SplitKV: Splitting IO Paths for Different Sized Key-Value Items with Advanced Storage Devices

Shukai Han, Dejun Jiang, Jin Xiong
SKL Computer Architecture, ICT, CAS, University of Chinese Academy of Science
{hanshukai, jiangdejun, xiongjin}@ict.ac.cn

Abstract

Modern advanced storage devices, such as modern NVMe SSD and non-volatile memory based persistent memory (PM), provide different access features when data size varies. Existing key-value stores adopt unified IO path for all key-value items, which cannot fully exploit the advantages of different advanced storage devices. In this paper, we propose to split IO paths for different sized key-value items. We let small key-value items be written directly to PM and then migrated to SSD. Meanwhile, large key-value items are directly written to SSD. We present and discuss design choices towards challenging issues of splitting IO paths. We build SplitKV, a key-value store prototype to show the benefits of splitting IO paths. The preliminary results show SplitKV outperforms RocksDB, KVell, and NoveLSM under small-large KV mixed workloads.

1 Introduction

Persistent Key-Value (KV) stores are widely deployed as a key component of storage infrastructure in today's data centers. Persistent KV stores usually serve interactive applications, such as web search [7, 20], e-commerce [9], and social networks [12]. Thus, KV stores are required to provide fast access of data for ensuring service quality for upper-level applications [10, 17, 26]. A few recent works have analyzed the real-world KV store workloads and reported that the size of key-value items varies from a couple of bytes to hundreds of kilobytes [4, 20, 33]. More importantly, small key-value items are dominant in real-world workloads [4]. For example, the average value sizes of three workloads in Facebook, including social graph, distributed KV store, and AI/ML services, are 126.7 B, 42.9 B, and 46.8 B.

Over the past decade, a number of research efforts are made to design and optimize block device based key-value stores [6, 23, 25, 27, 28, 30]. Conventional block devices, such as hard disk drives and low-end/medium-end solid state drives (SSDs), provide better performance when serving sequential access with large granularity, Log-Structured Merge Tree (LSM-Tree) structure [24] is widely adopted in existing KV stores [12, 14]. LSM-Tree converts random writes to sequential writes to make full use of maximum bandwidth.

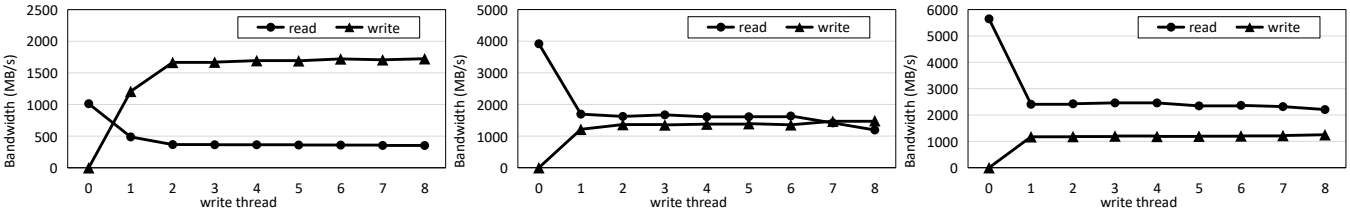
Recently, advanced storage devices exhibit different features compared to conventional block devices, which provides new design choices for building high-performance KV stores. On one hand, non-volatile memory based persistent memory, such as Intel DC persistent memory (Intel DC PM [1]), provides byte-addressable access and sub-microsecond latency for directly persisting data. A few recent works propose to adopt persistent memory to remove write-ahead log and reduce (de-)serialization overhead [18, 19]. On the other hand, modern NVMe SSD, such as Intel Optane SSD [3], sustains high throughput with sub-millisecond latency and provides similar performance for both sequential and random accesses. This motivates research effort to batch KV requests and write to SSD directly without commit log and sorting on disk [21].

In this paper, we take the insight that KV items with different sizes benefit differently from PM and modern fast SSD (e.g. Optane SSD). Table 1 shows the write latencies of PM and Optane SSD. Writing 64 B data to PM outperforms writing it to Optane SSD by 79.2x. However, the write latency gaps between PM and SSD decrease when the data size increases. For example, when writing 16 KB or above, the gap decreases under 3x. Existing KV stores adopt unified IO path for all KV items. However, this design choice cannot fully exploit the advantages of these advanced storage devices. Thus, we explore to split IO paths for different sized KV items. We let small KV items be written to PM directly and migrated to SSD later. As for large KV items, we directly write them to SSD.

Although providing different IO paths to different sized KV items sounds intuitive, applying it to build KV store faces several challenges. First, the size boundary to distinguish small and large KV items needs to carefully decide. Writing KV items directly on PM benefits from lower write latency than writing to SSD. However, these KV items require to be flushed to SSD later at the cost of competing PM read bandwidth as well as SSD write bandwidth. Thus, it is necessary

	Random Write Latency (us)									
Access Size	64 B	256 B	1 KB	4 KB	16 KB	64 KB	128 KB	1 MB	4 MB	16 MB
Optane SSD	14.09	14.09	14.09	14.09	21.44	45.79	145.58	532	2091	8223
Optane DC PM	0.18	0.20	0.43	1.05	3.90	15.50	61.88	247.25	1440	6840
Ratio	79.2	70.5	33.0	13.4	5.50	2.9	2.4	2.2	1.45	1.2

Table 1: **Random write latencies (us) of Optane DC Persistent Memory and Optane SSD.** *The performance gap between PM and SSD decreases when the access size increases.*



(a) 1 read thread. (b) 4 read threads. (c) 8 read threads.

Figure 1: **Trends of PM bandwidth.** *These figures show the trends of read and write bandwidth of Optane PM with changing read and write threads.*

to carefully decide the size boundary of small/large KV items to balance the benefit of writing PM and the migration cost.

Second, due to the PM space limitation, small KV items still need to be migrated to SSD. On one hand, accessing small KV items in SSD suffers from read/write amplification. Especially, it causes degraded performance when reclaiming invalid small KV items on SSD. Byte-addressable PM allows in-place-update for small KV items. Thus, one needs to carefully figure out cold KV items and migrate them to SSD to help reduce read/write amplification. On the other hand, KV migration affects the foreground query operations on both PM and SSD. It is necessary to design well-controlled migration to avoid negative impact on KV store performance.

Third, taking the querying process of a KV store into account, index searching and updating contribute a significant part to the accessing performance, especially when KV item accessing only needs sub-microsecond on PM. Thus, it is non-trivial to build a highly-efficient indexing for SplitKV for managing small KV items on PM and large ones on SSD.

In this paper, we build a key-value store prototype SplitKV to explore the advantages of splitting IO paths for different sized key-value items. We explore the design choices for the above first and second issues and discuss the third one. We show the preliminary results of SplitKV against conventional LSM-Tree based RocksDB, Optane SSD based KVell, and PM based NoveLSM. Under small-large mixed workloads, SplitKV improves throughput by up to 7.8x ,and reduce average write latency by up to 14.4x.

2 Background and Related Works

Existing KV stores make design choices to fully adapt to the access features of underlying storage devices.

LSM-Tree based KV store. Considering the performance of sequential access is faster than that of random access in earlier storage devices (e.g. hard disk drives and earlier solid state drives), Log Structured Merge Tree is widely adopted in KV stores to convert random writes to sequential writes [24]. LSM-Tree first writes KV items to in-memory buffer. When the buffer is full, the KV items are sorted and flushed to disk. This is beneficial to write-intensive workloads, which fully exploits the fast sequential access of earlier storage devices. However, LSM-Tree based KV store suffers from costly data compaction and slow data read. Recently, a number of research efforts are made to optimize LSM-Tree based KV stores [6, 13, 23, 25, 27, 30].

Modern NVMe SSD based KV store. Compared to earlier storage device, modern NVMe SSD (e.g. Intel Optane SSD) provides higher bandwidth and lower access latency. Especially, modern NVMe SSD provides similar performance for random and sequential accesses. Conventional LSM-Tree based KV stores show CPU bottleneck instead of storage device bottleneck when running on Optane SSD [3]. Thus, KVell [21] proposes several design choices to reduce operations incurring CPU overhead. For example, KVell adopts exclusive data structures to reduce conflicts between threads, and batches requests to reduce the number of syscalls. Moreover, KVell does not sort data to avoid costly CPU operations.

Persistent Memory based KV store. Non-Volatile Memories (NVMs), such as 3D XPoint [2], Phase Change Memory (PCM) [29], and Resistive Memory [5], provide low latency and byte-addressable access. Thus, a number of persistent KV stores are proposed to build on NVM-based persistent memory. Some works [15, 31, 32, 34] focus on optimizing indexing in KV stores. Meanwhile, a few works instead explore applying PM to reduce the overheads of logging and

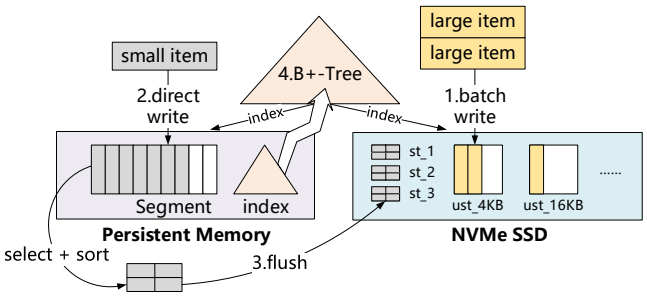


Figure 2: **SplitKV overview.** *(1) Large items are batch flushed to unsorted tables (ust) in SSD. (2) Small items are directly written to segments in PM. (3) Small items with low weights are migrated to SSD to generate sorted tables (st). (4) A global B^+-Tree index is used to index KV items in both PM and SSD. The B^+-Tree index is persisted in PM.*

(de-)serialization in LSM-Tree based KV stores [11, 18, 19].

Motivation Although making design choices to adapt to different features of different storage devices, existing KV store design a unified IO path for handling all KV items. As reported in [4], the size ranges of real-world KV workloads are from a couple of bytes to hundreds of kilobytes. As shown in Table 1, PM is friendly to serve small data (e.g. less than 1 KB) and meanwhile this advantage is weakened when data size increases. On the contrary, modern NVMe SSD suffers from read/write amplification when serving small data (e.g. less than 4 KB). This disadvantage does not exist when data is multiple block size[1] Moreover, modern NVMe SSD behaves similarly when serving sequential and random accesses as shown in Table 1. Thus, we are motivated to design different IO paths for different sized KV items in this paper.

3 SplitKV

In this section, we present the design of SplitKV. Figure 2 shows the system overview of SplitKV. SplitKV adopts different IO paths for small/large KV items. As for small KV items, SplitKV first stores them in PM. To reclaim PM space, SplitKV selects part of KV items and migrate them to SSD. Small KV items are stored in the form of sorted tables in SSD similar to LevelDB/RocksDB. Instead, SplitKV applies similar techniques as in KVell [21] to directly write large KV items to SSD without sorting them. Similar to SLM-DB [18], SplitKV adopts a global B^+Tree index to locate KV pairs in both PM and SSD. The global B^+Tree index is persisted in persistent memory. We guarantee its persistency and crash consistency as existed persistent B^+Tree index [15] do.

[1]Here the size of a block is 4 KB, which is the default block granularity for most block devices.

3.1 Size Boundary of KV Items

Unlike byte-addressable persistent memory, modern NVMe SSD provides block interface which usually requires data access with 4 KB alignment. Thus, accessing KV items that are less than 4 KB in SSD suffers from read/write amplification. Moreover, as shown in Table 1, the latencies of writing data less than 4 KB in PM are greatly improved by up to 79.2x compared to writing them in SSD. However, directly writing data with the sizes of 4 KB, 16 KB, and 64 KB in PM also achieves at least 2.9x faster performance compared to writing to SSD as in Table 1. In order to decide the size boundary of small/large KV items, we also take the migration cost into account. KV items still need to be migrated from PM to SSD due to the limited PM space. The migration introduces extra competition for both PM read bandwidth and SSD write bandwidth.

Access Size	256 B	1 KB	4 KB	16 KB
PM + SSD	1.48	4.47	15.70	27.57
SSD	23.35	25.39	14.79	21.33
Ratio	15.82	5.68	0.94	0.77

Table 2: **Average write latencies (us) of writing PM and writing SSD.** ***PM + SSD*** *indicates first writing KV items in PM and then migrating them to SSD when 80% of PM is filled.* ***SSD*** *indicates directly writing KV items to SSD.*

Table 2 shows the average write latencies when adopting different IO paths for different sized KV items. The ratios indicate the write latency benefit of writing KVs in PM first and then migrating to SSD. As shown in Table 2, the benefit starts to decrease when a KV items is equal to or greater than 4 KB. This is because the background migration competes for more PM bandwidth for migrating larger KV items.

Thus, SplitKV sets the size boundary of a small key-value pair to be 4 KB. Any KV pair whose size is equal to or greater than 4 KB is considered to be large one. The IO path for small KV items is first writing in PM and then migrating to SSD. This not only accelerates accessing small KV items, but also provides the opportunity of applying in-place-update in PM. A KV items is directly updated in case of existing in PM. Otherwise, the updated one is written in PM as log-structuring does. Moreover, small KV items can be sorted and clustered into 4 KB blocks when migrated to SSD. When serving scan queries, SplitKV is able to read these sorted small KV pairs with a couple of 4 KB blocks and thus reduces read amplification. We show the improved scan performance of SplitKV in Section 5.

3.2 Direct Writing Small KV Items

Batch writing is widely used in block devices by increasing IO queue depth. Batch writing helps to increase KV store throughput. However, when applying batch writing to PM,

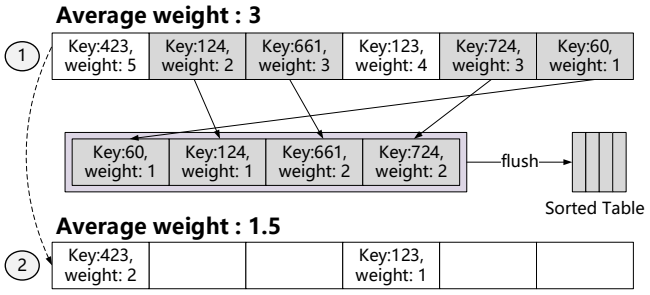


Figure 3: **A simple example of migrating KV items from PM to SSD.**

one requires to buffer KV items first in DRAM, and then flush them into PM in a batch. As reported in some works [16, 22], the write latency of PM is close to that of DRAM. Instead of bringing throughput benefit, batch writing to PM causes additional memory copy overhead. Moreover, batching data in DRAM before flushing them to SSD takes the risk of data lost in case of power failure. In order to provide data reliability, extra efforts are required, such as data logging. Thus, SplitKV directly writes a KV item into PM with IO queue depth of 1.

3.3 Hotness-aware KV Migration

Due to the limited space of PM, KV items in PM need to be migrated to underlying SSD for reclaiming PM space. As SplitKV applies appending write to small KV items in SSD, any update to an in-SSD small KV brings expensive garbage collection. Thus, SplitKV adopts hotness-aware KV migration and figures out KV items that are less frequently accessed for migration. The hotness-aware KV migration actually works like a cache, and existing or new cache policies can be explored to filter cold KV items. In this paper, we apply a simple policy for demonstration purpose. We set a weight for each KV item by using its access count. SplitKV runs a background thread to periodically calculate the average weight of all KV items in PM. When the PM capacity usage goes beyond a certain threshold (e.g. 80%), SplitKV selects the KV items whose weights are lower than the average one. These selected KV items are then migrated to SSD. For a KV item whose weight is greater than the average one, SplitKV instead updates its weight by subtracting the average one. Figure 3 shows an example. There exist 6 KV items with different weights in PM, and the average weight is 3. SplitKV selects the items with the keys of 124, 661, 724 and 60 as the cold ones as their weights are less than 3. These four KV items are flushed to SSD. The weights of the KV items with keys 423 and 123 are updated by being subtracted by 3.

The background KV migration from PM to SSD introduces extra reads to PM. In order to reduce its impact on foreground queries, one needs to control the bandwidth competition by the background migration. However, slow migration in turn results in slowed PM space reclamation. Figure 1 shows the read bandwidth changes of Optane DC PM as write threads increase. We can see that about 2 write threads can saturate the write bandwidth. Meanwhile, the write bandwidth can still be sustained around 1.2 GB/s even co-running 8 read threads. However, when the number of write threads increases, the read bandwidth is first reduced and then can also sustain stable. Moreover, the read bandwidth increases with increased read threads. Thus, we by default adopt 1 background thread to execute KV migration. On one hand, this has limited impact on PM writes. On the other hand, one still can increase the background threads in case of urgent PM space reclaiming.

4 KV Operations

Put When serving KV item insertion, SplitKV adopts different IO paths for different sized KV items. As for large KV items, SplitKV batches items in a queue, and then submit the whole queue. Otherwise, SplitKV directly writes small KV items in PM. After successfully writing KV item(s) in either PM or SSD, SplitKV creates the index entry(ies) in B^+-Tree and returns success to upper-level applications.

Update When updating a large KV item, SplitKV applies in-place-update by executing read-modify-write to the target KV item. In case of updating a small KV item, SplitKV applies in-place-update to the target KV item if it is located in PM. Otherwise, the KV item is already migrated to SSD, and SplitKV adopts out-of-place update by directly writing the updated one in PM. Then, SplitKV updates the location of the KV item in the global B^+-Tree index. The KV item in SSD is marked as invalid and then recycled. We leave the garbage collection process as the future work.

Get/Scan To search a KV item or a range of KV items, SplitKV first searches the B^+-Tree index to locate the item(s). Then, SplitKV fetches the item(s) according to its location(s) from either PM or SSD.

5 Evaluation

5.1 Experiment Setup

System and hardware configuration. We conduct all experiments on a server equipped with two Intel Xeon Gold 5215 CPU (2.5GHZ), 64GB memory, one Intel Optane SSD P4800 and one Intel Optane DC PM. We run CentOS Linux release 7.6.1810 with 4.18.8 kernel.

Compared systems. We compare SplitKV against RocksDB [12], KVell [21] and NoveLSM [19]. We set MemTable size of RocksDB to 64MB. Note that, RocksDB does not use PM and adopts asynchronous write ahead log. KVell does not sort data on disk, and uses non-shared data structures and batches I/O requests to reduce CPU overhead. NoveLSM uses PM to reduce (de-)serialization cost and is implemented on LevelDB. We set up 8 GB persistent

KV Store	Uniform						Zipfan					
	A	B	C	D	E	F	A	B	C	D	E	F
NoveLSM	96.69	69.77	61.04	64.56	476.19	145.14	48.35	34.89	30.52	32.28	445.83	72.57
RocksDB	21.11	26.13	26.08	25.89	529.10	43.27	17.47	21.82	21.72	21.13	497.02	35.19
KVell	17.86	14.02	13.31	13.80	670.69	23.09	11.76	8.60	8.64	9.20	609.38	14.12
SplitKV	8.81	12.78	12.77	9.22	346.02	13.87	3.81	4.65	4.56	4.56	306.65	5.05

Table 3: **Average latencies of different KV stores.** *This table shows the average latency (us) comparison of different KV stores with single thread under YCSB mixed workloads.*

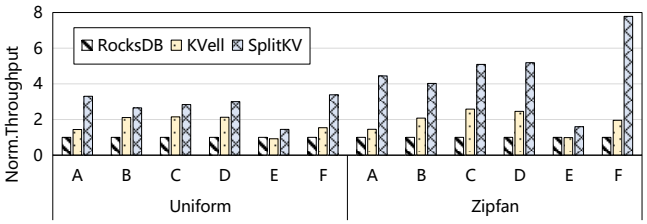


Figure 4: **Normalized throughputs of different KV stores.** *The figure shows the throughput of different KV stores with four threads under YCSB mixed workloads.*

MemTable and 64 MB in-DRAM MemTable for NoveLSM. For SplitKV, we set 8 GB PM for it.

Workloads. We use the 6 default workloads in YCSB [8] for evaluation. Workload A performs 50% reads and 50% updates; Workload B performs 95% reads and 5% updates; Workload C performs 100% reads; Workload D performs 95%reads for latest keys and 5% inserts; Workload E performs 95% range queries and 5% inserts; Workload F performs 50% reads and 50% read-modify-writes. We evaluate both uniform and zipfian distributions. All workloads include small KV items with 256 B value and large KV items with 4 KB value. All keys have a length of 8 B. We first warm up a 200 GB database and then run workloads A, B, C, D, E in turn. Each workload handles 128 GB data set, in which small/large KV items account for half respectively. We execute 5 runs for each experiment and use average results.

5.2 Evaluation Results

Table 3 shows latency results of different KV stores using single thread, and meanwhile Figure 4 shows the throughput results normalized to RocksDB using four threads[2]. For workloads A and F, SplitKV reduces latency by 14.4x, 6.9x, and 3.1x compared to NoveLSM, RocksDB and KVell under zipfan workloads. As for throughput, SplitKV outperforms RocksDB and KVell by 5.8x and 1.6x respectively. LSM-Tree based NoveLSM and RocksDB perform compaction to ensure ordered data and recycle invalid data. This causes high write amplification and reduces performance. KVell applies in-place-update for all KV items to avoid sorting overhead to achieve better performance. However, KVell suffers from read/write amplifications for small KV items. Instead, SplitKV writes small KV items in PM to accelerate KV accessing. Moreover, SplitKV exploits PM to conduct in-place-update for small KV items to reduce read/write amplifications. Thus, SplitKV further outperforms KVell.

For read-intensive workloads B, C and D, SplitKV and KVell achieved better performance than NoveLSM and RocksDB due to the adoption of the global B^+-Tree index. SplitKV is able to fetch small KV items from PM instead of SSD. Thus, SplitKV improves both latency and throughput compared to KVell.

For workload E, SplitKV outperforms RocksDB and KVell by 1.6x and 1.7x respectively under zipfan workloads in terms of throughput. Meanwhile, SplitKV achieves lowest latency among all KV stores. KVell does not sort small KV items in SSD. This introduces read amplification to KVell when serving scan query by reading a plenty of blocks. Thus, KVell suffers from the highest latency. NoveLSM and RocksDB performs better than KVell as they sort all KV items in SSD. SplitKV does not sort large KV items in SSD which reduces CPU overhead. Meanwhile, SplitKV flushes sorted small KV items into SSD, which helps to accelerate scan performance. Therefore, SplitKV further outperforms NoveLSM and RocksDB.

Note that, compared to the results under zipfan workloads, the latency and throughput improvements of SplitKV decrease under uniform workload. This is because the current hotness-aware migration policy is difficult to figure out cold items under uniform workloads.

6 Conclusion

Modern NVMe SSD and persistent memory provide different access features when serving small/large data. In this paper, we propose SplitKV to provide different IO paths for different sized KV items for building KV stores with such advanced storage devices. SplitKV let small KV items be written in PM directly and then selectively migrated to SSD. As for large KV items, SplitKV directly writes them to SSD. Our preliminary evaluations show SplitKV outperforms state-of-the-art KV stores in terms of both throughput and latency.

[2]Note that NoveLSM only supports single thread.

7 Discussion Topics

Exploration of advanced storage devices in KV stores. Earlier storage devices prefer sequential access with large granularity than small random writes. This results in the widely adopted design choice of LSM-Tree in existing KV stores. However, LSM-Tree suffers from expensive data compaction and slow reads. A number of research efforts propose optimizations based on LSM-Tree [6, 13, 23, 25, 27]. With the development of advanced storage devices, such as Intel Optane SSD and Intel Optane DC PM, it is desirable to fully discuss the impact of these devices on KV stores as well as explore their usages on designing KV stores. A few recent works, such as KVell [21] and NoveLSM [19]/SLM-DB [18], have explored the design choices of KV stores towards Optane SSD and Optane PM respectively.

We argue splitting IO paths for different sized KV items to match different features of advanced storage devices. Although the preliminary results show SplitKV outperforms existing KV stores, several interesting topics deserve further discussion.

(1) Handling small items. As for large KV items, KVell proposes an effective approach to handle with modern NVMe SSDs. However, small KV items cannot benefit much from such design choices. SplitKV writes small KV items first in PM to shorten its write path, but the organization and update of small KV items in SSD are still open issues. LSM-Tree may not be a well-suited choice for modern NVMe SSDs. An alternative approach is keep small items in PM as much as possible. SplitKV explores filtering KV items before they are migrated to SSD with a simple policy. This policy can be further studied to distinguish different types of KV items according to features of PM and SSD. In such doing, PM friendly KV items can be figured out and kept in PM to accelerate KV access. Meanwhile, this helps to reduce IO pressures to SSD.

(2) Efficient indexing. Table 1 shows that PM improves write latency by almost 2 orders of magnitude compared to SSD with small granularity. However, we observe in the system level, the small-item write latency of SplitKV is only 10x faster than that of KVell which directly writes KVs to SSD (as shown in Table 2). This is because the indexing contributes more overhead when accessing KV items with underlying PM. One explorable direction is to design hybrid index for data in PM and SSD. For example, a global index is adopted to locate data in both PM and SSD to provide fast read, and meanwhile a PM friendly and highly efficient index is designed to search items in PM. We leave this for our future work.

(3) Flow control of PM. Migrating KV items from PM to SSD competes for PM bandwidth when reading data and meanwhile competes for SSD bandwidth when writing data. A carefully designed flow control or request scheduling is still an open issue.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. We thank Ying Wang and Huan Zhang for useful discussions. This work is supported by National Key Research and Development Program of China under grant No.2018YFB1003303, Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDB44030200.

References

[1] Big memory breakthrough for your biggest data challenges. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[2] Intel and micron produce breakthrough memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[3] Intel optane ssd 9 series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series.html.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale, editors, *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, June 11-15, 2012*, pages 53–64. ACM, 2012.

[5] I. G. Baek, M. S. Lee, S. Seo, M. J. Lee, D. H. Seo, D. . Suh, J. C. Park, S. O. Park, H. S. Kim, I. K. Yoo, U. . Chung, and J. T. Moon. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004.*, pages 587–590, Dec 2004.

[6] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019. USENIX Association, 2018.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154. ACM, 2010.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[10] Facebook. Myrocks. http://myrocks.io/.

[11] Facebook. Nvmrocks. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/.

[12] Facebook. Rocksdb. http://rocksdb.org/.

[13] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, April 21-24, 2015*, pages 32:1–32:14, 2015.

[14] Google. Leveldb. https://github.com/google/leveldb.

[15] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200. USENIX Association, 2018.

[16] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[17] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315. USENIX Association, 2015.

[18] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205. USENIX Association, 2019.

[19] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005. USENIX Association, 2018.

[20] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, May 2015.

[21] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, October 27-30, 2019*, pages 447–461. ACM, 2019.

[22] Jihang Liu and Shimin Chen. Initial experience with 3d xpoint main memory. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, April 8-12, 2019*, pages 300–305, 2019.

[23] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1):5:1–5:28, March 2017.

[24] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[25] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 497–514. ACM, 2017.

[26] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156. USENIX, 2013.

[27] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228. ACM, 2012.

[28] WiredTiger.2019. Wiredtiger caching and eviction. http://source.wiredtiger.com/.

[29] H. . P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[30] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82. USENIX Association, 2015.

[31] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362. USENIX Association, 2017.

[32] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181. USENIX Association, 2015.

[33] zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223. USENIX Association, February 2020.

[34] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476. USENIX Association, 2018.