 5.4 demonstrates the effectiveness of IMDs using real-world scientific workloads. We show related work in Section 5.5 and summarize in Section 5.6.

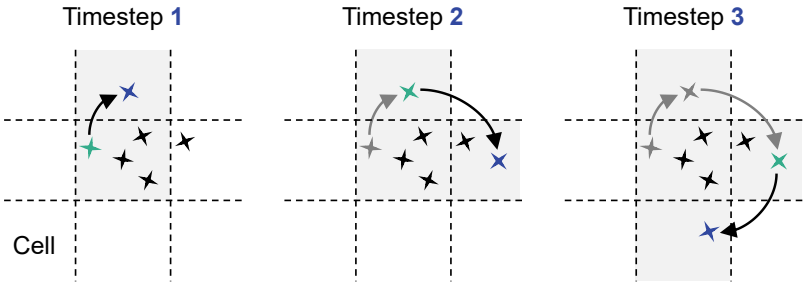


Figure 5.1: Illustration of a typical VPIC particle simulation. The simulation space is divided into cells. Each VPIC process manages a cell. Tracing the state of a particle over time is non–trivial as particles move in an unpredictable way during a simulation and can be saved at different storage locations at different time.

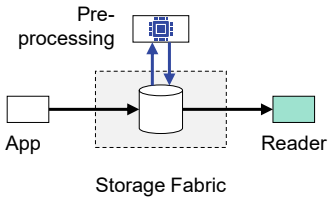
5.1 Motivation

The motivation for a new data processing mechanism came from difficulties a team of Los Alamos National Laboratory (LANL) scientists encountered when trying to use the Vector Particle-In-Cell (VPIC) simulation framework on larger and more data intensive problems.

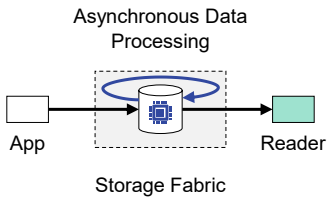
The Needle-In-a-Haystack Problem of VPIC. VPIC is an open-source parallel particle simulation framework developed at LANL [116]. In a VPIC simulation, the simulation code divides the simulated space into cells and distributes ownership and management of each cell among the processes in the simulation. Within each cell a process manages a set of moving particles based on underlying principles from physics. Individual particles often move between cells as the simulation progresses. This is done by transferring the particle state between two processes managing neighboring cells. Large-scale VPIC simulations powered by the world's largest high-performance computing platforms manage the state of trillions of particles across hundreds of thousands of CPU cores [39].

VPIC-based simulations run in timesteps. Every few timesteps VPIC stops and writes the state of all particles to storage. Typically, the analysis of a VPIC simulation run occurs after the simulation concludes. The problem the team of LANL scientists are looking at involves the trajectories of a tiny subset of particles that end a simulation with an unusually high energy. The trajectory of a particle includes its travel path through the simulated space over time and its state (e.g. energy-level) for each step of the path, as shown in Figure 5.1. High energy particles of interest are identified at the conclusion of a simulation.

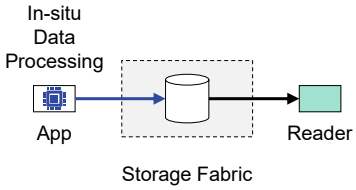
Finding the trajectories of a few high energy particles in a large simulation



(a) The Pre-processing Approach runs a separate data processing program to improve read performance but resulting in delays experienced by the reader application.



(b) The Current State-of-the-Art hides delays by offloading data processing to storage and performing it early and asynchronously. Performance is improved provided sufficient compute resources at the storage.



(c) Our in-situ Data Processing Approach takes advantage of the idle compute resources available on the compute nodes of the writer application to process data so data can be processed early to minimizes delays while not being limited by the processing power of the storage.

Figure 5.2: Comparison of three different data processing approaches to speeding up post-hoc data analysis queries. Our example consists of a writer application on the left and a followup reader application on the right. The writer application writes data to storage. The reader application executes queries which read data from storage. The best read performance is achieved when the data on storage is stored in a format that is optimized for the queries of the reader application. a) The pre-processing approach improves read performance by pre-transforming data to a read-optimized format before reads take place. The cost is the time the reader application has to wait before it can access data efficiently. b) To hide such delays, modern computing platforms utilize the compute resources within the storage to process data so data can be processed early and asynchronously while the writer application writes it to storage. Nevertheless, their abilities to hide delays are ultimately limited by the total amount of compute resources the storage possesses. c) Our work utilizes the idle compute resources available on the compute nodes of the writer application to process data and dynamically transforms the data to a read-optimized format as the writer application writes it to storage. Our approach has the advantage of not being subject to the processing power of the storage while allowing data to be processed early to minimize delays.

is a challenge for several reasons. First, the identity of the high energy particles of interest is not known in advance, so they cannot be marked or traced when the simulation starts. Second, as particles migrate between simulation processes during the course of a simulation, a particle's state is scattered across the nodes in the cluster running the simulation. Third, once the simulation completes and high energy particles are identified, the entire simulation output needs to be read back and scanned in order to extract the needed trajectories. Reading back an entire simulation output and filtering out relevant information is becoming prohibitively expensive as simulation size grows.

In some sense, tracing the trajectories of a small number of particles in a large simulation output is like finding a needle in a haystack: both are characterized by a high query selectivity and a large amount of data.

The reason this VPIC use case is interesting is two-fold. First, it represents

a common class of I/O problems for which existing data management schemes fall short when data size is large. We show this aspect in Section 5.1.1 and Section 5.1.2. Second, the massive amount of compute resources on modern computing platforms provide opportunities to process data early before data reaches storage while overcoming limitations of existing data management schemes. We show this aspect in Section 5.1.3.

5.1.1 Pre-Processing Data Before Queries

The difficulties VPIC scientists experience represent a common challenge in data analytics. Many data-intensive applications output data without necessarily considering the efficiency of the queries following the writes. This is especially common when an application's output consists of a large number of small objects and these small objects are batched together and appended to storage using large sequential writes. While doing so allows the underlying storage bandwidth to be more fully utilized, data is not always appended in the optimal order for subsequent queries. As a result, processing a query may require reading back an excessive amount of data from storage, which can be extremely inefficient and time-consuming.

One way to speed up queries is to pre-process data before queries. This allows data to be transformed to a format that is optimized for the upcoming inquiry. Today, data transformation is typically done by launching a separate data processing program on the main computing platform after the main application exits. As Figure 5.2a shows, the main application writes data to storage. A followup data processing program reorganizes the data, speeding up subsequent data analysis.

Reorganizing data requires reading back data from storage, processing it on client nodes, and writing the transformed data to storage. As data size grows, data reorganization done in the form of a separate data processing program can be extremely costly. This is due to the large amount of I/O involved in streaming the data from and to storage and the delay caused by the program to perform the data reorganization computation. As a result, a user may have to wait an extended amount of time before they can access data in a read-optimized format.

5.1.2 The Current State-Of-the-Art: Asynchronous Data Processing Near Data at Rest

To accelerate data analysis, modern computing platforms take advantage of the dedicated compute resources within their storage to perform data operations. This includes both the compute resources on a storage server and those within storage devices. It improves performance for two reasons:

First, as Figure 5.2b shows, offloading work to storage leaves room for more aggressive latency hiding through asynchronous data processing. For example, transformation of the current timestep's data could take place in the background while the application itself moves to the simulation computation of the next timestep. Processing data in the background can effectively hide delays. These include both the delay associated with I/O and the delay associated with performing the data processing computation.

Second, directly processing data on storage enables more efficient data access. This is because that the available I/O bandwidth inside the storage may be significantly higher than the bandwidth of shared, higher-level I/O channels outside the storage. This allows data to be processed faster, provided that processing data is primarily bottlenecked on I/O.

While offloading data processing to storage introduces performance benefits, two limitations exist:

First, despite the ability to hide latency through asynchronous processing, the overall processing power of the storage is still limited by the total amount of compute resources it possesses. When the available processing power is insufficient for the demands of the incoming workload, a user may still experience significant delays.

Second, even near media, reading and writing data in large amounts are expensive. Critically, data processing remains costly if performing it requires reading back all the data from storage and writing the processed data to storage. As the gap between compute and storage continues to rise [42], the performance of a modern computing platform is best exposed when data is processed using the minimum amount of I/O and applications experience as few data processing delays as possible [10, 13].

5.1.3 Processing Data Early Before Data Reaches Storage

To address the first limitation of the current state-of-the-art, we notice that the computing cycles needed for data processing may be available on the main computing platform while an application writes data to storage. This is because that the writing process is expected to be blocked on storage (and limited by it), which leaves idle CPU cycles on the compute nodes of the application performing the writes. These idle CPU cycles, potentially in massive amounts and allocated in proportion with the application's problem size, can then be utilized to perform storage operations, as we illustrate in Figure 5.2c.

We call this in-situ data computation, as the computation takes place right on the compute nodes of an application and happens when the application writes data to storage. Compared with today's on-storage data computation, in-situ computation is not limited by the computing capability defined by the underlying storage and has the ability to aggregate a massive amount of compute resources outside the storage to perform data computation.

The best performance of in-situ data computation is seen when an application writes data synchronously. This happens when the application runs without overlapping compute with I/O so the execution of the application program is effectively paused during the application's I/O phases, as the example we show in Figure 5.3.

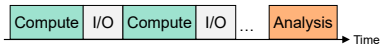


Figure 5.3: Illustration of a typical scientific workflow consisting of a bulk-synchronous parallel simulation application acting as a data writer and a subsequent data analysis program acting as a data reader with the execution of the writer further divided into iterations of non-overlapping compute and I/O phases. A writer application chooses not to overlap its compute with its I/O because overlapping does not always reduce total run time. The idle CPU cycles available on the compute nodes of such a writer application during its I/O phases can then be utilized to perform storage operations, accelerating subsequent data analysis.

Applications write data synchronously because overlapping is not always a better option:

First, the in-memory state needed for I/O (the state of all particles in the case of VPIC) may be modified during the next computation phase so overlapping I/O with computation would require making a second copy of the state in memory for I/O. As applications often set their problem sizes to use all available memory, it would be inconvenient to store two copies of state in memory reducing overall memory utilization.

Second, the NICs on the compute nodes may be used by the application to perform inter-process communication during its computation phases. Thus overlapping I/O with computation introduces contention in the network, which may increase run time and reduce overall application performance.

To address the second limitation of the current state-of-the-art, we notice

that the I/O cost of data processing can be significantly reduced if the processing is done on the fly while an application writes data to storage. The reason is two-fold. First, it allows data to be processed early before it reaches storage so processing data does not involve reading back all the data from storage and then writing the processed data to storage. This minimizes total I/O. Second, it allows data to be processed in parallel with its writing so that the data processing latency can be conveniently hidden by the writing of the data to storage. This minimizes the total data processing delay applications obverse.

We propose DeltaFS IMDs, a scalable method for parallel applications to dynamically reorganize their writes for fast subsequent reads. To achieve this, DeltaFS IMDs reuse the idle CPU cycles available on the compute nodes of their applications to process data and dynamically transform data to a read-optimized format as these applications write the data to storage. While our work is inspired by the VPIC use case, we expect our techniques to be generally useful in handling sequential writes and random reads (needle in a haystack) problems, such as anomaly detection and software debugging in network monitoring and event tracing systems [117, 118].

5.2 System Overview

DeltaFS IMDs are client middleware to be embedded inside the processes of a parallel data application for in-situ data computation. As Figure 5.4 illustrates, when the application writes data to storage, DeltaFS IMDs dynamically transform the data to a read-optimized format and write the transformed data to storage, speeding up subsequent queries. In this section, we present an overview of this online data transformation service.

5.2.1 Target Application and Query Types

Applications that benefit most from DeltaFS IMDs are ones with data output that occurs in bursts. We imagine these applications to be massively parallel programs in which each application process outputs data and data is written to a shared underlying storage system such as a distributed parallel filesystem running on a dedicated set of storage nodes.

We model data as simple key-value (KV) pairs. Keys are written in an

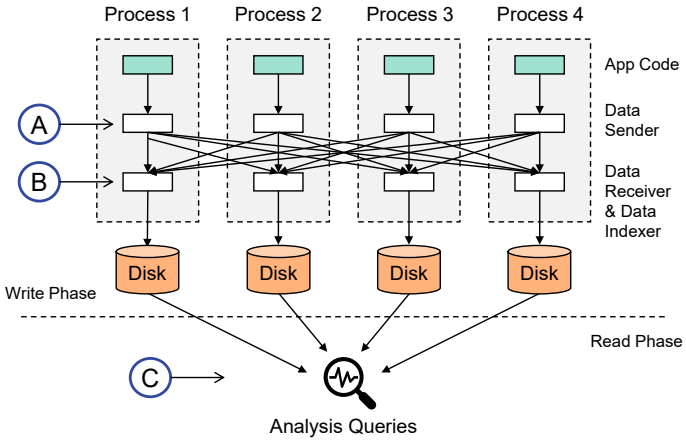


Figure 5.4: Illustration of performing in-situ data computation on the write path of a parallel data application for speeding up queries on the read path. Data computation takes place within each application process. Data is processed on the fly as it is written by the application to storage. Processing data consists of (**A**) online data partitioning via all-to-all data shuffling and (**B**) online per-partition data indexing. Each application process is a data partition, and acts as both a sender and a receiver of data. Indexed data is written to a shared underlying storage system. Analysis queries are done by a followup reader program (**C**), which queries data directly against the underlying storage.

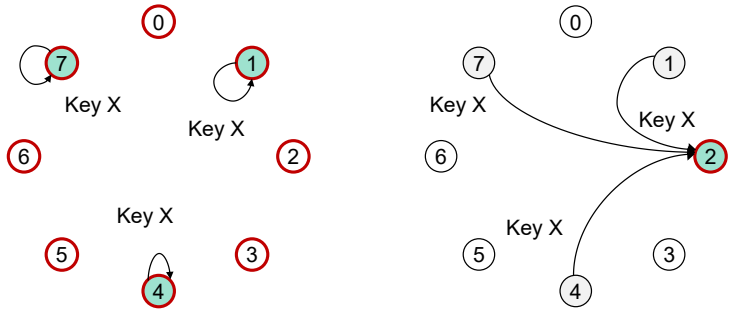
arbitrary order with each key possibly appearing more than once. When this happens, subsequent values on the same key append data to the existing value rather than overwriting it. We expect each application to be followed by a reader program which performs analysis on the data produced by the former. Data analysis takes place after all data is written to storage. We consider two types of queries: looking up a key for a specific append or looking up all data that has been appended to a key. We call the first type of query point queries and the second small-range queries.

5.2.2 Write Path

The write path of DeltaFS IMDs is designed as code that transforms data on the fly as data is written to storage. Transforming data is a two-step process. The first step partitions data among the processes of the writer application. The second step builds per-partition data indexes.

An important reason for partitioning data is to assign a home region to each key so that looking up a key does not require searching all data. Figure 5.5a shows a case where data is not on the fly partitioned and is directly written to per-process output files in the shared underlying storage. Because any process may write any key, a followup reader program will have to check all files in order to find the data of a key.

On the other hand, with data partitioning each key is sent to its home process for writing so reading back a key requires searching only the output of one specific process, which better bounds the work of a reader program,



(a) Without data partitioning, data is directly written to per-process output files and reading back a key requires searching ALL files.

(b) With data partitioning, each key is sent to its home process for writing so reading back a key requires searching only the output of one specific process (process 2 in the example figure).

Figure 5.5: Processes (numbered as 0, 1, 2, ..., 9) of a parallel application writing data output with and without data partitioning. Note that for both cases we assume that all data is written to a shared underlying storage system and that a followup reader is able to access all files.

as Figure 5.5b shows.

To partition data, each process of a parallel DeltaFS IMD writer application is assigned a partition. A hash function is used to map keys to partitions such that each partition is responsible for a disjoint range of keys. When the parallel writer application writes data to storage, data not belonging to the local process is sent to the remote process responsible for the key. Each process is a sender of data, and may receive data from all other processes. When a process receives a key, an index entry is dynamically generated for the key, speeding up data lookups within that data partition.

The overall write path of an DeltaFS IMD can be viewed as an in-situ data processing pipeline embedded inside the processes of a parallel writer application. The parallel writer application streams data to the pipeline, the pipeline processes the data, and streams the processed data to storage. Streaming data processing takes place on all processes of the writer application, and consists of an online data partition step and an online data indexing step. All work is done through reusing the idle CPU cycles available during the writing of the data to storage. After processing, data is stored as indexed per-process log objects in the shared underlying storage. We present more details of this in-situ data processing pipeline in Section 5.3.

5.2.3 Read Path

The read path of DeltaFS IMDs is designed as code that processes queries on behalf of a reader program. It utilizes the data partitions and indexes created at the write phase to quickly recall keys and values. Processing both types of queries (point queries and small range queries) requires first determining the partition responsible for the key and then utilizing the indexes of that partition to locate the data of the key. As we will demonstrate in Section 5.4, because only a small amount of data needs to be read per query, DeltaFS IMDs are able to keep queries fast even when data size is large. This is as opposed to cases in which query latency is kept low by having large numbers of compute nodes read back data in parallel, as we sometimes see in practice [39, 119–121].

5.2.4 Programming Interface

DeltaFS IMDs are designed such that they can be accessed like a regular filesystem directory. Providing a filesystem-like programming interface makes it easier for us to integrate with different user applications. The directory is designed to be opened either for reading or writing. When opened for writing, the directory operates as an in-situ data processing pipeline as described in Section 5.2.2. Each file write is turned into a KV pair sent to the pipeline. The name of the file serves as key. The data of the file serves as value. The final data streamed out of the pipeline is written to an underlying storage container representing the directory.

When opened for reading, the directory operates as a serial query processor. Each file read becomes a query keyed by the name of the file. The directory processes the query as described in Section 5.2.3, and returns data as if it was read from a physical filesystem file. We expect most reads to be sequential fetches of entire files, so each read reads all data of a key.

5.3 Challenges and Techniques

The key concept behind DeltaFS IMDs is the use of idle CPU cycles available on the compute nodes of an application to perform data computation. While the potential to process data across a large number of idle compute nodes

enables us to carry out massive computation, scaling an embedded data computation service within a parallel data application can be drastically different from scaling a traditional storage service:

First, traditional storage software may use all of a machine's memory to achieve scaling whereas an embedded data service can only use as much memory as the application can live with.

Second, traditional storage software is able to take advantage of ownership of resources to schedule as much work as possible in the background whereas an embedded data service must avoid impacting application performance by scavenging only idle resources and processing data as optimally as possible.

In this section, we present the techniques that DeltaFS IMDs use for fast online data partitioning and indexing. To address the challenges of embedded in-situ data processing, DeltaFS IMDs index data in a single pass (Section 5.3.1), efficiently partition data across the processes of a parallel application (Section 5.3.2), and frugally use memory for all-to-all data communication (Section 5.3.3).

5.3.1 Indexing Data In a Single Pass

A key component of our in-situ data processing pipeline is the online indexing of data at each data partition. While indexing data as it is written speeds up followup queries, it may also significantly increase the write time of an application making it less efficient overall. Therefore the first challenge of our work is to device an indexing mechanism that improves the query performance at the read phase while not slowing down the application at the write phase.

Must Index Data in One Pass! One way to structure data for fast reads is to sort it by key. This enables queries to quickly rule out regions in the storage that do not contain a key and directly jump to the data of interest. To dynamically sort data by key as data is written to storage, one uses a self-balancing data structure such as an LSM-Tree [58, 69].

Figure 5.6 shows the internal workings of a simplified LSM-Tree. An LSM-Tree is made of two on-disk components. One of the two components is

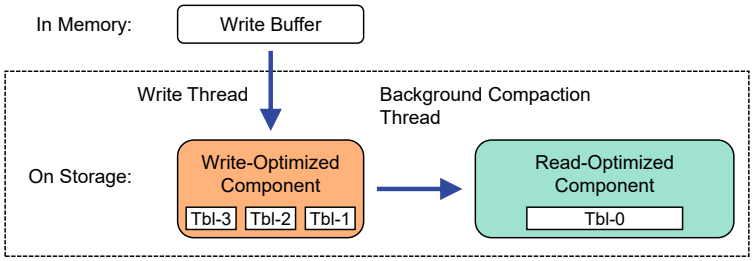


Figure 5.6: Illustration of a simplified LSM-Tree. An LSM-Tree consists of an in-memory write buffer, a write-optimized on-storage component made of a series of logged tables (Tbl's) of sorted KV pairs, and a read-optimized on-storage component consisting of a single large sorted table. Tables are sorted at the time they are written. Each table is sorted independently. User data is first written to the in-memory buffer space of the tree and is flushed to storage when the buffer is full. Each buffer flush writes a new table in the write-optimized on-storage component of the tree. Querying a key from an LSM-Tree requires performing searches on tables starting from the most recent table of the tree (Tbl-3 in the example) to the least recent table (Tbl-0). To reduce the number of table searches per query, a background compaction thread is run by the tree to asynchronously migrate data from the write-optimized component to the read-optimized component. Migration is done through merge-sorting tables of the two components. The best read performance is achieved when all data is merged into the read-optimized component so that all queries search no more than a single table. The cost of achieving fast reads, on the other hand, is the massive data rereads and rewrites needed to migrate data from the write-optimized component of the tree to the read-optimized component, which often require an order of magnitude more I/O than the initial writing of data to the write-optimized component.

write-optimized. It consists of a series of logged tables of KV pairs. Each table is independently sorted by key at the time it is logged to storage. The other on-disk component is read-optimized, consisting of a single large table of sorted KV pairs. During writing, user data is first staged at an in-memory buffer space of an LSM-Tree. When the buffer is full, the data in the buffer will be sorted and then logged to the write-optimized on-disk component of the tree as a new table. A mapping structure is maintained to record the storage locations of all tables in a tree.

During reads, a reader process uses the indexing information in the mapping structure to locate tables and performs searches in the reverse order of time (from the most recent table to the least recent). In the worst case, a reader process will have to search all tables in a tree in order to find the data of a key. To improve read performance, a separate compaction thread is run by the LSM-Tree in the background to migrate data from the write-optimized component to the read-optimized component. This is done by merging tables of the two components. The best read performance is achieved when all data is merged into the read-optimized component so that each query searches no more than a single table.

While LSM-Trees are widely used in modern computing systems for dynamically transforming data from a write-optimized format to a read-optimized format for fast reads, they are extremely expensive for online data indexing in the context of being embedded inside the write path of a parallel data application for streaming processing. First, the migration

of data from the write-optimized component of the LSM-Tree to the read-optimized component requires massive storage rereads and rewrites, which can significantly increase the run time of an application due to increased I/O and computation. Second, operating as client middleware embedded inside the processes of a parallel application prevents us from being able to efficiently hide the latency of background storage operations through asynchronously processing. When the background compaction activities cannot keep up with the foreground data operation, either one risks leaving data in a query-unfriendly format or the foreground data operation must be rate limited or blocked to experience the data processing delays.

To prevent such bottlenecks, we need a mechanism that is capable of having data reorganized and indexed in one pass and does not require merging data in the background.

Switching to Filters. The reason merging improves performance is that it reduces the number of places a reader has to search in order to find the data of a key. To improve read performance without it, we use filters [64]. Filters are a special type of data structure whose canonical use involves membership management. In these applications, one inserts keys into a set and then asks if a key is in the set. A filter returns False when a key is not in the set or True when the key may be in the set. Compared with an index, filters inform a reader of where *not* to look at rather than the storage locations worthy of reading.

Filters are effective for two reasons. First, filters are small compared with the data they filter so writing and storing them in addition to data does not introduce a significant I/O and storage overhead. Second, by creating a filter for each table, a reader process is able to leverage information in the filters to rule out tables that do not contain a key and only perform searches on the rest of the tree. The total number of tables a reader needs to search for a query is now bounded by the overall performance of the filters rather than the progress of background merging. The former can be improved through creating more computationally expensive filters whereas the latter is largely a function of the available storage bandwidth. Filters are a better approach when writing data to storage is primarily bottlenecked on storage and less on the CPU cycles on the writing nodes.

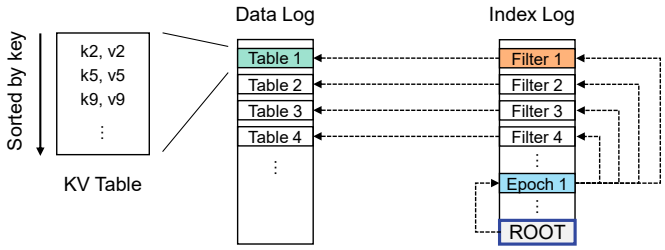


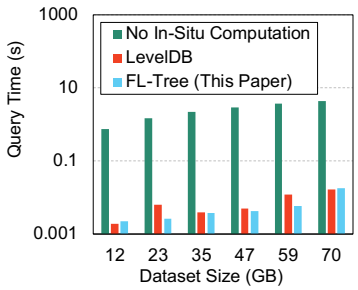
Figure 5.7: Illustration of the on-storage format of an FL-Tree consisting of a data log and an index log.

A Flattened LSM-Tree. We propose FL-Tree, a flattened LSM-Tree that transforms data in a single pass. We achieve this through aggressive filtering and being free of merging. The data structures of an FL-Tree consist of an in-memory write buffer and a series of logged KV tables on storage. Like LSM-Trees, writing data requires first staging the data in the in-memory buffer and then flushing it when the buffer is full. Each buffer flush writes a new table. Unlike LSM-Trees, FL-Trees do not rely upon merging to improve read performance. Instead, queries are made efficient primarily through filtering and an on-disk data format that minimizes the number of storage seeks per query. The former is achieved by creating a filter for each table. The latter is achieved by packing all filters in a single log file for efficient retrieval.

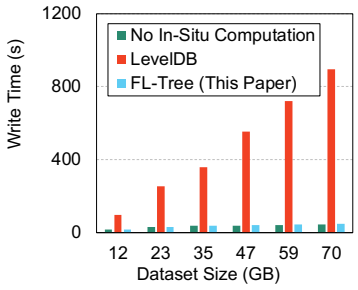
Figure 5.7 shows the on-disk format of an FL-Tree. It consists of two log files: a data log and an indexing log. The data log is a simple concatenation of all tables logged in the tree. The index log contains the filter for each table and two types of indexes: per-epoch indexes and a root index. A per-epoch index is appended to the index log at the end of each epoch (such as a timestep of a scientific simulation). It records the storage locations (log offsets) of all tables and their associated filters created during that epoch. The root index is appended to the index log at the conclusion of an application run. It records the total number of epochs and for each epoch the storage location of its per-epoch index.

Reading back a key from an FL-Tree is a three-step process. First, the reader program reads the index log of the tree. Next, the reader program uses the information in the index log to select and locate the tables of interest. Finally, the reader program reads and searches those tables to obtain the data of the key.

Measurements. To demonstrate the effectiveness of our design, we compare our techniques with the current state-of-the-art. We use 32 compute



(a) Average Latency of querying a KV pair. Performing in-situ data compaction as data is written drastically increases subsequent query performance. An FL-Tree's read performance is on par with LevelDB.



(b) Total Write Time (including both data insertion time and compaction time). An FL-Tree does not read back data from storage for compaction as data is written. It finishes writes almost as quickly as running no in-situ computation.

Figure 5.8: Read and write performance of an FL-Tree, LevelDB, and running no in-situ data compaction. a) Carefully laying out and packing data on storage allows an FL-Tree to answer queries almost as efficiently as the current state-of-the-art LevelDB. b) Free of compaction and background data merging allows an FL-Tree to absorb writes as efficiently as performing no in-situ data operations.

nodes. Each compute node consists of 32 CPU cores and 128GB RAM. Storage is provided through a remote parallel filesystem shared by all compute nodes. We developed a simple KV benchmark. It runs as a parallel program. We run a benchmark process on each CPU core. Each process produces a number of random KV pairs. We fix keys at 8 bytes and values at 40 bytes. These KV pairs are partitioned on-the-fly by the benchmark program and then indexed by a per-partition indexing mechanism. The resulting data is written to the remote parallel filesystem. We use LevelDB to represent the current state-of-the-art technique for per-partition data indexing [57, 60]. LevelDB is a general-purpose implementation of an LSM-Tree. It relies on background compaction (merging) to achieve good read performance. LevelDB is widely used by many storage systems we see today [26, 51, 78, 99, 122].

Our experiments compare FL-Tree with the LSM-Tree in LevelDB. Each our run consists of a write phase followed by a read phase. We focus on total write time and average query latency. To achieve good read performance, LevelDB runs LSM-Tree compaction (background data merging and reorganization) as data is inserted. Our total write time includes both data insertion time and data compaction time. LevelDB performs compaction in parallel with data insertion. The total write time we report does not double count the overlapped portion of time. In addition to LevelDB, our experiments also include a configuration where we directly write data to storage without performing any in-situ data computation.

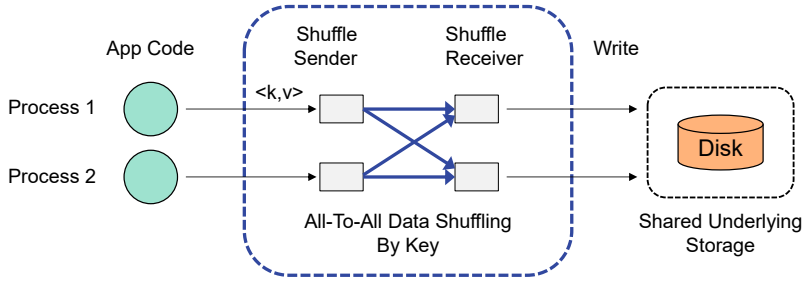


Figure 5.9: Illustration of performing online data partitioning across the processes of a parallel data application while these processes simultaneously write data to storage. The goal of data partitioning is to send each key to its home data partition. Data partitions are spread across all application processes. Each application process is responsible for a data partition, which maps to a disjoint range of keys. As these processes write data to storage, data not belonging to the local process is sent over the network to the remote process responsible for the key. Each process may send data to each other process, forming an all-to-all data shuffling process. The best performance is achieved when partitioning data does not slow down writes so that the overall writing process continues to be blocked on storage (rather than becoming bottlenecked on the added data partitioning process).

Results. Figure 5.8a shows average query latency as a function of total data size. All our queries read data from the underlying parallel filesystem. We run 100 queries per configuration. Each query starts with a cold cache and targets a random key. Without any in-situ processing at the write phase, querying a key requires scanning all data. Results show that latency can be high even when data is read back in parallel (our experiments used all the CPU cores to scan data in parallel). By partitioning and indexing data on the fly as data is written, both LevelDB and FL-Tree runs have data dynamically optimized for fast reads. They both exhibit good read performance. An FL-Tree is able to answer queries almost as efficiently as the LSM-Tree in LevelDB. This is because FL-Trees carefully lay out and pack data on storage so that data can be efficiently queried without performing a large number of storage seeks, and without requiring sorting at the write phase.

Figure 5.8b shows the total write time of each configuration. FL-Trees index data on the fly without reading back data from storage for merging. So they finish writes almost as quickly as running no indexing at all. LevelDB uses compaction to keep reads efficient. Performing compaction as data is written requires repeatedly reading back data from storage for sorting. This significantly increases an application's total write time when compaction cost cannot be hidden by asynchronous processing using dedicated resources.

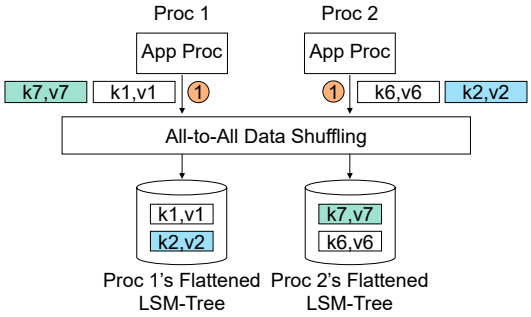
5.3.2 Efficiently Partitioning Data Over the Network

DeltaFS IMDs run inside the processes of a parallel data application and perform in-situ computation on data as the application writes it to storage. In addition to online data indexing, another key component of DeltaFS IMDs is the online partitioning of data, as we illustrate in Figure 5.9.

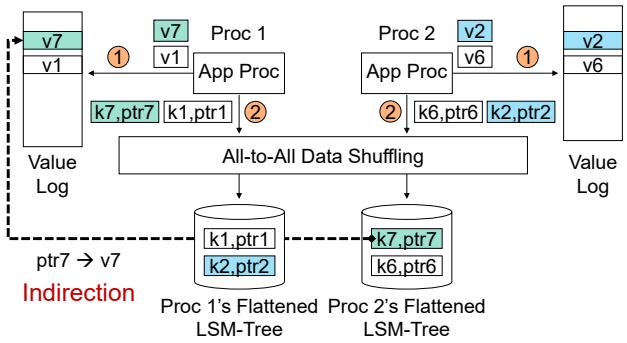
Online data partitioning requires transferring data over the network so that data can be processed by the partition responsible for the key. We expect that the bulk of our applications to be high entropy such that all data needs to be sent to a remote partition during data partitioning. However, when the interconnection network bandwidth of the parallel application does not dominate the bandwidth of storage, performing online data partitioning during application writes could drastically slow down the writes, making it prohibitive due to increased write time. Thus the second challenge of our work is to be able to efficiently partition data over the network such that the overall data partitioning overhead remains low even when the network-to-storage bandwidth ratio decreases.

DeltaFS IMDs model data as KV pairs. To efficiently move KV pairs over the network, a trivial optimization is to batch multiple KV pairs within the payload of one RPC. Assuming that RPC size is fixed at a large number (e.g., 32KB), online data partitioning efficiency then depends on the amount of data exchanged. We show a novel data shuffling mechanism that drastically reduces the amount of network traffic needed for online KV partitioning, making the process significantly less subject to network performance.

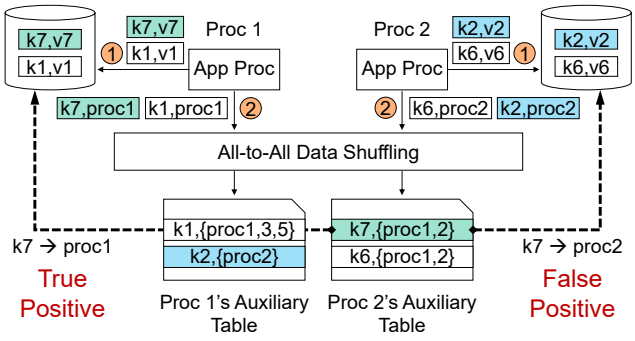
Simple Data Indirection Doesn't Work. The current state-of-the-art uses data indirection to reduce KV movement when directly moving KV pairs is too costly [26, 123]. With data indirection, one moves keys with pointers to values rather than moving complete KV pairs. Figure 5.10 shows all-to-all data shuffling both before (Figure 5.10a) and after (Figure 5.10b) applying data indirection. To shuffle data with indirection, an application process writes the value component of a KV pair to a per-process log file. We call this log file a Value Log. Next, the offset of the write and the ID of the process is encoded into a pointer. The process then sends the key with the pointer (offset + process ID) to the partition to which the key belongs. Thus instead of receiving a KV pair, the destination partition receives a Key-Pointer (KP) pair. This KP pair is then inserted into the FL-Tree of that partition. To recall a KV pair, a reader program first retrieves the



(a) The Base scheme shuffles intact KV pairs. Shuffle overhead may be high on computing platforms with less capable networks.



(b) The Current State-of-the-Art shuffles keys with pointers to values reducing network traffic. But pointers add significant I/O overhead (12 Bytes/Key) when value size is small (e.g., less than 64 Bytes).



(c) Our Lossy Format stores partial pointers. Each pointer may point to one or more data locations (e.g., k7 is mapped to both process 1 and 2). Note that not all processes or keys are shown in the figure (only process 1 and 2 are drawn). Our scheme reduces write cost while only slightly slowing down reads.

Figure 5.10: Illustration of 3 different data partitioning schemes. a) The base format shuffles intact KV pairs so potentially lots of data is exchanged over the network. b) By shuffling keys with only pointers to values, simple indirection (the current state-of-the-art) moves less data over the network but storing pointers in addition to values adds a significant amount of I/O to storage when value size is small. c) Our scheme encodes pointers in a lossy format so storing pointers requires transferring fewer bits to storage. Our technique reduces write overhead while still allowing KV pairs to be efficiently queried.

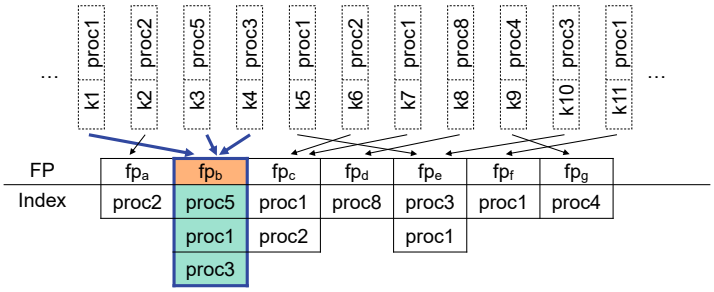


Figure 5.11: Auxiliary Table Design consisting of a fingerprint (FP) layer and an index layer. These two layers work hand in hand to map keys to their source processes. Raw keys (k1, k2, k3, ...) are lossily converted to tiny hash fingerprints (fp_a, fp_b, fp_c) before they are inserted into the fingerprint layer. Due to hash collisions, it is possible for multiple keys to be stored as one fingerprint. When this happens, the source processes of these keys will be listed at the index layer under that fingerprint. In this example, <k1, proc1>, <k3, proc5>, and <k4, proc3> are clustered under fp_b, and stored as <fp_b, {proc5, proc1, proc3}>. Each query to k1, k3, or k4 will return {proc5, proc1, proc3} resulting in false positives.

corresponding KP pair, and then dereferences the pointer to read back the value of the key.

The advantage of sending KP instead of KV pairs is a reduction in the amount of data exchanged over the network. However, storing pointers in addition to the original KV data increases total data size and has the disadvantage of increasing an application's total I/O time. While this overhead is negligible when the size of pointers is dwarfed by the size of data, this is not always the case. Values smaller than 250 bytes are reported to be the norm for Facebook's Memcached [124]. It is also common for scientific applications to output objects that are smaller than 50 bytes [116, 125]. In these cases, applying indirection may end up adding more overhead to storage (in the form of increased I/O time) than is removed from the network. To more efficiently apply data indirection when value size is small, we have developed LossyKV.

The LossyKV Data Shuffling Scheme. The key to improving performance beyond the current state-of-the-art is to make pointers less costly when value size is small. Recall from Figure 5.10b that with simple data indirection only KP pairs are shuffled across the network. Values are directly written to per-process value logs. Recovering a KV pair requires first retrieving the corresponding KP pair and then using the pointer to read back the value. Each pointer identifies the value log to which the value is written and the offset in the log file where the value resides. To achieve so, pointers typically add a 12-byte I/O and storage overhead per key, with each pointer consisting of a 4-byte file ID and an 8-byte file offset. Our goal is to significantly reduce this overhead while still facilitating fast reads.

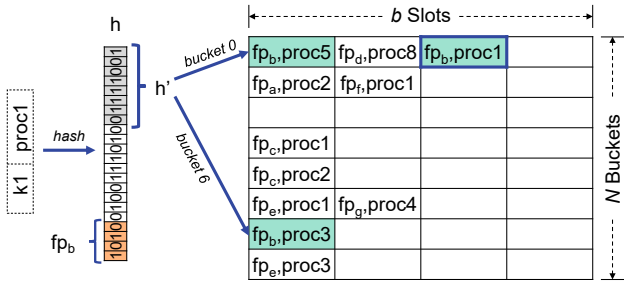


Figure 5.12: Auxiliary Table Implementation using Partial-Key Cuckoo Hash Tables. A partial-key cuckoo hash table consists of a number of buckets (currently sized at 6 in this example). Each bucket holds up to b data slots ($b = 4$ in this example). Each key is mapped to m buckets ($m = 2$ in this example) and can be stored at any of the empty slots in either of the m buckets. Each slot stores a key's fingerprint (partial-key) and its source process ID. k1 is mapped to bucket 0 and 5. Because different keys may share fingerprints, a key can be mapped to multiple source processes. In this example, k1 is fingerprinted as fp_b and is mapped to proc1, 3, and 5.

Our approach, named LossyKV, uses lossy pointers to reduce pointer overhead. A pointer is said to be lossy when its interpretation may match multiple data locations. This is as opposed to the usual case in which pointers are encoded with exact location information. Reducing pointer accuracy enables us to store pointers using fewer amounts of bits reducing their I/O and storage cost.

Figure 5.10c depicts how LossyKV works. LossyKV shuffles Key-ID (KID) pairs, rather than KV or KP pairs. When shuffling a KV pair, an application process first writes the intact KV pair to the process's own FL-Tree. It then sends another copy of the key along with its process ID (a KID pair) to the key's destination partition. Each KID pair serves as a pointer mapping a key back to the process who wrote the key and the process's FL-Tree that stores the value of the key. After data partitioning, LossyKV produces two types of data structures. One is the FL-Tree storing intact KV pairs. The other, called Auxiliary Tables, stores KID pairs in a lossy way and serves as a lossy index mapping keys to their source FL-Trees. Because KID pairs are all-to-all shuffled according to their partitions, each auxiliary table indexes a disjoint range of keys.

Recovering a KV pair from a LossyKV is a two-step process. First, the source FL-Tree of the key is determined through a responsible auxiliary table. Second, the corresponding FL-Tree is read returning the value of the key. Since auxiliary tables store data lossily to reduce I/O and storage overhead, it is possible for a key to be mapped to multiple source FL-Trees. In these cases, a reader program searches all these trees until it finds the target key. As FL-Trees are packed with intact KV pairs, a reader knows when it hits a key.

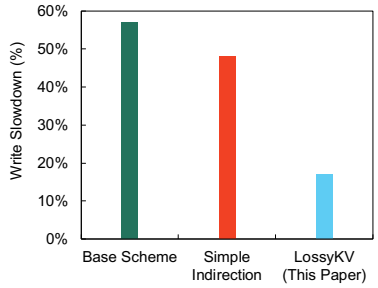
Achieving Lossiness. We use compact hash data structures to achieve lossiness. Figure 5.11 shows a logical view of our auxiliary tables. Each auxiliary table consists of a fingerprint layer and an index layer. The fingerprint layer is made up of a set of fingerprints. Fingerprints are partial keys; they store partial information about keys (such as a prefix of the hash of a key) rather than their full byte information. The index layer records the source processes that are mapped to each fingerprint (key). Multiple keys may be stored as one fingerprint. So each fingerprint may map to multiple source processes, with each of these source processes represents a key that is stored as the fingerprint.

A Cuckoo Hash Based Implementation. To implement our design, we use Partial-Key Cuckoo Hash Tables [126–128]. These are a cuckoo hash table variant that stores fingerprints of keys (partial-keys) instead of full keys [129, 130]. Figure 5.12 shows an example. Each partial-key cuckoo hash table consists of an array of buckets (the array size is 8 in our example). Each bucket holds b (fixed at 4 in our example) data slots.

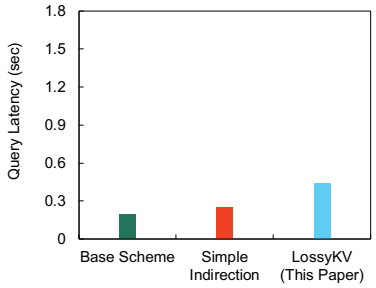
When a KV pair is inserted into a partial-key cuckoo hash table, the key is hashed into a partial key. The resulting partial KV pair is then assigned to m (2 in our example) candidate buckets in the table and can be placed at any of the empty slots in either of the buckets. When all such slots are taken, a random slot from one of the m buckets will be selected to hold the incoming key. The current resident of the slot will be evicted and then relocated to its alternative locations in the table. This relocation process continues recursively until an empty slot can be found, or fails after a large number (typically 500) of attempts and causes the table to be resized.

In practice, partial-key cuckoo hash table sizes are powers of 2, so each resize doubles the size of a table [127]. Mapping each key to $b \times m$ potential locations in the table allows for high levels of table space utilization before a table must be resized [63]. But because not all slots are necessarily filled after all data is inserted into the table, a partial-key cuckoo hash table may leak space in the data structure, leading to unnecessary memory and storage overhead.

To minimize such overhead, our implementation uses a side table when the primary one is full. For example, rather than resizing a 1-million-slot table to 2 million, our implementation combines a 1-million-slot table with an 128K-slot table to hold 1.1 million keys. This keeps space utilization at



(a) Performance of partitioning and indexing 32 billion keys. LossyKV reduces network traffic while simultaneously keeping I/O overhead low resulting in the best write performance.



(b) Performance of reading a KV pair from a 2TB dataset. LossyKV shows comparable performance while requiring reading back more data and performing more lookups due to indirection and lossiness.

Figure 5.13: Read and write performance of different online data shuffling mechanisms. We compare the base format where intact KV pairs are shuffled, current state-of-the-art with data indirection, and LossyKV with both data indirection and a compact lossy format for storing pointers. a) LossyKV reduces write slowdown by up to 3.3x. b) LossyKV only slightly increases query overhead (200ms per query).

about 95% in practice, while only slightly slowing down reads.

Measurements. We run experiments to evaluate the read and write performance of LossyKV. We use 64 compute nodes. Each compute node has 68 CPU cores and 96GB RAM. Reading and writing data is through a remote parallel filesystem. Performing RPC operations on these compute nodes is expected to be about 4x more expensive in latency and 3x in bandwidth compared with the compute nodes we tested in Section 5.3.1, making balancing network and I/O overhead and balancing read and write performance critical [131, 132].

Our experiments are driven by a parallel benchmark program. Each run consists of a write phase and a read phase. In the write phase, the benchmark program generates 32 billion keys. We fix keys at 8 bytes and values at 56 bytes. Data is on the fly shuffled across 4096 benchmark processes and then indexed using a FL-Tree at each process before written to storage. About 2TB of raw data is generated per run. Our read phase consists of 100 independent queries. Each query starts with a cold cache, and reads a random KV pair.

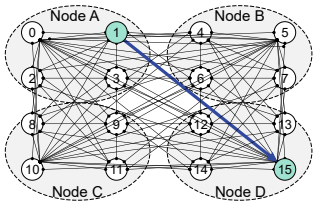
We compare performing all-to-all data shuffling using the base format, simple indirection (the current state-of-the-art), and LossyKV (our solution). We use Write Slowdown to gauge the total data partitioning overhead during the writing of data to storage. It is measured as the additional time each run must spend to finish writing all the data. We use median query latency to measure read performance.

Figure 5.13a compares the write slowdown of different online data shuffling schemes. Results show that LossyKV is able to reduce total write slowdown by 3.3x compared with the base format and by 2.8x compared with the current state-of-the-art. This is because LossyKV shuffles smaller KID pairs instead of KV or KP pairs (so less network traffic), and uses a lossy format to store pointers (so less I/O overhead). While the current state-of-the-art mechanism (simple indirection) also reduces network traffic, the increased I/O overhead caused by storing raw pointers makes this approach less effective overall.

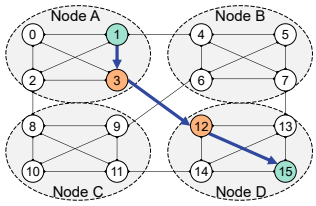
Figure 5.13b compares the median query latency of different schemes. The base format delivers the best read performance because reading a KV pair requires only searching one FL-Tree. The current state-of-the-art uses data indirection to reduce data movement within network. The cost of applying indirection is one extra read operation per query which increases its median query latency from 190ms to 250ms in these runs. Finally, with a compact lossy storage format for fast data shuffling, each LossyKV query must first read an entire auxiliary table (roughly 18MB each) and then attempt reads at multiple data partitions due to false positives (about 1.88 partitions per query in these runs). As such, LossyKV has the highest median read latency (440ms) among all three data management schemes. Though overall LossyKV shows comparable read performance with the base format and the current state-of-the-art, while being about 200ms slower.

5.3.3 Frugally Using Memory for All-to-All Data Communication

Efficiently exchanging data over network is essential to fast online data shuffling. But even with an efficient shuffle mechanism, performing data shuffling among hundreds of thousands of application processes can be challenging. Critically, direct N-N routed messages delayed for efficient transfer use too much memory for RPC writeback buffering. This excessive memory usage prevents us from running inside a parallel data application. Thus the third challenge of our work is to have a scalable communication mechanism that better bounds the memory needed for efficient all-to-all data shuffling.



(a) Direct N-N Routing (1-Hop). Each core talks with all other cores.



(b) 3-Hop Routing. Sending a message is done via up to 2 intermediate forwarder cores such that each core only talks with a subset of remote cores.

Figure 5.14: Illustration of different all-to-all data communication mechanisms. a) Direct N-N routed messages delayed for efficient network transfer use too much memory for RPC writeback buffering at scale. b) Routing messages in multiple hops drastically reduces per-core RPC destinations. Having each core serve as a partial representative load balances all cores and prevents representatives from becoming bottlenecks.

Direct N-N Routing Uses Too Much Memory. As a key component of our in-situ data computation, we need to efficiently support online data partitioning for both small- and large-scale data applications. Online data partitioning requires data to be all to all shuffled among the processes of a parallel application. Thus each application process communicates with each other process.

Unfortunately, even with today's fastest interconnection networks, the cost of large-scale all-to-all communication can be high if frequent communication consists of small payloads that prevent us from fully utilizing the network's bandwidth. To efficiently transfer data, one must buffer adequate data (e.g., 32KB) before sending it from one application process to another for all-to-all data shuffling. Efficiency is further improved when network operations are performed asynchronously so that concurrent data computation may be overlapped with network communication.

If all-to-all data communication were implemented by having each process directly send RPC messages to each other process (direct N-N routing), all processes would have to buffer data to be sent to all other processes. We assume running an application process on each CPU core. So direct N-N routing is effectively direct core-to-core communication. As Figure 5.14a illustrates, it directly follows that for large-scale application runs with hundreds of thousands of CPU cores the total size of RPC writeback buffers required per process for efficient use of the network will become unacceptable to applications with which we share memory. This makes direct N-N routing infeasible for in-situ data communication at scale.

Multi-Hop Routing. To restrict memory use in large-scale application runs, we have chosen to route messages via multiple hops. That is, we forward each RPC message through one or more intermediate application

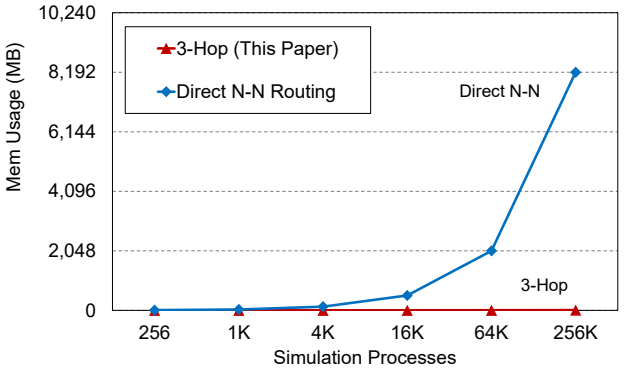


Figure 5.15: Projected memory usage for running a parallel application and performing all-to-all data shuffling among of processes of that application. Memory usage includes only the RPC writeback buffers an application process allocates for efficient transfer to other processes. It does not include system memory usage. The 3-hop line is close to zero is the figure.

processes before sending the message to its final destination. This is as opposed to directly sending each message to its final destination in one hop. With multi-hop routing, a process acts as a shuffle sender, a receiver, and an intermediate message forwarder simultaneously. Each process may forward some of its messages to other processes in the application for message delivery. Sharing and consolidating communication routes allows processes to directly communicate with only a small subset of their peers. Thus each process maintains fewer RPC writeback buffers, and these writeback buffers can be filled more quickly. This improves overall shuffle efficiency and better bounds the total amount of buffer memory needed at each process.

As Figure 5.14b shows, our current multi-hop routing implementation consists of 3 hops. To send a message our protocol first forwards the message to a local Representative process on the sender node, and then to a remote representative on the receiver node, which then forwards the message to the final destination. The inter-node communication step is bypassed if a message is aimed at a process on the same node. To reduce communication cost we expect all intra-node communication to be performed through shared memory. One problem of this approach is that the representative process on each node tends to become a bottleneck.

To prevent such bottlenecks, our implementation has each process on a node act as a representative for only a subset of the remote nodes. This reduces the connection state per representative, and distributes communication load more evenly among all local processes and CPU cores.

Figure 5.14b shows an example of 3-hop routing with 4 nodes and 16 shuffle processes. Each process is both a shuffle sender and a shuffle receiver. On each node, 3 processes are selected to act as the local representatives

for one remote node each. Thus, each process only needs to maintain 3 local connections and at most 1 remote connection. In contrast, direct communication would require that each process maintains 15 connections. In general, given M nodes and C cores per node 3-hop routing only requires $O(M/C)$ remote connections per process on average, while direct N-N routing would require $O(MC)$. We consider this C^2 reduction important, because it suggests that if the number of cores per node increases faster than the number of nodes in a cluster, the amount of required communication state is further reduced. We expect this to be the case in the future, as higher counts of lightweight or specialized cores become more widespread.

Case Study. The best way to demonstrate the scalability of 3-hop routing is to run it on a big machine. We do this in Section 5.4. Here, we show the memory saving of 3-hop routing compared with direct N-N routing. We consider running a parallel data application on the LANL's Trinity supercomputer, the one we will use in Section 5.4. Each Trinity compute node has 32 CPU cores ($C = 32$). We imagine running an application process on each CPU core and performing all-to-all data shuffling among the processes of that application. We assume a 32K RPC size so that an application process buffers 32K of data before it sends the data to a destination process. RPC writeback buffers are allocated independently at a source process for each destination process. Figure 5.15 projects the total amount of memory needed per process for RPC writeback buffering as a function of job sizes. Three-hop routing is able to bound memory usage at a lower level (<16MB) as the job size grows, whereas with direct N-N routing memory usage quickly rises to prohibitive.

5.4 End-to-End Evaluation

This section evaluates the end-to-end performance of DeltaFS IMDs. DeltaFS IMDs dynamically reorganize the output of a parallel writer application to speed up queries of a subsequent reader program. Our evaluation shows that DeltaFS IMDs can effectively accelerate reads while only slightly slowing down the writer application for streaming data reorganization.

In this section, we start with discussing our VPIC driver application in Section 5.4.1. We then describe our experiment and show results in Section 5.4.2.

5.4.1 Driver Application: VPIC

VPIC is a parallel particle code widely used for simulating kinetic plasmas [116]. In a VPIC simulation, each process manages a region of cells in the simulation space. These simulation processes track particles as they move through their corresponding cells. Every few timesteps the simulation stops and every simulation process writes a per-process file containing the state of all the particles currently managed by the process. State for each particle is 48 bytes.

Data analysis takes place after a simulation concludes. Our analysis involves reading back the trajectories of a tiny subset of particles. These particles exhibit unusual characteristics such as high energy which separates them from other particles. VPIC particles each have an unique ID. We configure VPIC to print the IDs of all particles to be queried at the end of a simulation.

Today, looking up a particle trajectory by its ID requires scanning up to an entire simulation output. This is because that there is no particular order in which particles are written during a simulation. Thus a particle can be written from different processes to different per-process files at different timesteps of a simulation. This, combined with a lack of a per-particle index, makes trajectory analysis prohibitively costly for large-scale VPIC simulations.

To demonstrate the effectiveness of DeltaFS IMDs, we use them to dynamically partition and index particles as the simulation writes them to storage. Our implementation leverages idle CPU cycles available during simulation I/O to perform data operations. Dynamically constructed per-particle indexes speed up queries while not significantly slowing down the writing of data during a simulation. For VPIC, retrieving the state of a particle at a specific timestep represents a point query. Retrieving the trajectory of a particle over a range of timesteps represents a small-range query. Small-range queries are more difficult. We focus on them in our experiments.

I/O Model. With a filesystem-like interface, it is easy for VPIC to use DeltaFS IMDs. To speed up particle queries, VPIC creates one IMD file for each particle it simulates, and appends all data of a particle to that particle's file. As discussed above, particle queries are known to be keyed on particle

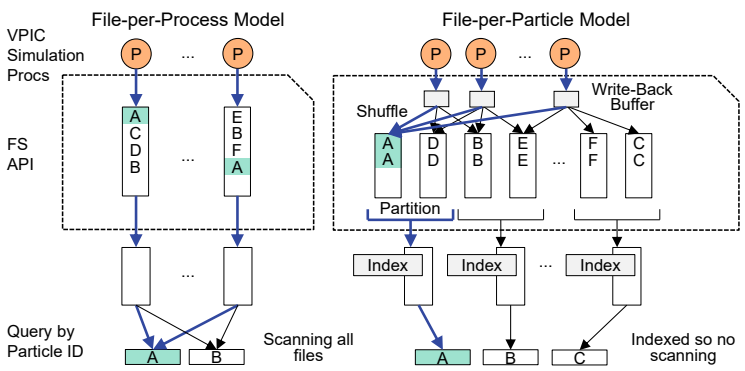


Figure 5.16: Comparison of the file-per-process model used by vanilla VPIC with the new file-per-particle model enabled by a DeltaFS IMD. Unmodified VPIC writes one file per process. To index VPIC particles with DeltaFS, we modify VPIC to write the state of each particle into a DeltaFS IMD using particle IDs as the filenames (A, B, C, ..., F). Dynamically created directory indexes keyed on filenames allow us to quickly retrieve per-particle information following a simulation. Critically, no massive data scans are expected. Indexed particle data is packed and stored by DeltaFS as large per-partition log objects in the underlying storage.

IDs. So VPIC uses particle ID to name all such files. To retrieve per-particle data, a reader program opens a DeltaFS IMD, and reads the corresponding particle file (using the ID of the particle as filename). Internally, the directory uses the indexes and partitions it creates during the write phase to quickly locate the data of the file. Both the indexes and the partitions are keyed on filenames making them capable of speeding up the data locating process. This process is transparent to the reader program. File data, which is opaque to the directory, is read by the reader program and interpreted by it as particle records.

Figure 5.16 compares the file-per-process model used by vanilla VPIC with the new file-per-particle model enabled by a DeltaFS IMD. Unmodified VPIC simulations write their output to an underlying filesystem using one file per process. Without pre-processing data before queries, retrieving the trajectory of a specific particle requires reading an entire simulation output (upwards of PBs of data). With DeltaFS IMDs, VPIC writes one file per particle. Each file represents a particle trajectory. These files are dynamically partitioned and indexed by the directory during simulation I/O so retrieving a file from a massive directory after a simulation requires reading mainly the indexes (typically only MBs of data) of one partition of the directory, followed by storage seeks that directly hit the file. No massive data readbacks are involved.

5.4.2 Experiment Design and Results

To measure performance, we run experiments on the LANL's Trinity supercomputer [32]. Each Trinity compute node has 128GB of DDR4 RAM and 32 Intel Xeon Haswell CPU cores with a base frequency of 2.3 GHz.

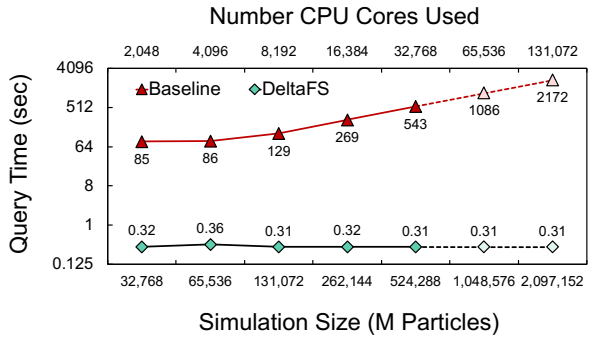
Our experiments consist of real VPIC simulation runs both with and without using DeltaFS IMDs for in-situ data processing. Each simulation run has one simulation process on each CPU core. For VPIC baseline runs, the simulation writes one output file per simulation process. For DeltaFS runs, the VPIC simulation writes into a DeltaFS IMD, with the directory dynamically partitioning and indexing the data, and writing the results as large per-process log objects. Our largest run simulated 2 trillion particles across 131,072 CPU cores.

Across all runs, simulation data is first written to a burst-buffer storage tier (made up of fast SSDs managed by the Cray's DataWarp software) and is later staged out to an underlying Lustre filesystem [9]. We keep the compute node to burst-buffer node ratio fixed at 32 to 1. Writing data from compute nodes to burst-buffer nodes is expected to be bottlenecked on the burst-buffer node's NIC bandwidth. Each burst-buffer node can absorb data at approximately 5.3GB per second. Our in-situ data techniques process data on the fly while fully utilizing available storage bandwidth minimizing write overhead.

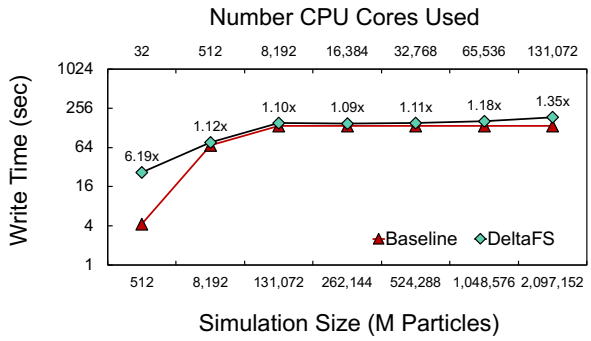
After each simulation, queries are executed directly from the underlying filesystem. Each query targets a random particle and reads all of its data. Particle data is written out over time as the simulation runs through timesteps. Each simulation is configured to output all particle data for 5 of those timesteps. Each our query therefore returns data from 5 distinct points in time. To retrieve the trajectory of a particle, the VPIC baseline reader reads an entire simulation output (thus time consuming) so all baseline queries are repeated up to 2 times. DeltaFS handles queries more efficiently. All DeltaFS queries are repeated 100 times, with each query starting with a cold data cache. We report the average query latency. DeltaFS uses a single CPU core to execute queries, whereas the baseline reader uses the number of simulation processes to read data in parallel.

Figure 5.17a shows the read performance. While the baseline reader used all the CPU cores to run queries, a single-core DeltaFS reader was still up-to 1,740x faster. This is because without an index for particles, the baseline reader reads all the particle data so its query latency is largely bounded by the underlying storage bandwidth. As DeltaFS builds indexes in-situ, it is able to quickly locate per-particle information after a simulation and maintain a low query latency (about 300ms in these experiments) as the simulation scales.

Figure 5.17b shows the I/O overhead DeltaFS adds to the simulation's I/O phases for building the data indexes. Part of the overhead comes from writing the indexes in addition to the original simulation output. The rest is due to the reduced I/O efficiency resulting from DeltaFS performing the in-situ indexing work. DeltaFS had large but decreasing overheads for the first 5 runs. This is because those jobs are not large enough to saturate the burst-buffer storage, so the system is dominated by the extra work DeltaFS performs to build the indexes. Starting from the sixth run the jobs began to bottleneck on the storage, and there is a modest DeltaFS slowdown of about 10%. For the last 2 runs, the job sizes are deliberately increased to demonstrate the performance at scale, and there is a slowdown of 20%-35%.



(a) Query Speedup of reading a VPIC trajectory from a 96TB dataset in the underlying storage. At 524,288 million particles, DeltaFS delivers a 1,740x speedup compared with the baseline. DeltaFS uses 1 CPU core to execute queries whereas the baseline uses all CPU cores (shown on the top of the figure).



(b) Overhead of performing in-situ data computation within a simulation application. Our techniques keep write overhead low (10% for medium runs, 20-30% for big runs) while efficiently transforming data to a read-optimized format, all while powered by only idle CPU cycles on the main computing platform.

Figure 5.17: Results from real VPIC simulation jobs on LANL's Trinity hardware. Our biggest job at the write phase used 4,096 compute nodes, 131,072 CPU cores, simulated 2 trillion particles, and wrote 96TB of data per timestep. Our biggest job at the read phase covered 524,288 million particles (jobs beyond that would require burning an excessive amount of computing hours on national lab resources). Results beyond that are projected reasonably. While our baseline VPIC reader used all the CPU cores to search particles in parallel, all DeltaFS queries were executed on a single CPU core.

5.5 Related Work

Filter data structures are used by many storage systems to improve read performance. Unlike indexes which directly map keys to data locations, filters speed up queries by indicating where not to read thus saving the query process from performing potentially a large number of unnecessary storage reads [133]. When the key space of an application is bounded, filters can be implemented using compressed bitmaps [134, 135]. When the key space is unbounded, filters are typically implemented through hash-based data structures such as the Bloom filter [64], cuckoo filter [126, 127], and quotient filter [136, 137]. Recently, we have also seen filters implemented using tries such as SuRF [138] and using perfect hash functions such as the ECT structure in SILT [126]. These filter implementations may also be used to implement LossyKV.

Many systems have proposed variants of LSM-Trees to improve performance. WiscKey [123] reduces the I/O amplification associated with compaction by storing keys and values separately and only performing compaction on the keys. Both Monkey [139] and SlimDB [128] use analytical models to generate optimized filter layouts that balance per-filter performance with available memory. LSM-Trie [140] uses an incremental compaction scheme [141] to reduce compaction overhead, and uses clustered indexes to improve query performance. VT-Tree [71] uses a customized compaction procedure that avoids re-sorting in-order data. The DeltaFS's custom LSM-Tree implementation presented in this paper is inspired by these reorganizations, particularly the LSM-Trie, though DeltaFS is primarily optimized for point and small-range queries.

DeltaFS employs a hash function to partition its data for efficient lookup. Hashing allows for efficient point queries, while maintaining fairly even loads across all partitions for arbitrary workloads. The tradeoff is that hashing places adjacent keys into distant buckets, and the data layout therefore has no locality. Therefore, certain queries such as range queries and prefix matching can not be supported efficiently. In practice, storage systems such as HyperDex [142] and HBase [143] reorganize data once written, to create an ordered storage layout. Parallel sorting algorithms such as SDS-Sort [144] also recreate order by making multiple passes over data once written. However, making multiple I/O passes can be extremely inefficient when dealing with large datasets and finite storage bandwidths – which is often the case with large simulations. Augmenting DeltaFS with

an in-situ partitioning capability that preserves locality, balances loads, and works with arbitrary distributions is something we're currently working on.

Rich in-transit data processing capabilities are provided by multiple middleware libraries such as PreDatA [145], GLEAN [146, 147], NESSIE [148], and DataSpaces [149]. These systems all use auxiliary nodes to provide analysis tasks. Similarly, systems such as Damaris [150] and Functional Partitioning [151] co-schedule analysis, visualization, and de-duplication tasks on compute nodes, but require dedicated cores. DeltaFS embeds indexing computation directly within the application processes and performs the processing during the application's regular output methods.

The GoldRush runtime [152] provides an embedded in-situ analytics capability by scheduling analysis tasks during idle periods in simulations using an OpenMP threaded runtime. The analysis tasks leverage the FlexIO [153] capability within ADIOS [154] to create shared memory channels for generating analysis tasks inputs to execute during idle periods of application execution. The embedded in-situ framework within DeltaFS instead co-schedules analysis tasks (i.e. partitioning and indexing) with the application's I/O output phase. While Goldrush is extremely effective at scavenging idle resources within the OpenMP runtime model, DeltaFS instead focuses on co-scheduling analysis tasks for single-threaded bulk-synchronous applications.

The SENSEI in-situ analysis framework [155] provides a generic library capable of running computationally efficient in-situ tasks on dedicated or shared resources. Their studies included instrumenting a variety of codes and mini-apps. Additionally, they concluded that most in-situ analysis tasks require little memory overhead. DeltaFS is able to use only 3% of the system memory to do effective latency hiding for in-situ operations even though the analysis requires shuffling and indexing the entire output dataset.

FastQuery [119, 120], a popular indexing and query library for scientific data, uses parallel, compressed bitmap indexes similar to the bitmap indexing described by FastBit [135], and has been deployed as part of in-situ indexing service to accelerate subsequent reads [156]. DeltaFS creates a similar compressed bitmap index following the shuffle phase to quickly filter data tables from within a partition. By customizing a bitmap index for partitioned particle data, DeltaFS is able to reduce the overall index size

and reduce storage overhead.

The distributed data partitioning and indexing capability of MDHIM [122] is similar to that of DeltaFS, though there are several key differences. First, DeltaFS uses an LSM-Tree that is more optimized for small value retrieval and in-situ scenarios. Second, DeltaFS uses a POSIX-like file system abstraction while MDHIM uses a key-value store abstraction. Finally, MDHIM relies on MPI for inter-process communication while DeltaFS uses Mercury RPC [87, 157] to run seamlessly across platforms supporting different network transports [158–160]. The Mercury RPC layer allows DeltaFS to run seamlessly across platforms supporting MPI, TCP/IP [158], InfiniBand [159] and OpenFabrics Interfaces [160].

Byna et al. have published the largest petascale particle simulations using vpic [39–41]. With two trillion particles and 2000 time steps of simulation the authors produced 350 TBs of data (including checkpoints) and detail the series of optimizations required to use a single shared HDF5 file output model. Some of the difficulties encountered while analyzing the resulting particle outputs motivated the creation of the DeltaFS embedded in-situ indexing pipeline for vpic.

5.6 Summary

Not all storage bottlenecks are caused by slow media. On the contrary, one might have fast media but the storage is bottlenecked on the server carrying the media, with its application being blocked on it not fully utilizing its compute node resources. DeltaFS IMDs harvest idle compute, memory, and network resources on the compute nodes of an application to perform data computation, hiding server bottlenecks. One uses DeltaFS IMDs to dynamically reorganize the writes of a writer application, speeding up the queries of a followup reader program.

In this chapter, we described a set of techniques that enabled the scaling of DeltaFS IMDs to more than a hundred thousand processes. The lessons we learned designing and applying these techniques can be used to address scalability challenges in a variety of in-situ data computation middleware. We distinguish our techniques between those that provide improvements to the scalable shuffling of data and those improving the efficiency of data indexing.

Latency hiding and efficient bandwidth utilization are critical for scalable shuffling. Our analysis shows that careful buffer management is the key to keeping latency low and bandwidth high. Buffers must be large enough to make efficient use of the network without being so large as to waste memory. In our configuration 32KB buffers are sufficient, but we anticipate that larger buffers may be required with future more lightweight processor cores. To slow the increase in the number of buffers as the system scales we introduced our 3-hop all-to-all communication technique. By limiting the number of off-node connections per process, we believe the applicability of the 3-hop technique will increase for future computing platforms if intra-node parallelism increases faster than inter-node parallelism.

An efficient data shuffling protocol that contracts network traffic is also essential to achieving scalable online data partitioning. We use indirection to reduce the total amount of data we need to send and receive over the network (when partitioning data) so that the overall data partitioning process becomes less subject to the hosting platform. We then strive to use the minimal amount of physical indexes to manage data indirection so that per-key overhead can be kept low and we can achieve good performance even when KV size is tiny. Critically, performing the latter distinguishes us from the current state-of-the-art that only exploits simple data indirection techniques.

Our indexing techniques demonstrate that on-storage data reorganization (e.g. LSM-Tree compaction) is not necessary if the dominant access regimes are point and small-range queries. In particular, clustered indexes can be efficiently constructed and accessed on modern computing platforms, and space-efficient KV filters are able to balance efficient searching with optimal storage system access.

For large-scale data indexing capabilities in particular, we believe in-situ indexing embedded within application provides a compelling advantage in its ability to scavenge temporarily available resources to improve the efficiency of post-hoc analysis. Although embedded in-situ processing introduces scalability challenges, we believe that these challenges are manageable. The techniques described in this paper demonstrate efficient scaling to a hundred thousand processes. We believe that additional techniques exist to improve embedded in-situ scaling even further. In addition to further scaling techniques, it is clear to us that improving the performance of queries when partitioning functions cannot provide an evenly balanced distribution is important to furthering the adoption

of our techniques. Support for multiple simultaneous indexes to enable multivariate analysis is also important to diverse types of scientific analysis. Adding this capability to our embedded in-situ pipeline will enable new classes of scientific applications to leverage DeltaFS.

6 Conclusion

It has been a tradition that, every once in a while, we stop and reassess the way we build filesystems. One previous effort was made by the NASD project [19], which urged people to stop coupling filesystem data communication with metadata management and instead use object storage devices for scalable data access. While the NASD principle has underpinned almost every modern parallel filesystem that we see today, this thesis informs people of what it takes to further advance parallel filesystem performance in order to better keep with up the rapidly increasing scale of today's massively-parallel computing environments.

Existing filesystem clients communicate too frequently with their servers. We showed how the relaxed consistency requirements of modern HPC workflows can be utilized to reduce filesystem metadata synchronization and serialization and how LSM-Tree data structures can be used to enable efficient logging and deferring of filesystem metadata changes. Existing filesystem metadata performance depends too much on dedicated server resources. We showed how filesystem namespace services can be dynamically instantiated on client nodes to achieve scalable performance and how inter-application communication can be efficiently implemented on top of immutable filesystem namespace snapshots with unrelated applications never having to communicate. Finally, modern scientific inquiry uses costly post-processing to ensure good read performance. We showed how data can be in-situ indexed for fast queries as an application writes it to storage and how this in-situ computation can be efficiently carried out through available compute resources on application compute nodes without requiring dedicated compute resources and without requiring post-processing.

Putting all of them together, we imagine future HPC storage to have no parallel filesystems as we see today. Instead, a simple object store serves as shared storage and on top it applications dynamically instantiate services for scalable filesystem namespace management and data processing. As a result, filesystem design and provisioning decisions can be separated from the overall design of a computing cluster leading to fewer bottlenecks. At the same time, applications running on a large computing platform can be

less restricted by the underlying storage to attain high performance.

Our work consists of the following key trace-offs. First, modern distributed filesystem metadata is optimized for interactive data accesses. Filesystem namespace updates made by one client are immediately synchronized and made visible to all clients in a computing cluster. Our work is primarily optimized for non-interactive batch workloads. Rather than immediately integrating every metadata change, our work delays it through deep client-side writeback caching until subsequent bulk integration for high metadata write performance. However, one cost of such a delay is the extra work an interactive reader application needs to pay for reading and merging information cached at clients and the increased complexity for handling errors out of the original filesystem call site when client cached metadata updates cannot later be applied. Second, today's parallel filesystems use dedicated servers for filesystem metadata management whereas our work advocates the use of dynamically instantiated client services for scalable filesystem metadata performance. While we have showed that shifting away from dedicated servers has many benefits, an important cost of it is the lack of a big, warmed-up cache for low-latency read accesses and the increased work an application may have to do for merging namespace logs potentially repeatedly. Finally, our Indexed Massive Directory work features a classic trace-off between reads and writes. In-situ indexing speeds up reads. The cost of it is the increased write time for computing and writing the indexes and the increased storage space for storing indexes in addition to data.

The idea of streaming indexing data as it is written is based on a model where data writeback buffered at application process memory can be dynamically indexed when the previous indexed buffer is written to storage. The key is to overlap indexing computation with storage writes such that an application process is always able to send the next write when the previous write returns provided that the client CPU core can process data faster than storage can absorb. As a result, storage is kept busy all the time and streaming data indexing overhead is reduced to minimum.

Streaming indexing data consists of first partitioning data across application process cores and then indexing data within each data partition. In practice, we find that the cost of data partitioning done through an all-to-all data shuffling process at scale can be significantly higher than per-partition data indexing. The reason is two-fold. First, direct all-to-all routed messages delayed for efficient network transfer will use too much application process memory for sender-side writeback buffering. To mitigate this problem,

multi-hop routing can be used to reduce per-process destinations at the cost of increased total network communication — each message is sent and received more than one times. Another reason all-to-all data shuffling is costly at scale is due to the total amount of data that must be transferred across the network. Multi-hop routing exacerbates this problem by sending each message multiple times. To mitigate this problem, data indirection can be used to transfer only data indexes across the network rather than the actual data at the cost of one extra data lookup at the read time and the extra storage space needed for storing the indexes. The size of the latter can be further reduced by using filter data structures with additional, additional read overhead.

At exascale and beyond, synchronization of anything global should be avoided, even if the changes needed to do so are drastic. Conventional parallel filesystems, with fully synchronous and consistent namespaces, mandate synchronization with file create and other filesystem metadata operations. This must 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. By shifting away from constant global synchronization, transforming dedicated filesystem metadata servers to per-job client software, and adopting a log-structured filesystem metadata representation for deep metadata writeback caching and merging, this thesis breaks established filesystem designs and identifies the set of changes that are needed for scalable parallel filesystem metadata performance. At the same time, by designing and implementing Indexed Massive Directories, this thesis demonstrates the promise of utilization of client-side storage writeback buffers and a potentially massive amount of client CPU cycles for a streaming data processing pipeline to unlock new ways of query acceleration, and shows how to perform such operations efficiently at scale. With the convergence of HPC and big data, the changes put forth in this thesis may prove necessary for other areas of computing as well.

Future Work

The ideas of this thesis may be extended in the follow directions.

Richer Object Storage Interface The idea of a richer object storage is not new. For example, PVFS discussed the possibility of having object

storage execute multiple data operations in a batch for better concurrency control and atomicity guarantees. The Google Filesystem (version 2) also used rich semantics at the object storage levels to ease file block management. With filesystem metadata completely served through transient client services, there is new room for the underlying storage to better serve a distributed filesystem. Critically, DeltaFS requires the underlying storage to provide reference counting to support garbage collection. Our current implementation assumed a standard object storage interface and had the clients to perform reference counting in namespace manifest logs. It is more secure, efficient, and less error-prone for the underlying storage to take over such responsibility and provide scalable reference counting for parallel filesystem metadata management.

Better Scheduling and Parameter Tuning for Filesystem Metadata. With filesystem metadata no longer managed through the HPC platform, applications take responsibility for tuning filesystem parameters and arrange compaction and other filesystem maintenance activities to attain high performance. While best practice and field knowledge may be communicated through filesystem manuals, per-site documentation, and rumors among application programmers, it is necessary to have an analytic model to better guide programmers to make the best decision.

Range Queries in IMDs. Our current Indexed Massive Directories (IMDs) implementation works only for queries against a single key. This is because that we use hashing to partition data across application process cores and to ensure that each query only searches data within one data partition. To extend the capability of IMDs to range queries and iterator functions, it is important that IMDs are paired with an ordered key-value pair abstraction. That is, keys are no longer converted to hashes before partitioning. Instead, keys are directly partitioned and partitioning them requires assigning key ranges to data partitions. Partitioning ordered keys might be done efficiently when the distribution of keys can be known beforehand. When this is not the case, we imagine a streaming process where key distribution is dynamically sampled in mini-phases and per-partition key ranges are dynamically adjusted on an as-needed basis.

Securely Executing User Functions at Shared Storage. In this work, we have focused on utilizing client compute resources for scalable filesys-

tem namespace management and scientific query acceleration. While dynamically instantiating storage functions on client nodes allows applications to efficiently group a large amount of compute resources to quickly absorb bursts of storage workloads, it is also critical that compute resources near data at rest, especially in the form of computational storage, are utilized to perform data operations. We imagine two forms of compute in which these backend compute resources are utilized by user applications. First, we imagine a domain language that application programmers can use to express their data processing needs and can submit their computation for direct, secure execution at the shared storage end. One example of such a mechanism is the Berkeley Packet Filter (BPF) technology where application logic is securely executed by the operating system for network package filtering and notification. Another form of such computation is to group storage resources as pools and then enable applications to allocate resource pools for data computation. Today, we already see solutions where pooled burst-buffer storage resources can be dynamically allocated by applications for increased I/O bandwidth. We imagine future technologies to extend this capability to enable applications to allocate not only storage space and bandwidth but the compute resources associated with the storage as well.

Bibliography

[1] Richard M. Russell. "The CRAY-1 Computer System." In: *Commun. ACM* 21.1 (Jan. 1978), pp. 63–72. ISSN: 0001-0782. DOI: 10.1145/359327.359336 (cited on page 3).

[2] *TOP 500*. https://www.top500.org/lists/top500/2020/11/. 2020 (cited on page 3).

[3] *LANL ASC Platforms*. https://www.lanl.gov/asc/platforms/index.php/ (cited on page 3).

[4] Jeffrey S. Vetter. *Contemporary High Performance Computing: From Petascale toward Exascale*. Vol. 1-2-3. CRC Press, 2019 (cited on page 3).

[5] W. Daniel Hillis and Lewis W. Tucker. "The CM-5 Connection Machine: A Scalable Supercomputer." In: *Commun. ACM* 36.11 (Nov. 1993), pp. 31–40. ISSN: 0001-0782. DOI: 10.1145/163359.163361 (cited on page 3).

[6] Wittawat Tantisiriroj et al. "On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS." In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*. 2011, 67:1–67:12. DOI: 10.1145/2063384.2063474 (cited on page 3).

[7] Frank B. Schmuck and Roger L. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*. 2002, pp. 231–244 (cited on pages 3, 5, 10, 29, 30, 33, 43, 45, 49).

[8] Brent Welch et al. "Scalable Performance of the Panasas Parallel File System." In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08)*. 2008, 2:1–2:17 (cited on pages 3–5, 11, 28, 29, 33, 46, 48, 49).

[9] Philip Schwan. "Lustre: Building a File System for 1000-Node Clusters." In: *Proceedings of the 2003 Linux Symposium*. 2003, pp. 380–386 (cited on pages 3, 5, 29, 45, 94).

[10] John Bent, Brad Settlemyer, and Gary Grider. "Serving Data to the Lunatic Fringe: The Evolution of HPC Storage." In: *USENIX ;login:* 41.2 (June 2016) (cited on pages 3, 4, 11, 46, 47, 70).

[11] R. A. Coyne, H. Hulen, and R. Watson. "The High Performance Storage System." In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (SC 93)*. 1993, pp. 83–92. DOI: 10.1145/169627.169662 (cited on page 3).

[12] Jeff Inman et al. "MarFS, a Near-POSIX Interface to Cloud Objects." In: *USENIX ;login:* 42.1 (Jan. 2017) (cited on pages 3, 47).

[13] LANL, NERSC, SNL. *Crossroads Workflows*. https://www.lanl.gov/projects/crossroads/__internal/__blocks/xroads-workflows.pdf. July 2018 (cited on pages 4, 6, 11, 14, 46, 70).

[14] John W. Young. "A First Order Approximation to the Optimum Checkpoint Interval." In: *Commun. ACM* 17.9 (Sept. 1974), pp. 530–531. ISSN: 0001-0782. DOI: 10.1145/361147.361115 (cited on page 4).

[15] N. El-Sayed and B. Schroeder. "Checkpoint/restart in practice: When 'simple is better'." In: *Proceedings of the 2014 IEEE International Conference on Cluster Computing (CLUSTER 14)*. 2014, pp. 84–92. DOI: 10.1109/CLUSTER.2014.6968777 (cited on page 4).

[16] David A. Patterson, Garth Gibson, and Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." In: *SIGMOD Rec.* 17.3 (June 1988), pp. 109–116. ISSN: 0163-5808. DOI: 10.1145/971701.50214 (cited on page 4).

[17] B. Schroeder and G. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems." In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (Oct. 2010), pp. 337–350. DOI: 10.1109/TDSC.2009.4 (cited on page 4).

[18] Sarp Oral et al. "Best Practices and Lessons Learned from Deploying and Operating Large-scale Data-centric Parallel File Systems." In: *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*. 2014, pp. 217–228. DOI: 10.1109/SC.2014.23 (cited on page 4).

[19] Garth A. Gibson et al. "A Cost-effective, High-bandwidth Storage Architecture." In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 98)*. 1998, pp. 92–103. DOI: 10.1145/291069.291029 (cited on pages 4, 11, 33, 45, 102).

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System." In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)*. 2003, pp. 29–43. DOI: 10.1145/945445.945450 (cited on pages 4, 33).

[21] K. Shvachko et al. "The Hadoop Distributed File System." In: *Proceedings of the 2010 International Conference on Massive Storage Systems and Technologies (MSST 10)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972 (cited on pages 4, 33, 41).

[22] R. J. Feiertag and E. I. Organick. "The Multics Input/Output System." In: *SIGOPS Oper. Syst. Rev.* 6.1/2 (June 1972), pp. 35–38. ISSN: 0163-5980. DOI: 10.1145/850614.850622 (cited on page 4).

[23] Dennis M. Ritchie and Ken Thompson. "The UNIX Time-Sharing System." In: *Commun. ACM* 17.7 (July 1974), pp. 365–375. ISSN: 0001-0782. DOI: 10.1145/361011.361061 (cited on pages 4, 5).

[24] Sage A. Weil et al. "Ceph: A Scalable, High-performance Distributed File System." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006, pp. 307–320 (cited on pages 4, 20, 29, 30, 33, 46, 48, 49, 56).

[25] 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 06)*. 2006, pp. 321–334 (cited on pages 4, 20, 29, 46, 48).

[26] Kai Ren et al. "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion." In: *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*. 2014, pp. 237–248. DOI: 10.1109/SC.2014.25 (cited on pages 4, 12, 14, 17, 18, 33–38, 41, 46, 48, 53, 80, 82).

[27] Lin Xiao et al. "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems." In: *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 15)*. 2015, pp. 236–249. DOI: 10.1145/2806777.2806844 (cited on pages 4, 14, 33, 46, 48, 53).

[28] Peter F. Corbett and Dror G. Feitelson. "The Vesta Parallel File System." In: *ACM Trans. Comput. Syst.* 14.3 (Aug. 1996), pp. 225–264. ISSN: 0734-2071. DOI: 10.1145/233557.233558 (cited on pages 4, 5, 53).

[29] Philip H. Carns et al. “PVFS: A Parallel File System for Linux Clusters.” In: *Proceedings of the 4th Annual Linux Showcase & Conference (ALS 00)*. 2000 (cited on pages 4, 5, 29, 33, 45).

[30] Feiyi Wang et al. *Understanding Lustre Filesystem Internals*. Tech. rep. ORNL/TM-2009/117. http://wiki.lustre.org/images/d/da/Understanding_Lustre_Filesystem_Internals.pdf. Oak Ridge National Laboratory, Apr. 2009 (cited on pages 10, 25, 29).

[31] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. “VAXcluster: A Closely-Coupled Distributed System.” In: *ACM Trans. Comput. Syst.* 4.2 (May 1986), pp. 130–146. ISSN: 0734-2071. DOI: 10.1145/214419.214421 (cited on pages 10, 49).

[32] LANL. *The Trinity Supercomputer*. http://www.lanl.gov/projects/trinity/. 2016 (cited on pages 10, 64, 93).

[33] Qing Zheng, Kai Ren, and Garth Gibson. “BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers.” In: *Proceedings of the 9th Parallel Data Storage Workshop (PDSW 14)*. 2014, pp. 1–6. DOI: 10.1109/PDSW.2014.7 (cited on pages 10, 17).

[34] Qing Zheng et al. “DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers.” In: *Proceedings of the 10th Parallel Data Storage Workshop (PDSW 15)*. 2015, pp. 1–6. DOI: 10.1145/2834976.2834984 (cited on pages 10, 17).

[35] Michael Stonebraker and Ugur Cetintemel. “"One Size Fits All": An Idea Whose Time Has Come and Gone.” In: *Proceedings of the 21st International Conference on Data Engineering (ICDE 05)*. 2005, pp. 2–11. DOI: 10.1109/ICDE.2005.1 (cited on page 10).

[36] *Exascale Computing Project (ECP)*. https://www.exascaleproject.org/ (cited on page 10).

[37] *LANL Crossroads*. https://www.lanl.gov/projects/crossroads/. 2018 (cited on page 10).

[38] *ANL Aurora*. https://www.alcf.anl.gov/alcf-aurora-2021-early-science-program-data-and-learning-call-proposals. 2018 (cited on page 10).

[39] Surendra Byna et al. “Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation.” In: *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 12)*. 2012, 59:1–59:12. DOI: 10.1109/SC.2012.92 (cited on pages 11, 67, 75, 99).

[40] Suren Byna et al. “Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper.” In: *Proceedings of the 2013 Cray User Group (CUG 2013)*. https://cug.org/proceedings/cug2013_proceedings/includes/files/pap107-file2.pdf. 2013 (cited on pages 11, 99).

[41] Suren Byna et al. “Tuning Parallel I/O on Blue Waters for Writing 10 Trillion Particles.” In: *Proceedings of the 2015 Cray User Group (CUG 2015)*. https://cug.org/proceedings/cug2015_proceedings/includes/files/pap120-file2.pdf. 2015 (cited on pages 11, 99).

[42] Samuel Lang et al. “I/O Performance Challenges at Leadership Scale.” In: *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. 2009, 40:1–40:12. DOI: 10.1145/1654059.1654100 (cited on pages 11, 36, 70).

[43] J. Bent et al. “Storage Challenges at Los Alamos National Lab.” In: *Proceedings of the 2012 International Conference on Massive Storage Systems and Technologies (MSST 12)*. 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232376 (cited on page 11).

[44] John Bent et al. “PLFS: A Checkpoint Filesystem for Parallel Applications.” In: *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. 2009, 21:1–21:12. DOI: 10.1145/1654059.1654081 (cited on pages 11, 25, 36, 43).

[45] Sadaf R. Alam et al. “Parallel I/O and the Metadata Wall.” In: *Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW 11)*. 2011, pp. 13–18. DOI: 10.1145/2159352.2159356 (cited on page 11).

[46] Tim Shaffer and Douglas Thain. “Taming Metadata Storms in Parallel Filesystems with MetaFS.” In: *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 17)*. 2017, pp. 25–30. DOI: 10.1145/3149393.3149401 (cited on page 11).

[47] J. Lofstead et al. “DAOS and Friends: A Proposal for an Exascale Storage System.” In: *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 16)*. 2016, pp. 585–596. DOI: 10.1109/SC.2016.49 (cited on page 11).

[48] Chen Luo and Michael J. Carey. “LSM-based storage techniques: a survey.” In: *The VLDB Journal* 29.1 (Jan. 2020), pp. 393–418. DOI: 10.1007/s00778-019-00555-y (cited on page 11).

[49] Mendel Rosenblum and John K. Ousterhout. “The Design and Implementation of a Log-structured File System.” In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52. ISSN: 0734-2071. DOI: 10.1145/146941.146943 (cited on page 11).

[50] S. Patil, K. Ren, and G. Gibson. “A Case for Scaling HPC Metadata Performance through De-specialization.” In: *Proceedings of the 7th Parallel Data Storage Workshop (PDSW 12)*. 2012, pp. 30–35. DOI: 10.1109/SC.Companion.2012.372 (cited on page 12).

[51] Kai Ren and Garth Gibson. “TABLEFS: Enhancing Metadata Efficiency in the Local File System.” In: *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 145–156 (cited on pages 12, 18, 20, 22, 30, 34, 38, 46, 80).

[52] LANL. *Handling trillions of supercomputer files just got simpler*. https://www.lanl.gov/discover/news-release-archive/2019/March/0314-handling-trillions-of-supercomuters.php. 2019 (cited on page 17).

[53] Qing Zheng et al. “Software-defined Storage for Fast Trajectory Queries Using a DeltaFS Indexed Massive Directory.” In: *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 17)*. 2017, pp. 7–12. DOI: 10.1145/3149393.3149398 (cited on page 17).

[54] Qing Zheng et al. “Scaling Embedded In-situ Indexing with DeltaFS.” In: *Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 18)*. 2018, 3:1–3:15. DOI: 10.1109/SC.2018.00006 (cited on page 17).

[55] Q. Zheng et al. “Compact Filters for Fast Online Data Partitioning.” In: *Proceedings of the 2019 IEEE International Conference on Cluster Computing (CLUSTER 19)*. 2019, pp. 1–12. DOI: 10.1109/CLUSTER.2019.8890992 (cited on page 17).

[56] Qing Zheng et al. “Streaming Data Reorganization at Scale with DeltaFS Indexed Massive Directories.” In: *ACM Trans. Storage* 16.4 (Sept. 2020). ISSN: 1553-3077. DOI: 10.1145/3415581 (cited on page 17).

[57] Google. *LevelDB*. https://github.com/google/leveldb/. 2013 (cited on pages 18, 38, 80).

[58] Patrick O'Neil et al. "The Log-structured Merge-tree (LSM-tree)." In: *Acta Inf.* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903. DOI: 10.1007/s002360050048 (cited on pages 18, 20, 21, 76).

[59] Peter Corbett et al. "Overview of the MPI-IO Parallel I/O Interface." In: *Input/Output in Parallel and Distributed Computer Systems*. Ed. by Ravi Jain, John Werth, and James C. Browne. Boston, MA: Springer US, 1996, pp. 127–146. ISBN: 978-1-4613-1401-1. DOI: 10.1007/978-1-4613-1401-1_5. URL: https://doi.org/10.1007/978-1-4613-1401-1_5 (cited on pages 19, 36).

[60] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006, pp. 205–218 (cited on pages 20, 21, 80).

[61] 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 11)*. 2011 (cited on pages 20, 21, 30, 33, 37, 46, 48).

[62] B. Welch and G. Noer. "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions." In: *Proceedings of the 2013 International Conference on Massive Storage Systems and Technologies (MSST 13)*. 2013, pp. 1–12. DOI: 10.1109/MSST.2013.6558449 (cited on page 20).

[63] M. Mitzenmacher. "The power of two choices in randomized load balancing." In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (Oct. 2001), pp. 1094–1104. DOI: 10.1109/71.963420 (cited on pages 20, 86).

[64] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors." In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692 (cited on pages 22, 78, 97).

[65] Atul Adya et al. "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment." In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. 2002, pp. 1–14 (cited on page 29).

[66] Konstantin V. Shvachko and Yuxiang Chen. "Scaling Namespace Operations with Giraffa File System." In: *USENIX ;login:* 47.2 (Apr. 2017) (cited on page 29).

[67] *Apache HBase*. https://hbase.apache.org/ (cited on pages 29, 31).

[68] *OrangeFS*. http://www.orangefs.org/ (cited on page 30).

[69] Russell Sears and Raghu Ramakrishnan. "bLSM: A General Purpose Log Structured Merge Tree." In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 12)*. 2012, pp. 217–228. DOI: 10.1145/2213836.2213862 (cited on pages 30, 76).

[70] Andy Twigg et al. "Stratified B-Trees and Versioned Dictionaries." In: *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage11)*. 2011, p. 10 (cited on page 30).

[71] Pradeep Shetty et al. "Building Workload-independent Storage with VT-trees." In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. 2013, pp. 17–30 (cited on pages 30, 97).

[72] John Esmet et al. "The TokuFS Streaming File System." In: *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 12)*. 2012 (cited on pages 30, 34, 46).

[73] Andrew W. Leung et al. “Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems.” In: *Proccedings of the 7th Conference on File and Storage Technologies (FAST 09)*. 2009, pp. 153–166 (cited on page 30).

[74] P. Carns et al. “Small-file access in parallel file systems.” In: *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 09)*. 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161029 (cited on page 30).

[75] Doug Beaver et al. “Finding a Needle in Haystack: Facebook’s Photo Storage.” In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI10)*. 2010, pp. 47–60 (cited on page 30).

[76] Adam Silberstein et al. “Efficient Bulk Insertion into a Distributed Ordered Table.” In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 08)*. 2008, pp. 765–778. DOI: 10.1145/1376616.1376693 (cited on page 31).

[77] Roshan Sumbaly et al. “Serving Large-Scale Batch Computed Data with Project Voldemort.” In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST12)*. 2012, p. 18 (cited on page 31).

[78] Siyang Li et al. “LocoFS: A Loosely-coupled Metadata Service for Distributed File Systems.” In: *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*. 2017, 4:1–4:12. DOI: 10.1145/3126908.3126928 (cited on pages 34, 46, 80).

[79] Salman Niazi et al. “HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases.” In: *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST 17)*. 2017, pp. 89–103 (cited on pages 34, 46).

[80] George C. Necula and Peter Lee. “Safe Kernel Extensions without Run-Time Checking.” In: *SIGOPS Oper. Syst. Rev.* 30.SI (Oct. 1996), pp. 229–243. ISSN: 0163-5980. DOI: 10.1145/248155.238781 (cited on pages 40, 44).

[81] Benjamin Braun et al. “Verifying Computations with State.” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 13)*. 2013, pp. 341–357. DOI: 10.1145/2517349.2522733 (cited on pages 40, 44).

[82] H. T. Kung and John T. Robinson. “On Optimistic Methods for Concurrency Control.” In: *ACM Trans. Database Syst.* 6.2 (June 1981), pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567 (cited on pages 40, 43).

[83] Austin T. Clements et al. “The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors.” In: *ACM Trans. Comput. Syst.* 32.4 (Jan. 2015). ISSN: 0734-2071. DOI: 10.1145/2699681 (cited on page 40).

[84] Murali Vilayannur et al. *Extending the POSIX I/O interface: a parallel file system perspective.* Tech. rep. Argonne National Lab, 2008 (cited on page 41).

[85] *IOR/mdtest*. https://github.com/hpc/ior (cited on page 41).

[86] Thomas E. Anderson et al. “Serverless Network File Systems.” In: *ACM Trans. Comput. Syst.* 14.1 (Feb. 1996), pp. 41–79. ISSN: 0734-2071. DOI: 10.1145/225535.225537 (cited on pages 43, 49).

[87] J. Soumagne et al. "Mercury: Enabling remote procedure call for high-performance computing." In: *Proceedings of the 2013 IEEE International Conference on Cluster Computing (CLUSTER 13)*. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702617 (cited on pages 43, 99).

[88] N. Ali et al. "Scalable I/O Forwarding Framework for High-performance Computing Systems." In: *Proceedings of the 2009 IEEE International Conference on Cluster Computing (CLUSTER 09)*. 2009, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289188 (cited on page 43).

[89] N. Liu et al. "On the Role of Burst Buffers in Leadership-Class Storage Systems." In: *Proceedings of the 2012 International Conference on Massive Storage Systems and Technologies (MSST 12)*. 2012, pp. 1–11. DOI: 10.1109/MSST.2012.6232369 (cited on page 43).

[90] James J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System." In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 3–25. ISSN: 0734-2071. DOI: 10.1145/146941.146942 (cited on page 43).

[91] D. B. Terry et al. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 172–182. ISSN: 0163-5980. DOI: 10.1145/224057.224070 (cited on page 43).

[92] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store." In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP 07)*. 2007, pp. 205–220. DOI: 10.1145/1294261.1294281 (cited on page 43).

[93] Philip Carns et al. "A Case for Optimistic Coordination in HPC Storage Systems." In: *Proceedings of the 7th Parallel Data Storage Workshop (PDSW 12)*. 2012, pp. 48–53. DOI: 10.1109/SC.Companion.2012.19 (cited on page 44).

[94] P. Carns, R. Ross, and S. Lang. "Object storage semantics for replicated concurrent-writer file systems." In: *Proceedings of the 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS 10)*. 2010, pp. 1–10. DOI: 10.1109/CLUSTERWKSP.2010.5613095 (cited on page 44).

[95] S. Lang et al. "Interfaces for coordinated access in the file system." In: *Proceedings of the 2009 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS 09)*. 2009, pp. 1–9. DOI: 10.1109/CLUSTR.2009.5289152 (cited on page 44).

[96] Tal Garfinkel et al. "Terra: A Virtual Machine-Based Platform for Trusted Computing." In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 193–206. ISSN: 0163-5980. DOI: 10.1145/1165389.945464 (cited on page 44).

[97] Emin Gün Sirer et al. "Logical Attestation: An Authorization Architecture for Trustworthy Computing." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*. 2011, pp. 249–264. DOI: 10.1145/2043556.2043580 (cited on page 44).

[98] Tirthak Patel et al. "Uncovering Access, Reuse, and Sharing Characteristics of I/O-Intensive Files on Large-Scale Production HPC Systems." In: *FAST*. 2020, pp. 91–101 (cited on page 46).

[99] Sage A. Weil et al. "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters." In: *Proceedings of the 2Nd International Workshop on Petascale Data Storage (PDSW 07)*. 2007, pp. 35–44. DOI: 10.1145/1374596.1374606 (cited on pages 47, 56, 80).

[100] Murthy Devarakonda et al. "Evaluation of Design Alternative for a Cluster File System." In: *Proceedings of the 1995 USENIX Annual Technical Conference (USENIX ATC 95)*. 1995 (cited on page 49).

[101] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. "Frangipani: A Scalable Distributed File System." In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP 97)*. 1997, pp. 224–237. DOI: 10.1145/268998.266694 (cited on page 49).

[102] Edward K. Lee and Chandramohan A. Thekkath. "Petal: Distributed Virtual Disks." In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. 1996, pp. 84–92. DOI: 10.1145/237090.237157 (cited on page 49).

[103] Zhenhan Liu, Xiaoxuan Meng, and Lu Xu. "Lock Management in Blue Whale File System." In: *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology Culture and Human (ICIS 09)*. 2009, pp. 1142–1147. DOI: 10.1145/1655925.1656133 (cited on page 49).

[104] Mike Burrows. "The Chubby Lock Service for Loosely-coupled Distributed Systems." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006, pp. 335–350 (cited on page 49).

[105] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010 (cited on page 49).

[106] K. W. Preslan et al. "A 64-bit, shared disk file system for Linux." In: *Proceedings of the 16th IEEE Symposium on Mass Storage Systems in cooperation with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies (MASS 99)*. 1999, pp. 22–41. DOI: 10.1109/MASS.1999.829973 (cited on page 49).

[107] Mark Fasheh. "OCFS2: The Oracle Clustered File System, Version 2." In: *Proceedings of the 2006 Linux Symposium*. 2006, pp. 289–302 (cited on page 49).

[108] Steven Whitehouse. "The GFS2 Filesystem." In: *Proceedings of the 2007 Linux Symposium*. 2007, pp. 253–260 (cited on page 49).

[109] Krishna Kunchithapadam et al. "Oracle Database Filesystem." In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 11)*. 2011, pp. 1149–1160. DOI: 10.1145/1989323.1989445 (cited on page 49).

[110] *ORNL Summit*. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. 2018 (cited on page 49).

[111] *LLNL Sierra*. https://hpc.llnl.gov/hardware/platforms/sierra/. 2018 (cited on page 49).

[112] Charles P. Wright et al. "Versatility and Unix Semantics in Namespace Unification." In: *ACM Trans. Storage* 2.1 (Feb. 2006), pp. 74–105. ISSN: 1553-3077. DOI: 10.1145/1138041.1138045 (cited on page 52).

[113] *OverlayFS*. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt (cited on page 52).

[114] S. A. Brandt et al. "Efficient Metadata Management in Large Distributed Storage Systems." In: *Proceedings of the 2003 International Conference on Massive Storage Systems and Technologies (MSST 03)*. 2003, pp. 290–298. DOI: 10.1109/MASS.2003.1194865 (cited on page 53).

[115] Sage A. Weil et al. "Dynamic Metadata Management for Petabyte-Scale File Systems." In: *Proceedings of the 2004 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 04)*. 2004, pp. 4–. DOI: 10.1109/SC.2004.22 (cited on page 56).

[116] K. J. Bowers et al. “Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation.” In: *Physics of Plasmas* 15.5 (2008), p. 7 (cited on pages 67, 84, 92).

[117] Peter J. Desnoyers and Prashant Shenoy. “Hyperion: High Volume Stream Archival for Retrospective Querying.” In: *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC 07)*. 2007 (cited on page 72).

[118] D. Bigelow et al. “Mahanaxar: Quality of service guarantees in high-bandwidth, real-time streaming data storage.” In: *Proceedings of the 2010 International Conference on Massive Storage Systems and Technologies (MSST 10)*. 2010, pp. 1–11. DOI: 10.1109/MSST.2010.5496975 (cited on page 72).

[119] J. Chou, K. Wu, and Prabhat. “FastQuery: A Parallel Indexing System for Scientific Data.” In: *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER 11)*. 2011, pp. 455–464. DOI: 10.1109/CLUSTER.2011.86 (cited on pages 75, 98).

[120] Jerry Chou et al. “Parallel Index and Query for Large Scale Data Analysis.” In: *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*. 2011, 30:1–30:11. DOI: 10.1145/2063384.2063424 (cited on pages 75, 98).

[121] Tiankai Tu et al. “Accelerating Parallel Analysis of Scientific Simulation Data via Zazen.” In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST 10)*. 2010 (cited on page 75).

[122] Hugh N. Greenberg, John Bent, and Gary Grider. “MDHIM: A Parallel Key/Value Framework for HPC.” In: *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 15)*. 2015 (cited on pages 80, 99).

[123] Lanyue Lu et al. “WiscKey: Separating Keys from Values in SSD-conscious Storage.” In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 133–148 (cited on pages 82, 97).

[124] Berk Atikoglu et al. “Workload Analysis of a Large-scale Key-value Store.” In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 12)*. 2012, pp. 53–64. DOI: 10.1145/2254756.2254766 (cited on page 84).

[125] Jacqueline H Chen et al. “Terascale direct numerical simulations of turbulent combustion using S3D.” In: *Computational Science & Discovery* 2.1 (2009) (cited on page 84).

[126] Hyeontaek Lim et al. “SILT: A Memory-efficient, High-performance Key-value Store.” In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*. 2011, pp. 1–13. DOI: 10.1145/2043556.2043558 (cited on pages 86, 97).

[127] Bin Fan et al. “Cuckoo Filter: Practically Better Than Bloom.” In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT 14)*. 2014, pp. 75–88. DOI: 10.1145/2674005.2674994 (cited on pages 86, 97).

[128] Kai Ren et al. “SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data.” In: *Proc. VLDB Endow.* 10.13 (Sept. 2017), pp. 2037–2048. ISSN: 2150-8097. DOI: 10.14778/3151106.3151108 (cited on pages 86, 97).

[129] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo Hashing.” In: *J. Algorithms* 51.2 (May 2004), pp. 122–144. ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.002 (cited on page 86).

[130] Xiaozhou Li et al. "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing." In: *Proceedings of the Ninth European Conference on Computer Systems (EuroSys 14)*. 2014, 27:1–27:14. DOI: 10.1145/2592798.2592820 (cited on page 86).

[131] Douglas Doerfler et al. "Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC." In: *Proceedings of the 2017 Cray User Group (CUG 2017)*. https://cug.org/proceedings/cug2017_proceedings/includes/files/pap117s2-file1.pdf. 2017 (cited on page 87).

[132] A. Sodani et al. "Knights Landing: Second-Generation Intel Xeon Phi Product." In: *IEEE Micro* 36.2 (Mar. 2016), pp. 34–46. DOI: 10.1109/MM.2016.25 (cited on page 87).

[133] Ashok Anand et al. "Cheap and Large CAMs for High Performance Data-intensive Networked Systems." In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI10)*. 2010 (cited on page 97).

[134] Jianguo Wang et al. "An Experimental Study of Bitmap Compression vs. Inverted List Compression." In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*. 2017, pp. 993–1008. DOI: 10.1145/3035918.3064007 (cited on page 97).

[135] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. "Optimizing Bitmap Indices with Efficient Compression." In: *ACM Trans. Database Syst.* 31.1 (Mar. 2006), pp. 1–38. ISSN: 0362-5915. DOI: 10.1145/1132863.1132864 (cited on pages 97, 98).

[136] Prashant Pandey et al. "A General-Purpose Counting Filter: Making Every Bit Count." In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*. 2017, pp. 775–787. DOI: 10.1145/3035918.3035963 (cited on page 97).

[137] Michael A. Bender et al. "Don'T Thrash: How to Cache Your Hash on Flash." In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1627–1637. ISSN: 2150-8097. DOI: 10.14778/2350229.2350275 (cited on page 97).

[138] Huanchen Zhang et al. "SuRF: Practical Range Query Filtering with Fast Succinct Tries." In: *Proceedings of the 2018 International Conference on Management of Data (SIGMOD 18)*. 2018, pp. 323–336. DOI: 10.1145/3183713.3196931 (cited on page 97).

[139] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. "Monkey: Optimal Navigable Key-Value Store." In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*. 2017, pp. 79–94. DOI: 10.1145/3035918.3064054 (cited on page 97).

[140] Xingbo Wu et al. "LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data." In: *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 71–82 (cited on page 97).

[141] H. V. Jagadish et al. "Incremental Organization for Data Recording and Warehousing." In: *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 97)*. 1997, pp. 16–25 (cited on page 97).

[142] Robert Escriva, Bernard Wong, and Emin Gün Sirer. "HyperDex: A Distributed, Searchable Key-Value Store." In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications Technologies Architectures and Protocols for Computer Communication (SIGCOMM 12)*. 2012, pp. 25–36. DOI: 10.1145/2342356.2342360 (cited on page 97).

[143] Tyler Harter et al. "Analysis of HDFS under HBase: A Facebook Messages Case Study." In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. 2014, pp. 199–212 (cited on page 97).

[144] Bin Dong, Surendra Byna, and Kesheng Wu. "SDS-Sort: Scalable Dynamic Skew-Aware Parallel Sorting." In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 16)*. 2016, pp. 57–68. DOI: 10.1145/2907294.2907300 (cited on page 97).

[145] F. Zheng et al. "PreDatA - preparatory data analytics on peta-scale machines." In: *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 10)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470454 (cited on page 98).

[146] V. Vishwanath, M. Hereld, and M. E. Papka. "Toward simulation-time data analysis and I/O acceleration on leadership-class systems." In: *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*. 2011, pp. 9–14. DOI: 10.1109/LDAV.2011.6092178 (cited on page 98).

[147] V. Vishwanath et al. "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems." In: *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*. 2011, pp. 1–11. DOI: 10.1145/2063384.2063409 (cited on page 98).

[148] Ron A. Oldfield et al. "Trilinos I/O Support Trios." In: *Sci. Program.* 20.2 (Apr. 2012), pp. 181–196. ISSN: 1058-9244. DOI: 10.1155/2012/842791 (cited on page 98).

[149] J. C. Bennett et al. "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis." In: *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 12)*. 2012, pp. 1–9. DOI: 10.1109/SC.2012.31 (cited on page 98).

[150] M. Dorier et al. "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O." In: *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER 12)*. 2012, pp. 155–163. DOI: 10.1109/CLUSTER.2012.26 (cited on page 98).

[151] M. Li et al. "Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures." In: *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 10)*. 2010, pp. 1–12. DOI: 10.1109/SC.2010.28 (cited on page 98).

[152] F. Zheng et al. "GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution." In: *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 13)*. 2013, pp. 1–12. DOI: 10.1145/2503210.2503279 (cited on page 98).

[153] F. Zheng et al. "FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics." In: *Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 13)*. 2013, pp. 320–331. DOI: 10.1109/IPDPS.2013.46 (cited on page 98).

[154] J. Lofstead et al. "Adaptable, metadata rich IO methods for portable high performance IO." In: *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 09)*. 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5161052 (cited on page 98).

[155] Utkarsh Ayachit et al. "Performance Analysis, Design Considerations, and Applications of Extreme-scale in Situ Infrastructures." In: *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 16)*. 2016, 79:1–79:12 (cited on page 98).

[156] J. Kim et al. "Parallel In Situ Indexing for Data-intensive Computing." In: *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*. 2011, pp. 65–72. DOI: 10.1109/LDAV.2011.6092319 (cited on page 98).

[157] Andrew D. Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls." In: *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP 83)*. 1983, pp. 3–. DOI: 10.1145/800217.806609 (cited on page 99).

[158] P. Carns et al. "BMI: a network abstraction layer for parallel I/O." In: *Proceedings of the 2005 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 05)*. 2005, pp. 1–8. DOI: 10.1109/IPDPS.2005.128 (cited on page 99).

[159] S. Atchley et al. "The Common Communication Interface (CCI)." In: *Proceedings of the 2011 IEEE Annual Symposium on High-Performance Interconnects (HOTI 11)*. 2011, pp. 51–60. DOI: 10.1109/HOTI.2011.17 (cited on page 99).

[160] P. Grun et al. "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency." In: *Proceedings of the 2015 IEEE Annual Symposium on High-Performance Interconnects (HOTI 15)*. 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19 (cited on page 99).