PTierDB: Building Better Read-Write Cost Balanced Key-Value Stores for Small Data on SSD

Li Liu[†] and Ke Zhou[†‡*]
[†]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
[‡]Key Laboratory of Information Storage System, Ministry of Education of China, Wuhan, China
lillian_hust@hust.edu.cn, zhke@hust.edu.cn

***Abstract*—The popular Log-Structured Merge (LSM) tree based Key-Value (KV) stores make trade-offs between write cost and read cost via different merge policies, i.e., leveling and tiering. It has been widely documented that leveling severely hampers write throughput, while tiering hampers read throughput. The characteristics of modern workloads are seriously challenging LSM-tree based KV stores for high performance and high scalability on SSDs. In this work, we present PTierDB, an LSM-tree based KV store that strikes the better balance between read cost and write cost for small data on SSD via an adaptive tiering principle and three merge policies in the LSM-tree, leveraging both the sequential and random performance characteristics of SSDs. Adaptive tiering introduces two merge principles: prefix-based data split which bounds the lookup cost and coexisted merge and move which reduces data merging. Based on adaptive tiering, three merge policies make decisions to merge-sort or move data during the merging processes for different levels. We demonstrate the advantages of PTierDB with both microbenchmarks and YCSB workloads. Experimental results show that, compared with state-of-the-art KV stores and the KV implementations with popular merge policies, PTierDB achieves a better balance between read cost and write cost, and yields up to 2.5x improvement in the performance and 50% reduction of write amplification.**

I. Introduction

Persistent key-value (KV) stores play a crucial role in modern data-intensive applications, such as messaging [1], e-commerce [2], search indexing [3], [4] and advertising [5], [6]. A large variety of modern large-scale KV stores are built on Log-Structured Merge tree (LSM-tree) [7] to benefit from the design of sequential writes, including LevelDB [3], RocksDB [5], Cassandra [8] and bLSM [9], etc.

LSM-tree is an optimized index structure for write-intensive workloads on hard-disk drives (HDDs). As the storage landscape changes, solid-state drives (SSDs) are gradually supplanting HDDs in many use cases to boost the throughput of KV stores. However, there are several characteristics of LSM-tree and modern workloads that are seriously challenging LSM-tree based KV stores for high performance and high scalability on SSDs.

First, LSM-tree is structured into multiple levels (L_0, L_1,..., L_{k-1}) which is the root of write amplification and read amplification [10]. Especially, as the LSM-tree grows in size, it necessitates merge operations to reclaim invalid space and enable efficient lookups, resulting in large amounts of write I/O under write-intensive workloads [3], [5]. The high write amplification can significantly degrade the write performance and reduce the lifetime of SSD. Second, the size of KV items is closely related to performance and storage space efficiency. Recent studies show that the KV items are small [11], [12]. Specifically, in typical persistent KV-stores use cases which store social graph data, the metadata of objects in storage and the profile data used for AI/ML services, the average size of KV items is less than 200 bytes [12]. Third, read dominates the KV operations in some real-world workloads, such as social graph data and the metadata of objects in storage [12].

A large body of research efforts have been conducted to decrease write amplification [10], [13]–[17] from various perspectives and utilize the characteristics of SSD to improve write performance [18]–[23]. However, most of them pursue the improvement of write amplification and write performance without affecting read performance [14]–[16], [19], [22], and some of them work well for large KV items [10], [17].

In this paper, we present PTierDB, an LSM-tree based key-value store that strikes the better balance between read cost and write cost for small data on SSD leveraging the random I/O characteristic of SSDs. To achieve this, PTierDB designs a novel merge principle called adaptive tiering, including two techniques: prefix-based data split and coexisted merge and move. Based on adaptive tiering, three merge policies are applied to the LSM-tree. First, merge and split data from L_0 to L_1 so as to bound the key range of the data files, which can bound the lookup cost. Second, merge and move data from L_i to L_{i+1} ($0 < i < k-1$), which can further reduce data rewrites. Third, trigger an intra-level merge in L_{k-1} to enable efficient lookups at the largest level.

We implement PTierDB upon LevelDB (Version 1.22) [3]. We extensively compare the performance of PTierDB with three KV implementations based-on three popular merge policies (i.e., leveling, tiering and lazy leveling [24]) and three state-of-the-art KV stores, including PebblesDB [25], SILK [26] and Wisckey [10] (*vLog* implementation in the prototype of HashKV [17]). Experimental results show that with microbenchmarks, PTierDB achieves a better balance between write cost and read cost than the other six KV stores. The write throughput is close to tiering while the read throughput is 1.5x-2x faster than the other six KV stores. Under YCSB macrobenchmarks, PTierDB outperforms all other KV stores for small data on SSD. The average latency of leveling is better than that of tiering. Compared with the KV stores with leveling, the average latency of PTierDB is reduced by 1.2x-

978-3-9819263-5-4/DATE21/©2021 EDAA

2.5x, and the write amplification is reduced by 30%-50%.

II. Background and Motivation

This section gives the necessary background on LSM-tree based key-value stores and the popular merge policies.

a) Log-structured Merge Tree: An LSM-tree is a persistent structure that provides good performance by transforming small random writes into sequential writes in sorted order. An LSM-tree is composed of two or more tree-like components (C_0 to C_k) of exponentially increasing sizes. C_0 is memory-resident and update-in-place, while others are disk-resident and append-only. When C_0 fills up, LSM-tree merges data in C_0 to the nearest disk component (C_1). The merge operation arouses data rewrites to maintain the data in sorted order in disk components so as to (1) bound the number of disk accesses of a lookup, and to (2) remove obsolete data to reclaim space. The multiple disk components are used to amortize the overhead of merge operations.

LevelDB [3] is a representatively LSM-tree based KV store. In LevelDB, the disk components are structured in multiple levels, denoted by L_0, L_1,. . ., L_{k-1} (e.g., 7 levels by default). The capacities of levels are exponentially increased, and the size ratio of the capacities between adjacent levels is configured as 10 by default. Each level contains a number of data files each of which stores KV items in sorted order, called Sorted String Table (SSTable).The size of SSTables usually sets as a few MBs (e.g., 2 MB by default). SSTables at the same level have disjoint key ranges, besides that at L_0 since L_0 is a buffer level for the memory component. Thus, each lookup only has to probe at most once at each level (besides L_0). When the memory component fills up, it is flushed to L_0. When the number of SSTables at L_0 exceeds its limit, it triggers a background merge operation called compaction to merge-sort the overlapped SSTables at L_0 and L_1, and generate new SSTables into L_1. Once the total size of L_i ($0 < i < k-1$) exceeds its limit, the compaction thread chooses one SSTable from L_i, merge-sorts the target with all the overlapped SSTables at L_{i+1}, and generates new SSTables into L_{i+1}.

b) Merge Policies: The merge policy controls the number of times a KV item gets merged across levels. Most of LSM-tree based KV stores use either one of two merge policies: tiering [27] or leveling [7]. The main difference between the two merge policies is that leveling organizes data in sorted order at each level to achieve efficient lookups while tiering allows key range overlapping at each level to reduce write amplification and achieve good write performance. Thus, leveling has strictly better read cost and strictly worse write cost than tiering.

We compare the two policies in Fig. 1. When L_0 reaches its capacity, with tiering, we merge-sort all the data at L_0 and write the merged data into L_1 ; with leveling, we merge-sort all the data in the overlapping key range at L_0 and L_1 and write the merged data into L_1. There is no size limit of the data files. Thus, with tiering, there are several large SSTables at each level (the number depends on the size ratio),

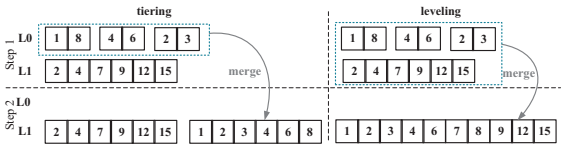

Fig. 1. Before and after a merge operation with tiering and leveling.

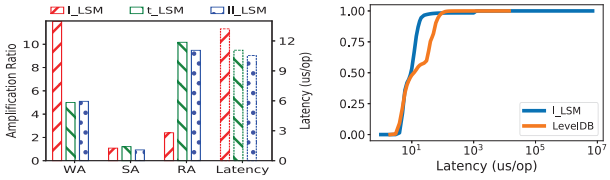

Fig. 2. Experimental results of the KV implementations with leveling, tiering, lazy leveling and fragmented leveling respectively.

while with leveling, there is one large SSTable at each level besides L_0. Large files are conducive to the endurance of SSDs. Because even with the parallel I/O characteristic of SSDs, larger files can fill the entire block. After deleting a file, SSD will directly erase these blocks without invoking garbage collection, thereby reducing the write amplification in SSD and mitigating the impact of garbage collection in SSD on the write performance of KV stores. In addition, using the sequential write performance of SSDs can reduce the overhead of merging large files.

Compaction in LevelDB is considered as fragmented leveling [24], which can smooth out performance slumps due to long merge operations at larger levels. Notably, most point lookup I/Os target the largest level. Thus, lazy leveling is presented to reduce lookup cost by using leveling for the largest level and using tiering for others [24].

To quantitatively understand the impacts of merge policies on LSM-tree based KV stores, we have implemented leveling, tiering and lazy leveling upon LevelDB (Version 1.22) respectively, called *leveled LSM* (**l_LSM**), *tiered LSM* (**t_LSM**) and *lazyleveled LSM* (**ll_LSM**). We use single-threaded YCSB benchmarks to evaluate LevelDB and the three implementations on 100 M random writes and 100 M YCSB A (50% updates and 50% reads). The size of KV items is of 256 B. To reveal the differences between tiering and lazy leveling, we set the size ratio of capacities between adjacent levels as 6 to enable data arriving at the largest level. Fig. 2 illustrates the results of the ratio of write amplification (**WA**), space amplification (**SA**), read amplification (**RA**) and latency of the three implementations (left) as well as the difference between leveling and fragmented leveling in cumulative distribution function (CDF) figure (right).

As shown in Fig. 2 (left), leveling has strictly worse write amplification and strictly better read amplification than tiering. Lazy leveling has slightly worse write amplification and slightly better read amplification than tiering. Because write dominates the KV operations in this evaluation, the latency of leveling is the worst. The CDF figure shows that the latency of almost half of the operations in leveled LSM is lower than

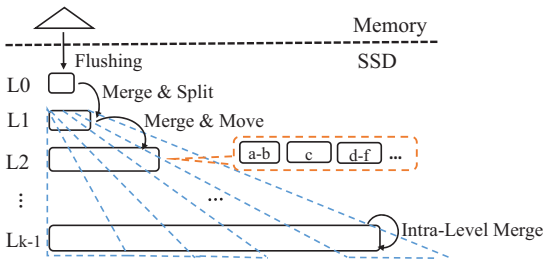

Fig. 3. The overall architectural view of PTierDB.

LevelDB, but that in leveled LSM has a long tail distribution which means part of operations have very bad latency.

III. PTierDB

This section introduces the design of PTierDB. The overarching goal is to achieve the better balance between read cost and write cost for small data on SSD. In the following, we describe the design details.

A. Overall Design

Fig. 3 demonstrates the overall architectural view of P-TierDB. PTierDB is designed atop tiered LSM, allowing key ranges between SSTables to overlap. PTierDB presents an adaptive tiering principle to split the merged data based on the prefix of keys and provide a merge and move coexisted principle to further reduce data rewrites. Data is divided into a few SSTables at each level according to the prefix of the keys (as shown by the yellow dotted box). Therefore, the data organization on the disk can be considered as a prefix-based data grouping (as shown by the blue dotted line). Based on adaptive tiering, PTierDB uses three merge policies for different levels, respectively. When the number of SSTables at L_0 exceeds its limit, merge and split all the SSTables from L_0 to L_1. When the total size of L_i $(0 < i < k-1)$ exceeds its limit, merge and move all the SSTables from L_i to L_{i+1}. When the number of accesses to an SSTable at L_{k-1} exceeds its limit, merge all the overlapped SSTables at L_{k-1} with the target and generate a new SSTable at L_{k-1}.

B. Adaptive Tiering

Adaptive tiering is a merge principle designed based on tiering. Adaptive tiering inherits the advantages on write performance and SSD endurance of tiering, and further limits read cost and reduces data rewrites via two principles of prefix-based data split and coexisted merge and move. The detrimental effect of tiering on read performance also becomes a legacy issue. Adaptive tiering attempts to reduce read cost through the random I/O characteristic of SSDs and the prefix-based data split principle.

1) Prefix-based Data Split: We provide a prefix-based data split principle to divide the merge data into SSTables according to the prefix of keys during a merge operation in order to bound the key range of SSTables. We set a threshold size T_s for SSTables (e.g., 4 MB) to prevent generating many small files so as to mitigate the fragmentation and garbage collection overhead of SSD. When writing the merged KV items into SSTables, we acquire the prefix of each KV item. If the prefix of current KV item is different from that of the predecessor, there are possible three situations according to the size of current SSTable C_s, as follows:

- $C_s < T_s$, keep appending the KV item;
- $C_s >= T_s$, if the size of the remaining KV items is less than T_s,
  - ✓ Keep appending the KV item;
  - × Finish current SSTable and create a new SSTable.

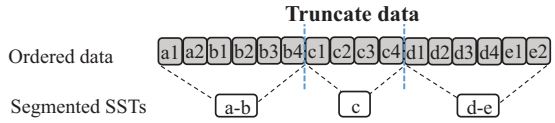

Fig. 4. A view of the data split principle based on the prefix of keys.

Fig. 4 gives an example of how to truncate the ordered data into segmented SSTables according to the data split principle.

2) Coexisted Merge and Move: The prefix-based data split principle bounds the key range of SSTables which can reduce the probability of overlapped SSTables, based on that, we design a coexisted merge and move principle to further reduce data rewrites.

Suppose there are N overlapped SSTables, including M multi-prefix SSTables and S single-prefix SSTables ($M+S=N$). We provide several rules to help make merge or move decisions among SSTables, as follows:

- $N = 1$, no overlap, move the SSTable;
- $N = 2$,
  - * $M = 2$ or $S = 2$, merge the two SSTables;
  - * $M = 1$ and $S = 1$, move the two SSTables;
- $N >= 3$,
  - * If a prefix appears three times or more, merge the SSTables containing this key prefix;
  - * If a prefix appears twice and one is a single-prefix SSTable, move the single-prefix SSTable.

We give an example of how the coexisted merge and move principle guides the merge process in the next section as shown in Fig. 5.

C. Three Merge Policies

Based on adaptive tiering, three merge policies are applied to different levels in LSM-tree, called merge and split, merge and move and intra-level merge, respectively. To better understand the three merge policies, Fig. 5 demonstrates the view of merge states for each merge policy.

In the LSM-tree, the SSTables at each level are sorted in order according to their smallest key. When the number of SSTables in L_0 exceeds its limit, it triggers a merge and split operation which chooses the overlapped SSTables at L_0, merge-sorts the data in these SSTables and splits the merged

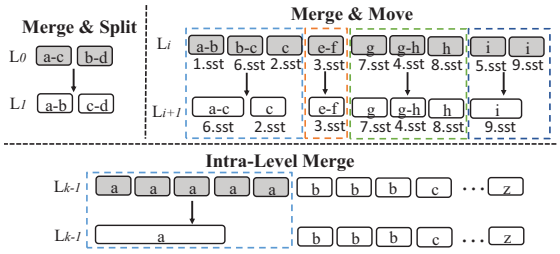

Fig. 5. Before and after a merge operation with the three merge policies at different levels.

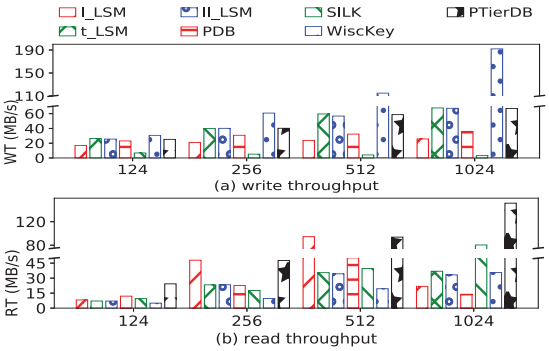

Fig. 6. Average throughput of seven KV stores under various KV item sizes.

data into new SSTables at L_1 according to the prefix-based data split principle.

As the LSM-tree grows in size, there accumulates large volume of data at each level (besides L_0), which is organized in a number of SSTables. When the total size of L_i $(0 < i < k-1)$ exceeds its limit, it triggers merge and move operation which iterates over all the SSTables at L_i and makes decisions of merge or move for each SSTable according to the coexisted merge and move principle. Some of the SSTables are merged into new SSTables at L_{i+1} while others are directed moved to L_{i+1}, as shown in Fig. 5. Notably, when merging SSTables into the next level, we should use the newest file number of the merging SSTables as the file number of the new SSTable to keep that SSTables at larger levels are always numbered older than lower levels so as to guarantee the data consistency.

At the largest level (L_{k-1}), there will exist lots of overlapped SSTables resulted from the merge and move operation. Since most lookup I/Os target L_{k-1}, it necessitates efficient lookups at L_{k-1} to achieve better balance between read cost and write cost. Thus, we provide intra-level merge at L_{k-1} which trades write amplification for read cost. When the number of accesses to an SSTable at L_{k-1} exceeds its limit, it triggers intra-level merge which chooses the SSTables overlapped with the target, merges the SSTables and generates a new SSTable at L_{k-1}.

IV. Evaluation

We implement PTierDB on top of LevelDB [3]. We conduct a set of experiments to evaluate PTierDB and compare it against the KV implementations with widely-used merge policies and state-of-the-art LSM-tree based and KV separated stores, including *leveled LSM*, *tiered LSM*, *lazyleveled LSM*, *PebblesDB* [25], *SILK* [26] and *Wisckey* [10], [17].

A. Experimental Setup

The evaluation is performed on a machine running 64-bit Ubuntu 20.04 with the Linux 5.4.0 kernel, 480 GB SSD Intel D3-S4510, 64 GB of RAM and an Intel Xeon with two 10-core 2.20GHz processors. All the experiments are run through the built-in microbenchmarks *db_bench* [3], [5] and YCSB macrobenchmarks [28].

For the six LSM-tree based KV stores, the size of the memory components is set as 32 MB for each (i.e., one active and one immutable), and the bloom filter is 10 bits per entry. For other parameters, such as the size of SST and the capacity of each level, we use the default values in the experiments. For Wisckey, the parameters of the LSM-tree are default and the size of write buffer is set as 1 MB. Since the maximal file size on modern file systems is big enough (e.g., 8 EB on xfs) for Wisckey to run a long time without garbage collection, we allocate sufficient storage space for Wisckey which enables Wisckey to achieve its best write performance [10], [17].

B. Microbenchmarks

This section evaluates the performance of the seven KV stores under various sizes of KV items which vary from 124 B to 1 KB using single-threaded microbenchmarks. The workload generation involves 50 M random writes and 50 M random reads, and the keys follow uniform distribution. We adjust the KV item size by only changing the value size and keeping the key size fixed at 16 B. The experimental results are as shown in Fig.6.

Random Write: As shown in Fig.6 (a), Wisckey outperforms other KV stores, especially when the KV item size is larger than 512 B. The write throughput of PTierDB is consistently close to that of tiered LSM and lazyleveled LSM, and is better than that of leveled LSM and SILK.

Random Read: As shown in Fig.6 (b), PTierDB outperforms other KV stores when the KV item size is of 124 B and 1024 B, and is close to that of leveled LSM when the KV item size is of 256 B and 512 B. Wisckey has the worst read throughput when the KV item size is less than 1024 B.

The experimental results of write and read throughput show that PTierDB achieves a better balance between write cost and read cost than the other six KV stores.

C. Macrobenchmarks

This section use the full YCSB benchmarks to evaluate the performance of the seven KV stores in a zipfian key distribution. The workload generation involves a loading phase which inserts 100 M unique KV items and six core workloads of YCSB, each workload performs 100 M operations. We use 16 B keys and 100 B values.

YCSB benchmarks have two runs, first run performs in the following sequences: load, YCSB A (50% update, 50% reads),

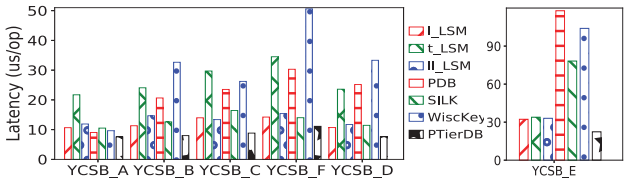

Fig. 7. Average latency of seven KV stores in YCSB.

TABLE I
WRITE AMPLIFICATION AND DB SIZE.

	l_LSM	t_LSM	ll_LSM	PDB	SILK	Wisckey	PTierDB
Write_amp	25.34	4.91	7.3	6.32	17.54	2.8	12.34
DB_size(GB)	7.91	14.84	7.9	12.7	10.5	27.2	8.1

YCSB B (5% updates, 95% reads), YCSB C (100% reads), YCSB F (50% read-modify-write, 50% reads) and YCSB D (5% updates, 95% reads); second run performs load and YCSB E (5% updates, 95% scans; scan length 100 elements).

Performance: Fig.7 shows the average latency of seven KV stores under six YCSB workloads. PTierDB consistently outperforms other KV stores. In read-dominated workloads (YCSB B, C, D and E), the performance advantage of PTierDB is more significant than that in write-dominated workloads (YCSB A and F), which means PTierDB has strictly better read cost than other KV stores. Lazyleveled LSM is slightly better than leveled LSM besides in YCSB C and tiered LSM consistently worse than leveled LSM, especially in YCSB C and YCSB F, which show that read cost is greater than write cost. The latency of tiered LSM and PebblesDB is consistently worse than leveled LSM and SILK, which means leveling works better than tiering for small data on SSD. The results of Wisckey show that the key-value separation design does not work for small data on SSD.

Write Amplification: Table I shows the write amplification and space amplification (evaluated by the DB size) of the seven KV stores in the first run. The volume of actual writes is about 22 GB. Wisckey has the best write amplification and the worst space amplification without garbage collection. Tiered LSM and PebblesDB have worse space amplification and better write amplification, while leveled LSM and SILK have worse write amplification but better space amplification. Lazyleveled LSM has a better balance between write amplification and space amplification. PTierDB trades write amplification for read performance via the intra-level merge policy at the largest level, which also brings good space amplification.

V. RELATED WORK

LSM-tree [7] is a popular index structure for persistent key-value stores. Much work has gone into optimizing the LSM-tree from different perspectives [16], [25], [29]–[31]. Most of them focus on improving write performance, and few studies focus on the improvement of read performance and the trade-off between read and write costs.

bLSM [9] introduces a new merge scheduler to minimize write latency and reduce its negative impact on write performance. VT-tree [29] avoids unnecessary data copying for data that is already sorted using an extra level of indirection. cLSM [31] introduces a new algorithm for increasing concurrency with a goal of increasing scalability. LSM-trie [13] combines hash and trie-tree to organize KV pairs, proposes an efficient compaction strategy and builds stronger Bloom filters for fast searches on disk, but gives up range queries. FloDB [30] inserts an additional fast in-memory buffer on top of Memtable to achieve better write performance. Yue et al. [14] proposes a skip-tree to push KV pairs to non-adjacent, larger-capacity levels to reduce the data traffic from the write buffer to bottom level. TRIAD [16] uses a holistic combination of the optimizations at the memory component level, the storage level and the commit log level to improve the throughput by reducing the write amplification. PebblesDB [25] proposes a new data structure FLSM with the notion of guards, and avoids rewriting data in the same level to reduce write amplification. Mei et al. [32] implements an direct storage system based on LevelDB to fully utilize the LSM-tree features and provide high write performance. LOCS [20] exposes internal flash channels and improves LSM compaction by exploiting the internal parallelism of open-channel SSDs. KVSSD [19] presents a close integration of LSM trees and the FTL to manage write amplifications from different layers. WiscKey [10] introduces key-value separation design to reduce the write amplification caused by compaction for SSD and utilizes the random performance and parallelism characteristics of SSD to provide efficient read and range query performance. HashKV [17] builds hash-based data grouping to map values to storage space atop KV separation, to make GC efficient and provide high write performance under update-intensive workloads. SEALDB [33] collects and groups participating data of each compaction into sets and creates dynamic bands to eliminate write amplification for SMR drives. SILK [26] figures out that the interference between client writes, flushes and compactions results in the high tail latency of LSM-tree based KV stores, and introduces an I/O scheduler to manage external client load and internal LSM maintenance work. LDC [34] proposes a lower-level driven compaction method for LSM-tree KV stores to optimize both the tail latency and the system throughput. These studies focus on optimizing write amplification and write performance for LSM-tree itself and on various storage media.

SlimDB [23] introduces a space-efficient semi-sorting storage, combined with the Cuckoo filter and an slection model for index and filter to improve read and write performance for SSD. NoveLSM [35] utilizes the NVM features to redesign LSM for NVM by implementing byte-addressable skiplist, direct mutability of persistent state, and opportunistic read parallelism. ElasticBF [36] presents a fine-grained heterogeneous Bloom filter management scheme with dynamic adjustment according to data hotness to speed up read performance. Dostoevsky [24] introduces Lazy Leveling and Fluid LSM-tree to adaptively remove superfluous merging by navigating the Fluid LSM-tree design space based on the application workload and hardware so as to achieve better space-time trade-offs. These studies involve read performance improvement and trade-

offs between read and write cost. Different from Dostoevsky, PTierDB attempts to achieve a better balance between read and write cost through the adaptive tiering principle and three merge policies at different levels in the LSM-tree.

The above KV stores and PTierDB are stand-alone KV stores that can serve as the foundation of distributed storage systems (e.g., Cassandra [8], Dynamo [2] and HBase [37]).

VI. Conclusion

PTierDB strikes the better balance between read cost and write cost for small data on SSD via an adaptive tiering principle and three merge policies at different levels in the LSM-tree. Based on adaptive tiering, the three merge policies can bound the key range of SSTables at each level to reduce read cost, further reduce data rewrites to mitigate write cost, and reduce the overlapped hot SSTables at the largest level to reduce read cost. Compared with state-of-the-art KV stores and the KV implementations with popular merge policies, experimental results show that, for small data on SSD, P-TierDB achieves a better read and write cost balance under microbenchmarks and outperforms all other KV stores under YCSB macrobenchmarks.

Acknowledgment

We thank the anonymous reviewers for all their helpful comments and suggestions. This work is supported by the Innovation Group Project of National Natural Science Foundation of China (Grant No.61821003).

References

[1] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, and et al, "Analysis of HDFS under hbase: a facebook messages case study," in FAST, Santa Clara, CA, USA, February 17-20, 2014, pp. 199–212.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, and et al, "Dynamo: Amazon's highly available key-value store," Acm Sigops Operating Systems Review, vol. 41, no. 6, pp. 205–220, 2007.

[3] LevelDB, 2006. [Online]. Available: https://github.com/google/leveldb/

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, and D. A. W. et al, "Bigtable: A distributed storage system for structured data," in OSDI, November 6-8, Seattle, WA, USA, 2006, pp. 205–218.

[5] RocksDB, 2006. [Online]. Available: https://github.com/facebook/rocksdb

[6] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, and et al, "Pnuts: Yahoo!'s hosted data serving platform," Proceedings of the Vldb Endowment, vol. 1, no. 2, pp. 1277–1288, 2008.

[7] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (lsm-tree)," Acta Informatica, vol. 33, no. 4, pp. 351–385, 1996.

[8] Cassandra, 2006. [Online]. Available: http://cassandra.apache.org

[9] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in SIGMOD, Scottsdale, AZ, USA, May 20-24, 2012, pp. 217–228.

[10] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in FAST, Santa Clara, CA, USA, February 22-25, 2016, pp. 133–148.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in SIGMETRICS, London, United Kingdom, June 11-15, 2012, pp. 53–64.

[12] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in FAST, Santa Clara, CA, USA, February 24-27, 2020, pp. 209–223.

[13] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items," in ATC , July 8-10, Santa Clara, CA, USA, 2015, pp. 71–82.

[14] Y. Yue, B. He, Y. Li, and W. Wang, "Building an efficient put-intensive key-value store with skip-tree," TPDS, vol. 28, no. 4, pp. 961–973, 2017.

[15] Y. Ting, W. Jiguang, H. Ping, H. Xubin, G. Qingxin, and et al, "A Lightweight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores," MSST, 2017.

[16] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, and et al, "TRIAD: creating synergies between memory, disk and log in log structured key-value stores," in ATC, Santa Clara, CA, USA, July 12-14, 2017, pp. 363–375.

[17] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in KV storage via hashing," in ATC, Boston, MA, USA, July 11-13, 2018, pp. 1007–1019.

[18] B. K. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," Proc. VLDB Endow., vol. 3, no. 2, pp. 1414–1425, 2010.

[19] S. Wu, K. Lin, and L. Chang, "KVSSD: close integration of LSM trees and flash translation layer for write-efficient KV store," in DATE, Dresden, Germany, March 19-23, 2018, pp. 563–568.

[20] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, and et al, "An efficient design and implementation of lsm-tree based key-value store on open-channel SSD," in EuroSys, Amsterdam, The Netherlands, April 13-16, 2014, pp. 16:1–16:14.

[21] P. Menon, T. Rabl, M. Sadoghi, and H. Jacobsen, "Cassandra: An SSD boosted key-value store," in ICDE, Chicago, IL, USA, March 31 - April 4, 2014, pp. 1162–1167.

[22] R. Thonangi and J. Yang, "On log-structured merge for solid-state drives," in ICDE, San Diego, CA, USA, April 19-22, 2017, pp. 683–694.

[23] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "Slimdb: A space-efficient key-value storage engine for semi-sorted data," PVLDB, vol. 10, no. 13, pp. 2037–2048, 2017.

[24] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging," in SIGMOD, Houston, TX, USA, June 10-15, 2018, pp. 505–520.

[25] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in SOSP, Shanghai, China, October 28-31, 2017, pp. 497–514.

[26] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: preventing latency spikes in log-structured merge key-value stores," in ATC, WA, USA, July 10-12, 2019, pp. 753–766.

[27] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in VLDB, Athens, Greece, August 25-29, 1997, pp. 16–25.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in SoCC, Indianapolis, Indiana, USA, June 10-11, 2010, pp. 143–154.

[29] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in FAST, San Jose, CA, USA, February 12-15, 2013, pp. 17–30.

[30] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi, "Flodb: Unlocking memory in persistent key-value stores," in EuroSys, Belgrade, Serbia, April 23-26, 2017, pp. 80–94.

[31] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in EuroSys, Bordeaux, France, April 21-24, 2015, pp. 32:1–32:14.

[32] F. Mei, Q. Cao, H. Jiang, and L. Tian, "Lsm-tree managed storage for large-scale key-value store," TPDS, vol. 30, no. 2, pp. 400–414, 2019.

[33] T. Yao, Z. Tan, J. Wan, P. Huang, Y. Zhang, and et al, "SEALDB: an efficient lsm-tree based KV store on SMR drives with sets and dynamic bands," TPDS, vol. 30, no. 11, pp. 2595–2607, 2019.

[34] Y. Chai, Y. Chai, X. Wang, H. Wei, N. Bao, and Y. Liang, "LDC: A lower-level driven compaction method to optimize ssd-oriented key-value stores," in ICDE, Macao, China, April 8-11, 2019, pp. 722–733.

[35] S. Kannan, N. Bhat, A. Gavrilovska, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in ATC, Boston, MA, USA, July 11-13, 2018, pp. 993–1005.

[36] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in USENIX ATC , Renton, WA, USA, July 10-12, 2019, pp. 739–752.

[37] HBase, 2006. [Online]. Available: http://hbase.apache.org/