

Elevating Performance of LSM-Tree-Based Key-Value Stores with Gradient Data Hierarchy

Hui Sun and Jinfeng Xu

*School of Computer Science and Technology
Anhui University
Hefei, China
sunhui,jinfengx@ahu.edu.cn*

Xiao Qin

*Department of Computer Science and
Software Engineering
Auburn University, AL, USA
xqin@auburn.edu*

Abstract—Key-value stores are a key player of managing large-scale unstructured data in storage systems. Performance improvement of the LSM-tree structure has been extensively investigated, but current work primarily focuses on cache structural optimization rather than hot-and-cold data properties. Moreover, existing and external memory components of LSM-tree rarely have uniform hot and cold attributions. In this study, we make use of the gradient and hierarchy mechanism to optimize the components catering for cache data. We design an adaptive data migration method according to hot and cold data in the cache. We reform and expand a gradient cold-hot data hierarchy (GDH) mechanism that replaces the in-memory data structure to address the problem of missing hot and cold data attributes. The hot and cold data are placed in separate cache partitions to store hot data as far the high hierarchy as possible, reducing I/O accesses. When it comes to frequently accessed hot data, we advocate for a hotness-aware technique for data stored on a disk, where read-write performance and the cache hit rate are revamped. The experiment results reveal that our proposed GDH achieves a high cache-hit ratio and low access latency under a wide range of workloads.

Index Terms—Key-value store, Cache, Hot and cold data hierarchy, GPU Acceleration

I. INTRODUCTION

NoSQL was proposed to meet the requirements of large-scale concurrency requests. LSM-tree-based key-value stores – KV stores – in NoSQL offer an efficient management solution coping with distributed data. LSM-tree is a hierarchical and ordered data structure. The write performance of KV stores is highly competent because random writes are ordered sequentially to bolster write throughput. The high performance of KV stores, however, become possible at the expense of storing redundant data. The compaction procedure converts SSTables with unordered and overlapping key ranges into ordered and non-overlaps ones in KV stores. To prevent redundant keys between levels and ensure efficient read performance, this approach gathers invalid data. The LSM-tree has gained widespread adoption in the industry, with examples including LevelDB, multi-dimensional and multi-threaded RocksDB, Wiskey that separates keys and values, and PebblesDB with its guard structure. With the rapid increase in storage device transmission speeds, such as NVMe SSDs and PCIe SSDs, the performance gap between sequential and random write I/Os has significantly narrowed.

Wiskey [1] is designed to store keys and values separately, with the keys kept in an LSM-tree and placed in an additional log file. This approach is particularly useful when dealing with large value sizes, and it employs a crash consistency and garbage collection (GC) strategy to manage the log file. However, for small-size values, there are some limitations. One such limitation is that Wiskey does not optimize the cache component, which means that it is unable to accelerate reads under a typical WPI workload (i.e., 75% lookups and 25% writes). In pursuit of high read performance, we explore in this study the optimization of memory components. In KV stores, the hit rate of the cache is influenced by the hot and cold properties of the data. It is thus essential to accurately differentiate between hot and cold data, particularly since hot data constitutes the majority of user requests. To this end, we advocate for using a cascading-loop bloom filter to classify data as hot or cold. In addition, we develop a Hash Table-based index mechanism to replace the in-memory skip list. It is worth noting that we also devise a hash structure that is slated to optimize both read and write performance, with an approximate time complexity of $O(1)$.

The prevalent memory structures currently in use typically rely on a B/B+ tree structure as a substitute for an in-memory skip table. The reason for this choice is that the B/B+ tree is a well-balanced structure that enhances both read and write speed. Nonetheless, the adoption of the B/B+ tree leads to additional memory usage. Furthermore, in certain scenarios where the data is frequently accessed, the B/B+ tree can cause uneven I/O operations. One solution to this problem is to transfer unordered hash data to a GPU, which can exploit its superior computational capacity to sort the data. Moreover, it is possible to cache frequently accessed key-value (KV) pairs in GPGPU memory. This approach enables GDH to prioritize storing new KV pairs in memory, rather than on disk. As long as the requested KVs are found in either CPU or GPU memory, the system avoids issuing requests to the disk. As a result, this technique enhances read performance and minimizes unnecessary I/O operations.

In this study, our propose the GDH system to offer the following contributions to cache management in KV stores.

- We design both a filter and an in-memory component that can effectively identify hot and cold data, which

dramatically enhances compaction performance, as well as reads and write operations.

- We implement a hotness-based cross-level data migration scheme in the KV store. This scheme allows for hierarchical storage of hot and cold data by enabling the mobility of data across different storage levels. By doing so, we can take advantage of new hardware that joins the LSM-tree system architecture, thereby enhancing the performance of the system.
- We design a GPU sort algorithm that leverages the computational resources available on a GPU. By storing data in GPU memory, we can optimize read performance, resulting in faster data processing.
- We optimize the garbage collection (GC) process in KV stores using a key-value separation method. Additionally, we propose an in-situ GC strategy that minimizes overhead and reduces the amount of invalid space generated during GC operations.
- We implement GDH-enabled KV store and conduct extensive experiments. The results unveil that, the memory component we implemented achieved a memory read hit ratio of 36.87%. We achieved a maximum reduction of 94.9% in average latency under different hot and cold loads, and a maximum reduction of 96.7% in P99.9 latency. Under different read/write ratios, we achieved a maximum reduction of 67.3% in average latency, and a maximum reduction of 91.84% in P99.9 latency.

II. BACKGROUND AND RELATED WORK

We shed light on the background of existing KV stores, which served as inspiration for our proposed solution.

A. Background

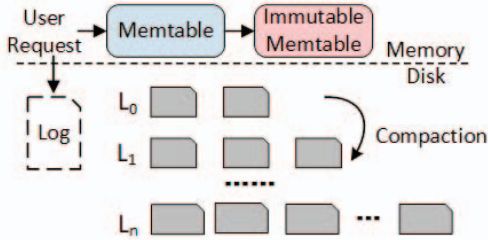


Fig. 1: Structure of the typical LSM-tree.

The LSM-tree is an ordered and hierarchical data structure that is specifically designed for disk-based storage. LSM tree-based KV stores are well-suited for workloads that involve large-scale write and retrieval operations. Taking LevelDB [2] as an example, we outline the structure of KV stores in Fig. 1. Memtable is stored in memory, and the skiplist is used as the structure. Data is flushed to disk when the immutable is full. Once the number of SSTables in L_k exceeds a threshold, one SSTable at higher level L_k compacts with SSTables that have overlapped key-ranges at lower level L_{k+1} . This procedure, which constructs newly sorted SSTables at level L_{k+1} and deletes invalid SSTables, is referred to as compaction.

There have been numerous studies focused on improving the performance of data storage systems. For example, FloDB [3]

and Accordion [4] optimize in-memory components through a multi-level approach. FloDB provides two distinctive data structures in levels to orchestrate big in-memory components: a small hash table receives fast user writes at the first level, and the second level is a large skiplist. Data in the hash table are ordered, and migrating to the skiplist averts random writes in main memory. Accordion partitions more levels to manage the in-memory components. At the high level, the mutable receives write; when the mutable is full, data is flushed into an immutable component. The large memory component can primarily suppress the I/Os overhead by merging multiple immutables in memory.

LSM-tree optimization based on SSDs/NVMs. Software frameworks, such as tiering merge and key-value separation, were designed to fully utilize the high-performance storage devices. These leading-edge techniques improve write and read performance with a prolonged device lifetime. Non-volatile memories (NVMs) are widely used in hybrid memory systems to achieve high throughput and low latency. MatrixKV [5], based on a hybrid NVM memory structure, applies the byte-addressability of NVM to store the entire L_0 in NVM: the matrix container manages the L_0 level. MatrixKV also performs fine-grained column compaction for data at levels L_0 and L_1 , minimizing write stalls. SpanDB [6] improves NVMe SSD and commodity SSD-based KV storage. SpanDB selectively allocates critical and common data in a hybrid storage system. For example, large amounts of data are stored on cost-effective and high-performance SSDs. Write-ahead log files (WAL) and data at the top level of the LSM tree are reallocated to a fast NVMe SSD with small capacity, thereby increasing the throughput of the KV stores. Hybrid KV stores (KVell[7] and MioDB [8]) have become a popular configuration for storage systems.

I/O patterns in workloads. The I/O patterns of workloads primarily affect the performance of Memcached. A growing number of existing studies investigated the I/O patterns in Memcached clusters, such as production workload in Facebook [9][10], the findings of which can aid in the design of in-memory KV store hybrid systems. It is essential to characterize workloads in KV stores. We conduct extensive experiments on db_bench in LevelDB and all the default loads in YCSB, see Table I. We discover that more than 40% of hot data (read more than once) and more than 10% of hot data is accessed more than three times under db_bench: the trend means that there are more hot data in workloads generated by db_bench. For the YCSB workloads, the proportion of hot data in all the workloads is in a range between 20% and 40%. In workload C, for instance, the hot-data proportion reaches 42.15%. These hot data listed in Table I suggest that it is desirable to consider identifying hot data in design of KV stores because hot data that are frequently accessed may remain hot for a extended time. Therefore, we ought to cache these hot data to boost read performance. Besides, storing hot data in cache improves cache utilization, as frequently accessed data are readily available in the cache.

TABLE I: Characteristics of db_bench YCSB-C (Zipfian = 0.99) Workloads

db_bench	Access frequency of keys(5 million)						The ratio of access frequency of keys(%)					
	1	2	3	4	5	≥ 5	1	2	3	4	5	≥ 5
fillrandom	1840943	921612	305822	76159	15133	2929	58.21	29.14	9.67	2.41	0.48	0.09
YCSB-C	1	2	3	4	5	≥ 5	1	2	3	4	5	≥ 5
Load	996851	2579	295	86	53	136	99.69	0.26	0.03	0.01	0.01	0.01
Workload-A	76193	21002	7923	3884	2086	7593	64.2	17.7	6.68	3.27	1.76	6.4
Workload-B	17354	1928	610	291	187	686	82.4	9.16	2.9	1.38	0.89	3.26
Workload-C	103910	35404	15748	8319	4913	16530	57.85	19.71	8.77	4.63	2.74	9.2
Workload-D	207104	32026	11444	5568	3360	13409	78.1	12.08	4.32	2.1	1.27	5.06
Workload-F	76193	21002	7923	3884	2086	7593	64.20	17.7	6.68	3.27	1.76	6.4

III. MOTIVATION

Although numerous solutions have been proposed to enhance the performance of LSM-Tree-based KV stores, we identify several challenges that still need to be addressed.

Optimizing Read and Write Performance. The conventional LSM-tree has a skiplist memory component with an average read/write overhead, lookup, cost of $\mathcal{O}(\log N)$. To optimize both read and write operations, we use a hash table as a primary data structure for the memory component aiming to achieve an approximate speedup for the insert and delete operations $\mathcal{O}(1)$. Hashing, a storage space-efficient scheme to large-scale data volumes, has very low time complexity. In this study, we explore the use of GPU sort as an alternative to flushing operations in the hash table. This is because flushing operations require sorting the entire table, which adds real-time overhead to the CPU and adversely impacts read and write performance.

Low Cache Read Hits. The lack of a hot and cold mechanism in the cache data management results in low cache read hits. Hot data needs to be cached as much as possible for in-memory caching. Because of the limited memory space available, the Hash Table is unable to store all keys simultaneously. To address this issue, it is necessary to distinguish between hot and cold data and allocate cache space accordingly. To differentiate between hot and cold data, the hot data is typically stored in a separate memory partition or cache, while the cold data is stored in a different location. Keeping track of hot and cold data metrics can help inform storage decisions and improve efficiency. The hot and cold hierarchy needs to be made on disk, which enables hot data to be as close to the upper level as possible and press cold data into the lower level. This idea prioritizes the most frequently accessed data and reduces the overhead associated with accessing data that is not as frequently used.

The Limitations of Existing Key-Value Separation. To reclaim invalid space, wisckey uses a combination of full table scan and HashKV[11] partitioned scans. In the HashKV partitioned scans, double pointers are used to move invalid areas to valid areas where there is relatively more available space. When reclaiming invalid space, it is important to consider how that space will be managed and reused in the future. We configure a dedicated locate table to record the start location and size of the invalid space. When a value is written, we look up an area from the located table to write the value, freeing the invalid space for reuse. Since SSDs (Non Volatile

Memory express is better) have similar performance for both random writes and sequential writes, it may be advantageous to replace append writes with random writes in the Value Store File on disk under certain conditions.

What Makes Our Solution Superior. We adopt a hash table to replace the SkipList in memory. Our design enjoys a swift read/write accesses with time complexity of $\mathcal{O}(1)$ through hash table. To better manage hot data of different frequencies, we incorporate a cascaded loop bloom filter into our system. To address the significant computational resources consumption associated with CPU sorting, we implement GPU sort in our system. In the disk, hot and cold values are brought into a file, which is dumped according to its hotness. To address the issue of the key-value separation leading to a large invalid space, we implement an in-place write strategy.

IV. GDH DESIGN AND IMPLEMENTATION

The GDH is composed of memory, GPU, and disk management modules, The workflow is shown in the Fig. 2.

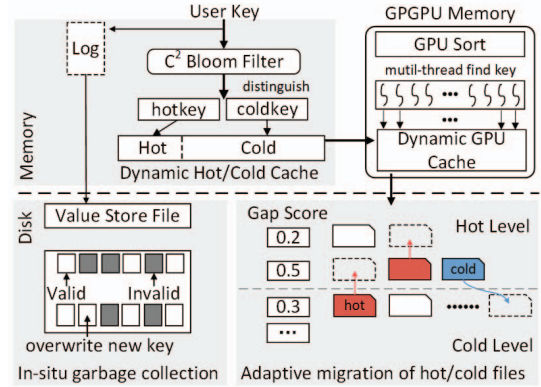


Fig. 2: Overall System of GDH.

Fig. 2 depicts that when a user request arrives, the key-value pairs are first separated, the value is written to the value store file, and the key goes through a cascaded loop bloom filter to differentiate between hot and cold. Data is stored in two buffers: the hot buffer and the cold buffer. When the hot buffer reaches its capacity, the hot data is transferred to the cold buffer. Similarly, when the cold buffer is filled, the cold data is sent to a heterogeneous device for sorting and caching. The LSM-tree stores the processed keys. As keys are compiled into an SSTable, their hotness is tallied. As the SSTable is written to disk, the metadata records these hotness values. Using this

metadata, the system can then efficiently move hot and cold data, achieving a hierarchical storage of the data. Whenever an updated key is added to the KV store, it is inserted into the invalid space of value store file rather than using the append strategy. The technique known as in-situ GC.

A. Hot and cold data identification mechanism

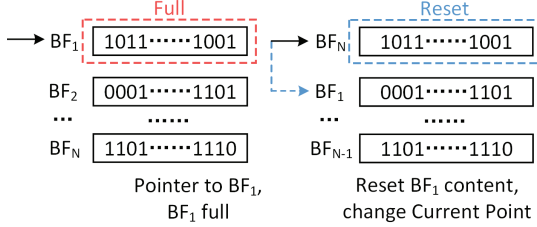


Fig. 3: Cascaded and Cyclic Bloom Filter in GDH.

Data has both temporal and spatial characteristics. Hot data that is currently being accessed may continue to be accessed in the future. Simply distinguishing between hot and cold data is not enough. We develop a precise approach for identifying and differentiating between hot and cold data: this solution allows us to perform operations based on the temperature of the data, such as partitioning the storage space for hot and cold data and placing them on separate storage devices. We propose a management module that takes advantage of multi-stage cyclic Bloom filter. The module consists of N (the default value, $N = 3$) same-size Bloom filters that form a cyclic queue for filtering hot and cold data. The head of the queue is indicated by a current pointer, and we assign weights to nodes from the head to the tail of the queue.

Fig. 3 plots a set of same-sized bloom filters form a cyclic queue. The bloom filter referenced by the head pointer is BF_1 , and the others in turn are $BF_2, \dots, BF_N$. The bloom filter BF_1 is located at the start of the queue and is initially filled up. Once it becomes full, the cyclic bloom filter initiates the subsequent operations. Initially, the content of BF_1 is cleared, and all bits are set to 0. Next, the head pointer is moved to BF_2 , and all numbers are reset to the default value of one bit. When the bloom filter pointed to by the head pointer becomes full, the above process is repeated. This procedure is the working mechanism of cascading bloom filters, which are efficient in identifying hot and cold data and occupy a fixed-size memory space.

We can determine the hotness value of a cascaded bloom filter at any level by defining the set $S = 1, 2, 3, \dots, N - 1$. Then, the hotness value (H) of the key at level k can be computed as follows:

$$H = \begin{cases} \sum_{k=1}^{N-1} 2 * (k - 1), & k \in S \\ Max, & k = N \end{cases}$$

The filtering rules are implemented as follows. To prevent the bloom filter from becoming too large, we constrain the number of keys it can hold. This involves checking whether the number of keys in the first bloom filter indicated by the current pointer has surpassed the limit. If it has, we reset the

contents of the current bloom filter to zero and redirect the current pointer to the next bloom filter. When a key arrives, it is inserted into the first bloom filter if the key does not already exist there, and its hotness value is H_1 : all keys have an initial hotness value of H_1 . If the key already exists in the first bloom filter, the system checks the next bloom filter. If the key is found there, it moves on to the following bloom filter; otherwise, the key is added to that bloom filter, and its hotness value is $(H_1 + H_2)$. If the key does not already exist, it is added, and its hotness value is set to $(H_1 + H_2 + H_3)$. If it does exist, its hotness value is immediately set to the default maximum value. In addition, if the number of key occurrences exceeds the number of Bloom filters, the key will be extremely hot. Then, we merely set the hotness value to the default maximum. Nevertheless, this extreme case is rare in the real world. And the extremely hot data are handled separately by increasing the number of Bloom filters.

A higher frequency of data access and shorter time intervals between frequent visits result in higher hotness values. For simplicity, let us assume that the time interval between each bloom filter is a default of 2 (as the weight mentioned above). In this case, we can gauge the hotness value based on the number of visits and the time interval. Accurately distinguishing between hot and cold data using the cascaded loop Bloom filter can provide quantitative support for caching and adaptive movement of data. This precise differentiation allows for better utilization of the cache, which has a limited capacity. By keeping more hot data in memory instead of cold data, we maximize cache utilization and improve read performance. In contrast, failing to accurately identify hot and cold data can result in the cache being occupied by cold data, consuming valuable space without boosting read performance. Accurate hot and cold values are essential for efficiently moving files on disk based on heat measures. When the hotness of a file is inaccurately estimated, meaningless file movement will leads to wastage of I/O bandwidth. Therefore, precise differentiation between hot and cold data is crucial for optimizing disk space utilization and improving overall system performance.

B. Dynamic hot and cold cache partitioning

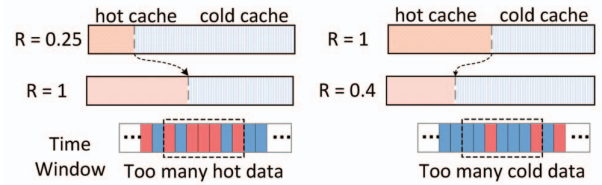


Fig. 4: Dynamic hot and cold cache partition.

When data has specific hot and cold labels, it's necessary to partition the cache space to optimize caching for hot data and reduce memory usage. To achieve this, we propose a dynamic cache partitioning method with a hot/cold boundary. It's important to note that the ratio of hot to cold data is not fixed and can vary over time: for example, when data

is accessed intensively, the ratio of hot data is high. We introduce a ratio parameter R , defined as the number of hot data divided by the number of cold data (default value is 0.25) to dynamically adjust the cache partitioning.

To achieve a hot and cold adaptive cache, we follow these steps within a time window of fixed size. Firstly, we calculate the ratio of hot and cold data within this window, which allows us to compute the R -value for the current time window. Next, we update the previous R -value based on the R -value computed for the latest time window. By using this scheme, we can avoid unnecessary disk I/Os that may occur when frequently accessing such data. It is worth noting that the size of the hot data cache should not exceed half of the overall cache size, which means that the proportional value of R should be no greater than 1. This is because, despite having a lot of hot data in the current time window, we still need to retire some of the old hot data and move it into the cold data area to make room for new hot data. Therefore, we must reserve some space in the cache for storing cold data. However, because the hot and cold data buffers are kept in memory, the lookup speed remains unchanged.

The hash table module is designed to optimize both read and write operations, with similar speed performance of $\mathcal{O}(1)$. The hash table is divided into two areas: hot and cold. The hot area contains only the keys with the highest usage frequency, while the remaining keys are stored in the cold area. Once the hot and cold data are sorted, we create two separate hash tables: one for keys with a hot value greater than zero and another for keys with a hot value of zero. Furthermore, we remove outdated hot data from the cold area by transferring half of the data based on the current contents of the hot area. Once the cold area is full, the data can be transferred to the module on the disk for sorting. Meanwhile, the value is written to the value store file by appending a write operation.

C. Heterogeneous Co-processing and Caching

The heterogeneous co-processing module comprises a GPU sort and a cache module. To take advantage of the vast memory space and computational capability of the GPU, we developed a GPU buffer that dynamically adjusts its size based on the GPU's maximum memory capacity. This approach is intended to achieve highly efficient read optimization. The GPU buffer is primarily designed for fast data retrieval. Whenever a key-value pair from a user request enters the write queue, the full pair is written to the available GPU buffer to ensure that the data in the GPU cache is up-to-date. If we attempt to insert more key-value pairs into the buffer than its capacity allows, we will discard the oldest key-value pairs using a First-In-First-Out (FIFO) scheme. Since the GPU cache maintains a complete record of key-value pairs, the query path of a point query will take longer during the query process in the GPU cache. The complete process is outlined as follows.

During a point query, we first search the memory component. If we find the target key in memory, we immediately return the result. However, if the target key is not found in memory, we trigger a GPU-side search in the local cache. This

search process can be parallelized in the GPU, leveraging its parallel computing capability. All GPU threads can perform a parallel search and return the results. Once the target keys are found, we return the corresponding results. In contrast, if the target key is not found in memory or in the local cache, we initiate a search procedure on the disk. It is worth noting that we only conduct this search for hot data, as cold data is less likely to be stored in the GPU cache. By doing so, we can enhance the GPU cache hit ratio in this scenario.

The GPU sort component is specifically designed to address the issue when the data in a hash table is naturally out of order. When the number of keys in the cold cache reaches its maximum capacity, the CPU sends all cold keys in the cold hash table to the GPU. The GPU leverages its massive computing power to sort the keys and remove any old keys using parallel computational resources. The GPU sort algorithm used in our system is a new GPU sort byte stream algorithm, which can significantly speed up the sorting process. Once the GPU has completed the sorting, the results are directly shipped back to memory. The CPU then controls the generation of SSTables and stores them on the disk. In our experiment, we reckon that the GPU accelerates the sorting performance by tens of times compared with CPU. In this paper, we employ the heterogeneous devices GPU in the system, in which this module is optional. Without GPU, the query and sorting performance tend to be reduced.

D. Data Hierarchy and Adaptive Movement

In our system, we assume that a database with a large amount of data is like a data ocean, with a hierarchy of hot and cold data based on the number of accesses. The hot and cold data can be exchanged with each other, and it is necessary to manage this data movement in order to move the more frequently accessed hot data up from the bottom level to a higher level, making it easier to access. We refer to this movement of hot data as an "upward" operation. In contrast, cold data that is not frequently accessed moves down the hierarchy to the bottom level, which we refer to as a "downward" operation. This approach enables us to dynamically adjust the cache to better reflect the changing access patterns of the database, improving overall performance.

To manage the key metadata SSTable and the sum of the values of written keys, the disk component uses a hot and cold hierarchy. The SSTable is marked to differentiate between hot and cold files. The files' hot values aid in the downward movement of cold data and upward movement of hot data. Storing data hierarchically on different-speed storage devices is practical for storage systems. The hot value facilitates hierarchical storage, enabling the storage of data with low hot values on slower storage devices such as HDD, and higher hot values on high-performance storage devices like SSD and NVM. Therefore, a hierarchy of hot and cold data storage can be achieved. Let us define the first three levels as hot and the latter as cold levels. Furthermore, we introduce a gap score that verifies the difference between the hottest file ($H_{max} = \max\{H_i, i \in \{1, 2, \dots, N\}\}$) and the average

hotness of the files ($\bar{H} = (\sum_{i=1}^N H_i)/N$) in this level. The gap score for this level can be expressed as $S_{gap} = \frac{H_{max}}{H} - 1$, where N is the number of files at this level. This equation has restrictions in calculating the gap score: the gap score function cannot calculate the score of level 0 because level 0 is located in the highest level in the KV store. The data stored at this level can no longer be upward. Similarly, It's impossible for a cold file at the lowest level to move "downward".

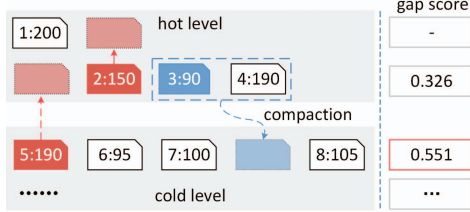


Fig. 5: Movement of hot and cold data. In value 1:200, '1' is the file number and '200' is the hotness of the file.

To determine when to move the hot and cold SSTables, we use the gap score. We examine the gap score of each level and select the one with the largest score. In that level, we move the hot and cold SSTables, and this action invokes the adaptive movement mechanism. Fig. 5 illustrates that it is not viable to compute the gap score at level 0. At the second and third levels, we calculate the corresponding gap score using the above formula. For the sake of clarity, we do not display the gap score of the remaining levels. We observe that the gap score of the third level is greater than that of the second level, which indicates that the SSTable with the highest hotness value in the third level has a greater difference in hotness value than the one in the second level. Therefore, it would be beneficial to transfer the SSTable from the third level to the second level, and we move the hottest SSTable to the third level.

Hot and cold SSTables are transferred before starting the compaction. If the key range of a hot SSTable does not overlap with the upper-level SSTables, its metadata will be moved to the upper level. Conversely, the low-level driver will trigger the bottom-up compaction process. We select the SSTable with overlapping hot data and generate a new SSTable after compaction, which is placed on the upper level. The sum of the hot values of the new SSTable is recorded. The procedure for cold data downward is simpler than that of hot data upward, as it involves only gap score calculation and metadata judgment. When compaction is kicked in, the file with the smallest hot value is included in the process, rather than calculating which level needs to move the cold files.

E. In-situ Garbage Collection

In Wiskey, the GC selects a fixed-size GC area and queries the LSM-tree to determine whether the KV pairs are valid. The valid KV pairs will be copy-rewritten to a new log file by append-write mode. Otherwise, the invalid data will be discarded directly. After the scan of the log file, the log file processed in GC is deleted. The following write KV pairs can only be written in a new log file in the append-write mode.

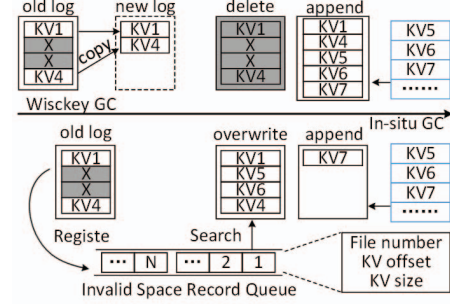


Fig. 6: Comparison of two GC methods.

In the proposed GDH, we propose a new GC method named in-situ GC, see Fig.6 to address the issue that invalid space cannot be reused in lightweight online GC procedure. We collect the invalid space in the log file by the metadata of invalid KV pairs in compaction. The metadata includes the log number, the offset of the invalid space, and the size of the invalid space. The information of KV pairs in the invalid space cannot need to be recorded. The metadata is stored in a queue; an atomic variable records the queue size. Metadata of the new invalid space is stored in the head of the queue while the size of the queue is plus one. We delete metadata at the end of the queue, and the queue size is subtracted by one. The data is written to the invalid space according to information of the invalid space in the metadata.

In our proposed queue model, the record and recycle of the invalid space becomes a producer and consumer mode where record and recycle procedures cannot affect each other. The thread-safe mechanism enables this method to achieve the feasibility in multi-thread scenarios. We must determine whether the invalid space size is enough to store the inserted KV pairs. When insufficient space exists, the KV pairs are written to the latest log file in the append-write mode. The data outside of the log file are randomly written in the space. In-situ GC, the file number of the invalid space cannot be equal to one of the latest log files. This is because the append-write mode used as the write mechanism cannot support random write. The large-size GC area requires a big log-file table, but the proposed queue method avoids the large-size log table used in the GC area. The queue size depends on the performance gap between producers and consumers. The in-place GC strategy avoids the accumulation of invalid space, lowering space amplification. Then, we find that the in-situ reduces the size of the KV store.

F. Discussion

In the proposed GDH, we spearhead to lessen the overhead of designed modules in implementing GDH. In the hot-and-cold data identification module, we determine the size of the cascading loop Bloom filter; meanwhile, data is reset when a single Bloom filter is filled with elements, with a goal to avoid increasing growth of the filter size. In the dynamic partitioning module, we configure a partition identifier and accomplish the adaptive dynamic adjustment of the hot and cold partition

sizes by adjusting the identifier's location. The data-adaptive movement module changes the logical level of metadata, and few compaction operations are conducted; thus, the cost of this module decreases. The benefit of the in-place GC module outweighs its overhead because this module curtails space amplification by up to 21.2% even though there is a 10% reduction in throughput.

The parameters configuration in the system are dynamically adjusted according to scenarios. For example, we select the first three levels in GDH based on the compaction frequency. This strategy is appealing because the compaction frequency at the first three levels is high; therefore, we choose the first three levels as hot ones. For instance, in a hybrid memory system, we place several levels of GDH-enabled KV stores in persistent memory or DRAM; these levels are hot. The hotness value factor can be adjusted according to users requirements because the hotness value is a system-level metric, and various parameters lead to different system behaviors. The time window - an empirical parameter - is customized by users because the time-window size varies with workloads. When burst I/O accesses are more intensive, a small window is set; otherwise, a large-sized window is configured. In the GDH, hotkeys can be identified when the hotness value is greater than zero, entailing that the hotkeys exist in the cascading loop Bloom filter.

This is because the compaction frequency at the first three levels is high; therefore, we choose the first three levels as hot ones. For example, in a hybrid memory system, we place several levels of GDH-enabled KV stores in PM or DRAM; these levels are hot. The hotness value factor can be adjusted according to the users requirements because the hotness value is a system-level metric, and various parameters lead to different system behaviors. The time window is an empirical parameter that needs to be customized by the user because the size of the time window varies with the workloads. When the burst I/O accesses are more intensive, a small window is set; otherwise, a large-size window is configured. In the GDH, hotkeys can be identified when the hotness value is greater than zero. It means that the hotkeys exist in the cascading loop Bloom filter.

V. EXPERIMENTAL EVALUATION

We compare the performance of GDH with the alternative KV stores under the db_bench and YCSB-C workloads. We run the tested KV stores on a server with two Intel(R) Xeon(R) CPUs with E5-2630 v4 @ 2.20GHz CPUs, 32-GB DDR4, and SATA-III 500-GB SSDs. The operating system for the tests is an Ubuntu 20.04. We deploy the NVIDIA TESLA P100 GPGPU [12] with 16-GB memory. We employ CUDA V11.4 and implement GDH by extending LevelDB V1.20, which is used as the baseline in this test. We configure tested KV stores with 4-MB SSTables and 8-MB Memtable. The remainder of the parameters are set to the default configurations. The evaluated competitor KV stores include RocksDB V6.10 [13], LUDA [14], HyperLevelDB, and PebblesDB [15], which are

configured with the default settings. Among the leading-edge KV stores, LUDA [14] is a GPGPU-enabled KV store.

A. DB_Bench

DB_bench is a default micro-benchmark in LevelDB. We gauge a wide range of performance metrics, including throughput under fillrandom, time cost per operation, accessed-level distribution under read-write workloads, and hit rate of the memory table. We test a total of 2,000,000 KV pairs with 1 KB, 8 KB, and 32 KB, respectively, and obtain the following experimental results.

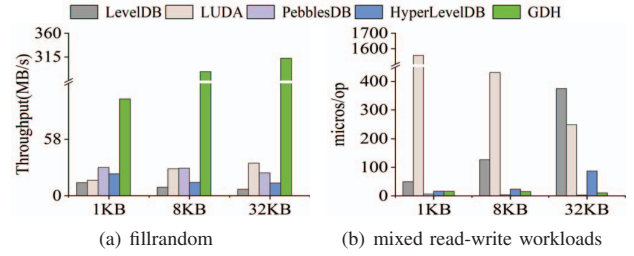


Fig. 7: Performance under DB_Bench.

Fig. 7(a) unveils the throughput of the tested KV stores under the fillrandom workload. According to the experimental results, GDH has a clear performance edge over the other KV stores when dealing with workloads that include 1 KB, 8 KB, and 32 KB KV pairs, achieving the highest throughput. Specifically, GDH demonstrates throughput improvements of 46.6x, 10.4x, 12.5x, and 24x over LevelDB, LUDA, PebblesDB, and HyperLevelDB, respectively. Fig. 7(b) depicts the time cost of each operation under a read-write workload for the KV stores. Notably, the number of reads varies among the different KV stores. For instance, while PebblesDB performs only a few thousand reads, the others conduct hundreds of thousands of reads. Consequently, the results reveal that PebblesDB has the lowest measure. By and large, GDH performs the best as its number of accesses is similar to those of the other KV stores excluding PebblesDB. Moreover, GDH achieves the lowest overhead given the same number of read operations. The performance discrepancy of KV stores becomes more pronounced under mixed read-write workloads. The performance of LevelDB decreases with large-size values. As the value size goes up, the SSTables move to a lower level, which searches data level by level to handle user requests, deteriorating system performance. LUDA optimizes write path without affecting read path. As the value size increases, LUDA processes fewer KV pairs using the same-size resources and processing the same task, which enhances the system performance.

In our investigation of cache hit ratios across various workloads with different KV pair sizes, we observe that GDH consistently delivers drastically higher cache hit ratios compared to the other KV stores, as summarized in Table II.

TABLE II: Cache hit ratio under workloads with 2,000,000 KV pairs.

	LevelDB	LUDA	PebblesDB	HyperLevelDB	GDH
value-1KB	2033	5747	4860	7989	737309
value-8KB	451	987	908	1676	737641
value-32KB	153	226	334	441	737913

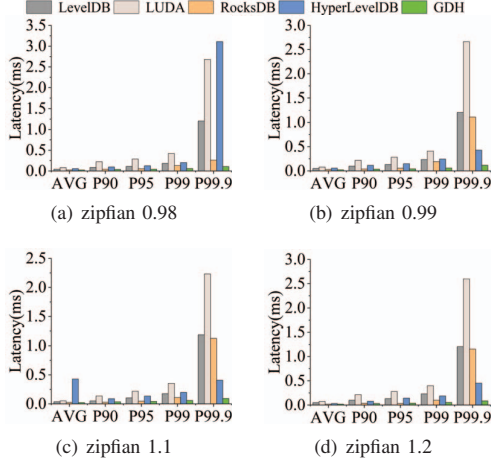


Fig. 8: Latency under Zipfian with different factors.

One reason for this improvement is that our cache scheme caches only keys, and since keys are generally smaller than their corresponding values, GDH manages to cache more information. Additionally, we utilize GPU memory as a cache to store more data, a distinctive feature not available to the alternative KV stores lacking heterogeneous devices, which further contributes to GDH's caching advantage.

B. YCSB-C

To delve into the performance comparisons of the KV stores, we employ the YCSB benchmark generating a diversity of workloads. Specifically, we configure two types of workloads each consisting of 20,000,000 1 KB key-value pairs. We use two types of workloads in the test. First, we configure workloads under the Zipfian distribution with different factors to study the distribution of hot data under various workloads. Second, we set up workloads with various ratios of mixed write-read to verify the low response latency of GDH.

1) *Average latency and tail latency under Zipfian distribution:* When the factor of Zipfian is larger under Zipfian-based workloads, there is more concentration of hot data, resulting in more frequent accesses to the hot data. Fig. 8(a) illustrates the performance of the KV stores under the evaluated workloads with a Zipfian factor of 0.98. Despite having less hot data volume than workloads with the default Zipfian factor, the separation of hot and cold data can impact performance. The experimental results unveil that GDH exhibits very low latency even with less hot data and less frequent accesses, thereby improving response time even under low-hot workloads. Notably, RocksDB's average response time is similar to GDH's P90 and P95 tail latency but 2.25x and 2.49x higher than GDH's P99 and P99.9 tail latency, respectively. In terms of P99.9 tail latency, LevelDB achieves 11.46x higher response time than GDH, while LUDA presents a 25.53x higher response time and HyperLevelDB's P99.9 tail latency is 29.64x higher than GDH. Overall, GDH with stable latency is superior to the other KV stores in terms of average and tail latency.

Under the workloads with a Zipfian factor of 0.99, the hot data volume increases with high access frequency. The

latency of GDH is shortened compared with that in the Zipfian case of 0.98 thanks to the hot-cold data cache management in GDH. The average latency of LevelDB is higher than the workload with the Zipfian factor of 0.98. The average latency of LUDA and HyperLevelDB remains unchanged, but RocksDB's average latency is enlarged. Under this workload, the P99.9 latency of KV stores is higher, but the latency of HyperLevelDB becomes lower.

We focus on the two workloads with the Zipfian factor of 1.1 and 1.2, which are higher than the default value – meaning that there is much more hot data under the two workloads, which can validate the hot and cold data identifying module in GDH. As the hot data volume expands, GDH's latency is curtailed, which indicates that the hot data is precisely cached in the hot data buffer to achieve high read performance and low read latency. Compared the workloads with Zipfian of 1.2 against that of Zipfian of 0.98, the average latency of GDH reduces by approximately 20%. Under the workloads with Zipfian of 1.1, HyperLevelDB's average latency and P99.9 tail latency is 19.71x and 4.45x higher than that of GDH, respectively.

The P99.9 latency of RocksDB is also 12.24x and 13.56x higher than GDH under workloads with Zipfian factors of 1.1 and 1.2, respectively. Although RocksDB mostly achieves low latency under workloads, LUDA maintained high latency throughout the tests. In summary, GDH accurately caches hot data by distinguishing between hot and cold data, resulting in low latency under workloads with more hot data. Compared to the other KV stores, GDH has lower P99 and P99.9 tail latency, which is attributed to its lightweight and efficient memory component. GDH – a design that is less expensive to operate – achieves low latency and high throughput under mixed workloads that have more hot spots. We conclude that GDH outperforms the other KV stores in terms of both average and tail latency, making GDH an efficient and effective choice for high-performance data storage and retrieval.

2) *Read-write and tail latency:* Under the load workloads, the highest average latency in the load phase is LevelDB because the average latency is related to the speed of the writes. LevelDB is a key-value database that uses a non-specialized, optimized LSM-tree structure. The average latency of the other KV stores is lower than that of LevelDB, but the competitors' latency measures are higher than GDH. More prosaically, the latency of RocksDB, HyperLevelDB, and LUDA is 1.86x, 1.96x, and 4.7x higher than that of GDH: GDH achieves much lower latency than the other KV stores at P90, P95, P99, and P99.9 tail latency. For instance, RocksDB's latency is 24x that of GDH at P90 tail latency. HyperLevelDB performs relatively smoothly. GDH benefits from high write performance thanks to the presence of an in-memory hash table and cache.

Fig. 9(b) shows the latency of KV stores under the 75%-write and 25%-read workload. The average latency of all the KV stores is more than twice that of GDH, with HyperLevelDB being 3.02 times higher. LUDA's performance sharply drops when the workload embraces mixed reads and writes, likely because the GPU potentially blocks read operations amid the compaction procedure. In contrast, Hyper-

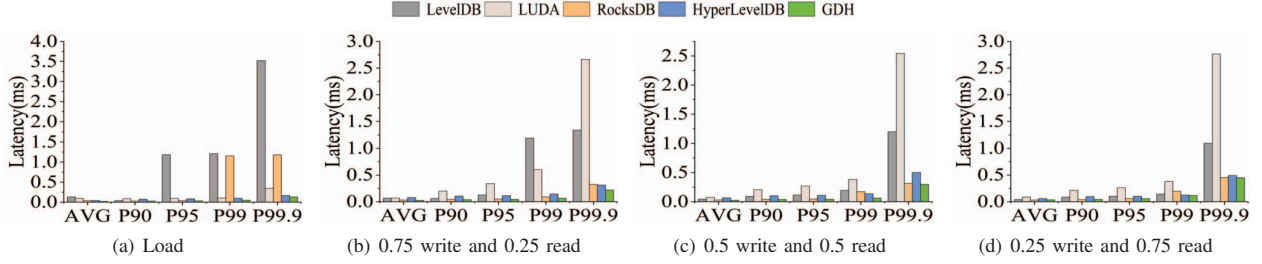


Fig. 9: Latency under workloads with various read-write ratios and a 1-KB KV pair.

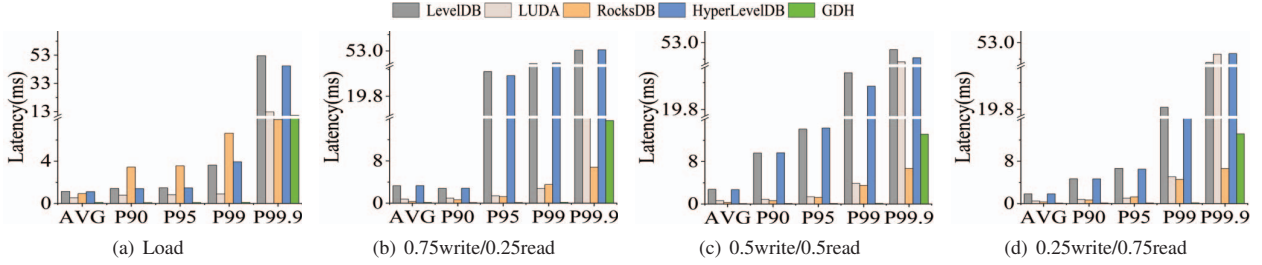


Fig. 10: Latency under workloads with various read-write ratios and a 32-KB KV pair.

LevelDB and RocksDB achieve low latency under the mixed read and write workloads. RocksDB's P99 latency becomes slightly worse, which is 2.81 times higher than GDH, and HyperLevelDB's P90 latency is 2.8 times higher than GDH.

Figs. 9(c) and 9(d) sketch the performance of KV stores under the workloads with 50% write-50% read and 25% write-75% read, respectively. As the read percentage jumps up, the average latency of all the KV stores, except for HyperLevelDB, surges as well. HyperLevelDB presents a downward trend, likely due to its read-and-write locking optimization, but HyperLevelDB is still not on par with GDH delivering good performance under workloads. Under the WPI workload, GDH optimizes the performance of HyperLevelDB, RocksDB, and LUDA by up to a factor of 2.31, 1.66, and 6.15, respectively.

In Figs 9(b), 9(c), and 9(d), the performance of HyperLevelDB has a similar tail latency to that of RocksDB. HyperLevelDB enjoys a performance edge over RocksDB under a mixed read-write workload. The P99.9 latency of GDH is almost the same as HyperLevelDB and RocksDB under high-read-ratio workloads. When GPU sorting operations are conducted, the performance of queries in GPU cache is slated to be time-consuming than that of disk-level queries.

When it comes to the mixed read-write workloads, the low latency of GDH is attributed to the combination of the memory and GPU components. The GPU component offers low read latency by storing frequently accessed data in the GPU cache. If a key being looked up in the GPU cache is found, the key can be returned without issuing any disk I/Os – resulting in low high latency. The hash table in memory furnishes an excellent write-read data structure under the mixed read-write workloads. Together, these components in GDH allow for efficient and low-latency data access under mixed workloads.

The average latency of GDH is largely lower than other KV stores under workloads with 32-KB KV pair. The advantage

of GDH decreases in terms of the P99.9 latency because GDH's read operations result in a few long-patch queries and GDH's read paths include GPU cache. In addition, GDH has the lowest latency. HyperLevelDB and LevelDB have higher average latency than other KV stores for all three read-write ratios. LUDA and RocksDB have stable average write latency. LUDA provides 8.92x higher than GDH (50% write, 50% read), and RocksDB has a maximum of 4.15x. Compared to alternative KV stores, the latency of GDH is the lowest at P99. LevelDB, LUDA, RocksDB, and HyperLevelDB achieve 279, 19.21, 24.22 and 283.19 times higher than GDH at 75% write and 25% read. In P99.9 latency, the RocksDB has the lowest latency. However, GDH's P99.9 performs better than LevelDB, LUDA, and HyperLevelDB.

VI. RELATED WORK

TRIAD [16] uses a frequency-based mechanism to identify hot data, where an extra four bytes are reserved to count the frequency of each key. The keys with the frequency higher than the average measure are identified as hotkeys, and when TRIAD updates the hot data, the hot data in the memtable is written back to the memtable rather than flushed to level 0. This technique ensures that hot data is stored in memory. HMCached [17], built on a hybrid storage system, adopts a statistical mechanism to identify hotspot data based on an object access counting mechanism. The identified hot objects are then migrated to DRAM. HMCached separates metadata and objects to avoid frequent accesses to non-volatile memory (NVM). FloDB [3] manages in-memory modules using a two-level structure: a small concurrent hash table at the upper level supports fast writes, whereas the large skiplist at the lower level undertake range queries.

MatrixKV [5] is utilized across multiple storage media, including DRAM, NVM-PMEM, and SSD. The first level, known as L_0 , is located in NVM-PEME, and compaction in

L_0 L_1 is performed at other levels to alleviate I/O competition. To avoid write stalls, NoveLSM [18] extends a component on NVMs to store writes when memory components become full. FD-tree [19] is constructed on SSDs with the purpose of minimizing random-write I/Os. As a result, read performance is enhanced in a cascading manner. To improve the performance of update and range scan KV stores, a hybrid index and a B+-tree index [20] were built in NVM and DRAM.

Numerous studies in recent years have sought to separate keys and values. For example, WiscKey [1] deploys a key-value separation mechanism to involve keys in compaction to reduce the compaction cost of values. WiscKey separates GC from key sorting to improve sorting performance. Meantime, GC can be performed either online or offline, mitigating its adverse impacts on system performance. Additionally, LSM-tree stores only keys in memory and aids in cache management, leading to a reduction in the number of accesses to the storage device. hash-based data grouping (HahKV) was implemented to hash keys into a fixed-size partition, and the ratio of valid values determines whether GC is provoked. HashKV distinguishes between hot and cold data stored in different areas; therefore, various GC strategies can be invoked to alleviate high write amplification. ALDC [21], designed to optimize both latency and throughput, incorporates a novel Lower-level Driven Compaction (LDC) method – an approach that deviates from the traditional upper-level driven compaction schemes and performs compaction actions bottom-up. The ALDC method minimizes compaction granularity for lower latency while lowering write amplification that benefits for high throughput.

VII. CONCLUSION

In this study, we adopted the gradient and hierarchy mechanism or GDH to optimize the components catering for cache data in KV stores. We devised an adaptive data migration method according to hot and cold data in the cache. We reformed and expanded the gradient cold-hot data hierarchy mechanism that replaces the in-memory data structure to address the problem of missing hot and cold data attributes. Our approach includes a cascaded loop Bloom filter that enables us to manage hot data with varying frequencies more effectively: the new design accurately classifies data as hot or cold based on how frequently data are accessed. To tackle the high computational expenses related to CPU sorting during the hash table flush to a disk, we successfully introduced GPU sort in our system. It is evident from the experiment results that our proposed GDH technique outperforms the other Key-Value stores under various workloads in terms of cache-hit ratio, average latency, and tail latency. Therefore, we conclude that GDH is a viable and efficient option for data storage and retrieval that requires high performance.

ACKNOWLEDGMENT

This work is supported in part by National Natural Science Foundation of China under Grants 61702004 and 62072001. Xiao Qin's work is supported by the U.S. National Science

Foundation under Grants IIS-1618669, OAC-1642133, CCF-0845257.

REFERENCES

- [1] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *USENIX Conference on File and Storage Technologies*, 2016, pp. 133–148.
- [2] Google, "LevelDB, levelDB is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values." <https://github.com/google/leveldb>, 2017.
- [3] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablatchi, "Flodb: Unlocking memory in persistent key-value stores," in *Twelfth European Conference on Computer Systems*, 2017, pp. 80–94.
- [4] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi, "Accordion: better memory organization for lsm key-value stores," *VLDB Endowment*, vol. 11, no. 12, pp. 1863–1875, 2018.
- [5] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *USENIX Annual Technical Conference*, 2020, pp. 17–31.
- [6] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage," in *Proc. USENIX Conference on File and Storage Technologies*, 2021, pp. 17–32.
- [7] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proc. ACM Symposium on Operating Systems Principles*, 2019, pp. 447–461.
- [8] Z. Duan, J. Yao, H. Liu, X. Liao, H. Jin, and Y. Zhang, "Revisiting log-structured merging for kv stores in hybrid memory systems," in *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023, pp. 674–687.
- [9] B. Atikoglu, Y. Xu, E. Frachtenberg, J. Song, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, 2012.
- [10] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. Huang, "Load balancing of heterogeneous workloads in memcached clusters," in *International Workshop on Feedback Computing*, 2014, pp. 1–8.
- [11] H. H. Chan, Y. Li, P. P. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in {KV} storage via hashing," in *2018 USENIX Annual Technical Conference (ATC18)*, 2018, pp. 1007–1019.
- [12] Nvidia, "Nvidia tesla p100, the world's first ai supercomputing data centre gpu," <https://www.nvidia.com/en-us/data-center/resources/pascal-architecture-whitepaper/>.
- [13] Facebook, "Rocksdb, a persistent key-value store for fast storage environments," <http://rocksdb.org/>, 2019.
- [14] P. Xu, J. Wan, P. Huang, X. Yang, C. Tang, F. Wu, and C. Xie, "LUDA: boost LSM key value store compactions with gpus," *CoRR*, vol. abs/2004.03054, 2020.
- [15] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Symposium on Operating Systems Principles*, 2017, pp. 497–514.
- [16] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "Triad: Creating synergies between memory, disk and log in log structured key-value stores," in *USENIX Annual Technical Conference*, 2017, p. 363375.
- [17] H. Jin, Z. Li, H. Liu, X. Liao, and Y. Zhang, "Hotspot-aware hybrid memory management for in-memory key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 779–792, 2019.
- [18] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *USENIX Annual Technical Conference*, 2018, pp. 993–1005.
- [19] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *VLDB Endowment*, vol. 3, no. 1-2, pp. 1195–1206, 2010.
- [20] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *USENIX Conference on Usenix Annual Technical Conference*, 2017, pp. 349 – 362.
- [21] Y. Chai, Y. Chai, X. Wang, H. Wei, and Y. Wang, "Adaptive lower-level driven compaction to optimize lsm-tree key-value stores," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 6, pp. 2595–2609, 2022.