Compact Filters for Fast Online Data Partitioning

Qing Zheng†, Charles D. Cranor†, Ankush Jain†, Gregory R. Ganger†, Garth A. Gibson†, George Amvrosiadis†,
Bradley W. Settlemyer‡, Gary Grider‡
†Carnegie Mellon University, ‡Los Alamos National Laboratory
{zhengq, chuck, ankushj, ganger, garth, gamvrosi}@cmu.edu, {bws, ggrider}@lanl.gov

***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, one partitions data by key across the CPU cores of a parallel job. Then each core indexes its subset as data is persisted. Online partitioning requires transferring data over the network so that it can be indexed and stored by the core responsible for the data. This approach is becoming increasingly costly as new computing platforms emphasize parallelism instead of individual core performance that is crucial for communication libraries and systems software in general. In addition to indexing, scalable online data partitioning is also useful in other contexts such as load balancing and efficient compression.**

We present FilterKV, an efficient data management scheme for fast online data partitioning of key-value (KV) pairs. FilterKV reduces the total amount of data sent over the network and to storage. We achieve this by: (a) partitioning pointers to KV pairs instead of the KV pairs themselves and (b) using a compact format to represent and store KV pointers. Results from LANL show that FilterKV can reduce total write slowdown (including partitioning overhead) by up to 3x across 4096 CPU cores.

I. 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 comb through all the data we are generating in order to extract insight [1–4]. Fortunately, computational power, as defined in FLOPS, continues to increase in each cluster [5–7]. This has led the research community to move towards performing computation on data ***in-situ***, i.e., as it streams to storage [8].

One example of such computation is the in-situ generation of data indexes [9]. In-situ data indexing trades off increased computation at write time for improved data access speed at read time. The best read performance is achieved when data is dynamically partitioned during the writing of it such that queries following the writes are restricted to small subsets of data. Recent work has shown that in-situ indexing of key-value (KV) pairs can be performed at the scale of hundreds of thousands of CPU cores [10]. In addition to indexing, online data partitioning is also useful in other contexts such as data compression and load balancing.

978-1-7281-4734-5/19/$31.00

The increase in computational power observed in modern clusters often comes from increasing parallelism at the cost of individual core performance [11]. GPUs and manycore CPUs [12] that use a large number of simpler, independent cores are becoming prevalent across the industry and in modern High Performance Computing (HPC) platforms [5, 6]. This is bad news for network-intensive workloads, as many network operations depend on the performance of individual cores [13]. 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 [14–16]. While HPC systems typically give applications direct access to the NIC, performance still depends on how fast CPUs can communicate with the NIC and how fast they can handle operations (e.g., tag matching) that are not processed by the NIC. Moreover, library code that directly accesses NIC buffers 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 when a program is executed on a GPU- or manycore CPU-based computing platform. As we show in Section II, 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. More importantly, the impact this will have on application performance will increase with the number of RPCs the application performs.

To reduce the total 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, thus moving it quickly off the network. Then partitioning is performed on a compact representation of the KV pairs. This compact KV pair representation consists of a prefix derived from the key and the ID of the process that generated the KV pair. In Section IV, we show that this representation is more compact than previous state-of-the-art work that moves keys with pointers to values [17, 18].

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 [19–22]. We partition all of this data in-situ and index it as it is written to storage. In practice, this means that our approach is tailored

TABLE I: 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 one needs to budget for each key to bound the number of data partitions per query per key to 2 (b2) or 10 (b10). We explain these numbers further in Section IV.

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

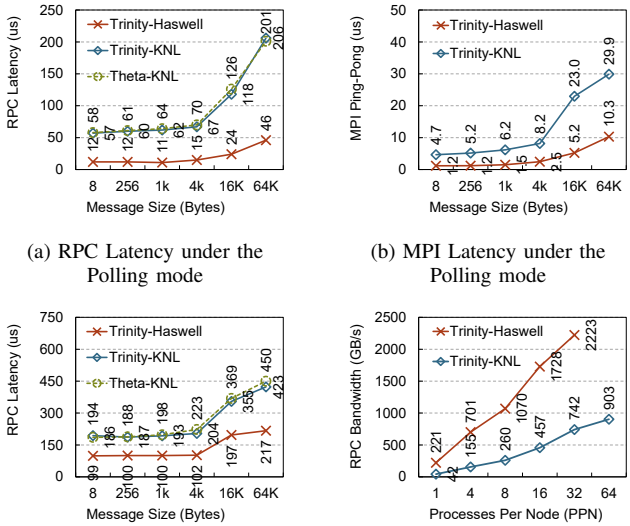
to applications with bursts of I/O activity where a partitioning can be decided on-the-fly while load balancing all the CPU cores. The reason we consider HPC simulations an interesting use case is because they routinely exhibit extreme entropy in the way they generate keys. This means that our work makes 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. Compared with moving entire KV pairs, we show that FilterKV can reduce total write slowdown by up to 3x across 4096 CPU cores depending on both available network bandwidth and available underlying storage bandwidth. Compared with the current state-of-the-art scheme that moves only keys and value pointers, FilterKV can also reduce write slowdown by up to 1.9x with a negligible increase in query latency.

The remainder of the paper is organized as follows. In Section II we provide results that motivate the need for a more compact data management scheme. In Section III and IV we present the design and implementation of FilterKV. Section V presents our evaluation of FilterKV compared with the current state-of-the-art. Finally, we present related work in Section VI and conclude in Section VII.

II. Motivation

As we build machines with increasing computational power, or FLOPS [6, 23], energy efficiency becomes increasingly important. This has led to the rising popularity of manycore processors [12]. Manycore processors feature more CPU cores per die, but each core operates at a lower frequency and is less aggressively optimized for single-thread performance (e.g., fewer reorder buffers or branch predictors for more cores). Because power consumption decreases approximately quadratically as CPU frequency decreases, machines equipped with manycore processors tend to be more energy efficient. As Table I shows, there are a growing number of computing platforms built partially or entirely with manycore processors. Examples of these systems include the Trinity computer at LANL [24], Cori at NERSC [25], and Theta at ANL [26].

The performance of these modern computing platforms can be best utilized through explicit parallel processing. This can be achieved by programming applications to use a large number of threads that each execute as many SIMD instructions



(a) RPC Latency under the Polling mode

(b) MPI Latency under the Polling mode

(c) RPC Latency under the Blocking mode

(d) RPC Bandwidth under the Blocking mode

Fig. 1: **Results from LANL (Trinity) and ANL (Theta) comparing RPC performance between an Intel multicore processor (Haswell) and two Intel manycore processors (KNL). For reference, we also show in Figure (b) MPI ping-pong latency numbers obtained using the OSU benchmark.**

as possible [12]. However, as a result of considerably reduced per-core performance, single-threaded request handlers found in existing systems software (which can be difficult to vectorize) are no longer fast enough to meet latency targets [11]. Inter-process data communication, perhaps surprisingly, is one immediate victim of this architectural shift.

To demonstrate the impact of modern manycore processors on inter-process data communication, we developed an RPC benchmark program [27]. Our program uses the Mercury RPC framework [28] and libfabric [29] to enable communication over various low-level interfaces including TCP, RDMA, and vendor-specific APIs such as the Cray's GNI API [30, 31]. Using this benchmark we ran tests on the Trinity and Theta computing platforms at LANL and ANL. Both are Cray machines equipped with the Aries interconnect [14]. 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) [12]. All Theta compute nodes use 1.3 GHz KNL processors. Our tests evaluate each of the 3 processor types. To measure RPC latency, we ran a sender and a receiver process on two 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 relinquish control over the CPU when no events are present. We report average RPC latency.

In Figure 1a, we see that the RPC latency measured on the two KNL platforms is noticeably higher (about 4x in our runs) than that on Haswell. Moreover, in Figure 1c we see that this reduction in performance is more significant (in absolute terms) when the RPC implementation does not poll for new events and thus may have multiple context switches when processing an incoming event. The stark performance

gap between KNL and Haswell processors is due in a large part to the fact that the latencies of RPC processing, context switching, and system call handling are all a function of the single-thread performance of the underlying processor rather than the throughput of it. Likewise, our results from LMbench [32] show that it can take roughly 6x longer to fork a process on a KNL node than on a Haswell node.

For reference, we show in Figure 1b MPI latency numbers obtained using the OSU benchmark [33]. KNL latency is still about 4x higher than Haswell. Note that while MPI code tends to have better network performance, it provides a different set of semantics (e.g., requiring all peers to bootstrap together) than that of RPC. Thus, RPC-based software code might not be able to leverage MPI development efforts.

When per-core performance is low, total performance may be improved by splitting a task over multiple cores and having all the cores progressing in parallel. Figure 1d shows the total RPC bandwidth we can achieve per node when performing all-to-all data shuffling across 32 Trinity compute nodes using different numbers of processes per node (PPN). We fix RPC message size at 16K bytes, which is the largest payload GNI supports without requiring bulk transfers [31]. Results show that while KNL nodes have twice as many CPU cores as Haswell nodes, per-node RPC bandwidth on KNL nodes is still roughly 3x lower than that on Haswell nodes. These results further exemplify the impact of modern computing processors on inter-process data communication.

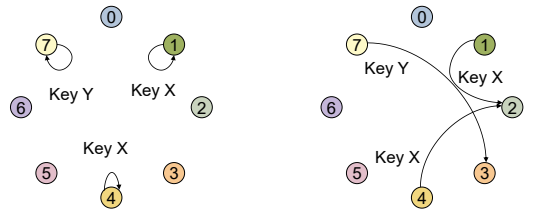
While our work is mainly motivated by the reduced network performance observed on certain recent computing platforms, this is not the only scenario where applications may have to undergo higher network communication cost. Communication cost increases as one scales from thousands of processes to hundreds of thousands of processes [10]. Interference from concurrent jobs and contention at network devices may also cause applications to experience reduced network performance [34, 35]. The fast data partitioning techniques discussed in this paper can be applied to these scenarios as well.

III. Fast Data Partitioning Designs

While HPC systems are often built using fast interconnects, the actual network performance of these computing systems can be significantly influenced by the compute node processor architecture and by other bottlenecks within the system. This can make online data partitioning potentially prohibitively expensive. We believe that the key to mitigating this problem is to greatly reduce the total amount of data exchanged over the network (when partitioning data) so that the overall data partitioning process is less subject to the hosting environment. In this section we state the data partitioning problem, describe the current state-of-the-art solution to it (simple data indirection), and present FilterKV, our own solution.

A. The Data Partitioning Problem

In this paper we model data as KV pairs [36–39]. We focus on the partitioning of data among the processes of a parallel application and its impact on performance. Each application process owns a data partition. Each data partition corresponds to a disjoint subset of keys. A data partitioning function is used by the application to determine the process responsible for a key subset. Each process is a sender of data, and we assume that any process can also be a receiver for a subset of keys. Our goal is to reduce the total amount of data that must be communicated to remote peers.



(a) Without Data Partitioning (b) With Data Partitioning

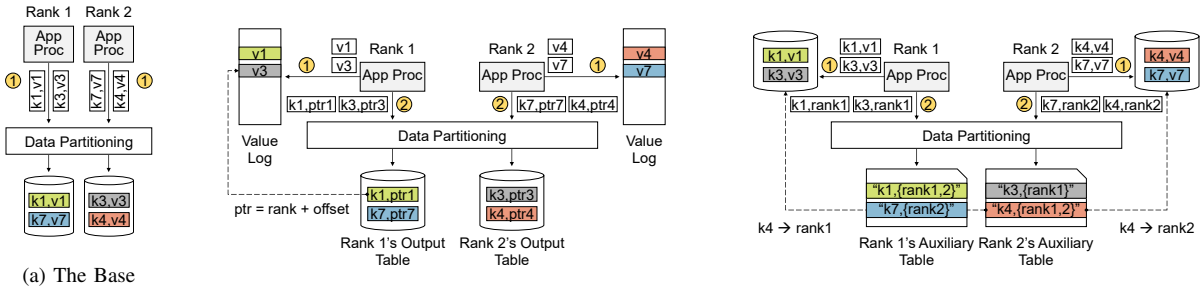
Fig. 2: **Distributed processes of a parallel application writing data output. Without data partitioning, data is streamed to per-process files so per-key data ends up in multiple files and reading a key may require searching all files. With data partitioning, each key is sent to a particular process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

Many applications, especially those in the scientific domain, output data without necessarily considering the efficiency of followup queries [3]. For these applications, online data partitioning can improve query performance by confining queries to individual partitions of a dataset. As an example, Figure 2a shows a case in which output data is not partitioned and is directly written to per-process files. As a result, a subsequent query may have to scan an entire dataset in order to recall the data of a key. However, with data partitioning all data of a key is grouped at a particular process before writing. Therefore, reading that key requires searching only a single data partition, as illustrated in Figure 2b.

In this paper we focus on data partitioning that takes place *in-situ* with application I/O. This is because in-situ processing partitions data as it is written so data can be reorganized without expensive readbacks [40–42]. For computing platforms that use shared remote storage as opposed to on-node storage [20, 21], writing data to storage consumes a portion of the compute node's total available network bandwidth. As such, the fraction of the compute node's network bandwidth not consumed by storage I/O can be utilized to perform in-situ data operations. We refer to this bandwidth as ***residual network bandwidth***. While our previous work considered cases where this bandwidth is high [9, 10], here we handle cases where residual network bandwidth is limited making online data partitioning prohibitive. Note that this limit on residual network bandwidth may be due to the processor being unable to fully saturate the NIC as opposed to a property of the network itself (e.g., low NIC bandwidth).

B. Simple Data Indirection

The current state-of-the-art uses data indirection to avoid overloading the network when it is slow. With data indirection, one sends keys and pointers to values rather than entire KV pairs. To do this, an application process writes the value portion of a KV pair to a per-process log file, as shown in



(a) The Base Format shuffles intact KV pairs

(b) The Current State-of-the-Art shuffles keys with pointers to values

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

Fig. 3: **Illustration of three 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 but storing pointers in addition to values can incur a significant amount of extra I/O to storage when value size is relatively small. (c) Finally, by persisting data pointers in a lossy manner, location information can be stored using less space, reducing total I/O while still permitting efficient reads.**

Figure 3b. Next, the offset of the write and the rank number of the process (used as a partition ID) is encoded into a pointer. The process then sends the composite pointer along with the key to the partition where the key belongs. With pointers and keys stored together at their respective 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 the actual data is a reduction in the total amount of data exchanged over the network. However, storing pointers in addition to the original 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 their respective KV pairs, this is not always the case. Values smaller than 250 bytes are reported to be the norm for Facebook's Memcached [43]. Similarly, scientific application output often consists of a large number of objects smaller than 50 bytes [44, 45]. In these cases, applying indirection may end up adding more overhead to the underlying storage (in the form of increased I/O time) than is removed from the network. We study this in more detail in Section V. While data compression helps, storage overhead may be high even after data compression, as we show in Section IV. To be able to more effectively attack this problem, we show that one can use filter data structures to more compactly represent data pointers so that they can be stored using considerably less space, even before compression.

C. FilterKV: Reducing I/O and Space Overhead

To improve performance beyond simple data indirection, one needs to reduce its I/O and space overhead when KV size is small. Recall from Figure 3b that with simple data indirection the write-path code only sends 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 are able to learn where to recover the value associated with a key by reading the pointer stored alongside the key. Each pointer identifies the log file to which the value is written and the offset in the log file where the value resides. In practice, indirection 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 recall per-key data.

Our approach, referred to as FilterKV, exploits lossiness to reduce data indirection overhead. Rather than recording a key's exact data location, we map each key to a list of candidate data locations of which only one truly stores the data corresponding to the key. The loss of "accuracy" here enables us to store less information. We show in Section V that this loss in accuracy does not significantly impact query performance.

We use filter data structures to reduce the accuracy of our data pointers and the total amount of extra data we must store. Figure 3c shows a high-level picture of our design: instead of writing values to per-process output files and sending keys with pointers to their respective cores, 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 responsible cores. The final data output consists of two types of tables. One stores the original KV pairs. The other maps keys to their source data locations. We refer to the first type of table as ***main table***, and the second type of table as ***auxiliary table***.

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 at the auxiliary table responsible for the key. The reader then goes to the corresponding main table to retrieve the data of the key. Due to intentionally reduced accuracy, it is possible for keys to be mapped to multiple source locations. In such cases, a reader program searches all these locations, potentially concurrently, until it finds the data of interest. Because main tables are packed with complete KV pairs, readers know when they hit a key.

IV. Implementation and Measurements

To avoid overloading the network when partitioning data, the current state-of-the-art method adds an index entry for every data record it handles. Then partitioning is performed on indexes instead of the original data. We call this simple data indirection, and it can add a significant I/O and storage overhead when data is small. To alleviate this overhead, we have designed FilterKV. It includes a compact mapping

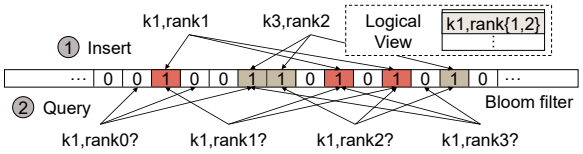


Fig. 4: **Compactly mapping keys to ranks using Bloom filters. Step 1 inserts two key-rank mappings into the filter, turning two subsets of the filter's bits from 0 to 1. Step 2 retrieves k1's mapping by exhaustively checking all possible mappings for k1. Due to false positives, a Bloom filter may map 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).**

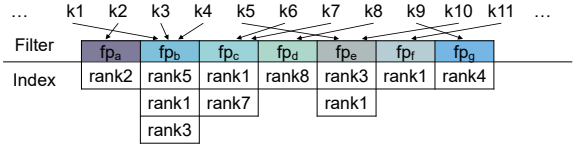


Fig. 5: **A hybrid auxiliary table design using both filters and indexes. Keys are stored as small fingerprints in a filter. Each fingerprint is mapped to one or more ranks at the index layer. False positives are still possible because multiple keys may be mapped to a single key fingerprint. Readers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

structure (Figure 3c) able to produce much less index data than the current state-of-the-art method does (Figure 3b). To reduce index size, we allow individual keys to be mapped to multiple data locations using our compact lossy auxiliary tables. In this section we present techniques for implementing these tables while keeping query latency low. Note that because keys can be mapped to multiple data locations, a query may have to check all of them in order to find the data of interest.

A. Our Initial Implementation using Bloom Filters

One way to structure auxiliary tables in a compact lossy format is to use ***filters***. Filters are typically dense data structures whose canonical use involves managing memberships. In these applications, one inserts keys into a set and then asks if a key is in the set. Typically a filter returns "False" when a key is *definitely not* in the 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 [46–49]. In this section we use the Bloom filter [46] as an example filter to implement our auxiliary tables.

In FilterKV, auxiliary tables are used to map keys to their source data locations. To store these key-location 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 (Figure 4). To recall the source location of a key, we execute a series of queries against the filter. Each query targets a distinct data location, testing if there *might* exist a mapping from the key to that particular data location. We try all data locations of a dataset. For example, Figure 4 shows a case in which there are 4 data partitions. So we run 4 different queries to test all of them (1 partition each). The total number of source data locations (including false positives) we get for a key is dictated by the filter data structure's false positive rate, which 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 false positives.

Return to Figure 3c. 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 I the 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 at 2 (b2) or 10 (b10). For each machine, we imagine running an application that consumes the entire machine such 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 index overhead borne by the current state-of-the-art as we analyzed in Section III-B.

B. A Filter-Index Hybrid Implementation for FilterKV

The core of FilterKV is an efficient data partitioning scheme that reduces the total amount of data exchanged over the network. We achieve this by maintaining an auxiliary table at every data partition to record the source location of each individual key. Auxiliary tables take extra space. To minimize I/O and space overhead, we need to additionally restrict the size of these auxiliary tables on storage and we must do so without considerably increasing query latency. In Section IV-A, we discussed representing auxiliary tables using compact filters such as the Bloom filter.

Unfortunately, while filters scale well for modern computing platforms with millions of CPU cores, they can be problematic for future exascale computing platforms [23] with tens or hundreds of millions of processing cores producing tens or hundreds of millions of data partitions. To see why, recall from Figure 3c that auxiliary tables are used to map keys to the data locations that *might* store the keys. 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 key-location mappings of the key. As the number of filter tests we need to perform approaches the number of

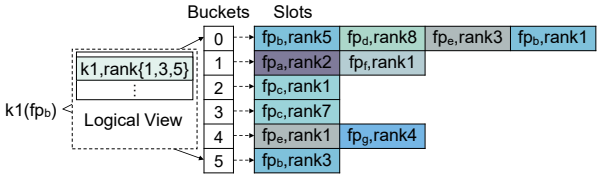


Fig. 6: **Compactly mapping keys to ranks with partial-key cuckoo hash tables. A cuckoo hash table consists of an array of buckets. Each bucket holds up to a certain amount of data slots (4 in this example). Each slot stores a partial-key (fingerprint) and the rank it maps to. This figure shows 6 buckets and 11 non-empty slots. Key k1 can be stored either at bucket 0 or 5. Because different keys may share fingerprints, 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.**

data partitions a dataset has, processing a query may require running an excessive amount of filter tests, and the latency of these tests may no longer be dwarfed by storage reads.

To attack this problem, we build upon our filter-based auxiliary table design discussed in Section IV-A 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 set of user keys 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 IV-A. On the other hand, directly remembering the source locations of each fingerprint as indexes prevents queries from having to perform a potentially large number of filter tests 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 IV-C.

To implement FilterKV auxiliary tables using this new hybrid design, we use partial-key cuckoo hash tables [47, 50], a cuckoo hash table [51, 52] 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 is transformed into 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 sizes are powers of 2, so each resize doubles the size of a table [47]. Mapping every key to two potential locations in the table allows for high levels of table space utilization before a table must be resized [53]. 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 unnecessary memory and on-storage space usage.

To minimize wasted space, our implementation creates a new table when the current 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.

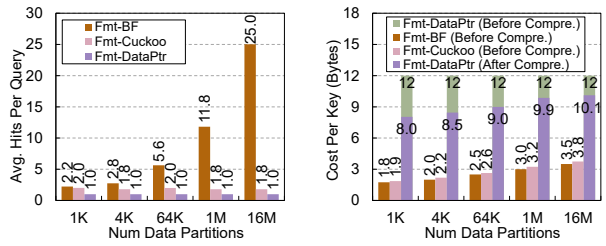
C. Measurements

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 this is that a query may have to perform additional lookups to find the data of a key. In this section we evaluate this tradeoff. We study 3 data partitioning schemes: data indirection as the current state-of-the-art (Fmt-DataPtr), FilterKV using Bloom filters (Fmt-BF), and FilterKV using partial-key cuckoo hash tables (Fmt-Cuckoo).

Test runs consist of generating 16 million keys and storing their indexing information using different formats. 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 1024 to 16 million (the largest current supercomputer have about 10 million CPU cores). For the current state-of-the-art (Fmt-DataPtr), we set each data pointer to be 12 bytes consisting of an 8-byte offset and a 4-byte rank number as per Section III-B. For the FilterKV using Bloom filters (Fmt-BF), we configure each Bloom filter to budget 4 + $\log(N)$ bits for each key so that it uses the same amount of space as its partial-key cuckoo hash table counterpart. Finally, for the FilterKV using partial-key cuckoo hash tables (Fmt-Cuckoo), we configure each partial-key (key fingerprint) to be 4 bits and its value to be $\log(N)$ bits so that it has enough bits to distinguish all partitions.

Figure 7a shows the read overhead associated with each data partitioning scheme. Note that each scheme can be viewed as a table mapping keys to the data partitions that *might* store the keys. We measure ***query amplification***, which counts the average number of data partitions a scheme returns for a key. Unsurprisingly, the number is always 1 for the current state-of-the-art (Fmt-DataPtr) as it keeps complete information for each key. For the Bloom filter format (Fmt-BF), its query amplification is essentially a function of *both* the filter's false positive rate and the total number of data partitions. While the filter's false positive rate keeps decreasing as we increase the number of Bloom filter bits (4 + $\log(N)$) we budget for each key, the decrease wasn't significant enough to compensate for the increase in the total number of data partitions (N). Consequently, the scheme's overall query amplification keeps increasing, though logarithmically, as the total number of data partitions increases. To bound query amplification, we could configure the filter to budget 4 + 1.44$\log(N)$ bits per key (according to Bloom filter math) rather than the 4 + $\log(N)$ bits we tested, at the cost of increased space overhead.

Unlike the Bloom filter format (Fmt-BF), the average number of data partitions the Cuckoo format (Fmt-Cuckoo) returns per query is bounded (roughly 2 in these tests), and does not increase as the total number of data partitions increases. This



(a) Query Amplification (b) Per-Key Space Overhead

Fig. 7: **Comparison of three different data partitioning schemes. Fmt-DataPtr is the current state-of-the-art. Figure (b) shows Fmt-DataPtr's space usage both before and after data compression is applied.**

is because Cuckoo directly maps keys to source data partitions so the scheme's query amplification is *only* a function of the filter's false positive rate (no exhaustive filter tests) and is *not* affected by the total number of data partitions (N).

Figure 7b shows each scheme's space overhead, measured as the number of bytes per key. In the current state of the art, the space overhead is shown both before and after data compression (overlaid on the same bar). We used Google's Snappy compression code to compress data [54]. As the current state-of-the-art (Fmt-DataPtr) keeps complete information for each key, it consumes the most space in spite of compression. Further, we see that the index entropy increases as the number of partitions is increased, thus reducing the benefits of compression at scale. Both FilterKV schemes are able to use much less space due to their compact, albeit lossy, storage representations even before data compression. The Cuckoo format (Fmt-Cuckoo) used slightly more space than the Bloom filter format (Fmt-BF). This is because none of the Cuckoo runs were able to fully utilize the space in their respective cuckoo hash tables, as Section IV-B explains.

V. Experiments

This section evaluates the end-to-end performance of different online data partitioning schemes. We compare data partitioning that shuffles full KV pairs (Fmt-Base), data partitioning that uses indirection (Fmt-DataPtr), and data partitioning that implements both data indirection and lossy data pointers using partial-key cuckoo hash tables (Fmt-FilterKV).

A. Microbenchmark Results

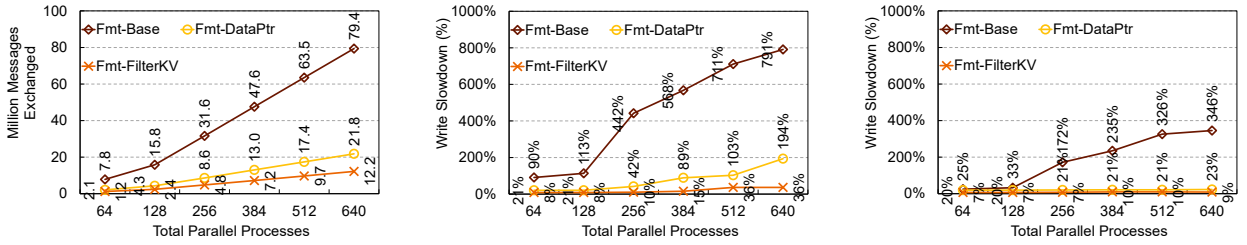
Our first group of experiments evaluate the performance of different data partitioning schemes under different job and KV size configurations. These experiments were performed on the Narwhal computing cluster at CMU [55]. Each Narwhal compute node consists of 4 CPU cores, 16GB memory, and an 1000Mbps NIC for network communication. These compute nodes are interconnected by a fat tree Ethernet with a 14:6 oversubscription ratio at the access layer and a 24:20 ratio at the distribution layer. Inter-process communication among a large number of Narwhal compute nodes is expected to be expensive in general, making Narwhal an ideal testbed for comparing the effectiveness of different data partitioning schemes in controlling and balancing their total network and I/O activities. In this particular group of experiments, network communication cost is high because the total network bandwidth available to each compute node is extremely limited. While HPC platforms are routinely paired with faster interconnects, their storage is faster too, making efficient data communication no less important. We study performance atop a real-world HPC cluster in Section V-B.

We developed a simple program to drive our tests. In each run, we start a certain number of parallel processes and have each process generate random KV pairs of a certain size. Each process is both a data sender and a receiver. In the first set of runs, we fix KV size at 64 bytes and vary the total number of processes from 64 to 640 using up to 160 Narwhal compute nodes. The results are shown in Figure 8. In the second set of runs, we fix the total number of processes at 256 using 64 Narwhal compute nodes and vary KV size from 16 to 192 bytes. The results are shown in Figure 9. Note that while KV sizes can differ across runs, all keys are fixed at 8 bytes. Also fixed is the total amount of data each process generates, which is set at 960MB per run. Our program buffers at most 16MB of data in memory before writing it to storage efficiently.

We compare two different levels of residual network bandwidth representing two different levels of network communication cost. To better understand this cost, we consider a real-world machine as a case study. The Trinity supercomputer at LANL features one burst-buffer storage node per 32 compute nodes [24]. All Trinity compute nodes and burst-buffer nodes are deployed within a single interconnection network. Both types of nodes carry the same type of NIC, and writing data from compute nodes to burst-buffer is largely bottlenecked at the NICs of the burst-buffer nodes. Therefore, the residual network bandwidth (total available network bandwidth - storage bandwidth) for each Trinity compute node is roughly 97% (1 - 1/32) of the node's total available network bandwidth. As shown in Figure 8 and 9, by configuring residual network bandwidth at 50% and 75% in our tests, we emulate cases where the actual network performance observed by an application is considerably lower than advertised.

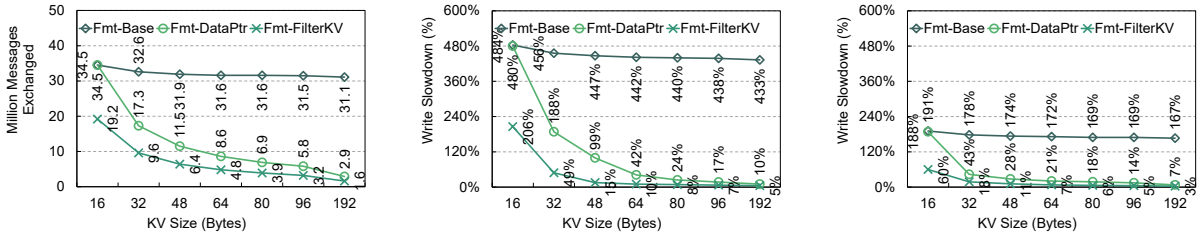
We use ***write slowdown*** to gauge the total data partitioning overhead during the writing of data to storage. This is measured as the additional time each run must spend to finish writing all the data. For example, write slowdown is said to be 100% if it takes a run twice of the time to write the data with data partitioning, as opposed to directly writing the data to storage without performing any online data operations. Note that this overhead includes both the overhead associated with network communication and the overhead incurred by writing filter and indexing information in addition to the original data.

Figure 8 compares performance as a function of job size. We also report the total number of RPC messages each run sends. The base format (Fmt-Base) shuffles intact KV pairs so a large amount of data is sent across the network in each run. The overall data partitioning overhead rises quickly as job size increases. The current state-of-the-art (Fmt-DataPtr) shuffles keys and pointers to values so much less network activities are needed to partition data. This leads to a much lower write slowdown (even though it writes more data) than the base



(a) Total RPC Messages Exchanged (b) 50% Residual Bandwidth (500Mbps) (c) 75% Residual Bandwidth (750Mbps)

Fig. 8: **Results comparing the performance of different online 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 KV pairs per process. Each KV pair is 64 bytes.**



(a) Total RPC Messages Exchanged (b) 50% Residual Bandwidth (500Mbps) (c) 75% Residual Bandwidth (750Mbps)

Fig. 9: **Results comparing the performance of different online data partitioning schemes under different KV sizes. We fix keys at 8 bytes and vary the total KV size (K + V) from 16 to 192 bytes. Each run uses 256 parallel processes (64 nodes) and generates the same amount of total KV data.**

format. Finally, by combining indirection with a lossy data indexing scheme, FilterKV is able to decrease the total amount of indexing data written to storage compared with the current state-of-the-art while sending even less data to the network (because no data offsets need to be sent), further reducing write slowdown.

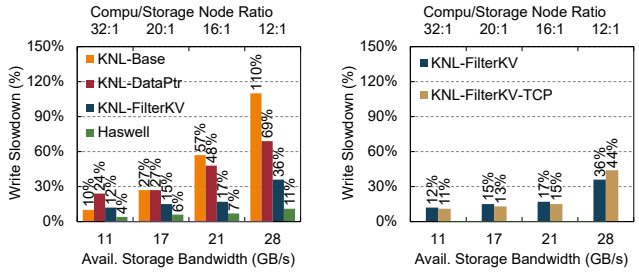
Figure 9 compares performance as a function of KV size. The base format shuffles and stores intact KV pairs. Thus its performance does not change with KV size. Both the current state-of-the-art and FilterKV use data indirection to reduce network communication. The indexes these two formats generate can cause a significant amount of extra storage I/O when KV size is small, and are less significant when KV size is large. As such, the overall data partitioning overhead of these two formats decreases as KV size increases. While FilterKV outperforms the current state-of-the-art in all cases, the advantage is most critical when KV size is between 32 and 64 bytes which is the data size of many scientific workloads [44, 45]. While we didn't test keys larger than 192 bytes, we expect the difference between the two formats to continue to shrink as the figure shows.

B. Macrobenchmark Results

Our second group of experiments evaluate the end-to-end performance of FilterKV under a real-world scientific use case: plasma physics. As a case study, our experiments use the Vector Particle-In-Cell (VPIC) simulation code from LANL to perform magnetic reconnection simulations [44, 56]. In such simulations, each simulation process manages a region of cells in the simulation space through which particles move. Every few timesteps the simulation stops and each process writes to storage the state of all the particles currently managed by the process. In our tests, state for each particle is 64 bytes. VPIC scientists are interested in the state of a tiny subset of particles at individual timesteps. The identities of these particles are not known until the end of a simulation. Unmodified VPIC directly writes data into per-process output files. But because particles move during a simulation, per-particle state often ends up in multiple output files. This means that without post-processing, reading back a particle would require searching an entire VPIC dataset. To speed up queries, we use our previous work, DeltaFS Indexed Massive Directories, to dynamically partition and index data as it streams to storage [9, 10]. To partition data, our previous implementation shuffles intact KV pairs. With this paper we have modified it to support both the DataPtr format and the FilterKV format.

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

Our test consists of running a VPIC simulation and then performing 100 independent queries. Each simulation uses 4096 processes and simulates a total of 32 billion particles. Approximately 2TB of data is produced per timestep. Particle data is first staged at a burst-buffer allocation and is later written to the platform's underlying filesystem. We use different numbers of burst-buffer nodes to test performance under different network-to-storage ratios. After simulation data is persisted, queries are executed from the underlying filesystem.



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

(b) Write performance of FilterKV on a fast transport (GNI) compared with running on a less performant transport (TCP)

Fig. 10: **Results from LANL Trinity running VPIC simulations and in-situ processing its data using different data management schemes and transports. (a) Higher storage bandwidth makes efficient data management more critical. (b) FilterKV reduces the total amount of data exchanged over the network, making running it on TCP almost identical to running it on GNI, performance-wise.**

Each query randomly targets a particle and a timestep and reads the state of that particle under that timestep.

Figure 10a compares the write slowdown of different online data management schemes as a function of available underlying storage bandwidth. The right-half of the figure focuses on the performance when the storage bandwidth available to a job is high. Higher storage bandwidth increases the impact of network performance on overall performance. As such, using an efficient data partitioning scheme that reduces the total amount of data exchanged over the network becomes crucial. Results show that FilterKV can reduce total write slowdown by up to 3.3x compared with the base format and by up to 2.8x compared with the current state-of-the-art, thus significantly closing the gap between KNL performance and Haswell performance.

The left-half of the figure shows the performance when the underlying storage bandwidth available to a job is low. Lower storage bandwidth means that the overall data writing process is more likely to be bottlenecked on storage. So reducing the total amount of data written to storage using a compact storage representation becomes crucial. Our results show that FilterKV can reduce total write slowdown by up to 2x compared with the current state-of-the-art. Nevertheless, because both FilterKV and the current state-of-the-art writes more data than the base format does, they tend to perform worse than it when available storage bandwidth is relatively low.

Figure 10b shows the performance when we use TCP rather than more efficient Cray GNI to perform low-level network operations. While we are by no means advocating TCP for production jobs, our results show that with FilterKV we can effectively run TCP jobs almost as fast as GNI jobs, suggesting that performance becomes less subject to the network.

Figure 11a-11c compare the read performance of different data management schemes. We report query latency, storage seeks, and total data fetched. Note that in these experiments, each partition is persisted as a flattened LSM-Tree by DeltaFS [9, 10]. The base format uses 3 or more read operations to read back a particle. The first read operation reads the footer of the partition containing the particle of interest. The second read operation loads the partition's indexes (roughly 12MB). The remaining read operations each read a data block until the reader finds the target particle (multiple of 4MB). On average, 3.1 read operations (or storage seeks) were executed per query (i.e., about 10% queries executed 4 or more read operations). The median query latency for the base format is 190 ms.

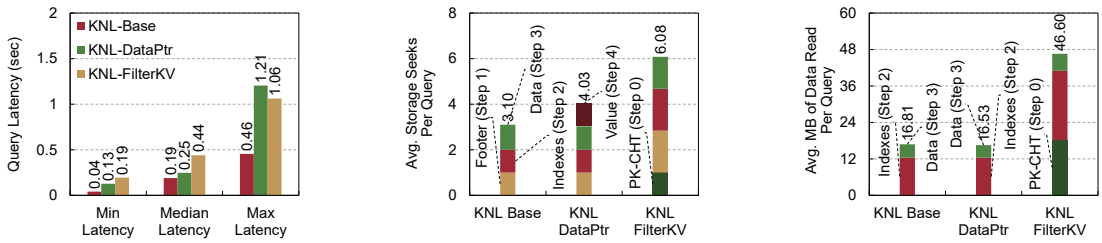
To reduce the total amount of data pushed to the network during the write phase, the current state-of-the-art partitions data using indirection. The cost of applying indirection is one extra read operation per query which increases median query latency from 190 ms to 250 ms in our tests (KNL-DataPtr).

Finally, with a compact lossy storage format for fast data partitioning, each FilterKV query must first read an entire auxiliary table (roughly 18MB each) and then attempt reads at multiple data partitions due to false positives (1.88 partitions per query in these runs). As such, FilterKV has the highest minimum (190 ms) and median (440 ms) read latency among all three data management schemes. Though overall FilterKV shows comparable read performance with the base format and the current state-of-the-art, while being about 200 ms slower.

VI. Related Work

Filter data structures are used by many storage systems to improve read performance. Unlike indexes which directly map keys to data locations, filters speed up queries by indicating where not to read thus saving the query process from performing potentially a large number of unnecessary storage reads [57]. When the key space of an application is bounded, filters can be implemented using compressed bitmaps [58, 59]. When the key space is unbounded, filters are typically implemented through hash-based data structures such as the Bloom filter [46], cuckoo filter [47, 50], and quotient filter [48, 49]. Recently, we have also seen filters implemented using tries such as SuRF [60] and using perfect hash functions such as the ECT structure in SILT [50]. These filter implementations may also be used to implement FilterKV.

The idea of data indirection is used by many LSM-Trees KV stores to reduce the I/O overhead associated with their background data reorganization operations. For example, WiscKey [18] reduces background data reorganization overhead by storing keys and values separately and only performing data reorganization on the keys. Similar use of this idea is also found in systems such as IndexFS [17] and Cassandra [39]. In addition to data indirection, systems such as Monkey [61] and SlimDB [62] use analytical models to generate optimized filter configurations that balance per-filter performance with available memory. This allows for minimizing its overall false positive rate given a fixed memory budget. Such designs work best on dedicated server machines whose entire memory can be used to serve data operations. Unlike these designs, this paper focuses on in-situ contexts in which memory available for data operations may be extremely limited. Finally, VT-Tree [63] 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



(a) Query Latency

(b) Avg. Storage Reads Per Query and Cost Breakdown

(c) Avg. Data Fetched Per Query and Cost Breakdown

Fig. 11: **Performance of reading individual particles from a 2TB dataset. FilterKV shows comparable latency with the base format and the current state-of-the-art. In (b) and (c), cost is categorized by the type of data read from storage. PK-CHT stands for partial-key cuckoo hash tables.**

data indirection. This paper focuses on using data indirection to reduce network communication as opposed to storage I/O. In addition, great emphasis is placed on using compact data representations to reduce space overhead.

Rich in-transit data processing capabilities are provided by multiple middleware libraries such as PreDatA [41], GLEAN [34, 35], NESSIE [64], and DataSpaces [65]. These systems all use auxiliary nodes to provide asynchronously data analysis functions. Similarly, systems such as Damaris [66] and Functional Partitioning [67] co-schedule analysis, visualization, and de-duplication tasks on compute nodes, but require dedicated cores. Unlike these systems, FilterKV processes data only during the writing of it without requiring dedicated computing resources. The GoldRush runtime [68] provides an embedded in-situ analytics capability by scheduling analysis tasks during idle periods in simulation execution using an OpenMP threaded runtime. The analysis tasks leverage the FlexIO [40] capability within ADIOS [42] to create shared memory channels for generating task inputs. Similar to these systems, FilterKV runs inside a parallel application and uses only the idle computing and network resources temporarily available during application I/O to process data.

Large-scale VPIC simulations have been conducted with trillions of particles, generating terabytes of data for each recorded timestep [1, 69, 70]. An important reason FilterKV is particularly useful to VPIC is that per-particle data in VPIC is extremely small so storing data pointers in addition to data can be prohibitively expensive.

VII. Conclusion

In this paper we described a novel data management mechanism, FilterKV, for fast 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.

To achieve space efficiency, we devised a lossy storage scheme for our data pointers. We allow each data pointer to reference multiple source data locations so that we can use less total bits to represent each key-location mapping. We use compact filters to implement this lossy data scheme. We compared two implementations: a filter-only implementation that uses Bloom filters, and a filter-index hybrid implementation that uses partial-key cuckoo hash tables. We expect the first to work well for jobs of moderate scales (e.g., < 1 million parallel processes). By eliminating the need for exhaustive filter tests, the second can handle jobs of even larger scales.

We have evaluated the performance of FilterKV using both microbenchamarks and macrobenchmarks. Results show that FilterKV works best when a job consists of a large number of parallel processes and when the effective network-storage ratio of a job is relatively low. While FilterKV outperforms the current state-of-the-art during the write phase, we expect the gain to be most critical when individual KV size is small. The cost of fusing lossiness into data indexes is increased query time. Our results show that FilterKV is able to provide comparable read performance with the base format and the current state-of-the-art, albeit requiring looking up and reading more data when processing a query.

As data size continues to grow, being able to process data in-situ becomes increasingly important. Online data partitioning, being one type of many possible in-situ operations, reorganizes data as it streams to storage,and improves the long-term value of data by storing it in a better format. With emerging HPC and high performance data analytics platforms combining multiple processor, memory, and storage technologies in new ways, it is also a good opportunity to revisit existing systems software designs and to adapt them to new computing environments. In this paper we revised established data partitioning schemes, better decoupling its performance from the performance of the underlying platform.

Acknowledgment

We thank Jerome Soumagne, Phil Carns, and Robert Ross for their guidance on the Mercury software. This material is based on work supported in part by the US DOE and LANL, under contract DE-AC52-06NA25396 subcontract 394903 (IRHPIT), and by the US DOE, Office of Science, Advanced Scientific Computing Research, under award DE-SC0015234. We also thank the member companies of the PDL Consortium (Alibaba, Amazon, Datrium, Dell EMC, Facebook, Google, HPE, Hitachi, IBM, Intel, Micron, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and Two Sigma). This paper is approved for unlimited release as LA-UR-19-28241.

References

[1] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, *et al.*, "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *SC*, 2012, 59:1–59:12. DOI: 10.1109/SC.2012.92.

[2] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. Ø. Jensen, J. L. Klepeis, *et al.*, "A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories," in *SC*, 2008, 56:1–56:12.

[3] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorszki, Q. Liu, *et al.*, "EDO: Improving read performance for scientific applications through elastic data organization," in *IEEE CLUSTER*, 2011, pp. 93–102. DOI: 10.1109/CLUSTER.2011.18.

[4] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, *et al.*, "Six degrees of scientific data: Reading patterns for extreme scale science IO," in *HPDC*, 2011, pp. 49–60. DOI: 10.1145/1996130.1996139.

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

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

[7] *LANL crossroads*, https://www.lanl.gov/projects/crossroads/, 2018.

[8] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao, I. Ahmad, R. H. Arpaci-Dusseau, *et al.*, "Data storage research vision 2025: Report on nsf visioning workshop held may 30–june 1, 2018," USA, Tech. Rep., 2018.

[9] Q. Zheng, G. Amvrosiadis, S. Kadekodi, G. A. Gibson, C. D. Cranor, B. W. Settlemyer, G. Grider, *et al.*, "Software-defined storage for fast trajectory queries using a DeltaFS indexed massive directory," in *PDSW-DISCS*, 2017, pp. 7–12. DOI: 10.1145/3149393.3149398.

[10] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, *et al.*, "Scaling embedded in-situ indexing with DeltaFS," in *SC*, 2018, 3:1–3:15. DOI: 10.1109/SC.2018.00006.

[11] T. Mudge and U. Holzle, "Challenges and opportunities for extremely energy-efficient processors," *IEEE Micro*, vol. 30, no. 4, pp. 20–24, Jul. 2010. DOI: 10.1109/MM.2010.61.

[12] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, *et al.*, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016. DOI: 10.1109/MM.2016.25.

[13] 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 *CUG 2017*, https://cug.org/proceedings/cug2017_proceedings/includes/files/pap117s2-file1.pdf, 2017.

[14] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," Cray Inc., Tech. Rep. WP-Aries01-1112, Nov. 2012, http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf.

[15] *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.

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

[17] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *SC*, 2014, pp. 237–248. DOI: 10.1109/SC.2014.25.

[18] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *FAST*, 2016, pp. 133–148.

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

[20] J. Bent, B. Settlemyer, and G. Grider, "Serving data to the lunatic fringe: The evolution of HPC storage," *USENIX ;login:*, vol. 41, no. 2, Jun. 2016.

[21] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage Challenges at Los Alamos National Lab," in *MSST*, 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232376.

[22] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *SC*, 2009, 40:1–40:12. DOI: 10.1145/1654059.1654100.

[23] *Exascale computing project (ECP)*, https://www.exascaleproject.org/.

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

[25] *NERSC Cori*, https://www.nersc.gov/users/computational-systems/cori/.

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

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

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

[29] 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 *IEEE HOTI*, 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19.

[30] S.-E. Choi, H. Pritchard, J. Shimek, J. Swaro, Z. Tiffany, and B. Turrubiates, "An implementation of ofi libfabric in support of multithreaded pgas solutions," in *PGAS*, 2015, pp. 59–69. DOI: 10.1109/PGAS.2015.14.

[31] H. Pritchard, E. Harvey, S.-E. Choi, J. Swaro, and Z. Tiffany, "The gni provider layer for ofi libfabric," in *CUG 2016*, https://cug.org/proceedings/cug2016_proceedings/includes/files/pap136s2-file1.pdf, 2016.

[32] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *USENIX ATC*, 1996, pp. 23–23.

[33] *Osu mpi benchmarks*, http://mvapich.cse.ohio-state.edu/benchmarks/.

[34] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *IEEE LDAV*, 2011, pp. 9–14. DOI: 10.1109/LDAV.2011.6092178.

[35] 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 *SC*, 2011, pp. 1–11. DOI: 10.1145/2063384.2063409.

[36] 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 *IEEE IPDPS*, 2017, pp. 1174–1183. DOI: 10.1109/IPDPS.2017.39.

[37] H. N. Greenberg, J. Bent, and G. Grider, "MDHIM: A parallel key/value framework for HPC," in *HotStorage*, 2015, pp. 10–10.

[38] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, *et al.*, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *IEEE IPDPS*, 2013, pp. 775–787. DOI: 10.1109/IPDPS.2013.110.

[39] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. DOI: 10.1145/1773912.1773922.

[40] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. A. Nguyen, *et al.*, "FlexIO: I/o middleware for

location-flexible scientific data analytics," in *IEEE IPDPS*, 2013, pp. 320–331. DOI: 10.1109/IPDPS.2013.46.

[41] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, *et al.*, "PreDatA - preparatory data analytics on peta-scale machines," in *IEEE IPDPS*, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470454.

[42] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *IEEE IPDPS*, 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5161052.

[43] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012, pp. 53–64. DOI: 10.1145/2254756.2254766.

[44] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 7, 2008.

[45] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, *et al.*, "Terascale direct numerical simulations of turbulent combustion using s3d," *Computational Science & Discovery*, vol. 2, no. 1, p. 015 001, 2009.

[46] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. DOI: 10.1145/362686.362692.

[47] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *CoNEXT*, 2014, pp. 75–88. DOI: 10.1145/2674005.2674994.

[48] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *SIGMOD*, 2017, pp. 775–787. DOI: 10.1145/3035918.3035963.

[49] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, *et al.*, "Don't thrash: How to cache your hash on flash," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1627–1637, Jul. 2012. DOI: 10.14778/2350229.2350275.

[50] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *SOSP*, 2011, pp. 1–13. DOI: 10.1145/2043556.2043558.

[51] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004. DOI: 10.1016/j.jalgor.2003.12.002.

[52] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *EuroSys*, 2014, 27:1–27:14. DOI: 10.1145/2592798.2592820.

[53] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001. DOI: 10.1109/71.963420.

[54] *Snappy*, https://github.com/google/snappy/.

[55] 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 *SC*, 2008, 1:1–1:11.

[56] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *SC*, 2008, 63:1–63:11.

[57] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and large cams for high performance data-intensive networked systems," in *NSDI10*, 2010, pp. 29–29.

[58] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *SIGMOD*, 2017, pp. 993–1008. DOI: 10.1145/3035918.3064007.

[59] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, Mar. 2006. DOI: 10.1145/1132863.1132864.

[60] 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 *SIGMOD*, 2018, pp. 323–336. DOI: 10.1145/3183713.3196931.

[61] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *SIGMOD*, 2017, pp. 79–94. DOI: 10.1145/3035918.3064054.

[62] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "Slimdb: A space-efficient key-value storage engine for semi-sorted data," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2037–2048, Sep. 2017. DOI: 10.14778/3151106.3151108.

[63] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *FAST*, 2013, pp. 17–30.

[64] R. A. Oldfield, G. D. Sjaardema, G. F. Lofstead II, and T. Kordenbrock, "Trilinos i/o support trios," *Sci. Program.*, vol. 20, no. 2, pp. 181–196, Apr. 2012. DOI: 10.1155/2012/842791.

[65] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, *et al.*, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC*, 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[66] 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 *IEEE CLUSTER*, 2012, pp. 155–163. DOI: 10.1109/CLUSTER.2012.26.

[67] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, *et al.*, "Functional partitioning to optimize end-to-end performance on many-core architectures," in *SC*, 2010, pp. 1–12. DOI: 10.1109/SC.2010.28.

[68] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, *et al.*, "GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *SC*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503279.

[69] 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.

[70] 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.