Streaming Data Reorganization at Scale with DeltaFS Indexed Massive Directories

QING ZHENG, CHARLES D. CRANOR, ANKUSH JAIN, GREGORY R. GANGER, GARTH A. GIBSON, and GEORGE AMVROSIADIS, Carnegie Mellon University, USA
BRADLEY W. SETTLEMYER and GARY GRIDER, Los Alamos National Laboratory, USA

Complex storage stacks providing data compression, indexing, and analytics help leverage the massive amounts of data generated today to derive insights. It is challenging to perform this computation, however, while fully utilizing the underlying storage media. This is because, while storage servers with large core counts are widely available, single-core performance and memory bandwidth per core grow slower than the core count per die. Computational storage offers a promising solution to this problem by utilizing dedicated compute resources along the storage processing path. We present DeltaFS Indexed Massive Directories (IMDs), a new approach to computational storage. DeltaFS IMDs harvest available (i.e., not dedicated) compute, memory, and network resources on the compute nodes of an application to perform computation on data. We demonstrate the efficiency of DeltaFS IMDs by using them to dynamically reorganize the output of a real-world simulation application across 131,072 CPU cores. DeltaFS IMDs speed up reads by 1,740× while only slightly slowing down the writing of data during simulation I/O for *in situ* data processing.

CCS Concepts: • **Information systems → Distributed storage**; **Stream management**; **Record and block layout**; **Point lookups**;

Additional Key Words and Phrases: *In situ* processing, computational storage

ACM Reference format:
Qing Zheng, Charles D. Cranor, Ankush Jain, Gregory R. Ganger, Garth A. Gibson, George Amvrosiadis, Bradley W. Settlemyer, and Gary Grider. 2020. Streaming Data Reorganization at Scale with DeltaFS Indexed Massive Directories. *ACM Trans. Storage* 16, 4, Article 23 (September 2020), 31 pages.
https://doi.org/10.1145/3415581

1 INTRODUCTION

Recent advances in solid-state storage media, high-speed interconnects, and systems software have greatly improved the ability of modern computing platforms to handle massive data. Nevertheless, applications leveraging sophisticated storage systems often experience performance far below

This manuscript has been approved for unlimited release and has been assigned LA-UR-20-26110. This work has been authored by an employee of Triad National Security, LLC, which operates Los Alamos National Laboratory under Contract No. 89233218CNA000001 with the U.S. Department of Energy/National Nuclear Security Administration. We thank the member companies of the PDL Consortium (Alibaba Group, Amazon, Datrium, Facebook, Google, Hewlett Packard Enterprise, Hitachi, Ltd., IBM Research, Intel Corporation, Microsoft Research, NetApp, Inc., Oracle Corporation, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, and Western Digital).
Authors' addresses: Q. Zheng, C. D. Cranor, A. Jain, G. R. Ganger, G. A. Gibson, and G. Amvrosiadis, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, USA; emails: {zhengq, chuck, ankushj, ganger, garth, gamvrosi}@cmu.edu; B. W. Settlemyer and G. Grider, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87544, USA; emails: {bws, ggrider}@lanl.gov.

1553-3077/2020/09-ART23 $15.00
https://doi.org/10.1145/3415581

what is provided by the underlying storage media and networks. Because of limited compute and memory resources on storage servers, complex storage software stacks providing data compression, indexing, and analytics are often unable to utilize all of the underlying device performance [1, 20]. While processors with large core counts are widely available, single task performance and memory bandwidth per task have grown more slowly than core counts.

To hide the processing delay associated with these rich software pipelines, storage systems featuring intensive background activities have emerged, designed to perform as many operations as possible asynchronously [61, 64, 77]. Asynchronous processing can effectively hide delays, but 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 storage server CPU cycles and memory resources are insufficient for the demands of the incoming workload, either the storage system must allow the full capabilities of the underlying storage media to go underutilized, or it must find a way to leverage the processing power and bandwidth of other parts of the system to make up the difference.

To accelerate storage operations, architectures that utilize additional compute resources along the storage processing path become attractive. Referred to as computational storage, these architectures improve performance by offloading data intensive operations to resources capable of matching the speed of the available storage media and the system's networking components. So far, computational storage research and practice have been concerned with the use of dedicated compute resources close to data at rest.

First, on-device computation allows data processing to occur nearest to the data. This is accomplished by reusing the controller and memory components of a device to perform data computation [7, 46, 52, 76]. Today, we call such a device a computational storage device (CSD) [6], and we have seen people leveraging these devices for compression [55, 106], query acceleration [36, 44, 92], data analytics [19, 29, 50, 83, 87], the implementation of an LSM-Tree [47, 54, 96], and the implementation of a key-value store [49, 51, 65]. In addition to on-device computation, another common form of computational storage involves the use of dedicated storage servers, or computational storage processors (CSPs) [6], to accelerate data processing. Examples include the Select capability in Amazon S3 [98], the SQL offload mechanisms in modern database storage engines [2, 3, 94], and recent Active Storage demonstrations based on parallel filesystems [72, 85], object-based storage servers [35, 79, 99], and the burst-buffer storage tier [15, 60] available on modern high-performance computing platforms [14, 27, 69].

In this article, we present a new approach to computational storage called DeltaFS Indexed Massive Directories (IMDs). Instead of utilizing dedicated compute resources that are coupled with storage, DeltaFS IMDs dynamically instantiate software storage services on compute nodes to process data and perform data computation on the fly without resorting to asynchronous processing to hide data processing delays. The core of each DeltaFS IMD is an *in situ* data computation mechanism that leverages idle compute, memory, and network resources of an application to perform data computation. DeltaFS IMDs use this mechanism to transform massive data from a write-optimized format to a read-optimized format as data flows from the client application to the backend storage servers. Our approach does not require dedicated accelerators within the storage, which may only be active when the storage system is under load. Instead, by leveraging temporarily idle resources on the compute nodes of a large computing platform, DeltaFS IMDs are able to aggregate potentially hundreds of thousands of CPU cores to perform data computation and better utilize the overall system. Our work can be viewed as a novel computational storage service (CSS) [6]. DeltaFS IMDs complement existing computational storage designs that enable the use of flexible offload mechanisms within the storage by also enabling the use of computational resources along the data processing path but outside the storage on an as-needed basis.

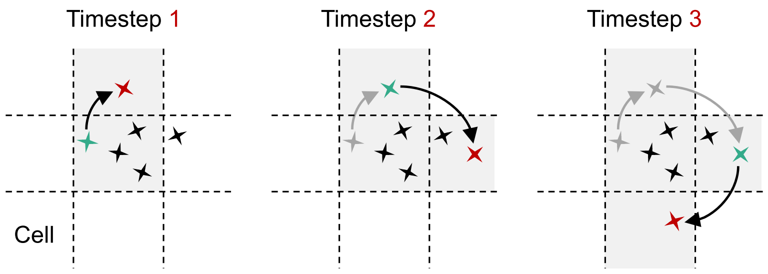


Fig. 1. Illustration of a typical VPIC particle simulation. Simulation space is divided into cells. Each VPIC process manages a cell. Tracing the state of a particle over time is non-trivial as particles move in an unpredictable way during a simulation and can be saved at different storage locations at different time.

We have implemented DeltaFS IMDs as a special directory type that runs on the clients of a distributed filesystem [104, 105]. To transform data to a read-optimized format, our implementation uses the popular key-value interface [33] to enable applications to define the data to be transformed. By combining our opportunistic approach to leveraging idle compute resources and a flexible key-value interface to perform storage operations, we demonstrate how computational storage can be integrated not just near storage devices, but throughout a distributed computing platform and how computational storage can make previously intractable data problems possible.

The rest of this article is structured as follows. Section 2 shows the background and the motivation behind this work. Section 3 presents the high-level design of our computational storage approach. Section 4 discusses challenges and a series of techniques for tackling these challenges. Section 5 shows experiment results. We show related work in Section 6 and conclude in Section 7.

2 MOTIVATION: *IN SITU* DATA COMPUTATION ON MASSIVE IDLE CPU CORES

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 in VPIC. VPIC is a parallel particle simulation framework that 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.

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 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 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 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 twofold. 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 2.1. 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 2.2.

2.1 Asynchronous Data Processing Near Data at Rest

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.

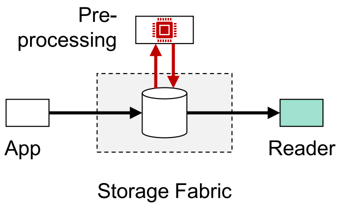
One way to speed up queries is to pre-process data before queries. This allows data to be transformed to a format that is optimized for the upcoming inquiry. Today, data transformation is often done by launching a separate data processing program on the main computing platform after the main application exits. As Figure 2(a) 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 due to the large amount of I/O involved in moving the data between the application program and the 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 data can be transformed to a read-optimized format.

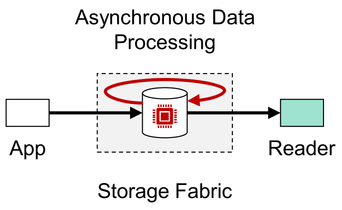
To accelerate data analysis, modern computing platforms take advantage of the dedicated compute resources within their storage to perform data operations (such as the transformation of data to a read-optimized format) on behalf of their client applications. This improves performance for two reasons. First, as Figure 2(b) 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, which allows data to be accessed much faster.

2.2 Processing Data Early Before Data Reaches Storage

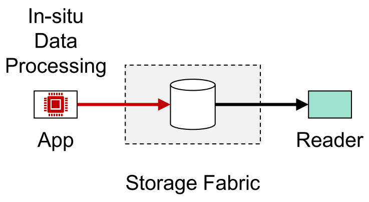
While offloading data processing to storage improves performance, two limitations exist. First, despite the ability to hide latency through asynchronous processing, the overall processing power of



(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.

Fig. 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 that 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.

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. 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 (the total computational power of a computing platform) and storage (the aggregate bandwidth of the underlying storage media) continues to rise, 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 [4, 15].

To address the first limitation, 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

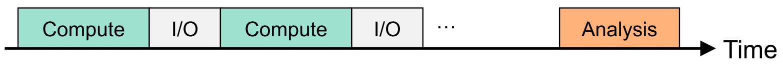


Fig. 3. Illustration of a typical scientific workflow consisting of a bulk-synchronous parallel simulation application acting as a data writer and a subsequent data analysis program acting as a data reader with the execution of the writer further divided into iterations of non-overlapping compute and 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.

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 2(c). 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 3. 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 interprocess 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, 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 twofold. First, it allows data to be processed early before it reaches storage so processing data does not require reading back all the data from storage and then writing the processed data to storage. This minimizes I/O. Second, it allows data to be processed in parallel with its writing so that the data processing delay can be overlapped with the writing of the data. This minimizes the total time an application experiences for writing and processing the data.

We propose DeltaFS IMDs, a scalable method for parallel applications to reorganize their writes for 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. DeltaFS IMDs demonstrate a new way of providing computational storage capabilities on modern computing platforms. We show that not only can data computation be pushed to the accelerators nearest to the media to enable flexible offload mechanisms, it can also be pushed to the compute nodes furthest from the media to achieve scalable software-defined storage services. 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 problems, such as anomaly detection and software debugging in network monitoring and

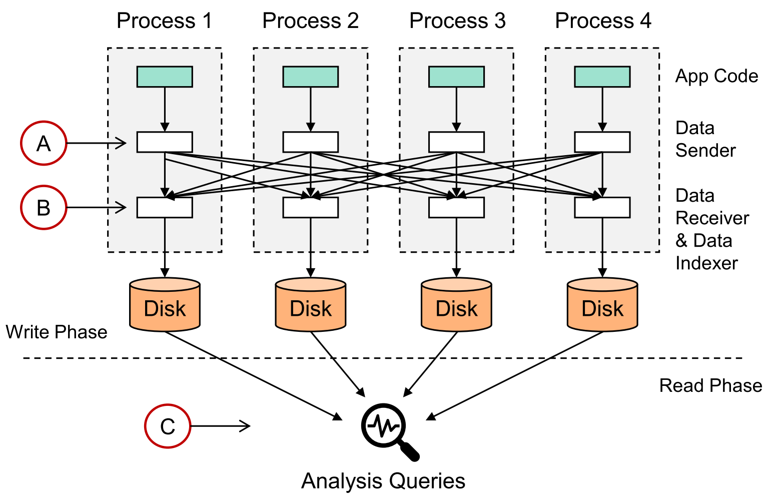


Fig. 4. Illustration of performing *in situ* data computation on the write path of a parallel data application for speeding up queries on the read path. Data computation takes place within each application process. Data is processed on the fly as it is written by the application to storage. Processing 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.

event tracing systems [16, 34]. The idea of reusing idle CPU cycles to perform data computation can be further explored to enable more types of data operations [78].

3 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 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.

3.1 Target Application and Query Types

Applications that benefit most from DeltaFS IMDs are ones with data output that occurs in bursts. These applications are typically massively parallel data programs that run on large computing platforms. Typically each application process outputs data, and data is written to a shared underlying storage system. This shared underlying storage system is often a large parallel filesystem running on a dedicated set of storage nodes.

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

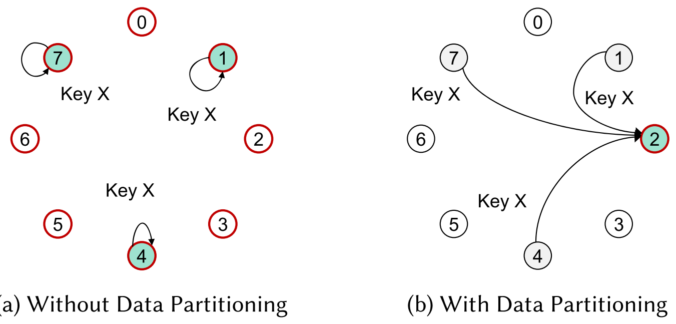


Fig. 5. Processes (numbered as 0, 1, 2, ...) of a parallel application writing data output. Without data partitioning, data is directly written to per-process files and reading back a key requires searching all files. With data partitioning, each key is sent to its home process for writing so reading back a key requires searching only the output of one specific process. 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.

3.2 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 looking up a key does not require searching all data. Figure 5(a) 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 to find the data of a key. As Figure 5(b) shows, 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.

To partition data, each process of the writer application is assigned a partition. A hash function is used to map keys to partitions. Each partition is responsible for a disjoint range of keys. When the 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 per-partition data lookups.

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 consisting of an online data partition component and an online data indexing component. 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. 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 4.

3.3 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, 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 [22, 30, 31, 88].

3.4 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 3.2. 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 3.3, 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.

4 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 while addressing these challenges. Our techniques enable us to index data in a single pass (Section 4.1), efficiently partition data across the processes of a parallel application (Section 4.2), and frugally use memory for all-to-all data communication (Section 4.3).

4.1 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 [68, 81].

Figure 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 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.

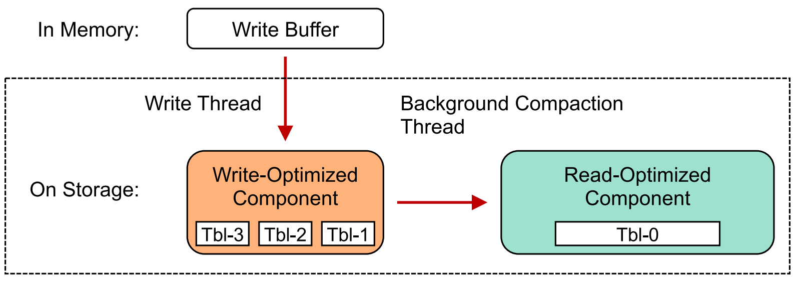


Fig. 6. Illustration of a simplified LSM-Tree. An LSM-Tree consists of an in-memory write buffer, a write-optimized on-storage component made of a series of logged tables (Tbl's) of sorted KV pairs, and a read-optimized on-storage component consisting of a single large sorted table. Tables are sorted 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, however, 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.

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 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 data rereads and rewrites for merging on-storage data, 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.

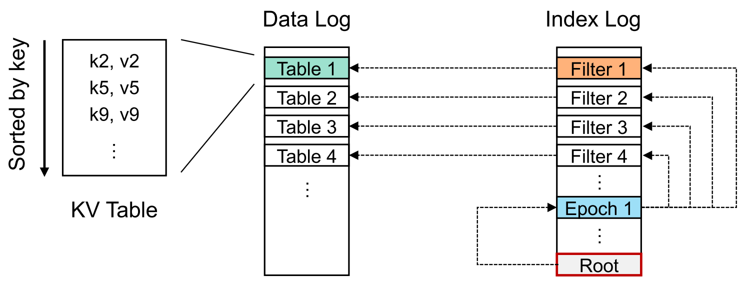


Fig. 7. Illustration of the on-storage format of an FL-Tree consisting of a data log and an index log.

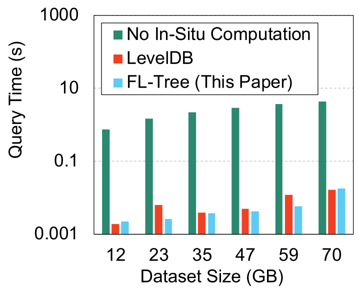
To prevent such bottlenecks, we need a mechanism that is capable of having data reorganized and indexed in one pass and does not require merging data in the background.

Switching to Filters. The reason merging improves performance is that it reduces the number of places a reader has to search to find the data of a key. To improve read performance without it, we use filters [18]. 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.

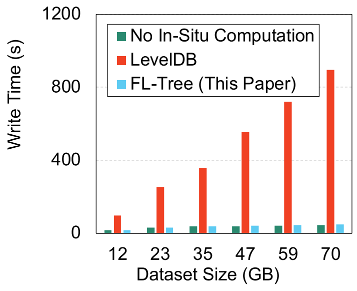
Filters are effective for two reasons. First, filters are small compared with the data they filter so writing and storing them in addition to data does not introduce a significant I/O and storage overhead. Second, by creating a filter for each table, a reader process is able to leverage information in 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.

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 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.



(a) Average Latency of querying a KV pair. Performing *In situ* data compaction as data is written drastically increases subsequent query performance. 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.

Fig. 8. Read and write performance of 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.

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 nodes. Each compute node consists of 32 CPU cores and 128 GB 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 [1, 26]. 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 [42, 57, 73, 75, 93].

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.

Results. Figure 8(a) 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

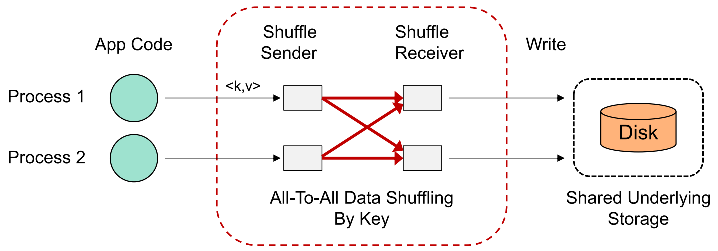


Fig. 9. Illustration of performing online data partitioning across the processes of an application while the application writes data to storage. Each application process represents a data partition. Data not belonging to the local process is sent to the remote process responsible for the key. Each process may send data to each other process. The best performance is achieved when the overall writing process is blocked on storage such that partitioning data does not slow down the writing of data.

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 8(b) 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.

4.2 Efficiently Partitioning Data Over the Network

In addition to online data indexing, another key component of our *in situ* computation is the online partitioning of data across the processes of a parallel data application. Online data partitioning requires transferring data over the network so that it can be processed by the partition responsible for the key. We expect the bulk of our applications to be high entropy such that all data needs to be sent to a remote partition during data partitioning.

Sending large amounts of data over network can be prohibitively expensive. The second challenge of our work is to be able to efficiently run our *in situ* data computation on computing platforms with less capable networks. Our *in situ* data computation service runs inside a parallel data application. It performs online data partitioning when an application writes data to storage. When the interconnection network bandwidth of the computing cluster is not significantly higher than that of storage, moving data across network may be no less costly than writing it to storage. Thus, partitioning data could drastically increase the write time of an application, significantly reducing its overall write efficiency, as we illustrate in Figure 9.

To more efficiently move KV pairs over the network, a trivial optimization is to batch multiple KV pairs within the payload of one RPC. Assuming RPC size is fixed at a large number (e.g., 32 KB), 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 it significantly less subject to network performance.

Simple Data Indirection Does Not Work. The current state-of-the-art uses data indirection to reduce KV movement when directly moving KV pairs is too costly [63, 75]. With data indirection,

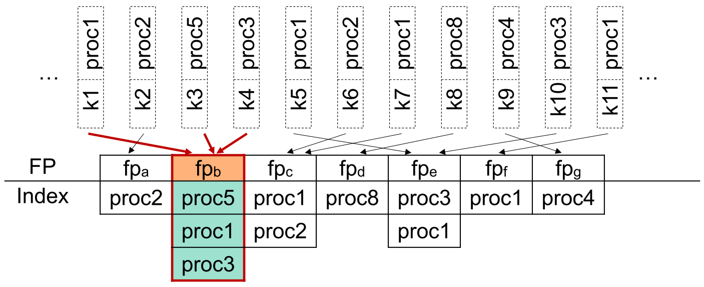


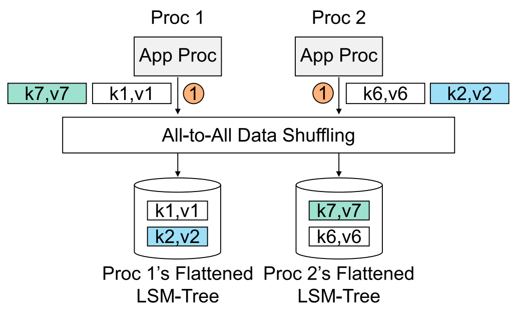
Fig. 10. Auxiliary Table Design consisting of a fingerprint (FP) layer and an index layer. These two layers work hand in hand to map keys to their source processes. Raw keys (k1, k2, k3, ...) are lossily converted to tiny hash fingerprints (fp$_a$, fp$_b$, fp$_c$) before they are inserted into the fingerprint 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.

one moves keys with pointers to values rather than moving KV pairs in their entirety. Figure 11 shows all-to-all data shuffling both before (Figure 11(a)) and after (Figure 11(b)) 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 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 where 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 corresponding KP pair, and then dereferences the pointer to read back the value of the key.

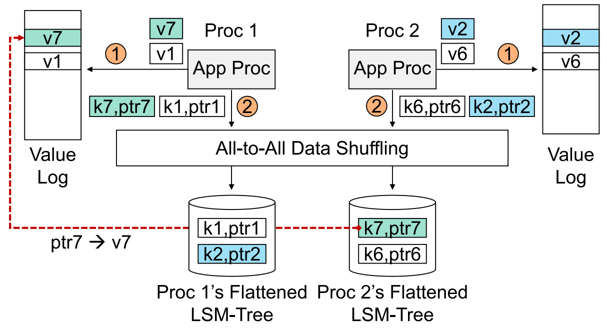
The advantage of sending KP instead of KV pairs is a reduction in the amount of data exchanged over the network. However, storing pointers in addition to the original KV data increases total data size and has the disadvantage of increasing an application's total I/O time. While this overhead is negligible 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 [10]. It is also common for scientific applications to output objects that are smaller than 50 bytes [21, 28]. 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.

LossyKV. 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 11(b) 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. In practice, pointers typically add a 12-byte I/O and storage overhead per key. Each pointer consists of a 4-byte file ID and an 8-byte file offset. Our goal is to significantly reduce this overhead while still allowing reads to be efficient.

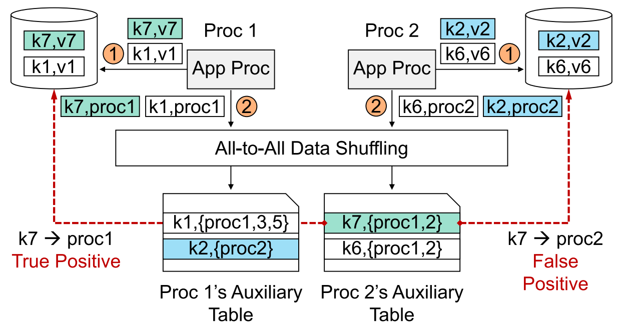
Our approach, named LossyKV, uses lossy pointers to reduce pointer overhead. A lossy pointer is one that may point to multiple data locations instead of just one. Figure 11(c) shows our high-level design. To partition data, an application process first writes an intact KV pair directly into its local partition's FL-Tree. It then sends its own process ID along with another copy of the key to the


Proc 1
Proc 2
App Proc
App Proc
k7,v7
k1,v1
1
1
k6,v6
k2,v2
All-to-All Data Shuffling
k1,v1
k2,v2
k7,v7
k6,v6
Proc 1's Flattened LSM-Tree
Proc 2's Flattened LSM-Tree

(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 >1 data locations (e.g., k7 is mapped to process 1 and 2; not all processes or keys are shown in the figure). Our scheme reduces write cost while only slightly slowing down reads.

Fig. 11. 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.

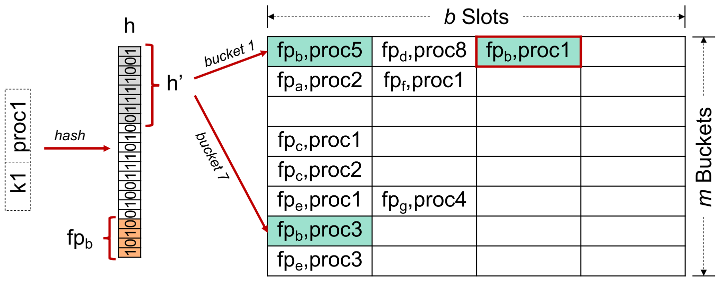


Fig. 12. Auxiliary Table Implementation using Partial-Key Cuckoo Hash Tables. A partial-key cuckoo hash table consists of a number of buckets (currently sized at 6 in this example). Each bucket holds up to b data slots ($b = 4$ in this example). Each key is mapped to m buckets ($m = 2$ in this example) 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.

key's destination partition. So instead of shuffling KV or KP pairs, LossyKV shuffles Key-ID (KID) pairs. Each KID pair serves as a pointer mapping a key back to the process who wrote the key. We refer to this as a key's source process. After data partitioning, LossyKV produces two types of data structures. One is the FL-Tree storing intact KV pairs. Note that FL-Trees now logically operate at the shuffle sender side rather than at the receiver side. The other, named an Auxiliary Table, maps keys to their source processes.

To reduce overhead, auxiliary tables store pointers (KID pairs) lossily such that one may map a key to multiple source processes. Reducing pointer accuracy enables us to store pointers using fewer amount of bits. Recovering a KV pair with LossyKV is a two-step process. First, a reader program uses the auxiliary table responsible for the target key to determine the source process. The reader then goes to the FL-Tree of the source process to read back the KV pair. Because pointers are stored lossily in an auxiliary table, it is possible for a key to be mapped to multiple source processes. In such cases, a reader program searches the FL-Trees of all these source processes until it finds the key. FL-Trees are packed with intact KV pairs. A reader knows when it hits a key.

Achieving Lossiness in Auxiliary Tables. We use compact hash data structures to achieve lossiness. Figure 10 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 an array of fingerprints. Fingerprints are tiny partial hashes (e.g., the leftmost 4 bits of a hash) of keys. Each fingerprint represents a key. Storing fingerprints instead of the original keys reduces I/O and storage overhead, and is the fundamental reason our auxiliary tables are lossy. The source process of each key is recorded in the index layer of an auxiliary table under the fingerprint that represents the key. It is possible for multiple keys to be hashed into one fingerprint. So each fingerprint effectively points to a set of source processes. Each of them comes from a key that is represented by the fingerprint.

To implement our design, we use Partial-Key Cuckoo Hash Tables [41, 59, 74]. These are a cuckoo hash table variant that stores fingerprints of keys (partial-keys) instead of full keys [58, 70]. Figure 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 [41]. 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 [66]. 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 about 95% in practice, while only slightly slowing down reads.

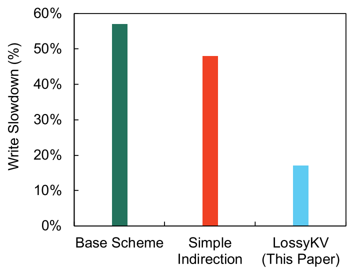
Measurements. We run experiments to evaluate the read and write performance of LossyKV. We use 64 compute nodes. Each compute node has 68 CPU cores and 96 GB RAM. Reading and writing data is through a remote parallel filesystem. Performing RPC operations on these compute nodes is expected to be about 4× more expensive in latency and 3× in bandwidth (compared with the compute nodes we used in Section 4.1), making balancing network and I/O overhead and balancing read and write performance critical [37, 84].

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 4,096 benchmark processes and then indexed using a FL-Tree at each process before written to storage. About 2 TB 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.

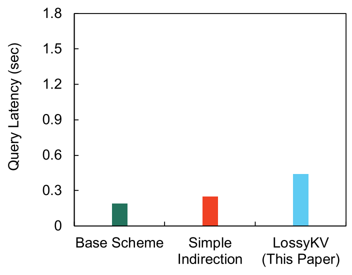
We compare performing all-to-all data shuffling using the base format, simple indirection (the current state-of-the-art), and LossyKV (our solution). We use Write Slowdown to gauge the total data partitioning overhead during the writing of data to storage. It is measured as the additional time each run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

Figure 13(a) compares the write slowdown of different online data shuffling schemes. Results show that LossyKV is able to reduce total write slowdown by 3.3× compared with the base format and by 2.8× 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 13(b) 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 190 to 250 ms 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 18 MB 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 (440 ms) 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 200 ms slower.



(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 2 TB dataset. LossyKV shows comparable performance while requiring reading back more data and performing more lookups due to indirection and lossiness.

Fig. 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 3.3×. (b) LossyKV only slightly increases query overhead (200 ms per query).

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

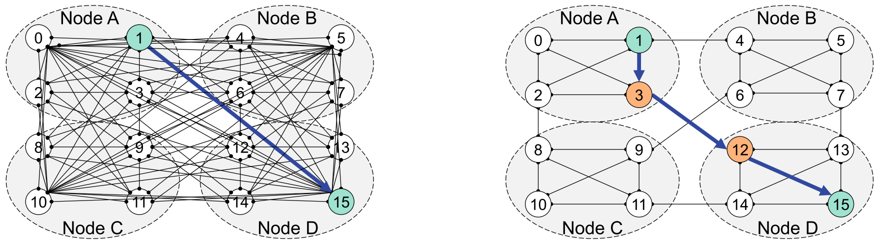
Efficiently exchanging data over network is essential to fast online data shuffling. But even with an efficient shuffle mechanism, performing data shuffling among hundreds of thousands of application processes can be challenging. Critically, direct N-N routed messages delayed for efficient transfer use 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.

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., 32 KB) 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), then 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 14(a) 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 processes before sending the message to its final destination. This is as



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

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

Fig. 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.

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 14(b) shows, our current multi-hop routing implementation consists of three 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 14(b) shows an example of three-hop routing with four nodes and 16 shuffle processes. Each process is both a shuffle sender and a shuffle receiver. On each node, three processes are selected to act as the local representatives for one remote node each. Thus, each process only needs to maintain three 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 three-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 three-hop routing is to run it on a big machine. We do this in Section 5. Here, we show the memory saving of three-hop routing

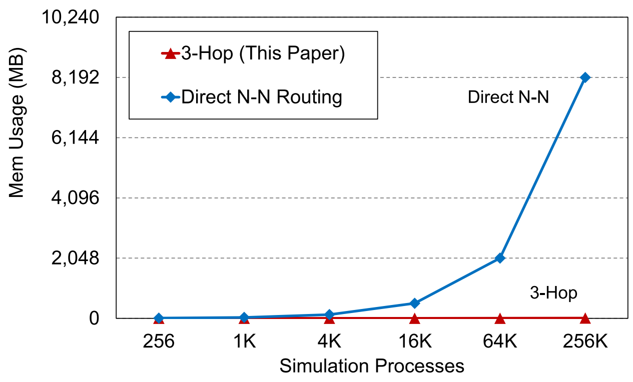


Fig. 15. Projected memory usage for running a parallel application and performing all-to-all data shuffling among of processes of that application. Memory usage includes only the RPC writeback buffers an application process allocates for efficient transfer to other processes. It does not include system memory usage. The three-hop line is close to zero is the figure.

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. 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 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 (<16 MB) as the job size grows, whereas with direct N-N routing memory usage quickly rises to prohibitive.

5 END-TO-END EVALUATION

This section evaluates the end-to-end performance of DeltaFS IMDs. DeltaFS IMDs dynamically reorganize the output of a 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 online data reorganization.

In this section, we start with discussing our VPIC driver application in Section 5.1. We then describe our experiment and show results in Section 5.2.

5.1 Driver Application: VPIC

VPIC is a parallel particle code widely used for simulating kinetic plasmas [21]. 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

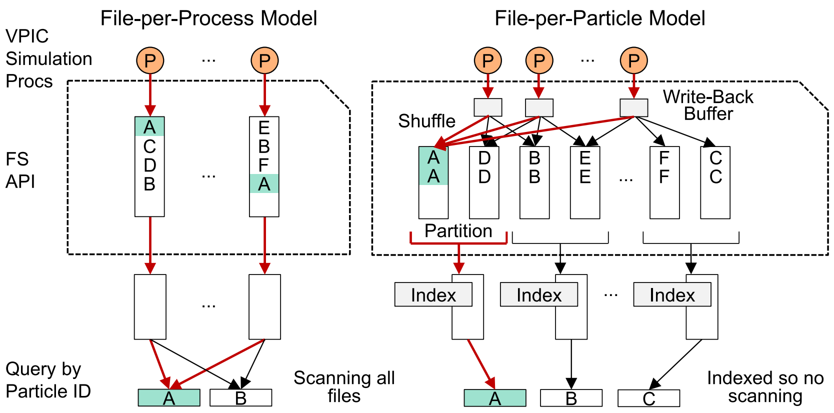


Fig. 16. Comparison of the file-per-process model used by vanilla VPIC with the new file-per-particle model enabled by a DeltaFS IMD. Unmodified VPIC writes one file per process. To index VPIC particles with DeltaFS, we modify VPIC to write the state of each particle into a DeltaFS IMD using particle IDs as the filenames (A, B, C, ...). 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.

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 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 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 MB of data) of one partition of the directory, followed by storage seeks that directly hit the file. No massive data readbacks are involved.

5.2 Experiment Design and Results

To measure performance, we run experiments on the LANL's Trinity supercomputer [5]. Each Trinity compute node has 128 GB 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 [80]. 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 5.3 GB per second. Our *in situ* data techniques process data on the fly while fully utilizing available storage bandwidth minimizing write overhead.

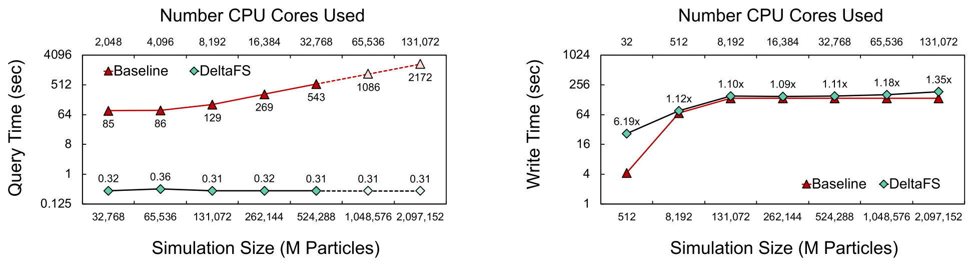
After each simulation, queries are executed directly from the underlying filesystem. Each query targets a random particle and reads all of its data. Particle data is written out over time as the simulation runs through timesteps. Each simulation is configured to output all particle data for five of those timesteps. Each our query therefore returns data from five 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 two 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 17(a) 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 1,740× 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 300 ms in these experiments) as the simulation scales.

Figure 17(b) 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 five 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 two runs, the job sizes are deliberately increased to demonstrate the performance at scale, and there is a slowdown of 20%–35%.

6 RELATED WORK

Filter data structures are used by many storage systems to improve read performance. Unlike indexes that 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 [8]. When the key space of an application is bounded, filters can be implemented using compressed bitmaps [91, 95]. When the key space is unbounded, filters are typically



(a) Query Speedup of reading a VPIC trajectory from a 96 TB 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.

Fig. 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 96 TB 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.

implemented through hash-based data structures, such as the Bloom filter [18], cuckoo filter [41, 59], and quotient filter [12, 71]. Recently, we have also seen filters implemented using tries like SuRF [100] and using perfect hash functions like the ECT structure in SILT [59]. These filter implementations may also be used to implement LossyKV.

Many systems have proposed variants of LSM-Trees to improve performance. WiscKey [63] reduces the I/O amplification associated with compaction by storing keys and values separately and only performing compaction on the keys. Both Monkey [32] and SlimDB [74] use analytical models to generate optimized filter layouts that balance per-filter performance with available memory. LSM-Trie [97] uses an incremental compaction scheme [48] to reduce compaction overhead, and uses clustered indexes to improve query performance. VT-Tree [82] uses a customized compaction procedure that avoids re-sorting in-order data. The DeltaFS's custom LSM-Tree implementation presented in this article 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, cannot be supported efficiently. In practice, storage systems like HyperDex [40] and HBase [45] reorganize data once written, to create an ordered storage layout. Parallel sorting algorithms like SDS-Sort [38] also re-create 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 [101], GLEAN [89, 90], NESSIE [67], and DataSpaces [13]. These systems all use auxiliary nodes to provide analysis tasks. Similarly, systems like Damaris [39] and Functional Partitioning [56] 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 [102] 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 [103] capability within ADIOS [62] 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 [11] 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 [30, 31], a popular indexing and query library for scientific data, uses parallel, compressed bitmap indexes similar to the bitmap indexing described by FastBit [95], and has been deployed as part of *in situ* indexing service to accelerate subsequent reads [53]. DeltaFS creates a similar compressed bitmap index following the shuffle phase to quickly filter subsequences 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 [42] 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 [17, 86] to run seamlessly across platforms supporting different network transports [9, 25, 43]. The Mercury RPC layer allows DeltaFS to run seamlessly across platforms supporting MPI, TCP/IP [25], InfiniBand [9], and OpenFabrics Interfaces [43].

Byna et al. have published the largest petascale particle simulations using VPIC [22–24]. With two trillion particles and 2,000 timesteps of simulation the authors produced 350 TB 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.

7 CONCLUSION

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 article, 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 32 KB 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 three-hop all-to-all communication technique. By limiting the number of off-node connections per process, we believe the applicability of the three-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 subsequence 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 article 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.

REFERENCES

[1] Google. 2012. LevelDB. Retrieved from https://github.com/google/leveldb/.

[2] Oracle. 2013. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. Retrieved from https://www.oracle.com/technetwork/database/exadata/exadata-dbmachine-x4-twp-2076451.pdf.

[3] IBM. 2014. IBM PureData System for Analytics Architecture, A Platform for High Performance Data Warehousing and Analytics. Retrieved from https://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf.

[4] LANL, NERSC, SNL. 2016. APEX Workflows. Retrieved from https://www.nersc.gov/assets/apex-workflows-v2.pdf.

[5] LANL. 2016. LANL Trinity. Retrieved from http://www.lanl.gov/projects/trinity/.

[6] SNIA. 2019. Computational Storage Architecture and Programming Model. Retrieved from https://www.snia.org/sites/default/files/technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.3R1.pdf.

[7] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active disks: Programming model, algorithms and evaluation. *SIGOPS Oper. Syst. Rev.* 32, 5 (Oct. 1998), 81–91. DOI: https://doi.org/10.1145/384265.291026

[8] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. 2010. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10).*

[9] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich. 2011. The common communication interface (CCI). In *Proceedings of the IEEE Annual Symposium on High-Performance Interconnects (HOTI'11)*. 51–60. DOI: https://doi.org/10.1109/HOTI.2011.17

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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)*. 53–64. DOI: https://doi.org/10.1145/2254756.2254766

[11] Utkarsh Ayachit, Andrew Bauer, Earl P. N. Duque, Greg Eisenhauer, Nicola Ferrier, Junmin Gu, Kenneth E. Jansen, Burlen Loring, Zarija Lukić, Suresh Menon, Dmitriy Morozov, Patrick O'Leary, Reetesh Ranjan, Michel Rasquin, Christopher P. Stone, Venkat Vishwanath, Gunther H. Weber, Brad Whitlock, Matthew Wolf, K. John Wu, and E. Wes Bethel. 2016. Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16)*. Article 79, 12 pages.

[12] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.* 5, 11 (July 2012), 1627–1637. DOI: https://doi.org/10.14778/2350229.2350275

[13] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. 2012. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'12)*. 1–9. DOI: https://doi.org/10.1109/SC.2012.31

[14] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. 2012. Jitter-free co-processing on a prototype exascale storage stack. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST'12)*. 1–5. DOI: https://doi.org/10.1109/MSST.2012.6232382

[15] John Bent, Brad Settlemyer, and Gary Grider. 2016. Serving data to the lunatic fringe: The evolution of HPC storage. *USENIX ;login:* 41, 2 (June 2016).

[16] D. Bigelow, S. Brandt, J. Bent, and H. B. Chen. 2010. Mahanaxar: Quality of service guarantees in high-bandwidth, real-time streaming data storage. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST'10)*. 1–11. DOI: https://doi.org/10.1109/MSST.2010.5496975

[17] Andrew D. Birrell and Bruce Jay Nelson. 1983. Implementing remote procedure calls. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP'83)*. 3–. DOI: https://doi.org/10.1145/800217.806609

[18] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. DOI: https://doi.org/10.1145/362686.362692

[19] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman. 2012. Active flash: Out-of-core data analytics on flash storage. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST 12)*. 1–12. DOI: https://doi.org/10.1109/MSST.2012.6232366

[20] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. *The Zettabyte File System.* Technical Report. Sun Microsystems.

[21] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Phys. Plasmas* 15, 5 (2008), 7.

[22] Surendra Byna, Jerry Chou, Oliver Rübel, Prabhat, Homa Karimabadi, William S. Daughton, Vadim Roytershteyn, E. Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, Arie Shoshani, Andrew Uselton, and Kesheng Wu. 2012. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. Article 59, 12 pages. DOI: https://doi.org/10.1109/SC.2012.92

[23] Suren Byna, Robert Sisneros, Kalyana Chadalavada, and Quincey Koziol. 2015. Tuning parallel I/O on blue waters for writing 10 trillion particles. In *Proceedings of the Cray User Group (CUG'15)*. Retrieved from https://cug.org/proceedings/cug2015_proceedings/includes/files/pap120-file2.pdf.

[24] Suren Byna, A. Uselton, D. Knaak Prabhat, and Y. He. 2013. Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper. In *Proceedings of the Cray User Group (CUG'13)*. Retrieved from https://cug.org/proceedings/cug2013_proceedings/includes/files/pap107-file2.pdf.

[25] P. Carns, W. Ligon, R. Ross, and P. Wyckoff. 2005. BMI: A network abstraction layer for parallel I/O. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'05)*. 1–8. DOI: https://doi.org/10.1109/IPDPS.2005.128

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 205–218.

[27] C. Chen, M. Lang, L. Ionkov, and Y. Chen. 2016. Active burst-buffer: In-transit processing integrated into hierarchical storage. In *Proceedings of the IEEE International Conference on Networking Architecture and Storage (NAS'16).* 1–10. DOI: https://doi.org/10.1109/NAS.2016.7549390

[28] Jacqueline H. Chen, Alok Choudhary, Bronis De Supinski, Matthew DeVries, Evatt R. Hawkes, Scott Klasky, Wei-Keng Liao, Kwan-Liu Ma, John Mellor-Crummey, Norbert Podhorszki, et al. 2009. Terascale direct numerical simulations of turbulent combustion using S3D. *Comput. Sci. Discov.* 2, 1 (2009), 015001.

[29] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. 2013. Active disk meets flash: A case for intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13).* 91–102. DOI: https://doi.org/10.1145/2464996.2465003

[30] Jerry Chou, Mark Howison, Brian Austin, Kesheng Wu, Ji Qiang, E. Wes Bethel, Arie Shoshani, Oliver Rübel, Prabhat, and Rob D. Ryne. 2011. Parallel index and query for large scale data analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11).* Article 30, 11 pages. DOI: https://doi.org/10.1145/2063384.2063424

[31] J. Chou, K. Wu, and Prabhat. 2011. FastQuery: A parallel indexing system for scientific data. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'11).* 455–464. DOI: https://doi.org/10.1109/CLUSTER.2011.86

[32] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17).* 79–94. DOI: https://doi.org/10.1145/3035918.3064054

[33] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI'04).*

[34] Peter J. Desnoyers and Prashant Shenoy. 2007. Hyperion: High volume stream archival for retrospective querying. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC'07).* Article 4, 14 pages.

[35] Ananth Devulapalli, Iyyappa Murugandi, Da Xu, and Pete Wyckoff. 2009. *Design of an Intelligent Object-based Storage Device.* Technical Report. Ohio Supercomputer Center.

[36] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13).* 1221–1230. DOI: https://doi.org/10.1145/2463676.2465295

[37] Douglas Doerfler, Brian Austin, Brandon Cook, Jack Deslippe, Krishna Kandalla, and Peter Mendygral. 2017. Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC. In *Proceedings of the Cray User Group (CUG'17).* Retrieved from https://cug.org/proceedings/cug2017_proceedings/includes/files/pap117s2-file1.pdf.

[38] Bin Dong, Surendra Byna, and Kesheng Wu. 2016. 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).* 57–68. DOI: https://doi.org/10.1145/2907294.2907300

[39] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. 2012. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'12).* 155–163. DOI: https://doi.org/10.1109/CLUSTER.2012.26

[40] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM Conference on Applications Technologies Architectures and Protocols for Computer Communication (SIGCOMM'12).* 25–36. DOI: https://doi.org/10.1145/2342356.2342360

[41] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT'14).* 75–88. DOI: https://doi.org/10.1145/2674005.2674994

[42] Hugh N. Greenberg, John Bent, and Gary Grider. 2015. 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).*

[43] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. 2015. A brief introduction to the openfabrics interfaces—A new network API for maximizing high performance application efficiency. In *Proceedings of the IEEE Annual Symposium on High-Performance Interconnects (HOTI'15).* 34–39. DOI: https://doi.org/10.1109/HOTI.2015.19

[44] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16).* 153–165. DOI: https://doi.org/10.1109/ISCA.2016.23

[45] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS under HBase: A Facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14).* 199–212.

[46] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. 2004. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04).*

[47] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20).*

[48] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental organization for data recording and warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97).* 16–25.

[49] Y. Jin, H. Tseng, Y. Papakonstantinou, and S. Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17).* 373–384. DOI: https://doi.org/10.1109/HPCA.2017.15

[50] Y. Kang, Y. Kee, E. L. Miller, and C. Park. 2013. Enabling cost-effective data processing with smart SSD. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST'13).* 1–12. DOI: https://doi.org/10.1109/MSST.2013.6558444

[51] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19).* 144–154. DOI: https://doi.org/10.1145/3319647.3325831

[52] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A case for intelligent disks (IDISKs). *SIGMOD Rec.* 27, 3 (Sept. 1998), 42–52. DOI: https://doi.org/10.1145/290593.290602

[53] J. Kim, H. Abbasi, L. Chacón, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu. 2011. Parallel in situ indexing for data-intensive computing. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV'11).* 65–72. DOI: https://doi.org/10.1109/LDAV.2011.6092319

[54] C. Lee, H. Kang, D. Park, S. Park, Y. Kim, J. Noh, W. Chung, and K. Park. 2019. iLSM-SSD: An intelligent LSM-tree-based key-value SSD for data analytics. In *Proceedings of the IEEE 27th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'19).* 384–395. DOI: https://doi.org/10.1109/MASCOTS.2019.00048

[55] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim. 2011. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Trans. Consumer Electr.* 57, 4 (Nov. 2011), 1732–1739. DOI: https://doi.org/10.1109/TCE.2011.6131148

[56] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. 2010. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'10).* 1–12. DOI: https://doi.org/10.1109/SC.2010.28

[57] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. LocoFS: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'17).* Article 4, 12 pages. DOI: https://doi.org/10.1145/3126908.3126928

[58] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14).* Article 27, 14 pages. DOI: https://doi.org/10.1145/2592798.2592820

[59] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11).* 1–13. DOI: https://doi.org/10.1145/2043556.2043558

[60] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST'12).* 1–11. DOI: https://doi.org/10.1109/MSST.2012.6232369

[61] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton. 2016. DAOS and friends: A proposal for an exascale storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16).* 585–596. DOI: https://doi.org/10.1109/SC.2016.49

[62] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. 2009. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09).* 1–10. DOI: https://doi.org/10.1109/IPDPS.2009.5161052

[63] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16).* 133–148.

[64] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: A survey. *VLDB J.* 29, 1 (Jan. 2020), 393–418. DOI: https://doi.org/10.1007/s00778-019-00555-y

[65] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. 207–219.

[66] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (Oct. 2001), 1094–1104. DOI:https://doi.org/10.1109/71.963420

[67] Ron A. Oldfield, Gregory D. Sjaardema, Gerald F. Lofstead, II, and Todd Kordenbrock. 2012. Trilinos I/O support trios. *Sci. Program.* 20, 2 (Apr. 2012), 181–196. DOI:https://doi.org/10.1155/2012/842791

[68] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Info.* 33, 4 (June 1996), 351–385. DOI:https://doi.org/10.1007/s002360050048

[69] Andrey Ovsyannikov, Melissa Romanus, Brian Van Straalen, Gunther H. Weber, and David Trebotich. 2016. Scientific workflows at datawarp-speed: Accelerated data-intensive science using NERSC's burst buffer. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS'16)*. 1–6.

[70] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algor.* 51, 2 (May 2004), 122–144. DOI:https://doi.org/10.1016/j.jalgor.2003.12.002

[71] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A general-purpose counting filter: Making every bit count. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. 775–787. DOI:https://doi.org/10.1145/3035918.3035963

[72] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. 2007. Evaluation of active storage strategies for the lustre parallel file system. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07)*. Article 28, 10 pages. DOI:https://doi.org/10.1145/1362622.1362660

[73] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. 145–156.

[74] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow.* 10, 13 (Sept. 2017), 2037–2048. DOI:https://doi.org/10.14778/3151106.3151108

[75] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'14)*. 237–248. DOI:https://doi.org/10.1109/SC.2014.25

[76] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. 2001. Active disks for large-scale data processing. *Computer* 34, 6 (June 2001), 68–74. DOI:https://doi.org/10.1109/2.928624

[77] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. DOI:https://doi.org/10.1145/146941.146943

[78] Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. 2020. Mochi: Composing data services for high-performance computing environments. *J. Comput. Sci. Technol.* 35, 1, Article 121 (2020), 23 pages. DOI:https://doi.org/10.1007/s11390-020-9802-0

[79] M. T. Runde, W. G. Stevens, P. A. Wortman, and J. A. Chandy. 2012. An active storage framework for object storage devices. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST'12)*. 1–12. DOI:https://doi.org/10.1109/MSST.2012.6232372

[80] Philip Schwan. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Ottawa Linux Symposium (OLS'03)*. 380–386.

[81] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. 217–228. DOI:https://doi.org/10.1145/2213836.2213862

[82] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 17–30.

[83] Hyogi Sim, Youngjae Kim, Sudharshan S. Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R. Butt, and Lavanya Ramakrishnan. 2015. AnalyzeThis: An analysis workflow-aware storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'15)*. Article 20, 12 pages. DOI:https://doi.org/10.1145/2807591.2807622

[84] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. 2016. Knights landing: Second-generation Intel Xeon phi product. *IEEE Micro* 36, 2 (Mar. 2016), 34–46. DOI:https://doi.org/10.1109/MM.2016.25

[85] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W. Liao, and A. Choudhary. 2010. Enabling active storage on parallel I/O software stacks. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST'10)*. 1–12. DOI:https://doi.org/10.1109/MSST.2010.5496981

[86] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. 2013. Mercury: Enabling remote procedure call for high-performance computing. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'13)*. 1–8. DOI: https://doi.org/10.1109/CLUSTER.2013.6702617

[87] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. 2013. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 119–132.

[88] Tiankai Tu, Charles A. Rendleman, Patrick J. Miller, Federico Sacerdoti, Ron O. Dror, and David E. Shaw. 2010. Accelerating parallel analysis of scientific simulation data via Zazen. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*.

[89] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. 2011. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*. 1–11. DOI: https://doi.org/10.1145/2063384.2063409

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

[91] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. 993–1008. DOI: https://doi.org/10.1145/3035918.3064007

[92] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16)*. Article 4, 7 pages. DOI: https://doi.org/10.1145/2933349.2933353

[93] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. RADOS: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW'07)*. 35–44. DOI: https://doi.org/10.1145/1374596.1374606

[94] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced SQL offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. DOI: https://doi.org/10.14778/2732967.2732972

[95] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* 31, 1 (Mar. 2006), 1–38. DOI: https://doi.org/10.1145/1132863.1132864

[96] S. Wu, K. Lin, and L. Chang. 2018. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *Proceedings of the Design Automation Test in Europe Conference Exhibition (DATE'18)*. 563–568. DOI: https://doi.org/10.23919/DATE.2018.8342070

[97] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. 71–82.

[98] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *Proceedings of the IEEE 36th International Conference on Data Engineering (ICDE'20)*. 1802–1805. DOI: https://doi.org/10.1109/ICDE48307.2020.00174

[99] Yulai Xie, K. Muniswamy-Reddy, D. Feng, D. D. E. Long, Yangwook Kang, Z. Niu, and Zhipeng Tan. 2011. Design and evaluation of Oasis: An active storage framework based on T10 OSD standard. In *Proceedings of the International Conference on Massive Storage Systems and Technologies (MSST'11)*. 1–12. DOI: https://doi.org/10.1109/MSST.2011.5937220

[100] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the International Conference on Management of Data (SIGMOD'18)*. 323–336. DOI: https://doi.org/10.1145/3183713.3196931

[101] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. 2010. PreDatA—Preparatory data analytics on peta-scale machines. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*. 1–12. DOI: https://doi.org/10.1109/IPDPS.2010.5470454

[102] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. 2013. GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'13)*. 1–12. DOI: https://doi.org/10.1145/2503210.2503279

[103] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. 2013. FlexIO: I/O middleware for location-flexible scientific data analytics. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'13)*. 320–331. DOI: https://doi.org/10.1109/IPDPS.2013.46

[104] Qing Zheng, George Amvrosiadis, Saurabh Kadekodi, Garth A. Gibson, Charles D. Cranor, Bradley W. Settlemyer, Gary Grider, and Fan Guo. 2017. 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 and Data Intensive Scalable Computing Systems (PDSW-DISCS'17)*. 7–12. DOI: https://doi.org/10.1145/3149393.3149398

[105] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW'15)*. 1–6. DOI: https://doi.org/10.1145/2834976.2834984

[106] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and how? In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14).*

Received February 2020; revised July 2020; accepted August 2020