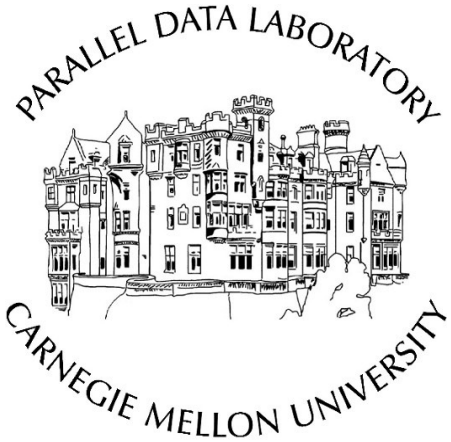


Compact Filter Structures for Fast Data Partitioning

Qing Zheng, Charles D. Cranor, Ankush Jain,
Gregory R. Ganger, Garth A. Gibson, George Amvrosiadis,
Bradley W. Settlemyer[†], Gary A. Grider[†]

[†]Los Alamos National Laboratory

CMU-PDL-19-104
June 2019

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

Acknowledgements: The authors would like to thank Phil Carns, Jerome Soumagne, Shane Snyder, and Robert Ross for their guidance on the *Mochi* software stack. This material is based on work supported in part by the US DOE and Los Alamos National Laboratory, under contract number DE-AC52-06NA25396 subcontract 394903 (IRHPIT), and by the US DOE, Office of Science, Advanced Scientific Computing Research (ASCR) under award number DE-SC0015234. We also thank the member companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, Micron, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Toshiba, Veritas, and Western Digital).

Keywords: In-situ data processing, data partitioning

Abstract

We are approaching a point in time when it will be infeasible to catalog and query data after it has been generated. This trend has fueled research on in-situ data processing (i.e. operating on data as it is streamed to storage). One important example of this approach is in-situ data indexing. Prior work has shown the feasibility of indexing at scale as a two-step process: first by partitioning data by key across CPU cores, and then by having each core produce indexes on its subset as data is persisted. Online partitioning requires that the data be shuffled over the network so that it can be indexed and stored by the responsible core. This is becoming more costly as new processors emphasize parallelism instead of individual core performance that is crucial for processing network events. In addition to indexing, scalable online data partitioning is also useful in other contexts such as efficient compression and load balancing.

We present FilterKV, a data management scheme for faster online data partitioning of key-value (KV) pair data. FilterKV reduces the amount of data shuffled over the network by: (a) moving KV pairs quickly off the network to storage, and (b) using an extremely compact representation to represent each KV pair in the communication occurring over the network. We demonstrate FilterKV on the LANL Trinity cluster, and show that it can reduce total write time (including partitioning overhead) by up to 1.9-3.0x across 4096 processor cores.

1 Introduction

The exponential growth of data continues unabated. At the same time, access times for capacity storage hard disks and flash drives remain almost constant year-to-year. An emerging reality we need to confront is that we are fast approaching a point in time when it will be infeasible to query all the data we are generating in order to extract intelligence [18, 56, 55, 40]. Fortunately, computational power, as defined in FLOPS, continues to increase in each cluster [47, 5, 33]. This has led the research community to move towards performing computation on data ***in-situ***, i.e., as it streams to storage [2].

One example of such computation in the scope of our work is the in-situ generation of data indexes [68]. In-situ data indexing is used to trade off computational power for data access speed. Recent work has demonstrated that in-situ data indexing can be performed at the scale of hundreds of thousands of CPU cores [69], but only after partitioning the data on-the-fly based on a given key. In addition to indexing, online data partitioning of key-value (KV) pairs is also useful in other contexts such as compression and load balancing while both writing and reading data.

The increase in computational power observed in modern clusters often comes from increasing parallelism at the cost of individual core performance [45]. GPUs and manycore CPUs [53] that opt for more lower-frequency cores per die are becoming prevalent across the industry and in modern supercomputers [47, 5]. This is bad news for network performance as many operations depend on the performance of individual cores [23]. We have found that state-of-the-art Network Interface Cards (NICs) from Intel, Mellanox, and Cray only expose one interrupt queue to the Operating System (OS) [1, 29, 21]. To achieve high efficiency, High Performance Computing (HPC) networks typically have communication bypass kernels and allow applications direct access to the NIC. Even so, communication performance is a function of how fast CPUs can process network events such as software tag matching. Library code that accesses NIC buffers directly can only poll as fast as the cores will let it. This suggests that the number of Remote Procedure Calls (RPCs) executed per time unit will be reduced compared to current, high-frequency multicore CPUs. As we show in Section 2, our experiments with multicore Intel Haswell and manycore Intel Knight's Landing (KNL) CPUs show a 3x difference in bandwidth and a 4x difference in latency. The impact this will have on application performance will increase with the number of RPCs the application performs.

To reduce the number of RPCs sent across the network, a trivial optimization is to batch multiple KV pairs within the payload of one RPC. Assuming the size of the payload is fixed, online data partitioning efficiency then depends on the amount of data exchanged. We present FilterKV, a data management scheme that reduces the amount of data moved through the network when performing online data partitioning. The key idea behind our approach is to persist each KV pair to local or shared storage directly, moving it quickly off the network. Then partitioning is performed on a compact representation of the KV pairs. The compact KV pair representation consists of a prefix derived from the key and the ID of the process that generated the KV pair. This representation is more compact than previous state-of-the-art work that moves keys and pointers to data [51, 41].

We demonstrate FilterKV on the Los Alamos National Lab's (LANL) Trinity cluster across 4096 CPU cores. Our evaluation is based on real HPC simulation workloads that periodically persist their in-memory state to storage [7, 13, 12, 32]. We partition all of this data in-situ and index it as it is persisted to storage. In practice, this means that our approach is tailored to applications with bursts of I/O activity where a partitioning can be decided before moving data in order to guarantee that each partition receives roughly the same load. The reason we consider HPC simulations an interesting use case is because they routinely exhibit extreme entropy in the way they generate keys. This means that our work makes no assumptions on the order in which keys are generated by any process. Furthermore, FilterKV can work for KV pair sizes ranging from tiny to large. We show that compared to moving entire KV pairs, FilterKV can reduce total write time by up to 3x across 4096 KNL CPU cores depending on both available network bandwidth and available underlying storage bandwidth. Compared to the state-of-the-art scheme that moves only keys and

	Machine		BF Bytes	
World Rank	Name (Organization)	CPU Cores	b2	b10
6	Trinity (LANL)	979,072	3.40	2.98
12	Cori (NERSC)	622,336	3.28	2.87
13	Nurion (KISTI)	570,020	3.26	2.84
14	Oakforest-PACS (JCAHPC)	556,104	3.26	2.84
16	Tera (CEA)	561,408	3.26	2.84
17	Stampede2 (TACC)	367,024	3.15	2.73
19	Marconi (CINECA)	348,000	3.13	2.72
24	Theta (ANL)	280,320	3.08	2.66

Table 1: Most powerful supercomputers that consist entirely, or in part, of manycore processors. Data from top500.org. We also show for each machine the number of Bloom filter (BF) bytes we need to budget per key in order to bound the number of data partitions per query per key to 2 (b2) or 10 (b10). These numbers will be explained in Section 4.

data pointers, we can also reduce total write time by up to 1.9x, with a negligible increase in query latency.

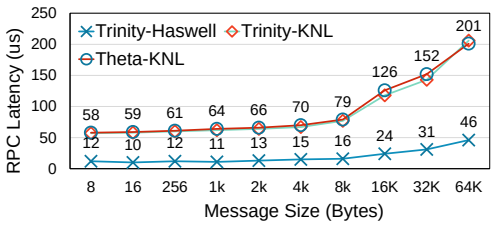
The remainder of the paper is organized as follows. In Section 2 we provide results that motivate the need for a more compact data management scheme. In Section 3 we present the design of FilterKV. Section 4 presents FilterKV implementation details and examines the performance of different data partitioning schemes. Section 5 presents our evaluation of FilterKV compared to the state of the art. Finally, we present related work in Section 6 and conclude.

2 Motivation

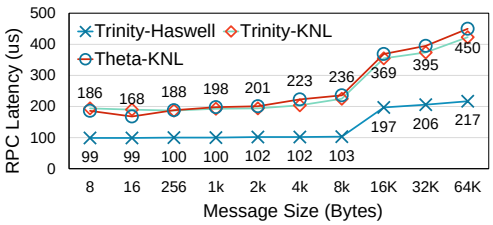
As we increase the computational power, or FLOPS, of the machines we build [25, 5], being able to efficiently use energy becomes increasingly important. This need has led to the rising popularity of manycore processors [53]. Compared with traditional multicore processors, manycore processors feature more CPU cores but with each core spinning in a lower frequency and optimized less extensively for single-thread performance (so less reorder buffers or branch predictors and more cores). Because power consumption decreases approximately quadratically as CPU frequency decreases, machines equipped with manycore processors are able to produce the same amount of computing bandwidth using much less power. As Table 1 shows, there has been a growing number of supercomputers built, partially or entirely, using manycore processors, such as the Trinity computer [34] at LANL and the Theta computer [6] at ANL.

While massive manycore platforms are good for high levels of parallel processing, single-threaded request handlers executed at critical regions are no longer fast enough to meet latency targets [45]. Inter-process data communication is one victim of this tradeoff, and our work is largely motivated by the impact of this processor architecture trend.

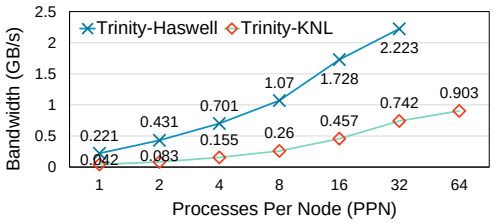
To demonstrate the impact of modern manycore processors on inter-process data communication, we developed an RPC benchmark program [43]. It uses the Mercury RPC framework [54] and libfabric [28] to enable communication over different low-level network transports including TCP, RDMA, and other vendor-specific APIs such as the Cray's GNI API. Using this benchmark we ran tests on the Trinity computing platform at LANL and Theta at ANL. Both are Cray machines using the ARIES interconnect [1]. Trinity consists of two types of compute nodes using either a traditional Intel Xeon multicore processor (Haswell) or an 1.4 GHz Intel Xeon Phi manycore processor (KNL) [53]. All Theta compute nodes use 1.3 GHz KNL processors. Our test evaluates each of the 3 processor types. Test runs consist of a sender and a



(a) RPC Latency under polling mode



(b) RPC Latency under blocking mode



(c) RPC Bandwidth

Figure 1: Results from LANL Trinity and ANL Theta comparing RPC performance between a multicore processor (Intel Haswell) and two manycore processors (Intel KNL). All RPC communication is handled by the Mercury RPC framework using Cray's GNI API via libfabric. We compare latency and bandwidth. For latency tests, we vary RPC message size from 8 to 64K bytes and distinguish two RPC modes: blocking and polling. For bandwidth tests, we use the blocking RPC mode, and vary PPN from 1 to 64, with RPC message size fixed at 16K bytes.

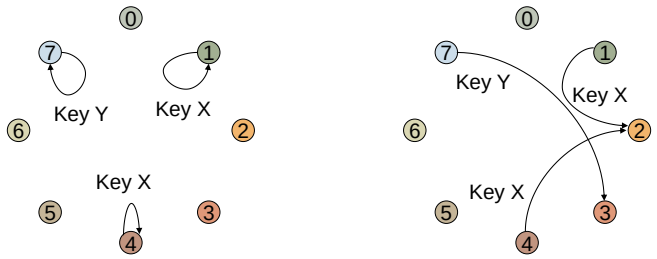
---

receiver process running on different nodes. We vary RPC message size from 8 to 64K bytes. We compare two different RPC modes: *polling* where network threads spin waiting for new events and *blocking* where network threads sleep when no new events are available. We report average RPC latency.

Figure 1 shows the results. With manycore processors being less optimized for single-thread performance, RPC latency measured on the two KNL platforms is noticeably higher than that on the Haswell platform. This reduction of performance is more significant when the RPC implementation does not poll for new events and requires one or more context switches to process an incoming event. Like RPC processing, context switch latency, along with the latency of many other system calls, is largely a function of the single-thread execution speed, not the throughput, of the underlying processor. For example, our results from LMbench [42] show that it takes only 141us to fork a process on a Trinity Haswell node while it takes about 6x longer to do so on a KNL node.

When per-core performance is low, total performance is not necessarily low when one splits a task over multiple processor cores and has all the cores progressing in parallel. Figure 1(c) shows the total RPC bandwidth we can achieve per-node while performing all-to-all data shuffling across 32 compute nodes using different number of processes per node (PPN). We keep RPC message size fixed at 16K, which is the biggest payload GNI allows without invoking bulk transfers. Unfortunately, despite KNL nodes having twice the number of CPU cores compared to Haswell nodes, per-node RPC bandwidth on KNL is still about 3x lower than that of Haswell nodes. This fact further exacerbates the impact of manycore processors on inter-process data communication activities.

While our work is mainly motivated by the reduced network performance seen on manycore platforms, there are other reasons that network bandwidth experienced by a parallel application can be lower than the capacity of the interconnection hardware. Interference from concurrent jobs and contention at network



(a) No Data Partitioning (b) With Data Partitioning

Figure 2: Distributed processes of a parallel application writing data output. Without data partitioning, data is streamed to per-process files; per-key data ends up in multiple files and reading a key may require searching all files. With data partitioning, each key is sent to a particular process so reading a key requires searching only a single data partition.

---

switches can cause applications to observe reduced performance [59, 58]. Inter-process communication cost also rises when applications scale from thousands of processes to hundreds of thousands [69]. Fast data partitioning techniques discussed in this paper can be applied to these cases as well.

3 Fast Data Partitioning Designs

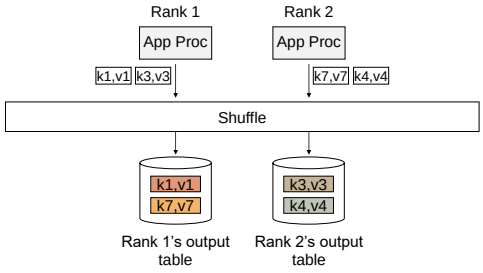
While fast interconnects are often found on HPC platforms, modern manycore processors, as well as bottlenecks within the interconnection network itself, can significantly increase the cost of network communication, making data partitioning prohibitively expensive. We believe that key to mitigating this problem is to greatly reduce the inter-process communication needed to partition the data so that the overall data partitioning process becomes less subject to the platform's hardware configuration and runtime status. In this section, we start by restating the data partitioning problem. We then describe the current state-of-the-art solution to this problem: simple data indirection, followed by our own solution: FilterKV.

3.1 The Data Partitioning Problem

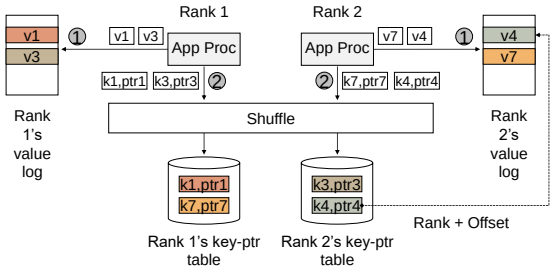
In this paper we model data as KV pairs [61, 27, 36, 31]. We focus on the partitioning of data among the distributed processes of a parallel application and its impact on performance. Each data partition corresponds to a disjoint range in the data key space. Each application process owns a data partition. A data partitioning function is used by the application to determine the process owning the range for a given key. Any process is a sender of data, and we assume that any process can also be a receiver for a range of keys. Our goal is to reduce the total amount of data that must be sent to remote processes.

Many applications, such as in the scientific domain, write their output in a different order than the way that followup queries want it. In that case, data partitioning can drastically reduce the total amount of data a query needs to search for a key of interest. As shown in Figure 2(a), without partitioning data is directly written to per-process files. This means that a subsequent analysis query may have to scan an entire dataset in order to recall the data of a specific key. With data partitioning, however, all data of a key is grouped at a particular process before writing so reading that key requires only searching a single data partition, as illustrated in Figure 2(b).

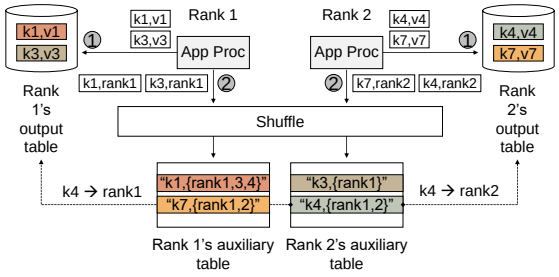
In view of the growing gap between compute and I/O [32], in this paper we focus on data partitioning that takes place *in-situ* with simulation I/O. This is because in-situ processing partitions data as it streams



(a) The Base Format shuffles intact KV pairs.



(b) Simple Indirection shuffles keys and pointers to the data.



(c) Lossy Auxiliary Tables reduce indexing I/O keeping reads efficient.

Figure 3: Illustration of three different data partitioning schemes. A) The base format shuffles keys with values so potentially lots of data is shuffled. B) By shuffling keys with only pointers to values, simple indirection moves less data but storing pointers in addition to values can incur a significant amount of extra I/O to storage when value size is small. C) By persisting lossy data pointers, indexes can be stored using less space, resulting in less I/O overhead while still permitting efficient reads.

---

to storage so expensive data post-processing can be avoided [67, 65, 39]. For computing platforms with shared network-attached storage as opposed to on-node storage [13, 12], writing data to storage consumes a portion of a compute node's network bandwidth potential. This potential is dependent on the interconnection network, the NIC, and the processor of the compute node. Usually, even with manycore processors, this potential is greater than the per-node storage bandwidth a compute node can get on HPC platforms. Because of this, the ***residual network bandwidth***, the fraction of a compute node's network bandwidth potential not consumed by storage I/O, can be utilized to perform in-situ data operations. While our previous work considered cases where this residual network bandwidth is high [69, 68], in this paper we handle cases where this residual bandwidth is limited.

3.2 Simple Data Indirection

The current state-of-the-art solution to avoid overloading the network when residual network bandwidth is low is to use simple data indirection. Rather than sending the entire KV pairs to the other side, as illustrated in Figure 3(a), one sends keys with only pointers to values. To achieve this, an application process writes the value portion of a KV pair to a per-process log file. The offset of the write along with the process's rank number (used as a partition ID) is then encoded into a pointer and sent with the key to the partition where the key belongs. This process is illustrated in Figure 3(b). With data pointers and keys stored at individual data partitions, a reader program is able to efficiently locate per-key information and traverse pointers to read back the actual data.

The advantage of sending pointers instead of actual data is a reduction in the total amount of data pushed over the network. However, storing data pointers in addition to the original data has the disadvantage of adding space overhead and thus increasing an application's total I/O time. While this overhead might turn out to be negligible when the size of the pointer is negligible compared to the size of the respective KV pair, this is not always the case. Values smaller than 250B are reported to be the norm for Facebook's Memcached [8], while scientific application output often consists of a massive number of objects smaller than 50B [15, 20]. In these cases, simple data indirection may end up adding more overhead in the form of I/O time to storage than is removed from the network layer. We measure this in more detail in Section 5. In Section 4 we also show that compression alone does not necessarily help. To more effectively attack this problem we utilize filter data structures that can more compactly represent data pointers and store them using considerably less space.

3.3 Reducing I/O and Space Overhead

To improve performance beyond what simple data indirection can provide we need to further reduce I/O and space overhead. With simple data indirection the write-path code only partitions and shuffles the keys and pointers. Values are written directly to per-process log files reducing the time that this data stays in the network (down to zero if the storage is directly attached). Readers will know exactly where to recover the data associated with each key by using the pointer stored with the key. To locate the data of a key, each data pointer identifies the log file to which the data is written as well as the offset in the log file where the data resides, as illustrated in Figure 3(b). In practice, data pointers can easily add a 12-byte I/O and storage overhead per key, with each pointer consisting of a 4-byte file ID and an 8-byte file offset. Our goal is to considerably reduce this overhead while still allowing readers to efficiently retrieve per-key data.

Our approach, referred to as FilterKV, is to store lossy data pointers instead. That is, rather than recording a key's exact data location, we map each key to a list of candidate data locations of which only one needs to contain the value corresponding to the key. The loss of precision here enables us to write less information. We show in Section 5 that this loss in precision does not significantly affect query performance.

We use filter data structures to reduce both the precision of our data pointers and the amount of data we must store. Figure 3(c) shows a high-level picture of our design: instead of writing values to per-process log files and sending keys with direct data pointers to the other side, each application process now writes complete KV pairs to a per-process KV table. It then sends a second copy of the keys along with the ID of the process to the other side. As a result, the final data output consists of two types of tables: one with the original KV pairs, and the other mapping keys to their source locations. We call the first type of tables ***main tables*** and the second type of tables ***auxiliary tables***.

Because information stored in auxiliary tables is partitioned, a reader program is able to quickly determine a key's source location by looking it up in one specific auxiliary table and then going to the corresponding main table to retrieve the data. Due to intentionally reduced precision, it is possible for keys to be mapped to multiple source locations. In such cases, a reader program must search one or more main tables until it finds the data of interest. Because main tables are packed with complete KV pairs, a reader program knows when it hits a key.

4 Implementation and Measurements

To avoid overloading the network we can write data to per-process log files and shuffle only data indexes. Because data indexes are persisted in addition to the original data, our goal is to minimize the total amount of indexing information we write to storage while not drastically disturbing query performance. With the FilterKV approach we presented in Section 3.3, we envision a compact auxiliary table representation able to

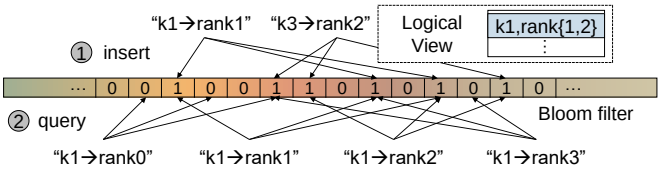


Figure 4: Compactly mapping keys to ranks using Bloom filters. Step 1 inserts two key-rank mappings into the filter, turning two subsets of the filter's bits from 0 to 1. Step 2 retrieves k1's mapping by exhaustively checking all possible mappings for k1. Due to false positives, a Bloom filter may map a key to one or more ranks. In this example, there are 4 ranks and k1 is mapped to both rank 1 (true positive) and 2 (false).

---

map keys to their source locations using only a small fraction of the indexing data needed by the current state-of-the-art. To make this feasible we allow individual keys to be mapped to more than one source locations. In this section we discuss techniques for realizing these lossy auxiliary tables while cost-effectively bounding query latency. Note that because keys can each be mapped to one or more source locations, a query may have to check all of them in order to find the data of interest.

4.1 Our Initial Implementation using Bloom Filters

One way to implement a lossy auxiliary table is to use a family of data structures known as ***filters***. Filters are compact data structures whose canonical use involves managing memberships. In these types of applications, one inserts keys into a set and then asks if a key is in the set. For example, a filter can be designed to return "*False*" when a key is ***definitely not*** in a set (i.e., no false negatives) or "*True*" when it ***may be*** in the set (i.e., false positives are possible). Compared with elementary data structures such as hash tables and binary search trees, filters can be extremely space-efficient. This is typically achieved by filters carefully converting keys to small fingerprints before storing them in their base data structures. There are many implementations of filters [14, 26, 49, 10]. In this section we use the Bloom filter [14] as an example filter to implement our auxiliary tables.

Auxiliary tables are designed to map keys to their source locations. To store these mappings using a Bloom filter, we treat each individual mapping as an opaque object and put the binary representation of it (e.g., the concatenation of the binary representation of the key and that of the source location) into the filter (shown in Figure 4). To read back the source location of a key we perform a set of independent queries on the filter. Each query targets a distinct source location, testing if there *might* exist a mapping from the key to that particular data location. To cover all possible source locations for a key we try all of them. For example, Figure 4 has 4 data partitions so we run 4 different queries to obtain all candidate source locations for each given key of interest. The total number of candidate locations a filter may return for a key is a function of the data structure's false positive rate, as we now discuss.

To achieve space efficiency, filters store in their base data structures small fixed-sized fingerprints instead of the actual data (which in our case is the opaque mapping objects we insert into the filter which can be of arbitrary size). The false positive rate of a filter is largely determined by the number of bits we set for each fingerprint. The more bits we set, the less the rate of false positives. For Bloom filters, each filter is built atop a bit vector and each incoming data is hashed into a subset of bits (the fingerprint) in the bit vector as shown in Figure 4. The more bits we budget for each fingerprint, the larger the size of the base bit vector and the less likely that different fingerprints collide and we get less false positives.

Returning to Figure 3(c), with auxiliary tables implemented using Bloom filters, bounding the total number of data partitions a query needs to search for a key is just a matter of configuring the filter to allocate enough bits for its fingerprints. To demonstrate the effectiveness of this approach we show in Table 1 the

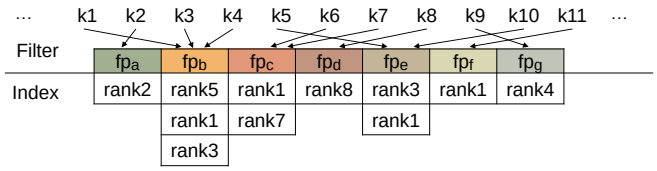


Figure 5: A hybrid auxiliary table design using both filters and indexes. Keys are stored as small fingerprints in a filter. Each fingerprint is mapped to one or more ranks at the index layer. False positives are still possible because multiple keys may be mapped to a single key fingerprint. Readers nevertheless can quickly retrieve the ranks mapped to a key using a single filter and index lookup; no exhaustive filter tests are needed.

---

minimum amount of Bloom filter bytes we need to budget for each key in order to bound the number of data partitions per query per key to 2 (b2) or 10 (b10). For each machine we image running an application that consumes the entire machine and that the total amount of data partitions is equal to the total amount of CPU cores the machine has. Results show that even for the world's largest machines we can ensure good query performance by spending only about 3 bytes per key on data indexes. This is considerably less than the 12-byte per-key overhead we see with the data indirection format we analyzed in Section 3.2.

4.2 A Filter-Index Hybrid Implementation for FilterKV

With FilterKV we are targeting an efficient data partitioning scheme that reduces the total amount of data shuffled through the network. FilterKV achieves this by maintaining an auxiliary table at every data partition to record the source location of each individual key. To minimize I/O and storage overhead, we need to restrict the size of FilterKV's auxiliary tables and we must do so without sacrificing fast queries. As Section 4.1 shows, we can achieve this by building auxiliary tables upon compact filter structures such as the Bloom filters.

Unfortunately, while filters scale well for modern computing platforms with millions of CPU cores, they can be problematic for future exascale computing platforms with tens or hundreds of millions of processing units [25] producing tens or hundreds of millions of data partitions. In Figure 3(c) we showed that auxiliary tables are responsible for mapping keys to their source data locations. With auxiliary tables implemented using filters, we insert into filters opaque mapping objects. Each mapping object identifies the source data location of a specific key, as illustrated in Figure 4. To recall the source location of a key, we test the existence of all possible mappings from the key to a particular data location. As the number of filter tests we need to perform equates the number of data partitions a dataset has, processing a query may require running an excessive amount of filter computation, and the latency of the computation may no longer be dwarfed by storage reads. For example, it can take about 1 second to perform 16 million filter operations on a 2.4GHz Intel Westmere CPU core whereas it may take less than 1 second to read a block from storage.

To attack this problem, we build upon our filter-based auxiliary table design discussed in Section 4.1 and propose a new hybrid implementation that uses both filters and indexes. Our new design, shown in Figure 5, consists of a filter layer and an index layer. The filter layer consists of an array of fingerprints. Each fingerprint represents a user key and is mapped to one or more source locations in the index layer. Storing fingerprints instead of the original keys allows for high space efficiency, an idea we inherit from Section 4.1. On the other hand, directly recording the source locations of each fingerprint as indexes prevents queries from having to perform a potentially large number of filter operations which can be time-consuming. The cost of improved query efficiency is increased space for storing auxiliary tables. We discuss and measure this tradeoff in more detail in Section 4.3.

To implement FilterKV auxiliary tables using this new hybrid design we use partial-key cuckoo hash

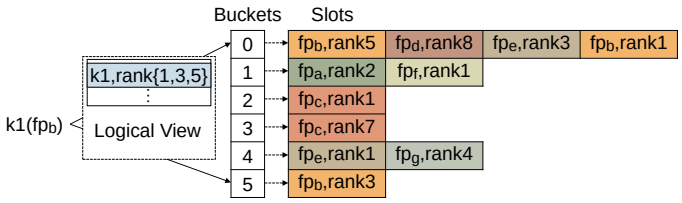


Figure 6: Compactly mapping keys to ranks with partial-key cuckoo hash tables. A cuckoo hash table consists of an array of buckets. Each bucket holds up to a certain amount of data slots (4 in this example). Each slot stores a fingerprint (partial-key) and the rank it maps to. This figure shows 6 buckets and 11 non-empty slots, and key k1 can be stored either at bucket 0 or 5. Because different keys may reuse a same fingerprint, a partial-key cuckoo hash table may map keys to more than 1 ranks. In this example, k1 is stored as fp_b and is mapped to ranks 1, 3, and 5.

---

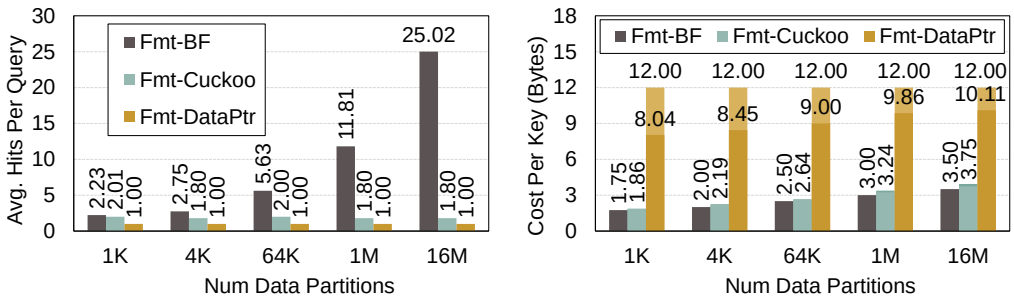
tables [38, 26], a cuckoo hash table [48, 37] variant that stores fingerprints of keys (partial-keys) instead of full keys. As illustrated in Figure 6, each partial-key cuckoo hash table consists of an array of buckets. Each bucket holds up to a certain amount of data slots. When a key-value pair is inserted into a partial-key cuckoo hash table, the key will be transformed to a partial key using a hash function. The resulting "<partial-key,value>" pair is then assigned to two candidate buckets in the table, and can be placed at any of the empty slots in either of the buckets. When no such slot is available, a random slot from one of the two buckets will be selected to hold the incoming key, with the current resident of the slot evicted and relocated to its alternative positions in the table. This relocation process continues until an empty slot can be found, or fails after a large number (e.g., 500) of recursive attempts and causes the table to be resized. In practice, partial-key cuckoo hash table size must be a power of 2 so each resize doubles the size of a table [26]. Mapping every key to two potential locations in the table allows for high levels of table space utilization before a table must be resized [44]. But because not all slots are necessarily filled after all keys have been inserted into the table, partial-key cuckoo hash tables can "leak" space in the data structure, leading to more space to be used than is necessary.

To minimize wasted space, our implementation creates a new partial-key cuckoo hash table when the current table is full. For example, instead of resizing an 1-million-slot table to 2 million, our implementation combines an 1-million-slot table with an 128K-slot table to hold 1.1 million keys. This allows us to keep space utilization at about 95% in practice.

4.3 Measurements

A key benefit FilterKV brings over the current state-of-the-art (simple data indirection) is a more compact representation of data indexes. The cost of it is that a query would incur additional lookups to find the data of a key. In this section we evaluate this tradeoff. We consider 3 data partitioning schemes: data indirection as the current state-of-the-art (Fmt-DataPtr), FilterKV with a Bloom Filter implementation (Fmt-BF), and FilterKV with a partial-key cuckoo hash table implementation.

Test runs consist of generating 16 million keys and inserting their indexing information into a data structure. All generated keys are random 8-byte integers. We vary the number of data partitions each data structure needs to index over, defined as N, from 1000 to 16 million, as the supercomputer with the most number of CPU cores in the world has about 10 million CPU cores. For the data indirection scheme (Fmt-DataPtr), we set each data pointer to be 12 bytes, consisting of an 8-byte offset and a 4-byte rank number, as discussed in Section 3.2. For the FilterKV with partial-key cuckoo hash tables (Fmt-Cuckoo), we configure each partial-key to be 4 bits and its value to be $\log(N)$ bits so it has enough bits to distinguish



(a) Query Amplification

(b) Per-Key Overhead

Figure 7: Comparison of three different data partitioning schemes. Fmt-DataPtr is the state-of-the-art baseline. Figure b) shows space overhead both before and after compression is applied.

---

individual partitions. For the FilterKV with Bloom filters (Fmt-BF), we configure our Bloom filters to budget $4 + \log(N)$ bits for each key. This way they use the same amount of storage as their partial-key cuckoo hash table counterpart.

Figure 7(a) shows the average number of data partitions a data structure returns for each key. This number is always 1 for the data indirection scheme (Fmt-DataPtr) as it stores exact index information. For the FilterKV with Bloom filters (Fmt-BF), the number of data partitions it returns increases, albeit slowly, as the total number of data partitions increases. This is because in this scheme the number of data partitions returned is a function of ***both*** the total number of data partitions and the filter's false positive rate, with the latter being a function of the number of bits we budget for each key. Unfortunately, each additional bit per key only causes the Bloom filter's false positive rate to be reduced by less then by 2x. In other words, the number of data partitions returned keeps increasing because the reduction in the filter's false positive rate cannot compensate for the increase in the total number of data partitions out there. To resolve this problem we could configure Bloom filters to budget $4 + 1.44\log(N)$ bits per key rather than the $4 + \log(N)$ bits we tested, at the cost of increased space overhead.

For the FilterKV with partial-key cuckoo hash tables (Fmt-Cuckoo), the number of data partitions the data structure returns for each key is around 2, and does not increase as the total number of data partitions increases. This is because in this scheme each key is directly mapped to data partitions so the number of data partitions returned is a function of ***only*** the filter's false positive rate; it is ***not*** affected by the total number of data partitions.

Figure 7(b) shows each scheme's space overhead, measured as the number of bytes per key, before and after we compress each data structure using the Google's Snappy compression library. Because with the data indirection scheme (Fmt-DataPtr) exact information is stored, it consumes the most space in spite of compression. Both our FilterKV schemes are able to use much less space due to their compact, albeit lossy, index representations, even before compression. The partial-key cuckoo hash table implementation (Fmt-Cuckoo) used slightly more space than the Bloom filter implementation because not all slots in a partial-key cuckoo hash table are eventually used, as we discussed in Section 4.2.

5 Experiments

This section evaluates the end-to-end performance of different data partitioning schemes at scale. We compare data partitioning that shuffles full KV pairs (Fmt-Base), data partitioning that uses indirection

(Fmt-DataPtr), and data partitioning that utilizes both indirection and partial-key cuckoo hash tables (Fmt-Cuckoo).

5.1 Microbenchmark Results

Our first group of experiments evaluate the performance of our cuckoo-based data partitioning scheme under different job scales and KV pair sizes. These experiments were performed at the Narwhal computing cluster at CMU. Each Narwhal compute node consists of 4 CPU cores and 16GB memory [9]. These compute nodes are interconnected with an Ethernet network. Each node is equipped with one 1Gbps Ethernet NIC for data communication. This limited network bandwidth makes Narwhal an ideal testbed for measuring the effectiveness of a data partitioning scheme in operating with reduced network communication cost. In this particular group of experiments, network communication is costly because the physical network bandwidth available to each compute node is low. Modern manycore processors can similarly make network communication costly, which we measure in Section 5.2.

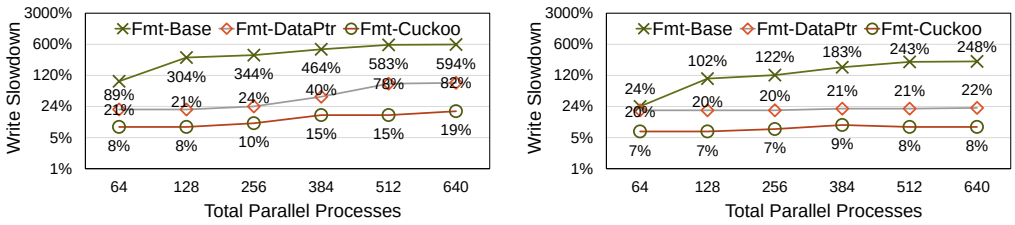
We developed a simple parallel benchmark program to drive our tests. In each test, we run a certain number of parallel processes, and have each process generate random KV pairs of certain sizes. In the first set of runs, we fix the key-value size at 64 bytes and vary the total number of parallel processes from 64 to 640, using 16 to 160 Narwhal compute nodes. The results are shown in Figure 8. In the second set of runs, we fix the total number of parallel processes at 128 and vary key-value size from 16 bytes to 192 bytes. The results are shown in Figure 9. While the total key-value size may vary across runs, the size of the key is always fixed at 8 bytes. Also fixed is the total number of data generated per parallel process, which is configured at 960MB per process across all runs.

We compare two different levels of residual network bandwidth, representing two different degrees of network communication cost. To better understand this cost, let us consider a real-world machine as a case study. The Trinity computer at LANL is configured with 1 burst-buffer node per 32 compute nodes [34]. The compute nodes and the burst-buffer nodes are connected within a single interconnection network. Both types of nodes are equipped with the same type of NIC, and writing data from compute nodes to the burst-buffer storage is bottlenecked at the NICs of the burst-buffer nodes. As such, the residual network bandwidth (total available network bandwidth - storage bandwidth) for Trinity compute nodes is roughly 97% (100% - 1/32). By configuring residual network bandwidth at 50% and 75% in our tests, we emulate cases in which network capability is considerably lower than advertised.

We use ***write slowdown*** to gauge the total data partitioning overhead during data writing. It is measured as the additional time each test needs to spend on writing all the data. As an example, the write slowdown is said to be 100% if it takes a test twice amount of the time to write the data with data partitioning, as opposed to directly writing the data to storage without performing any in-situ data operations. Note that this overhead includes both the overhead caused by performing additional network operations and the overhead incurred by writing filters or data pointers in addition to the original data.

Figure 8 shows the write slowdown of different data partitioning schemes as a function of job size. The base format (Fmt-Base) shuffles entire key-value pairs so a large amount of data is shuffled in each run and the overhead it incurs rises quickly as job size increases. Through the use of indirection (Fmt-DataPtr), data pointers are shuffled instead of the original data so much less network activities are performed. This leads to a much lower write slowdown compared to that of the base format. Finally, the use of partial-key cuckoo hash tables in addition to indirection (Fmt-Cuckoo) allows the total amount of data indexes written to storage to be minimized while shuffling less data, so the resulting write slowdown is even lower than that of using data indirection alone (Fmt-DataPtr).

Figure 9 compares write slowdown as a function of the size of key-value pairs. The base format (Fmt-Base) shuffles entire key-value pairs so its performance does not change with key-value sizes. For the other formats (Fmt-DataPtr and Fmt-Cuckoo), write slowdown decreases as key-value size increases. This



(a) 50% Residual Network Bandwidth (500Mbps) (b) 75% Residual Network Bandwidth (750Mbps)

Figure 8: Weak-scaling results comparing the performance of different data partitioning schemes under different job sizes. We vary job size from 64 (16 nodes) to 640 parallel processes (160 nodes). Each run generates 15 million key-value pairs per process. Each key-value pair is 64 bytes.

---

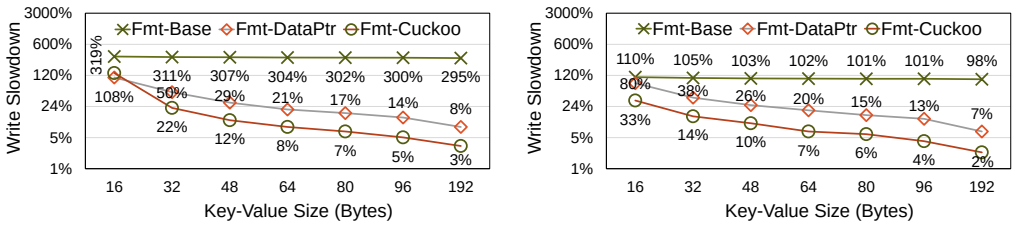
is because the additional data indexes these two formats generate may add a significant I/O and storage overhead when key-value size is small, and is insignificant compared to the total data size when key-value size is large. While our cuckoo format beats simple data indirection in all cases, the benefit is most welcome when key-value size is 32 to 64 bytes, which is the data size of many scientific workloads.

5.2 Macrobenchmark Results

Our second group of experiments evaluate the performance of FilterKV under a real scientific use-case. The scientific application we choose for our experiments is a Vector Particle-In-Cell (VPIC) simulation [15]. VPIC is a scalable particle simulation code developed at LANL. In a VPIC simulation, each simulation process manages a region of cells in the simulation space through which particles move. Every few timesteps the simulation stops and each simulation process writes out the state of all the particles currently managed by the process. In our experiments, state for each particle is 64 bytes. Each query involves retrieving the trajectory of a specific particle along the course of a simulation. Unmodified VPIC writes data directly into per-process output files. Because particles move during a simulation, they may end up in different per-process output files in different timesteps. This means that without post-processing to reorganize this data, retrieving the information of a specific particle would require searching an entire VPIC dataset. To speed up queries, we use our previous system, DeltaFS Indexed Massive Directories, to dynamically partition and index data as it streams to storage [68, 69]. To partition data our previous implementation shuffles entire KV pairs. With this paper we have modified our implementation to support data partitioning that shuffles keys with pointers to data, and data partitioning using the FilterKV scheme.

Our experiments were performed at the LANL's Trinity supercomputer. Recall from Section 2 that Trinity consists of two types of compute nodes equipped with either the Haswell multicore CPU, or the KNL manycore CPU. Each Haswell compute node consists of 32 CPU cores and 128GB memory. Each KNL compute node consists of 68 CPU cores and a total of 112GB memory with 96GB regular DDR4 and an additional 16GB high-bandwidth memory (HBM). Our runs only allocate memory from the main memory. Our results (not shown in this paper) indicate that using the 16GB HBM does not increase, or reduce, performance since the particular in-situ operations in our experiments are not bottlenecked on memory operations.

We compare performance both on Haswell nodes and on KNL nodes. Each run consists of a VPIC simulation, followed by 100 independent queries. Each VPIC simulation runs 4096 parallel processes and simulates a total of 32 billion particles. Each process on average manages 8 million particles. The aggregate state of all particles is approximately 2TB. Across all runs, simulation data was first pushed to burst-buffer storage and was later staged out to the platform's underlying filesystem. We vary the number of burst-



(a) 50% Residual Network Bandwidth (500Mbps) (b) 75% Residual Network Bandwidth (750 Mbps)

Figure 9: Results comparing the performance of different data partitioning schemes under different key-value size configurations. We fix keys at 8 bytes and vary key-value size from 16 to 192 bytes. Each run uses 128 parallel processes and generates the same amount of total data.

---

buffer nodes used in each run from 2 to 5 to test performance under different network-to-storage ratios. After simulation ends, queries were executed directly from the underlying filesystem. Each query randomly targets a particle and a timestep, and reads the state of that particle at that timestep.

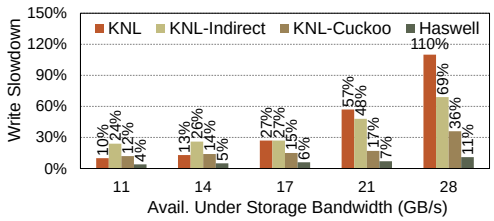
Figure 10(a) shows the overhead that in-situ data partitioning, as well as indexing, adds to the I/O phases of each VPIC simulation run. Looking at the right-half of the figure, it evaluates performance when available underlying storage bandwidth is high so hiding network communication cost is more critical to overall performance. In this region, using an efficient data partitioning scheme is crucial to the overall write performance. Results show that FilterKV can reduce total write time by up to 3x compared to the base format, and by up to 2x compared to the current state-of-the-art. Moving to the left-side of the figure, we evaluate performance when available underlying storage bandwidth is low so the overall writing process is more bottlenecked on the storage side and hiding network communication cost is thus less critical. In this region, using a compact representation that minimizes the size of data indexes is crucial for the overall write performance. Results show that FilterKV can reduce total write time by up to 2x compared to the current state-of-the-art.

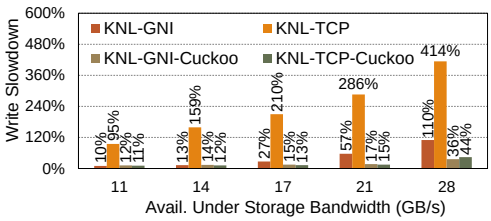
We have also tested how the performance of these data management schemes is affected by the network protocol. Figure 10(b) shows the write performance when we use TCP, rather than more efficient Cray GNI, to perform low-level network communications. While we are by no means advocating using TCP for production jobs, our results show that with the FilterKV scheme we can effectively run TCP jobs almost as fast as GNI jobs. On the other hand, with the base format, TCP jobs can be two-orders of magnitude slower than GNI jobs, which is not a surprise.

Figure 10(c) shows the read performance of all three data partitioning schemes. FilterKV has the highest minimum and median read latency because each query must first fetch a partial-key cuckoo hash table and then may have to attempt reads at multiple data partitions due to false positives. The base format has the lowest minimum and median read latency because each query can directly read data from a specific data partition. The current state-of-the-art shuffles keys and pointers to data so each query only performs one extra read and its latency is not as high as FilterKV in general.

6 Related Work

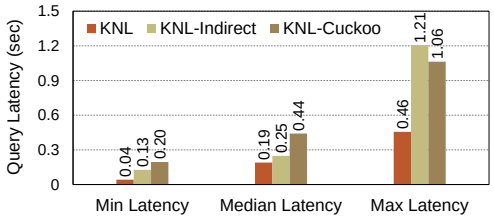
Filter data structures are used by many storage systems to achieve high read performance. Unlike indexes which directly map keys to their data locations, filters speed up queries by indicating where not to read and saving the query process from doing potentially a large number of unnecessary storage reads [3]. When the application's key space is bounded, filters are usually implemented by bitmaps and are compressed to





(a) Write performance of different data management schemes compared to running on a fast multicore CPU (Haswell).

(b) Write performance using different network protocols.



(c) Query latency.

Figure 10: End-to-end results from running the VPIC scientific application and partitioning its data using different data management schemes. All experiments were conducted on the LANL Trinity supercomputer. Higher storage bandwidth makes the effect of network performance more critical.

---

save space [60, 62]. When the key space is unbounded, filters can be implemented by hash-based data structures such as the Bloom filters [14], cuckoo filters [26, 38], and quotient filters [49, 10]. Recently, we have also seen filters implemented by tries such as SuRF [64] and the ECT structure in SILT [38]. These filter implementations can be used to implement our FilterKV schemes too.

The idea of data indirection is used by many LSM-Trees key-value stores to reduce the overhead of compaction operations. For example, WiscKey [41] reduces the I/O amplification associated with compaction by storing keys and values separately and only performing compaction on the keys. Similar use of this idea is also seen in systems such as IndexFS [51] and Cassandra [31]. In addition to data indirection, systems such as Monkey [22] and SlimDB [50] use analytical models to generate optimized filter layouts that balance per-filter performance with available memory. This allows for minimizing its overall false positive rate given a fixed memory budget. Such designs are typically optimized for dedicated storage nodes whose entire memory can be used to serve data operations. Both LSM-Trie [63] and SlimDB [50] use an incremental compaction scheme [30] to reduce compaction overhead. In this design, compaction overhead is kept low by performing compaction less aggressively, resulting in the tree having more levels. Such designs typically use larger filter structures to balance read performance. Finally, VT-Tree [52] features a design that allows in-order data to be linked into an ordered data structure instead of performing a direct merge-sort. This design can also be viewed as a form of data indirection. This paper focuses on using data indirection to reduce the total amount of data shuffled over the network as opposed to data compaction in LSM-Trees. In addition, with FilterKV we also focus on using compact data representations to reduce space overhead.

Rich in-transit data processing capabilities are provided by multiple middleware libraries such as PreDatA [65], GLEAN [59, 58], NESSIE [46], and DataSpaces [11]. These systems all use auxiliary nodes to provide analysis tasks. Similarly, systems such as Damaris [24] and Functional Partitioning [35] co-schedule analysis, visualization, and de-duplication tasks on compute nodes, but require dedicated cores.

The GoldRush runtime [66] 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 [67] capability within ADIOS [39] to create shared memory channels for generating analysis task inputs. These are processed during idle periods of application execution.

VPIC is a widely-used particle simulation code developed at LANL [15]. Large-scale VPIC simulations have been conducted with trillions of particles, generating terabytes of data for each recorded timestep [18, 17, 16]. While FilterKV helps VPIC simulations better partition particle data, it can help other applications as well. This is especially true when the size of data is small so storing data pointers in addition to data can be prohibitively expensive.

Conclusion

In this paper we show how to leverage filter data structures to produce more efficient data partitioning schemes that reduce the total amount of data transferred over the network. Our scheme makes data partitioning less subject to the configuration and architecture of the computing platform. Our results show that our mechanism is able to reduce in-situ data partitioning overhead by up to 3 times while only slightly increasing query latency.

So long as electricity bills still amount to a substantial portion of total operation cost, and so long as the laws of physics still have CPU power be roughly a quadratic function of CPU frequency, there is always a cause to switch to low-frequency platforms. And as a result, we will continue sacrificing single-thread performance for cost-effective high throughput. In fact, low-power data systems utilizing a large number of lightweight processors have been studied for a decade [57, 4, 19]. Recent efforts in deploying HPC filesystems on cost-effective ARM storage servers can also be viewed as a continuation of this idea. With efforts like these mainly targeting I/O nodes that do not need to perform heavy computation and communication, the recent shift from high-frequency multicore computing platforms to lightweight manycore platforms is a brave move to attempt cost-effective high throughput on the main computing platform. While highly energy-efficient manycore processors may become obsolete in the future, single-thread performance of the main computing platform is unlikely to increase and will more likely keep dropping slowly. This, coupled with the on-going trend of using software-defined storage services to manage on-platform storage, network, and compute resources, requires new storage formats to minimize data movement, index size, and our code's reliance on single-thread performance.

In this paper we have used filter data structures in novel ways to achieve both storage efficiency (by storing less data) and communication efficiency (by transmitting less data through the network). We found that filter data structures can be extremely effective in alleviating large-scale data partitioning bottlenecks. This includes Bloom filters that can be implemented in 100 lines of code. With emerging HPC and high performance data analytics systems combining multiple memory and storage technologies in new ways, it is possible for filters to be applied to other on-platform in-situ activities, and to be optimized to achieve further improvements in data management.

References

[1] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. *Cray XC Series Network*. Tech. rep. WP-Aries01-1112. http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf. Cray Inc., Nov. 2012.

[2] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao, I. Ahmad, R. H. Arpaci-Dusseau, F. Chen, Y. Chen, Y. Chen, Y. Cheng, V. Chidambaram, D. Da Silva, A. Demke-Brown, P. Desnoyers, J. Flinn, X. He, S. Jiang, G. Kuenning, M. Li, C. Maltzahn, E. L. Miller, K. Mohror, R. Rangaswami, N. Reddy, D. Rosenthal, A. S. Tosun, N. Talagala, P. Varman, S. Vazhkudai, A. Waldani, X. Zhang, Y. Zhang, and M. Zheng. *Data Storage Research Vision 2025: Report on NSF Visioning Workshop Held May 30–June 1, 2018*. Tech. rep. USA, 2018.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. "Cheap and Large CAMs for High Performance Data-intensive Networked Systems". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI10)*. 2010, pp. 29–29.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. "FAWN: A Fast Array of Wimpy Nodes". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP 09)*. 2009, pp. 1–14. DOI: 10.1145/1629575.1629577.

[5] *ANL Aurora*. https://www.alcf.anl.gov/alcf-aurora-2021-early-science-program-data-and-learning-call-proposals. 2018.

[6] *ANL Theta*. https://www.alcf.anl.gov/theta/.

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

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. "Workload Analysis of a Large-scale Key-value Store". In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 12)*. 2012, pp. 53–64. DOI: 10.1145/2254756.2254766.

[9] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. "Entering the Petaflop Era: The Architecture and Performance of Roadrunner". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 08)*. 2008, 1:1–1:11.

[10] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. "Don't Thrash: How to Cache Your Hash on Flash". In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1627–1637. DOI: 10.14778/2350229.2350275.

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

[12] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. "Storage challenges at Los Alamos National Lab". In: *Proceedings of the 2012 IEEE Conference on Massive Storage Systems and Technologies (MSST 12)*. 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232376.

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

[14] B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692.

[15] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. "Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation". In: *Physics of Plasmas* 15.5 (2008), p. 7.

[16] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol. "Tuning Parallel I/O on Blue Waters for Writing 10 Trillion Particles". In: *Cray User Group (CUG)*. https://cug.org/proceedings/cug2015_proceedings/includes/files/pap120-file2.pdf. 2015.

[17] S. Byna, A. Uselton, D. K. Prabhat, and Y. He. "Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper". In: *Cray User Group (CUG)*. https://cug.org/proceedings/cug2013_proceedings/includes/files/pap107-file2.pdf. 2013.

[18] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu. "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation". In: *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 12)*. 2012, 59:1–59:12. DOI: 10.1109/SC.2012.92.

[19] A. M. Caulfield, L. M. Grupp, and S. Swanson. "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. 2009, pp. 217–228. DOI: 10.1145/1508244.1508270.

[20] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki, et al. "Terascale direct numerical simulations of turbulent combustion using S3D". In: *Computational Science & Discovery* 2.1 (2009), p. 015001.

[21] *ConnextX-3*. http://www.mellanox.com/related-docs/user_manuals/ConnectX-3%20VPI_Single_and_Dual_QSFP+_Port_Adapter_Card_User_Manual.pdf.

[22] N. Dayan, M. Athanassoulis, and S. Idreos. "Monkey: Optimal Navigable Key-Value Store". In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*. 2017, pp. 79–94. DOI: 10.1145/3035918.3064054.

[23] D. Doerfler, B. Austin, B. Cook, J. Deslippe, K. Kandalla, and P. Mendygral. "Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC". In: *Proceedings of the 2017 Cray User Group (CUG 2017)*. https://cug.org/proceedings/cug2017_proceedings/includes/files/pap117s2-file1.pdf. 2017.

[24] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O". In: *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER 12)*. 2012, pp. 155–163. DOI: 10.1109/CLUSTER.2012.26.

[25] *Exascale Computing Project (ECP)*. https://www.exascaleproject.org/.

[26] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. "Cuckoo Filter: Practically Better Than Bloom". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT 14)*. 2014, pp. 75–88. DOI: 10.1145/2674005.2674994.

[27] H. N. Greenberg, J. Bent, and G. Grider. "MDHIM: A Parallel Key/Value Framework for HPC". In: *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 15)*. 2015, pp. 10–10.

[28] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency". In: *Proceedings of the 2015 IEEE Annual Symposium on High-Performance Interconnects (HOTI 15)*. 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19.

[29] *Intel Omni-Path Host Fabric Adapter 100 Series*. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/omni-path-host-fabric-interface-adapter-brief.pdf.

[30] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. "Incremental Organization for Data Recording and Warehousing". In: *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 97)*. 1997, pp. 16–25.

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

[32] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. "I/O Performance Challenges at Leadership Scale". In: *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. 2009, 40:1–40:12. DOI: 10.1145/1654059.1654100.

[33] *LANL Crossroads*. https://www.lanl.gov/projects/crossroads/. 2018.

[34] *LANL Trinity*. http://www.lanl.gov/projects/trinity/.

[35] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. "Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures". In: *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 10)*. 2010, pp. 1–12. DOI: 10.1109/SC.2010.28.

[36] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. "ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table". In: *Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 13)*. 2013, pp. 775–787. DOI: 10.1109/IPDPS.2013.110.

[37] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing". In: *Proceedings of the Ninth European Conference on Computer Systems (EuroSys 14)*. 2014, 27:1–27:14. DOI: 10.1145/2592798.2592820.

[38] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. "SILT: A Memory-efficient, High-performance Key-value Store". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*. 2011, pp. 1–13. DOI: 10.1145/2043556.2043558.

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

[40] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. "Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO". In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC 11)*. 2011, pp. 49–60. DOI: 10.1145/1996130.1996139.

[41] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "WiscKey: Separating Keys from Values in SSD-conscious Storage". In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 133–148.

[42] L. McVoy and C. Staelin. "Lmbench: Portable Tools for Performance Analysis". In: *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX ATC 96)*. 1996, pp. 23–23.

[43] *Mercury Runner*. https://github.com/pdlfs/mercury-runner.

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

[45] T. Mudge and U. Holzle. "Challenges and Opportunities for Extremely Energy-Efficient Processors". In: *IEEE Micro* 30.4 (July 2010), pp. 20–24. DOI: 10.1109/MM.2010.61.

[46] R. A. Oldfield, G. D. Sjaardema, G. F. Lofstead II, and T. Kordenbrock. "Trilinos I/O Support Trios". In: *Sci. Program.* 20.2 (Apr. 2012), pp. 181–196. DOI: 10.1155/2012/842791.

[47] *ORNL Summit*. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. 2018.

[48] R. Pagh and F. F. Rodler. "Cuckoo Hashing". In: *J. Algorithms* 51.2 (May 2004), pp. 122–144. DOI: 10.1016/j.jalgor.2003.12.002.

[49] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. "A General-Purpose Counting Filter: Making Every Bit Count". In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*. 2017, pp. 775–787. DOI: 10.1145/3035918.3035963.

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

[51] K. Ren, Q. Zheng, S. Patil, and G. Gibson. "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion". In: *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*. 2014, pp. 237–248. DOI: 10.1109/SC.2014.25.

[52] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. "Building Workload-independent Storage with VT-trees". In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. 2013, pp. 17–30.

[53] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36.2 (Mar. 2016), pp. 34–46. DOI: 10.1109/MM.2016.25.

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

[55] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu. "EDO: Improving Read Performance for Scientific Applications through Elastic Data Organization". In: *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER 11)*. 2011, pp. 93–102. DOI: 10.1109/CLUSTER.2011.18.

[56] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. Ø. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw. "A Scalable Parallel Framework for Analyzing Terascale Molecular Dynamics Simulation Trajectories". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 08)*. 2008, 56:1–56:12.

[57] V. Vasudevan, D. G. Andersen, M. Kaminsky, J. Franklin, M. A. Kozuch, I. Moraru, P. Pillai, and L. Tan. "Challenges and Opportunities for Efficient Computing with FAWN". In: *SIGOPS Oper. Syst. Rev.* 45.1 (Feb. 2011), pp. 34–44. DOI: 10.1145/1945023.1945029.

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

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

[60] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. "An Experimental Study of Bitmap Compression vs. Inverted List Compression". In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*. 2017, pp. 993–1008. DOI: 10.1145/3035918.3064007.

[61] T. Wang, A. Moody, Y. Zhu, K. Mohror, K. Sato, T. Islam, and W. Yu. "MetaKV: A Key-Value Store for Metadata Management of Distributed Burst Buffers". In: *Proceedings of the 2017 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 17)*. 2017, pp. 1174–1183. DOI: 10.1109/IPDPS.2017.39.

[62] K. Wu, E. J. Otoo, and A. Shoshani. "Optimizing Bitmap Indices with Efficient Compression". In: *ACM Trans. Database Syst.* 31.1 (Mar. 2006), pp. 1–38. DOI: 10.1145/1132863.1132864.

[63] X. Wu, Y. Xu, Z. Shao, and S. Jiang. "LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data". In: *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 71–82.

[64] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. "SuRF: Practical Range Query Filtering with Fast Succinct Tries". In: *Proceedings of the 2018 International Conference on Management of Data (SIGMOD 18)*. 2018, pp. 323–336. DOI: 10.1145/3183713.3196931.

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

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

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

[68] Q. Zheng, G. Amvrosiadis, S. Kadekodi, G. A. Gibson, C. D. Cranor, B. W. Settlemyer, G. Grider, and F. Guo. "Software-defined Storage for Fast Trajectory Queries Using a deltaFS Indexed Massive Directory". In: *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 17)*. 2017, pp. 7–12. DOI: 10.1145/3149393.3149398.

[69] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo. "Scaling Embedded In-situ Indexing with deltaFS". In: *Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 18)*. 2018, 3:1–3:15.