# KV-CSD: A Hardware-Accelerated Key-Value Store for Data-Intensive Applications

Inhyuk Park*, Qing Zheng†, Dominic Manno†, Soonyeal Yang*, Jason Lee†, David Bonnie†, Bradley Settlemyer‡, Youngjae Kim§, Woosuk Chung*, Gary Grider†
*SK hynix, †Los Alamos National Laboratory, ‡NVIDIA, §Sogang University
{inhyuk.park, soonyeal.yang, woosuk.chung}@sk.com, {bsettlemyer}@nvidia.com, {youkim}@sogang.ac.kr,
{qzheng, dmanno, jasonlee, dbonnie, ggrider}@lanl.gov

***Abstract*—Popular software key-value stores such as LevelDB and RocksDB are often tailored for efficient writing. Yet, they tend to also perform well on read operations. This is because while data is initially stored in a format that favors writes, it is later transformed by the DB in the background into a format that better accommodates reads. Write-optimized key-value stores can still block writes. This happens when those background workers cannot keep up with the foreground insertion workload.**

**This paper advocates for a hardware-accelerated key-value store, enabling performance-critical operations, like background data reorganization and queries, to execute directly on storage instead of a host as existing key-value stores do. This better hides background work latency, prevents it from blocking foreground writes, and improves overall I/O efficiency. Our prototype, called KV-CSD, is a key-value based computational storage device consisting of an NVMe SSD and a System-on-a-Chip (SoC) that implements an ordered key-value store atop the SSD. Through offloaded processing, KV-CSD streamlines data insertion, reduces host-device data movement for both background data reorganization and query processing, and shows up to 10.6× lower write times and up to 7.4× faster queries compared to the current state-of-the-art software key-value stores on a real scientific dataset.**

## I. Introduction

Demands for storage performance continue to grow due to rapidly increasing client performance, data size, and data-intensive applications such as simulation checkpointing, machine learning training, and large-scale data analytics [1–3]. To keep up with these demands, many recently deployed HPC systems — from Los Alamos' Trinity [4] supercomputer in 2016 to Oak Ridge's Frontier [5] and NERSC's Perlmutter [6] supercomputers in 2022 — have employed flash-based storage tiers to provide performance that matches the performance of their compute tiers. In these flash-accelerated systems, storage remains as block devices and applications continue to access storage using filesystems [7–10]. Sustained high bandwidth enabled by flash allows applications to quickly transfer a large amount of data between compute and storage nodes, which benefits applications such as simulation checkpointing [11, 12] that read and write datasets in their entirety (as opposed to selective reads) and do not require data to be converted to a different format for efficient retrieval at a later point in time [13]. Nevertheless, applications that do require a format conversion — either in the form of resorting data or building auxiliary indexes alongside it — tend to still experience long processing delays. This is especially the case when high

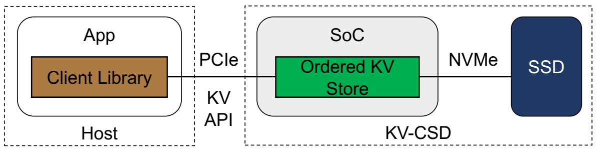


Fig. 1: Overview of a KV-CSD Computational Storage Device

volumes of small scientific data records previously written by a massively parallel scientific simulation are subsequently read for interactive data analytics with potentially very selective queries [14–16]. A scientist must either patiently wait for data to be converted into the right format or risk executing a query that reads back an excessive amount of data, leading to a very long run time.

To speed up applications with potentially highly selective data access patterns, embedded key-value stores such as LevelDB [17] and RocksDB [18, 19] have been increasingly explored in scenarios ranging from filesystem metadata management [20–23], block data management [24, 25], to large-scale data analytics [26–28]. By employing Log-Structured Merge (LSM)-Tree based data structures [29], these key-value stores achieve high data insertion speeds and rapid point and range query performance over primary keys. They do so by first writing data into logs and then sorting them in the background using a process known as *compaction*. Recently, we have also seen techniques that leverage LSM-based key-value stores for secondary indexes, allowing for efficiently answering queries with predicates that can span multiple data dimensions [30–33].

Unfortunately, even though LSM-based key-value stores have been increasingly deployed and can effectively transform data to read-optimized formats for rapid queries, they are still limited in their capacity to process data efficiently. First, an embedded key-value store runs inside an application process and depends solely on the host's compute resources to carry out all operations. This reliance constrains the store's ability to hide background work latency through asynchronous processing. As a result, in cases where the background compaction process of the key-value store becomes a bottleneck and fails to keep up with a foreground insertion application, the store may be unable to hide it — the insertion application can still experience long I/O delays and even be unable to make forward progress [34]. At the same time, with filesystems continuing to underpin modern data accesses for most HPC applications and workflows, an embedded key-value store is therefore also limited by the I/O interface and performance of the underlying filesystem or filesystems to deliver sufficient performance, leading to inefficiencies such as reduced storage media bandwidth utilization and read amplification, where more data than is needed by the application is transferred from storage to the application, slowing down reads and increasing query latency [35].

To better support highly selective data access patterns for HPC applications, this paper makes a case for a hardware-accelerated key-value store that overcomes the limitations of its software counterparts. As Figure 1 shows, our prototype, called KV-CSD, is a key-value based computational storage device [36] consisting of an NVMe SSD and a System-on-a-Chip (SoC) that implements an ordered key-value store on top of the SSD. User applications communicate with KV-CSD through a lightweight client library that exposes a key-value interface similar to that of a software key-value store. KV-CSD supports bulk inserts, bulk deletes, point and range queries over primary keys, and point and range queries over user-defined secondary index keys. All key-value operations and background tasks are handled primarily by the KV-CSD device, which is not limited by the host to deliver performance.

Compared with software-based key-value stores, KV-CSD possesses three unique advantages. First, by not relying on host compute resources to run potentially expensive background operations such as compaction, KV-CSD allows these expensive operations to be deferred and processed asynchronously in the device, hiding their latency from applications and preventing bottlenecks such as write stalls [34], where an application's foreground insertion workload cannot make progress due to pending background operations. Second, KV-CSD offers key-value based storage directly from a device, as opposed to being built on top of a host filesystem. This arrangement allows for directly translating high-level key-value operations, like GETs and PUTs, into low-level storage commands, such as NVMe read and write requests. By removing filesystem and the OS block layer code from the data processing path, KV-CSD shields applications from potential significant performance loss as data travels through these layers as prior work shows [24, 37]. Finally, by directly implementing key-value storage management in device, KV-CSD provides opportunities to leverage low-level storage interfaces, such as Zoned Namespace [38, 39], to optimize performance whereas a software key-value store must rely on the underlying filesystem and the operating system to adopt these optimizations accordingly.

To demonstrate these benefits, KV-CSD directly implements an LSM-Tree based key-value store atop an SSD using an SoC [40] that has 4 ARM CPU cores, 8GB RAM, and a Ubuntu OS. Applications send key-value operations to KV-CSD using direct memory access via PCIe through a lightweight client library. KV-CSD turns them into low-level NVMe requests to the underlying SSD. To minimize software overhead, KV-CSD's on-SoC key-value store is conveniently developed as a custom userspace block device driver using Intel's Storage Performance Development Kit (SPDK) [41], as Figure 3 shows. Because the key-value store itself is a device driver, it directly communicates with the underlying SSD through NVMe, completely bypassing the SoC OS kernel and its overhead.

**Deferred Compaction:** Prior work shows that data written by HPC applications such as simulation is often not accessed until analytics are performed. This has led to techniques such as deep write buffering and delayed filesystem metadata synchronization [11, 20, 22, 42] that drastically improve an application's write performance. KV-CSD hence similarly proposes deferred compaction, where application key-value pairs are first written as unordered log entries in the database. Applications subsequently invoke compaction to convert them into indexed, ordered data records for efficient queries. Data conversion is done asynchronously by the DB device — an application's foreground workload is not impacted.

**Key-Value over Zones:** Modern SSDs often exhibit reduced performance and long tail latency when used SSD blocks are garbage collected for incoming writes [38]. To address this bottleneck, Zoned Namespaces [39] have been recently introduced, allowing applications to explicitly manage SSD garbage collection for more predictable I/O performance. To convey this capability up to applications, KV-CSD introduces keyspaces. Applications dynamically create keyspaces for their datasets. Internally, KV-CSD will map these keyspaces to different SSD zones to streamline storage management and to better leverage available SSD bandwidth. Essentially, KV-CSD can be viewed as converting a zone-based NVMe block SSD into a key-value based active device featuring offloaded search and compaction functions, as we will show in Section IV.

**Primary and Secondary Indexes:** Finally, to enable rich query types, KV-CSD supports both point and range queries over primary and secondary index keys. When an application invokes compaction on a keyspace, KV-CSD sorts its data on keys, creating the primary index. Secondary indexes are created by applications specifying the byte range and the type of a certain part of value to serve as the secondary index keys. For example, an application can request creating a secondary index on the last 4 bytes of the values and have KV-CSD treat them as 32-bit integers.

Our experiments compare KV-CSD with RocksDB [18] — a popular software key-value store developed by Meta based on Google's LevelDB codebase [17]. While RocksDB is already popular due to its fast data insertion rates and swift queries, KV-CSD is able to outperform it by offering up to 10.6$\times$ lower write times and up to 7.4$\times$ faster queries on an actual scientific dataset. As we will discuss, KV-CSD achieves this by offloading performance critical tasks such as compaction and queries to storage, resulting in more effective latency hiding, less data movement, and more efficient data management.

The rest of this paper is structured as follows. Sections II and III describe the motivation and rationale behind our work. Sections IV and V detail our design. Section VI reports experiment results. Section VII discusses related work. Finally, Section VIII concludes.

## II. Motivation

Three factors motivate our work: (1) the growing need for new ways of accelerating highly selective data inquiries that minimize data movement, (2) the rising overhead of host software compared to next-generation performance tier flash devices, and (3) the promise of a hardware-accelerated key-value storage tier as computational storage is more practical than it was 25 years ago while new storage interfaces such as NVMe allow for innovation.

**Highly Selective Data Retrieval:** Scientific applications often generate a large amount of data. But during analytics, a scientist may be interested in only a small portion of a given dataset, leading to query patterns that can be highly selective and do not require reading back all data. For example, a scientist may just need to track the state of a few high energy particles along time or the very front of a shock wave traveling through a material [14, 15]. In these examples, the amount of data that needs looking at is only a small part of a large dataset, and only this small amount of data needs reading back. As future platforms enable larger scale applications and data sizes, the ratio between interesting data and overall data may further reduce, leading to potentially even more selective data inquires.

With searching for interesting data becoming increasingly selective, techniques that enable streaming back only necessary data becomes more and more important. This is because scientific applications such as simulations typically do not have the ability to optimize their data output (e.g.: sorting or building auxiliary indexes) for subsequent searches [13]. Instead, their primary goal when outputting is to output as fast as possible so that they can minimize their time spent on storage and maximize time spent on computation [3]. In the absence of a search optimized format on storage, a highly selective query may be unable to quickly locate the data being sought and therefore may have to read back all data, resulting in excessively long query times.

One way to speed up highly selective data inquiries is to pre-process the data before queries occur [15, 43, 44]. This allows data to be transformed into a query optimized format before it is queried. One problem of this approach is the large amount of I/O involved in transforming the data, which typically requires a full dataset read followed by a full dataset write. A user may have to wait an extended amount of time before data can be transformed to a desired format.

We have seen recently developed techniques that dynamically transform data to a read-optimized format as an application writes it to storage. For example, ADIOS [14, 45] allows redirecting an application's output to a separate set of compute nodes, where it asynchronously transforms data to a read-optimized format before writing it to storage. Techniques also exist that allow for data to be transformed potentially without additional compute nodes. One example is DeltaFS [46], which uses only idle CPU cycles found on job compute nodes to process data.

Unfortunately, even with these novel techniques, bottlenecks still exist. For example, ADIOS tends to be less useful when an entire computer is used to run a single large simulation such that no compute node will be left for processing data in situ. Similarly, while DeltaFS does not require additional compute nodes, it may only be able to index data on one dimension due to limited idle CPU resources on job compute nodes, leaving multidimensional queries still unaccelerated.

Computational storage provides new ways of addressing existing solution limitations. A hardware-accelerated key-value store can act upon data at rest. It does not need to transfer data to compute nodes or rely on their resources to scale performance.

**Host Software Overhead:** One promising development in addressing the I/O demands and performance needs of emerging scientific data analytics workloads is the enhanced performance of modern solid-state drives. Presently, a single high-performance flash SSD can achieve over 1 million small random read operations per second and offer multiple gigabytes per second of throughput [47–49]. Moreover, the NVMe transport, which is responsible for managing and accessing these drives, is equally adept at maximizing the potential of these high-speed storage devices.

However, even though high-speed storage solutions are becoming increasingly available at the device level, it remains crucial for host software, especially parallel filesystems, to effectively harness and present this performance to applications. Unfortunately, today's parallel filesystems have been increasingly unable to efficiently translate device performance increases to application performance increases [50]. The issue arises from the fact that many popular parallel filesystems today such as Lustre [51] and GPFS [52] were originally designed for hard disk drives, which offer significantly lower performance compared to flash-based storage devices like SSDs. Consequently, these systems were often designed with the premise of a virtually infinitely fast host compute system, resulting in deep and complex software stacks that can be too cumbersome and slow for today's performance tier flash devices. This leads to suboptimal utilization of available storage media bandwidth, as demonstrated in recent studies [24, 37, 53].

As filesystems are already unable to fully exploit the underlying device performance, layering a key-value abstraction — or anything else — on top of them only exacerbates this bottleneck. This stranded device performance is an important reason why we believe an embedded software key-value store can be insufficient. By contrast, directly providing storage services from an underlying device simplifies storage clients, reduces software layers, and allows for more fully utilizing available storage media performance.

**Towards Hardware-Accelerated Key-Value Storage:** To break I/O bottlenecks, computational storage architectures — in which smart devices or NICs are employed for near-data computing — become attractive. There are three major forms of computational storage: computational storage drives (CSDs) where one accelerator is attached to one storage drive (the form that KV-CSD takes), computational storage arrays (CSAs)

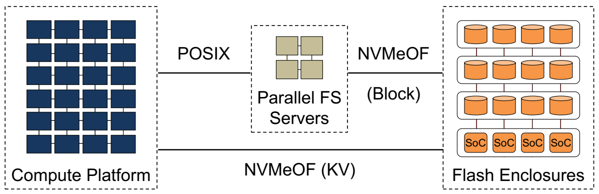


Fig. 2: KV-CSD Envisioned Next-Gen HPC Platform

where one accelerator is attached to many storage drives, and computational storage processors (CSPs) where the accelerator shares one or many storage drives with the host [36].

While HPC has mainly relied on host software to provide I/O services to applications, computational storage opens new ways of providing these services for improved performance. This is because computational storage allows processing to take place near data to minimize data movement, and can improve I/O efficiency by lowering the number of software layers needed due to hardware specialization.

Computational storage in the form of a hardware-accelerated key-value store can be particularly interesting because the key-value interface provides sufficient knowledge of data at the individual device level without having to resort to external metadata while the costly background compaction operations of the key-value store become a natural candidate for offloaded processing.

While we expect POSIX to remain the primary way of accessing storage, we envision various programming models and storage interfaces in future HPC platforms, as Figure 2 shows. Specifically, an analytics-intensive application may opt for a key-value (KV) based interface — rather than POSIX — for improved selective data retrieval performance. NVMe-over-fabric (NVMeOF) based flash enclosures, a recent innovation in data center storage, make it possible to efficiently share storage among compute nodes and parallel filesystem servers. NVMe namespaces [54] further allow for dynamically configuring storage enclaves for a given storage service. For example, an HPC site could devote 90% of its storage to serve parallel filesystems while the remaining 10% run as key-value stores. While our current prototype is a local PCIe device, nothing fundamental prevents us from extending it to NVMeOF for remote access.

## III. Enabling Technologies

Five overriding factors make developing efficient hardware-accelerated key-value stores such as KV-CSD possible.

**LSM-Trees:** leverage sorting to speed up queries. This approach differs from bitmap-based indexes — such as FastBit that are widely used in HPC communities [14, 15, 44] in that it does not require a potentially large amount of memory to build an index. While bitmap indexes can ultimately be very compact, their intermediate form during index construction can consume significant amounts of memory [55]. As a result, they may not be well suited to embedded computational storage environments. LSM-Trees enable a reduction in memory footprint at the cost of increased I/O time, as a consequence of performing multiple rounds of merge sorts. However, this overhead can be conveniently hidden through asynchronous processing within a computational storage device, like KV-CSD.

**NVMe:** enables the parallelism needed to fully utilize today's high-speed storage devices. Additionally, its command set specifications allow for extensions to accommodate new storage protocols such as key-value, making continuous innovation possible. While our current KV-CSD prototype requires a custom key-value client, internally it uses the standard NVMe key-value command set [56] for communication, along with KV-CSD's own extensions for commands not currently in the standard such as compaction and secondary index operations.

**Zoned Namespace (ZNS):** based SSDs use zones — as opposed to pages — as the basic unit of data management, each consisting of multiple NAND erase blocks. To improve performance, ZNS only permits sequential writes to zones, eliminating the need for in-device garbage collection at the cost of changes to applications, who become responsible for data placement and free-space reclamation over zones. ZNS plays a key role in simplifying KV-CSD's internal storage management and reducing garbage collection overheads when keyspaces created by applications are deleted, as prior work also finds [38].

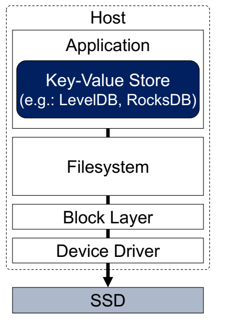
**Userspace Drivers:** Recent advancements in userspace block device drivers, exemplified by Intel's SPDK effort [41], have facilitated the development of high-performance storage applications that feature direct access from userspace to high-speed storage devices such as SSDs. This direct access bypasses the operating system kernel, allowing for fully leveraging the speed and I/O capabilities of the underlying device, leading to faster and more efficient storage access with minimum software overhead. By utilizing this technology, KV-CSD is able to efficiently implement its SoC-SSD communication, preventing unnecessary bottlenecks.

**Efficient Microprocessors:** The commoditization of the smartphone market has led to a significant decrease in the cost of embedded multi-core 64-bit processors, such as ARM, and a decrease in their power consumption. Additionally, the increase in storage density has made it more economical to incorporate additional computing resources within or near a storage device. These developments have made computational storage much more practical and cost-effective than it was 25 years ago when researchers first proposed it [57–59], making developing KV-CSD feasible.

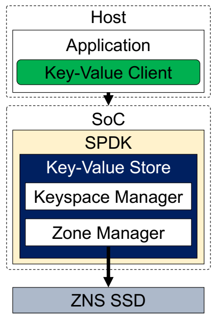
## IV. System Overview

KV-CSD is a computational storage device consisting of an NVMe ZNS SSD and a Linux-based SoC that implements an LSM-Tree based key-value store on top of the SSD in the form of a userspace block device driver using Intel SPDK.

As Figure 3 shows, this approach differs from modern software-based key-value stores such as LevelDB and RocksDB in that the key-value service is provided by the



(a) Software Key-Value Store



(b) KV-CSD

Fig. 3: KV-CSD vs Traditional Software-Based Key-Value Stores

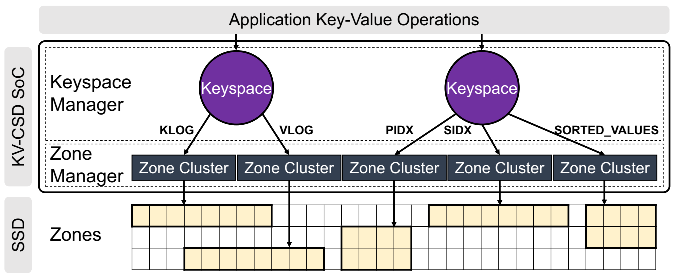


Fig. 4: Overview of KV-CSD's On-SoC Components

storage device rather than by code that runs inside an application above a filesystem. While KV-CSD also requires a client library, its primary job is to pack application function calls into requests that are sent to the underlying device, where the actual key-value based storage processing occurs.

For applications that cannot easily switch from POSIX to key-value in order to use KV-CSD, a lightweight shim layer may be used to translate file I/O into key-value operations as prior work such as TableFS [60] and DeltaFS [16] does.

KV-CSD's on-SoC key-value store is made up of a *keyspace manager* and a *zone manager*. Collectively, they are responsible for mapping and translating high-level application key-value operations into low-level NVMe read and write requests to the underlying ZNS SSD, as Figure 4 illustrates.

**Keyspace Manager:** In KV-CSD, applications store and manage their data through keyspaces. Keyspaces are containers of key-value pairs. Each is assigned a unique name specified by the application at the keyspace creation time. Keys within a keyspace must be unique while across keyspaces keys can be reused without causing conflicts. Once a keyspace is created, it can then be opened it for insertion, compaction, secondary index construction, and query operations.

Compared with managing all keys in one global keyspace, enabling separated keyspaces provides three key benefits. First, it prevents unrelated applications from having to frequently synchronize with each other in order to name their keys differently to avoid potential name conflicts. Second, it allows each keyspace to be compacted and indexed independently, and to have separate data formats and secondary index configurations. Finally, it streamlines KV-CSD's inner storage management by allowing for mapping different keyspaces to different zones. This prevents leaving "holes" in zones when created keyspaces are deleted, simplifying KV-CSD's internal garbage collection process, leading to more consistent I/O performance [38].

Each keyspace in KV-CSD can exist in one of the following four states: `EMPTY`, `WRITABLE`, `COMPACTING`, and `COMPACTED`. A newly created keyspace starts `EMPTY`. It reaches the `WRITABLE` state when an application opens it for writes for the first time. A keyspace enters the `COMPACTING` state the moment compaction is invoked upon it. This will prevent the keyspace from accepting any new writes, rendering it readonly. Once compaction is done, a keyspace reaches the `COMPACTED` state. Only keyspaces in this state are queryable. Additionally, one or more secondary indexes can be added to a keyspace when it is in the `COMPACTED` state. An application may delete a keyspace at any state, though deletion may be deferred due to on-going compaction or index operations.

The keyspace manager keeps track of the state and other metadata information (such as the number of key-value pairs, the minimum and the maximum keys, and the zone mapping information) of all live keyspaces. It does so by maintaining an in-memory keyspace table backed by a metadata zone in the underlying ZNS SSD for data persistence.

**Zone Manager:** is responsible for (1) allocating and deallocating zones as requested by the keyspace manager, and (2) grouping zones into clusters to enable parallel I/O across zones as Figure 4 depicts.

Zones are a logical subdivision of the available storage space on a ZNS SSD. In these SSDs, the storage space is exposed by the SSD firmware as an array of equal-sized zones, each with its own *write pointer*. Since data can only be written sequentially within a zone, the write pointer indicates the next allowable write position in the zone. It advances as writes are performed. When data in a zone is no longer needed, a *reset* operation can be performed on the zone, allowing its storage space to be reclaimed and its write pointer rewound.

The zone manager keeps track of all available zones and performs zone allocation and deallocation as needs arise. As Figure 4 shows, based on their usage there are five types of zones: `KLOG`, `VLOG`, `PIDX`, `SIDX`, and `SORTED_VALUES`. Keyspaces in the `WRITABLE` state use only `KLOG` and `VLOG` zones whereas keyspaces in the `INDEXED` state use only `PIDX`, `SIDX`, and `SORTED_VALUES` zones, as Section V will further explain.

Rather than allocating zones on a per-zone basis, KV-CSD allocates zones in groups that we call *zone clusters*. This enables striping I/O across multiple zones to better leverage available SSD bandwidth to improve overall I/O performance. Crucially, this I/O parallelism is enabled even when an application opens a single keyspace for read and write operations.

When multiple application process threads simultaneously write to KV-CSD, their writes may share some subset of SSD internal I/O channels even when they each write to a different keyspace. This leads to channel conflicts and reduced overall write throughput. To alleviate this bottleneck, KV-CSD associates a random number with each zone cluster to determine which zone to perform the next write within a zone

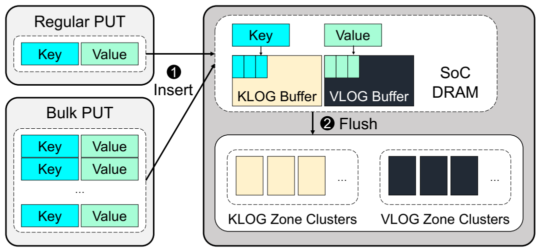


Fig. 5: Description of the Write Phase

cluster. This allows zone writes to be randomly distributed across all available I/O channels of an SSD, minimizing channel conflicts and maximizing SSD bandwidth utilization.

## V. Operational Flow

KV-CSD supports insertion, deletion, and point and range queries over both primary and secondary index keys. This is done by applications dynamically creating a keyspace, inserting keys into it, invoking compaction, configuring secondary indexes per workload needs, and then performing queries. An application deletes a keyspace when it is no longer needed.

The ordered key-value store abstraction provided by KV-CSD is designed to match the workflow of a typical simulation-based science pipeline consisting of a simulation phase and a post-analysis phase. Compared with popular software key-value stores such as RocksDB and LevelDB, KV-CSD enables more optimized data processing flow for scientific applications at the cost of being less flexible and more restrictive in the calling sequence of supported key-value operations. The LSM-Tree realized by KV-CSD can be viewed as a minimal, stripped-down version of its more feature-rich counterparts, and does not necessarily work for workloads that do not resemble a scientific simulation pipeline.

**Data Insertion:** Applications insert data into KV-CSD by first creating a keyspace and then writing data into it. Inserted data is first buffered at KV-CSD's SoC DRAM. When the DRAM buffer is full (192KB for the current prototype), it is then flushed to the SSD zone clusters that are mapped to the keyspace. KV-CSD stores keys and values separately: values are written to `VLOG` zone clusters while keys, along with pointers to the values, are written to `KLOG` zone clusters, as Figure 5 shows. Storing keys and values separately allows for sorting them in two separate steps (discussed later), reducing overall subsequent keyspace compaction overhead. If no zone clusters have been mapped to a keyspace, they will be dynamically allocated and mapped.

To minimize communication overhead, KV-CSD supports both regular PUT and bulk PUT operations. While a regular PUT operation only inserts a single key-value pair into a keyspace, bulk PUTs allows for inserting many key-value pairs using a single command to hide insertion latency, improving overall write throughput.

**Compaction:** When all data has been written to a keyspace, an application invokes compaction to have KV-CSD sort the keyspace, producing the primary index. Sorting a keyspace is done in two steps. First, KV-CSD sorts the keys. Then, KV-CSD uses the sorted keys to sort the values. Sorting is done by running multiple rounds of merge sorts, depending on available SoC DRAM space. Intermediate sorting results are stored in dynamically allocated zone clusters, which are released upon completion of the sort. Once a keysapce is sorted, its original unsorted data, stored in `VLOG` and `KLOG` zone clusters, is deleted and replaced with the newly formed `SORTED_VALUES` and `PIDX` zone clusters.The former store sorted values. The latter store sorted keys along with pointers to the values. Both store data as a series of 4KB data blocks. A small sketch of the `PIDX` data, consisting of a pivot primary index key and a block pointer for every constituent `PIDX` data block, is additionally built and stored as keyspace metadata in the keyspace manager's in-memory keyspace table. It serves as the starting point for all primary index queries — both point and range queries — against the keyspace.

**Secondary Index Construction:** Once the primary index has been built, an application may optionally build one or more secondary indexes according to workload needs. Unlike primary indexes, building a secondary index requires applications to specify the byte range and the type of a certain part of value to serve as the secondary index keys, as well as the name of the secondary index for future references.

Once a secondary index is configured, building it is a two-step process. First, KV-CSD performs a full scan of the keyspace data to extract all secondary index keys from the values, along with their associated primary index keys. This is done by reading back data from the `SORTED_VALUES` and `PIDX` zone clusters, extracting secondary index keys according to application-supplied configuration, and then writing them into temporarily allocated zone clusters as unordered "<secondary index key, primary index key>" pairs. Next, KV-CSD sorts these pairs in a manner similar to what it does for sorting the primary index keys, producing the secondary index stored in `SIDX` zone clusters.

Similar to the primary index case, a small sketch of the `SIDX` data, consisting of a pivot secondary index key and a block pointer for every constituent `SIDX` data block, is built and stored as additional keyspace metadata in the keyspace manager's in-memory keyspace table. It serves as the starting point for all point and range queries against this particular secondary index.

While our current design builds the primary index and each secondary index as separated operations, in future we expect to run these index construction operations in one single step to prevent from having to repeatedly reading back keyspace data into SoC DRAM, even though we can hide their latency through asynchronous processing in a computational storage device. One cost of consolidating all index construction into a single step is the increased SoC DRAM usage. We expect KV-CSD to resort back to separated index construction when DRAM resources become a bottleneck.

**Query Processing:** Once all indexes are constructed, a keyspace is then ready for queries. Leveraging fully sorted

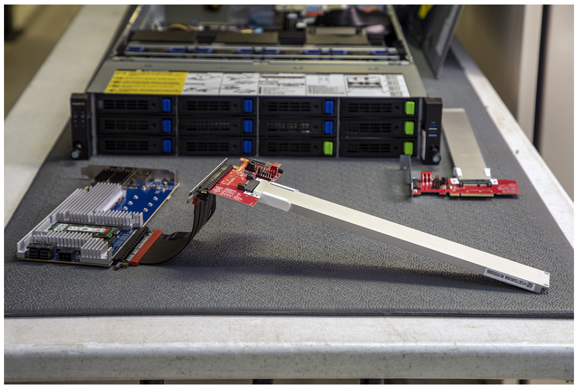


Fig. 6: KV-CSD in real world.

data in PIDX, SIDX, and SORTED_VALUES zone clusters, KV-CSD is able to efficiently answer point and range queries over both primary and secondary index keys. To handle a query, KV-CSD first identifies the keyspace from the keyspace manager's in-memory keyspace table. It then uses the keyspace's metadata to locate all related primary or secondary index data blocks on the SSD, and use them to process the incoming query. Because query is entirely processed in a computational storage device, only query results need to be transferred back to the application. This minimizes host-device data communication, reducing query latency especially when query selectivity is high.

**Keyspace Deletion:** KV-CSD allows applications to delete keyspaces that are no longer needed. Deleting a keyspace discards all data and metadata associated with it, and frees all SSD zones that are mapped to the keyspace, allowing their space to be reclaimed.

## VI. Evaluation

We implemented a prototype of KV-CSD using C and SoC hardware available from market, as illustrated in Figure 6. Our implementation consists of a client component and a server component. The client component is a lightweight userspace device driver written in C that runs inside an application process on a host. It translates high-level key-value operations invoked by the host to low-level I/O requests sent to the KV-CSD SoC, which acts as a "server". Both the translation and sending of the requests take place in userspace and completely bypass the host OS kernel. Our implementation uses efficient DMA methods to transfer data. This further reduces the amount of processing needed on host to execute a key-value operation.

TABLE I: Hardware Specification

| | **Host** | **KV-CSD CSD** |
|---|---|---|
| CPU | 32 AMD EPYC cores | 4 ARM Cortex A53 cores |
| RAM | 512GB DDR4 | 8GB DDR4 |
| OS | Ubuntu 18.04 | Ubuntu 16.04 |
| Storage | KV-CSD CSD | 15TB NVMe ZNS SSD |

The server component of KV-CSD is the SoC itself. We use a Fidus Sidewinder-100 (SW-100) development board [40] that carries an Xilinx Zynq UltraScale+ computing platform [61] with a quad-core ARM Cortex A53 processor and 8GB memory. An external E1.L NVMe ZNS SSD connects to the board using four PCIe Gen3 lanes. We had the board run a Ubuntu OS with its image backed by an embedded M.2 SSD that we installed on the board. Our KV-CSD software runs atop the Ubuntu OS. As discussed earlier, it is developed as a userspace block device driver using Intel's SPDK framework [41] that implements a simplified version of an LSM-Tree. It packs small application key-value pairs into large indexed data blocks stored on the ZNS SSD. By implementing an LSM-Tree near data, KV-CSD greatly reduces data movement during a query phase while allowing costly LSM-Tree compaction operations to run asynchronously in the device without relying on host compute resources and without potentially slowing down foreground application workloads.

Like RocksDB and others, KV-CSD uses write-ahead-logging to back in-memory data and supports explicit "fsync". We expect production applications to frequently disable write-ahead-logging though because many use checkpointing-restart for failure recovery making individual DB-level logging often redundant or overkill.

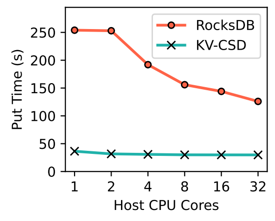
### A. Experimental Setup

All of our experiments were done on an AMD host with one of our prototype KV-CSD devices installed on it. As Table I shows, our host runs a Ubuntu OS and has 32 AMD EPYC CPU cores, 512GB memory, and a dedicated OS drive. Our prototype KV-CSD computational storage device (CSD) connects to the host using 16 PCIe Gen3 lanes.

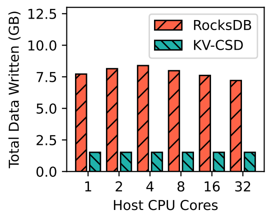
Our evaluation consists of both micro and macro benchmarks. In micro benchmarks, focuses on two basic DB operations — PUTs and GETs — which we utilize to highlight three key advantages of computational storage: enhanced I/O performance, reduced data movement, and reduced reliance on host compute resources. We test cases in which multiple application process threads share a single keyspace for reads and writes and cases in which multiple process threads spread keys across multiple keyspaces for more scalable performance.

In macro benchmarks, we use a real-world scientific dataset derived from VPIC [62] to showcase the potential of KV-CSD in expediting post-hoc data analytics when large amounts of data is inserted during simulation I/O and subsequently analyzed for insights with potentially highly selective queries. VPIC is a general-purpose, particle-in-cell simulation code widely used for modeling kinetic plasmas in one, two, or three spatial dimensions [13].

In both micro and macro benchmarks, we compare KV-CSD with RocksDB [18], a widely-used key-value store created by Meta, which we consider to represent the current state-of-the-art. RocksDB employs LSM-Trees for managing user data and supports both point and range queries. However, unlike KV-CSD, RocksDB operates on a POSIX filesystem



(a) Put time



(b) I/O statistics

Fig. 7: Time to insert 32M keys into a single keyspace using different amount of host compute resources.

and depends on host computing resources to carry out all database operations, including background compaction.

Due to physical space limitations, we were only able to mount a single KV-CSD device onto our host machine at present. We anticipate the ability to mount additional devices once hardware boxes that provide sufficient space to accommodate E1.L drives become available. Fortunately, even with a single device, we were able to demonstrate substantial performance enhancements compared to purely software-managed key-value stores.

## B. *Micro Benchmarks*

Current state-of-the-art key-value stores such as RocksDB are often host-based software that runs on top of a filesystem. Application data is turned into indexed, ordered table files entirely through host processing, which often results in significant data movement and subsequent slowing down of applications. With KV-CSD, we show that key-value stores scale better when they are offloaded for a computational storage based implementation which is closer to the data and capable of processing it more efficiently.

To demonstrate this, our first set of experiments compare KV-CSD with RocksDB and focus on their write performance.

**PUT Performance:** We start by examining a case where a scaling number of application process threads write data to a shared keyspace. To run it, we developed a multi-threaded program able to generate synthetic workloads according to a given configuration. A modular design was used such that the same code can run over both DB implementations. We ran this program using 1 to 32 threads. A total of 32M random key-value pairs are inserted in each run. We use 16B keys and 32B values.

For each KV-CSD run, we reset the device and insert keys into a newly-created keyspace. We use KV-CSD's bulk put interface to insert keys. Each bulk put message is 128KB. This 128KB space contains keys, values, and their respective sizes. For 16B keys and 32B values, each message carries up to 2570 key-value pairs and is 7x faster than regular puts. Once all keys are inserted, we invoke KV-CSD's background compaction process and exit — KV-CSD will run compaction asynchronously in the device for us. We report time to insert all keys. For each RocksDB run, we create a new DB instance on top of a newly-formatted ext4 and then insert keys into the DB. RocksDB runs background compaction as keys are inserted and may not be able to complete compaction when all keys are inserted. When this happens, our test program will wait until all compaction work concludes before exiting the program. We report time to insert all keys which *includes* additional wait time due to RocksDB compaction.

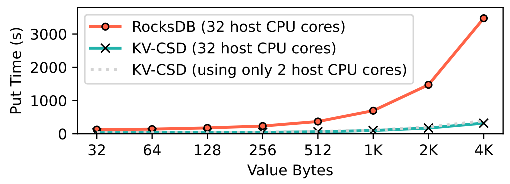


Fig. 8: Time to insert 32M keys with different value sizes into a single keyspace.

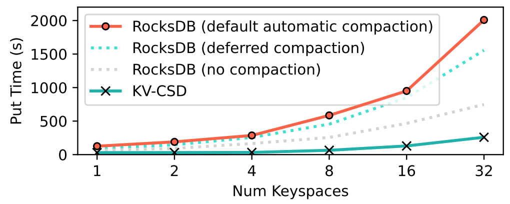


Fig. 9: RocksDB vs KV-CSD insertion time as keyspace count and data size increase.

To control host resource usage, we assigned each test thread to a specific CPU core for both KV-CSD and RocksDB runs. RocksDB creates two worker threads per DB instance to run background compaction tasks. We allow these threads to operate on any CPU core that had a test thread pinned on it.

Figure 7a shows the result. Because RocksDB is an embedded software key-value store, it relies on host compute resources to carry out all operations including compaction. As a result, a RocksDB user has to experience both write time and compaction time whereas a KV-CSD user only needs to experience write time, with compaction deferred and offloaded to the underlying device.

While RocksDB requires 32 dedicated host CPU cores to maximize performance, KV-CSD really only needs 2 CPU cores to reach peak performance, as it is the device that implements the DB rather than the host. At 32 CPU cores, KV-CSD is 4.2× faster than RocksDB. At two cores, KV-CSD is 7.9× faster.

Figure 7b reports the underlying I/O statistics. RocksDB runs compaction as data is inserted to dynamically transform it to a read-optimized format. This requires RocksDB to constantly read data from storage, re-sort it in memory, and write the results back to storage. Consequently, all RocksDB runs are marked by a multifold increase in storage I/O, resulting in longer I/O wait times as Figure 7a shows. In contrast, **KV-CSD postpones compaction and processes it asynchronously in the device, reducing I/O overhead during insertion and streamlining writes.**

*Larger Values:* Figure 8 shows a case where keys are inserted with different value sizes from 32B to 4KB. For RocksDB runs, we use 32 host CPU cores for maximum performance. For KV-CSD runs, we show results of using both 2 and 32 host CPU cores.

As value size increases, RocksDB becomes increasingly bottlenecked on data movement due to compaction. At 4KB values, KV-CSD using 32 host CPU cores is 10× faster than RocksDB. In fact, even limited to 2 host CPU cores, KV-CSD is still 8.9× faster than RocksDB using 32 cores, thanks to offloaded processing.

*Multi-Keyspaces:* Figure 9 investigates a case where multiple application threads each insert data into its own keyspace rather than sharing a global keyspace.

We run 1 to 32 threads. Each inserts 32M keys into a per-thread keyspace. For RocksDB runs, each thread inserts keys into a per-thread RocksDB instance created atop a shared ext4. We use 16B keys and 32B values. The biggest run uses 32 threads, creates 32 keyspaces or RocksDB instances, and inserts 1 billion keys.

To better quantify KV-CSD's performance improvements in comparison to the current state-of-the-art, we run RocksDB in three different modes: 1) default automatic compaction, 2) deferred compaction where compaction is manually held until after all keys are inserted, and 3) no compaction where compaction is disabled. Running RocksDB in three different modes helps us demonstrate the benefits of deferred compaction. For KV-CSD, we focus on deferred compaction because eager compaction tends only to do a disservice to scientific applications.

Result shows that KV-CSD scales well as keyspace and data size increases whereas RocksDB becomes increasingly bottlenecked on data movement. At 32 keyspaces, KV-CSD is 7.8×, 6.1×, and 2.9× faster than RocksDB with default automatic compaction, with deferred compaction, and with no compaction respectively. For RocksDB, deferred compaction improves performance because compaction is done in a single pass at the end of an insertion job, reducing total data movement. Turning off compaction prevents unnecessary data transfer, but comes with the trade-off of substantially lengthening the time required for future queries. **By offloading most of the insertion processing to the device, KV-CSD minimizes host processing, resulting in faster write speeds compared to RocksDB, even when the latter has background compaction disabled.** Moreover, by employing a deferred and asynchronous compaction process, KV-CSD enables rapid insertion rates without slowing down reads.

KV-CSD is faster due in part also to fewer software layers. To quantify this, we plan to add a run as future work where a modified RocksDB uses SPDK to communicate with storage such that RocksDB also benefits from a reduction of layers.

ZNS shows advantage when SSD space is heavily utilized making SSD-level garbage collection a performance bottleneck. While KV-CSD leverages ZNS to streamline storage space allocation and deallocation, we expect the ZNS effect to be insubstantial in these experiments because SSD space was only lightly utilized and because of a lack of overwrite operations. KV-CSD was faster due mostly to asynchronous

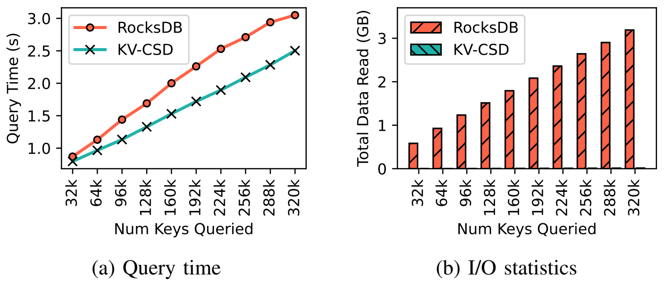


Fig. 10: Performance of random GET operations.

compaction processing by a computational storage device, which was a key advantage of KV-CSD.

**GET performance**: Figure 10 shows the performance when data inserted in Figure 9 is subsequently queried using random GET operations. We run GETs following a case where there are 32 keyspaces, 32M keys per keyspace, and a total of 1 billion keys. We run 32 query threads, each targeting a different keyspace. KV-CSD does not cache data in host or device memory. For RocksDB runs, we clean OS page cache at the beginning of each run.

Figure 10a reports the time it takes for the query program to finish 32K to 320K queries while 10b shows the background I/O statistics. **Through offloading queries to device, KV-CSD transfers only results back to clients, minimizing data movement and accelerating queries.** RocksDB exhibits high read inflation due to being a software key-value store running atop a filesystem — it must read back a significant amount of database file blocks before being able to process a query.

Results show that KV-CSD is up to 1.3× faster than RocksDB, even though both demonstrate extremely fast query performance thanks to compaction which reorganizes data into a read-optimized format. RocksDB query time improves as more keys are queried due to aggressive client-side caching. Nevertheless, we expect caching to be less effective in production environments due to their higher data-size-to-memory-size ratios. Similarly, we expect read inflation to be considerably more punishing in production environments due to remote storage and bottlenecks in networks.

## C. Macro Benchmarks

To demonstrate KV-CSD's potential in speeding up large-scale scientific data analytics in comparison to the current state-of-the-art software-based key-value stores, our macro benchmark examines a case where a real-world scientific dataset is converted to key-value pairs and then queried based on a secondary index key.

Our sample dataset is a partial VPIC simulation [62] dump consisting of 256M particles in the form of 16 binary files. Each VPIC particle is 48 bytes, consisting of a 16B particle ID and a 32B payload made up of 8 numeric attributes with one of them being the kinetic energy that we used for secondary index construction and queries.

We divide our experiment into a write phase and a query phase. In the write phase, we use a custom loader program to read VPIC particles from files and then insert them into a key-value store. Particles are written as one key-value pair per particle. We use particle IDs as keys and the rest as values. We run 16 threads to insert data concurrently. Each thread is responsible for loading one of the 16 available particle files and will write particles into a per-thread keyspace, creating a total of 16 keyspaces.

For KV-CSD runs, our loader program relies on the underlying device to run both compaction and secondary index construction. We report insertion time, compaction time, as well as secondary index construction time. For RocksDB runs, compactions are run by RocksDB as data is inserted. To create a secondary index on particle energies for subsequent queries, our loader program inserts auxiliary key-value pairs as it writes primary key-value pairs to the DB. These auxiliary key-value pairs use particle energies as keys and particle IDs as values. To search for particles with certain energy levels (e.g.: energy > 1.2), a reader program first queries against the auxiliary index keys, and then uses the returned particle IDs to run a secondary query on the primary key to read back the full particle. To distinguish auxiliary keys from primary keys, a small 1B prefix is prepended to each key. Automatic compactions run by RocksDB sort both auxiliary index keys and primary index keys, allowing for efficient queries on both dimensions. We report data insertion time as well as additional wait time due to RocksDB compaction, which — differs from the KV-CSD case — covers both indexes.

In the query phase, we use a multi-threaded reader program to run queries against both KV-CSD and RocksDB. We run 16 query threads, each targeting a different keyspace. We use different energy thresholds to drive different query selectivity levels. Our most selective query hits only 0.1% particles. Our least selective query hits as many as 20% particles. For KV-CSD runs, the device handles all queries and directly streams back all matching particles. For RocksDB runs, queries are run as a two-step process that involves querying two types of indexes. KV-CSD does not cache data. We clean OS page cache at the beginning of each RocksDB run.

Figure 11 shows the result for the write phase. **While it takes KV-CSD and RocksDB about the same amount of time to finish writing, compaction, and indexing, KV-CSD is able to run compaction and indexing asynchronously in the device without needing the host application to wait for it.** This makes KV-CSD effectively 10.6× faster than RocksDB in this particular experiment with its 66s effective write time compared to RocksDB's 704s effective write time.

In real world environments, simulations usually spend 85% time computing and 15% time writing. When write frequency increases significantly, KV-CSD may be unable to finish compaction before the next dump occurs. We expect production computational storage devices though to feature more optimized hardware such as FPGA such that it can process data much more quickly to accommodate extremer cases.

Figure 12 shows the results for the query phase. Through sorting and building secondary data indexes, both KV-CSD and RocksDB are able to efficiently handle queries over a

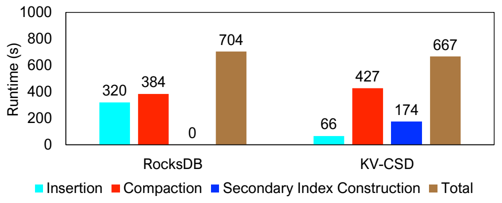


Fig. 11: Breakdown of KV-CSD and RocksDB insertion time.

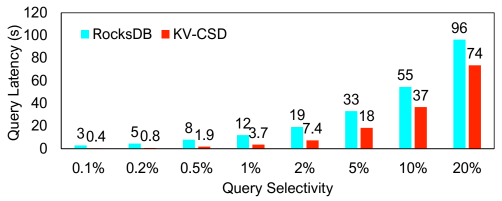


Fig. 12: KV-CSD vs RocksDB secondary index query time.

non-primary data attribute. KV-CSD shows up to 7.4× faster query performance compared to RocksDB due to offloaded processing which reduces not only the amount of data that needs reading back to host for query handling, but also the number of queries that the application needs to execute for a search. While KV-CSD directly handles a secondary index query, a RocksDB user has to query primary and secondary indexes separately. KV-CSD's query speedup drops as query selectivity reduces — from 7.4× in the 0.1% run to 1.3× in the 20% run — due to RocksDB client side caching which benefits queries that are less selective (queries that return more data). Because KV-CSD does not cache data, its query latency is always linear to the total number of particles returned.

Because we expect caching to be less effective in production environments due to their high data-size-to-memory-size ratios, and read inflation to be considerably more punishing, we anticipate KV-CSD, which minimizes data movement to accelerate queries, to outperform RocksDB, which employs client-side caching for query optimization, especially for situations involving highly selective data inquiries.

## VII. Related Work

Computational storage is a technique that reduces data movements between compute hosts and storage by bridging computation to the storage systems. This computation offloading approach offers several benefits, including reduced bandwidth requirements, lower latency, and energy-efficient computing. Previous studies on computational storage [63–78] have demonstrated its promise, and commercially available computational storage systems such as SmartSSD [64], ScaleFlux [78], Newport CSD [65, 74], Kinetic [79], and Catalina [74] have flourished over the past decade.

Several previous studies have proposed the design and implementation of Key-Value Store inside SSDs to accelerate the processing speed of key-value point queries, such as Put and Get operations using primary keys [80–86]. In particular, iLSM-SSD [82], Iso-KVSSD [83], and PinK [81] adopted LSM tree-based index structures, whereas KAML [80] uses a hash-based index data structure for their internal index implementation. Both iLSM-SSD [82], Iso-KVSSD [83], and PinK [81] adopted key-value separation techniques to overcome internal memory space constraints. Furthermore, PinK [81] manages to minimize tail latency by pinning low-level SSTables at DRAM.

Previously proposed key-value SSDs have been optimized for handling point queries based on primary keys, and are often not well-suited for HPC workloads due to their compaction overhead or hash-based collision issues when writing data. In comparison, KV-CSD has been carefully designed to cater specifically to HPC applications, and offers optimized search query services by using secondary attributes as a secondary key, enhancing scientific discovery services in HPC. In addition to KV-CSD, there are also hybrid approaches where secondary indexes are built by host on top of a hash based KVSSD [87, 88]. Finally, two recent surveys provide a comprehensive overview of various software-based LSM-Tree key-value store techniques and designs [89, 90].

## VIII. Conclusion

Rising data sizes and client performance expectations are driving an increasing demand for high-performance storage. Concurrently, the expansion of data is leading to a proliferation of highly selective data inquiry workloads across various scientific computing areas. With the performance of these intensive storage workloads becoming increasingly hampered by the overheads and limitations of host software, advancements in computational storage technology present a unique opportunity to overcome these challenges.

KV-CSD is a hardware-accelerated, key-value based computational storage device. It is designed to reduce the burden on host compute resources and enable highly efficient data queries of key-value records by providing both primary and secondary indexes for scientific applications. To support the proposed hardware-accelerated key-value store, KV-CSD implements a simplified LSM-Tree that allows for deferred compaction thus completing write operations before the expensive indexing process begins. This improves overall performance by enabling specialized processing of both writing and indexing data. When using real-world scientific application data to compare these methods with the current state-of-the-art, KV-CSD achieves ten times greater put performance and provides seven times speedup in query time. These techniques not only reduce the resources required by the host, but most importantly they decrease the amount of data being read and written out of the storage system.

KV-CSD is defined by three major features that accelerate data-intensive scientific applications. First, a hardware architecture that leverages computational storage and a ZNS SSD leading to an efficient solution that enables processing near storage while also removing bottlenecks due to expensive software layers. Second, index creation is optimized to occur asynchronously therefore providing optimal write performance without removing index capability or reducing index quality. Finally, KV-CSD provides secondary index construction to further minimize the amount of data retrieved thus improving performance for highly-selective queries.

## Acknowledgment

We thank the anonymous reviewers of our paper for their invaluable comments on improving the paper. This manuscript has been approved for unlimited release and has been assigned LA-UR-23-25314. This work has been authored by employees of Triad National Security, LLC which operates Los Alamos National Lab with the U.S. Department of Energy/National Nuclear Security Administration.

## References

[1] 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.

[2] J. Lofstead and R. Ross, “Insights for exascale IO apis from building a petascale IO api,” in *Proceedings of the 2013 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 13)*, 2013, 87:1–87:12. DOI: 10.1145/2503210.2503238.

[3] 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.

[4] *Trinity*, https://www.lanl.gov/projects/trinity/, 2017.

[5] *Frontier*, https://www.olcf.ornl.gov/frontier/, 2021.

[6] *Perlmutter*, https://www.nersc.gov/systems/perlmutter/, 2021.

[7] G. K. Lockwood, K. Lozinskiy, L. Gerhardt, R. Cheema, D. Hazen, and N. J. Wright, “Designing an all-flash lustre file system for the 2020 nersc perlmutter system,” in *Proceedings of the 2022 Cray User Group (CUG 2022)*, https://cug.org/proceedings/cug2019_proceedings/includes/files/pap131s2-file1.pdf, 2022.

[8] G. K. Lockwood, A. Chiusole, L. Gerhardt, K. Lozinskiy, D. Paul, and N. J. Wright, “Architecture and performance of perlmutter’s 35 pb clusterstor e1000 all-flash file system,” in *Proceedings of the 2021 Cray User Group (CUG 2021)*, https://cug.org/proceedings/cug2021_proceedings/includes/files/pap120s2-file1.pdf, 2021.

[9] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, “Architecture and design of cray datawarp,” in *Proceedings of the 2016 Cray User Group (CUG 2016)*, https://cug.org/proceedings/cug2016_proceedings/includes/files/pap105s2-file1.pdf, 2016.

[10] *Unify: Distributed burst buffer file system*, https://computing.llnl.gov/projects/unify, 2019.

[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, “PLFS: A checkpoint filesystem for parallel applications,” in *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*, 2009, 21:1–21:12. DOI: 10.1145/1654059.1654081.

[12] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, “On the Role of Burst Buffers in Leadership-Class Storage Systems,” in *Proceedings of the 2012 International Conference on Massive Storage Systems and Technologies (MSST 12)*, 2012, pp. 1–11. DOI: 10.1109/MSST.2012.6232369.

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

[14] J. Gu, S. Klasky, N. Podhorszki, J. Qiang, and K. Wu, “Querying large scientific data sets with adaptable io system adios,” in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds., Cham: Springer International Publishing, 2018, pp. 51–69.

[15] 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.

[16] 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. DOI: 10.1109/SC.2018.00006.

[17] *Leveldb*, https://github.com/google/leveldb, 2014.

[18] *Rocksdb*, https://rocksdb.org/, 2014.

[19] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, no. 4, 2021. DOI: 10.1145/3483840.

[20] 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.

[21] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, "LocoFS: A loosely-coupled metadata service for distributed file systems," in *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*, 2017, 4:1–4:12. DOI: 10.1145/3126908.3126928.

[22] Q. Zheng, C. D. Cranor, G. R. Ganger, G. A. Gibson, G. Amvrosiadis, B. W. Settlemyer, and G. A. Grider, "Deltafs: A scalable no-ground-truth filesystem for massively-parallel computing," in *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 21)*, 2021. DOI: 10.1145/3458817.3476148.

[23] W. Lv, Y. Lu, Y. Zhang, P. Duan, and J. Shu, "InfiniFS: An efficient metadata service for Large-Scale distributed filesystems," in *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST 22)*, Feb. 2022, pp. 313–328.

[24] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "The case for custom storage backends in distributed storage systems," *ACM Trans. Storage*, vol. 16, no. 2, 2020. DOI: 10.1145/3386362.

[25] D.-Y. Lee, K. Jeong, S.-H. Han, J.-S. Kim, J.-Y. Hwang, and S. Cho, "Understanding write behaviors of storage backends in ceph object store," in *Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology (MSST 17)*, vol. 10, 2017.

[26] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li, "Storage management in asterixdb," *Proc. VLDB Endow.*, vol. 7, no. 10, 841–852, 2014. DOI: 10.14778/2732951.2732958.

[27] Y. Matsunobu, S. Dong, and H. Lee, "Myrocks: Lsm-tree database storage engine serving facebook's social graph," *Proc. VLDB Endow.*, vol. 13, no. 12, 3217–3230, 2020. DOI: 10.14778/3415478.3415546.

[28] P. Desai and K. Leong, *Rockset concepts, design, and architecture*, https://rockset.com/Rockset_Concepts_Design_Architecture.pdf, 2022.

[29] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-structured Merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996. DOI: 10.1007/s002360050048.

[30] M. A. Qader, S. Cheng, and V. Hristidis, "A comparative study of secondary indexing techniques in lsm-based nosql databases," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD 18)*, 2018, 551–566. DOI: 10.1145/3183713.3196900.

[31] C. Luo and M. J. Carey, "Efficient data ingestion and query processing for lsm-based storage systems," *Proc. VLDB Endow.*, vol. 12, no. 5, 531–543, 2019. DOI: 10.14778/3303753.3303759.

[32] C. Tang, J. Wan, Z. Tan, and G. Li, "Accelerating range queries of primary and secondary indices for key-value separation," in *Proceedings of the 13th Symposium on Cloud Computing (SoCC 22)*, 2022, 226–239. DOI: 10.1145/3542929.3563479.

[33] J. V. D'silva, R. Ruiz-Carrillo, C. Yu, M. Y. Ahmad, and B. Kemme, "Secondary indexing techniques for key-value stores: Two rings to rule them all," in *International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*, 2017.

[34] C. Luo and M. J. Carey, "On performance stability in lsm-based storage systems," *Proc. VLDB Endow.*, vol. 13, no. 4, 449–462, 2019. DOI: 10.14778/3372716.3372719.

[35] F. Mei, Q. Cao, H. Jiang, and L. T. Tintri, "Lsm-tree managed storage for large-scale key-value store," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 17)*, 2017, 142–156. DOI: 10.1145/3127479.3127486.

[36] *Computational storage architecture and programming model v1.0*, https://www.snia.org/sites/default/files/technical-work/computational/release/SNIA-Computational-Storage-Architecture-and-Programming-Model-1.0.pdf, 2022.

[37] Y. Kang, R. Pitchumani, P. Mishra, Y.-s. Kee, F. Londono, S. Oh, J. Lee, and D. D. G. Lee, "Towards building a high-performance, scale-in key-value storage system," in *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR 19)*, 2019, 144–154. DOI: 10.1145/3319647.3325831.

[38] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis, "ZNS: Avoiding the block interface tax for flash-based SSDs," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*, Jul. 2021, pp. 689–703.

[39] *Nvm express zoned namespace command set specification*, https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-1.1c-2022.10.03-Ratified.pdf, 2022.

[40] *Sidewinder-100 pcie nvme storage controller*, https://fidus.com/wp-content/uploads/2019/01/Sidewinder_Data_Sheet.pdf, 2019.

[41] *Storage performance development kit*, https://spdk.io/, 2015.

[42] Y. Liu, Y. Lu, Z. Chen, and M. Zhao, "Pacon: Improving scalability and efficiency of metadata service through partial consistency," in *Proceedings of the 2020 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 20)*, 2020, pp. 986–996. DOI: 10.1109/IPDPS47924.2020.00105.

[43] B. Dong, S. Byna, and K. Wu, "Sds-sort: Scalable dynamic skew-aware parallel sorting," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 16)*, 2016, 57–68. DOI: 10.1145/2907294.2907300.

[44] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and R. D. Ryne, "Parallel index and query for large scale data analysis," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*, 2011, 30:1–30:11. DOI: 10.1145/2063384.2063424.

[45] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks," *Concurr. Comput. : Pract. Exper.*, vol. 26, no. 7, pp. 1453–1473, May 2014. DOI: 10.1002/cpe.3125.

[46] Q. Zheng, C. D. Cranor, A. Jain, G. R. Ganger, G. A. Gibson, G. Amvrosiadis, B. W. Settlemyer, and G. Grider, "Streaming data reorganization at scale with deltafs indexed massive directories," *ACM Trans. Storage*, vol. 16, no. 4, Sep. 2020. DOI: 10.1145/3415581.

[47] *Pe8000 series - sk hynix ssd*, https://product.skhynix.com/products/ssd/essd/pe8000.go, 2022.

[48] *Kioxia enterprise ssd data sheet*, https://americas.kioxia.com/content/dam/kioxia/shared/business/ssd/enterprise-ssd/asset/datasheet/EnterpriseSSD_DataSheet_E.pdf, 2022.

[49] *Samsung develops high-performance pcie 5.0 ssd for enterprise servers*, https://semiconductor.samsung.com/newsroom/news/samsung-develops-high-performance-pcie-5-0-ssd-for-enterprise-servers/, 2021.

[50] B. Settlemyer, G. Amvrosiadis, P. Carns, and R. Ross, "It's time to talk about hpc storage: Perspectives on the past and future," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 63–68, 2021. DOI: 10.1109/MCSE.2021.3117353.

[51] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, 2003, pp. 380–386.

[52] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002, pp. 231–244.

[53] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)*, 2019, 353–369. DOI: 10.1145/3341301.3359656.

[54] *Nvm express base specification*, https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf, 2022.

[55] 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.

[56] *Nvm express key value command set specification*, https://nvmexpress.org/wp-content/uploads/NVM-Express-Key-Value-Command-Set-Specification-1.0c-2022.10.03-Ratified-1.pdf, 2022.

[57] A. Acharya, M. Uysal, and J. Saltz, “Active disks: Programming model, algorithms and evaluation,” *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, 81–91, Oct. 1998. DOI: 10.1145/384265.291026.

[58] K. Keeton, D. A. Patterson, and J. M. Hellerstein, “A case for intelligent disks (idisks),” *SIGMOD Rec.*, vol. 27, no. 3, 42–52, Sep. 1998. DOI: 10.1145/290593.290602.

[59] E. Riedel, G. A. Gibson, and C. Faloutsos, “Active storage for large-scale data mining and multimedia,” in *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB 98)*, 1998, 62–73.

[60] K. Ren and G. Gibson, “TABLEFS: Enhancing metadata efficiency in the local file system,” in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 145–156.

[61] G. Steiner and B. Philofsky, *Managing power and performance with the zynq ultrascale+ mpsoc*, https://docs.xilinx.com/v/u/en-US/wp482-zu-pwr-perf, 2016.

[62] *Vpic*, https://github.com/lanl/vpic, 2008.

[63] Eideticom, *Noload computational storage processor*, https://www.eideticom.com/media/attachments/2020/06/03/noload-compression-zfs.pdf, 2020.

[64] Samsung Electronics, *Samsung Electronics Develops Second-Generation SmartSSD Computational Storage Drive With Upgraded Processing Functionality*, 2022. [Online]. Available: https://news.samsung.com/global/.

[65] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves, “Cost-Effective, Energy-Efficient, and Scalable Storage Computing for Large-Scale AI Applications,” *ACM Trans. Storage*, vol. 16, no. 4, 2020. DOI: 10.1145/3415580.

[66] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou, *In-storage Processing of I/O Intensive Applications on Computational Storage Drives*, 2021. eprint: arXiv: 2112.12415. [Online]. Available: https://arxiv.org/abs/2112.12415.

[67] A. HeydariGorji, S. Rezaei, M. Torabzadehkashi, H. Bobarshad, V. Alves, and P. H. Chou, “HyperTune: Dynamic Hyperparameter Tuning for Efficient Distribution of DNN Training over Heterogeneous Systems,” in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD ’20, Virtual Event, USA, 2020.

[68] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, “Active flash: Towards Energy-Efficient, In-Situ data analytics on Extreme-Scale machines,” in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, ser. FAST ’13’, Feb. 2013, pp. 119–132.

[69] C. Lukken and A. Trivedi, “Past, Present and Future of Computational Storage: A Survey,” *CoRR*, vol. abs/2112.09691, 2021. arXiv: 2112.09691.

[70] A. Barbalace and J. Do, “Computational storage: Where are we today?” In *CIDR*, 2021.

[71] A. Barbalace, M. Decky, J. Picorel, and P. Bhatotia, “BlockNDP: Block-Storage Near Data Processing,” in *Proceedings of the 21st International Middleware Conference Industrial Track*, ser. Middleware ’20, Delft, Netherlands, 2020, 8–15.

[72] C. Lukken, G. Frascaria, and A. Trivedi, “ZCSD: a Computational Storage Device over Zoned Namespaces (ZNS) SSDs,” *arXiv preprint arXiv:2112.00142*, 2021.

[73] G. Frascaria, “E2bpf: An evaluation of in-kernel data processing with ebpf,” Ph.D. dissertation, Universiteit van Amsterdam, 2021.

[74] M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, H. Bobarshad, V. Alves, and N. Bagherzadeh, “Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics,” in *Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP ’19, 2019, pp. 430–437. DOI: 10.1109/EMPDP.2019.8671589.

[75] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, “Biscuit: A Framework for Near-Data Processing of Big Data Workloads,” in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, ser. ISCA ’16, 2016, pp. 153–165. DOI: 10.1109/ISCA.2016.23.

[76] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, “NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’21, Virtual Event, USA, 2021, 262–272.

[77] S. Salamat, H. Zhang, Y. S. Ki, and T. Rosing, “NASCENT2: Generic Near-Storage Sort Accelerator for Data Analytics on SmartSSD,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 2, 2022.

[78] Scaleflux Inc, *Scaleflux*, "http://www.scaleflux.com/", 2022.

[79] Seagate, *Kinetic HDD*, "https://www.seagate.com/support/enterprise-servers-storage/nearline-storage/kinetic-hdd/", 2022.

[80] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, “KAML: A Flexible, High-Performance Key-Value SSD,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2017, pp. 373–384.

[81] J. Im, J. Bae, C. Chung, Arvind, and S. Lee, “PinK: High-speed In-storage Key-value Store with Bounded Tails,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, USENIX, 2020, pp. 173–187.

[82] C.-G. Lee, H. Kang, D. Park, S. Park, Y. Kim, J. Noh, W. Chung, and K. Park, “iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics,” in *Proceedings of the 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, 2019, pp. 384–395.

[83] D. Min and Y. Kim, “Isolating Namespace and Performance in Key-Value SSDs for Multi-tenant Environments,” in *Proceedings of the 13th USENIX Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage ’21, 2023.

[84] J. Bhimani, J. Yang, N. Mi, C. Choi, and M. Saha, “Fine-grained Control of Concurrency within KV-SSDs,” in *Proceedings of the 14th ACM International System and Storage Conference (SYSTOR)*, ACM, 2021, pp. 1–12.

[85] S.-M. Wu, K.-H. Lin, and L.-P. Chang, “KVSSD: Close Integration of LSM Trees and Flash Translation Layer for Write-efficient KV Store,” in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, pp. 563–568.

[86] *Samsung smart ssd*, https://samsungatfirst.com/smartssd-ocp/, 2018.

[87] C. Duffy, J. Shim, S.-H. Kim, and J.-S. Kim, “Dotori: A key-value ssd based kv store,” *Proc. VLDB Endow.*, vol. 16, no. 6, 1560–1572, 2023. DOI: 10.14778/3583140.3583167. [Online]. Available: https://doi.org/10.14778/3583140.3583167.

[88] M. Qin, Q. Zheng, J. Lee, B. Settlemyer, F. Wen, N. Reddy, and P. Gratz, “Kvrangedb: Range queries for a hash-based key-value device,” *ACM Trans. Storage*, 2023, Just Accepted. DOI: 10.1145/3582013.

[89] S. Sarkar and M. Athanassoulis, “Dissecting, designing, and optimizing lsm-based data stores,” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD ’22, Philadelphia, PA, USA: Association for Computing Machinery, 2022, 2489–2497. DOI: 10.1145/3514221.3522563. [Online]. Available: https://doi.org/10.1145/3514221.3522563.

[90] C. Luo and M. J. Carey, “Lsm-based storage techniques: A survey,” *The VLDB Journal*, vol. 29, pp. 393–418, 2018. [Online]. Available: https://doi.org/10.1007/s00778-019-00555-y.