An optimized implementation for concurrent LSM-structured key-value stores

1st Li Liu
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
lillian.wtu@gmail.com

2nd Hua Wang
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
hwang@hust.edu.cn

3rd Ke Zhou
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
k.zhou@hust.edu.cn

***Abstract*—Log-Structured Merge Trees (LSM) based key-value (KV) stores such as LevelDB and HyperLevelDB, use a compaction strategy which brings frequent compaction operations, to store key-value items in sorted order. However, large numbers of compactions impose a negative impact on write and read performance for random data-intensive workloads.**

To remedy this problem, this paper presents OHDB, an optimization of HyperLevelDB for random data-intensive workloads. OHDB implements two stand-alone techniques in the disk component of LSM structure to optimize the concurrent compactions. One is dividing KV items by prefix at the first level in the disk component, to reduce the frequency of overlapping in key range among data files, and thus reduces the amount of compactions. The other is separating the first level in the disk component from the rest levels, and organizing them in two disks individually, to increase parallelism of disk writes of compactions. We evaluate three OHDBs which are OHDB with each of the technique and OHDB with the combination of both respectively, using micro-benchmarks with random write-intensive and read-intensive workloads. Experimental results show that OHDB reduces the amount of compactions by a factor of up to 4x, and improves the write and read performance for random data-intensive workloads under various settings.

I. Introduction

Key-value stores play a crucial role in a variety of modern data-intensive applications, such as messaging [2]–[4], e-commerce [5], online gaming [6], search indexing [7] [8] [28] and advertising [2] [7] [9]. Some KV stores maintain data in memory (e.g., Redis [26] and Memcached [27]), while others employ solid-state drives (SSD) or rotating disks (HDD) (e.g., LevelDB [8], Cassandra [10] and LOCS [20]). Since the volume of data is larger than main memory in many applications in real world [4] [5] [7] [9], persistent KV stores are always required.

LSM [1] structure is employed in a wide range of persistent KV stores, such as LevelDB [8], RocksDB [2], Cassandra [10], cLSM [11], HyperLevelDB [12] and bLSM [13]. LSM structured KV stores (LSMs) can provide excellent write throughput for write-intensive workloads, at the cost of a small decrease in read throughput. However, LSM structure also brings some constraints to the performance of LSMs.

Generally speaking, LSMs consist of two components: a memory component and a disk component. The memory component has two parts residing in main memory, called Memtable and immutable Memtable [8]. Memtable is responsible for absorbing data (i.e., KV items), and organizing them in a skiplist [15]. The size of the Memtable is usually ranging from a few MBs to tens of MBs. When the size of Memtable exceeds its target size, it will turn into the immutable Memtable. The disk component is a persisting data storage, typically has four parts: a commit log, an operation log, metadata and data. The commit log is a write-ahead log, used to avoid data loss, the operation log is used to support checking the execution state of the KV store, and the metadata is used to recover the KV store. KV items are organized into multiple levels (L_0, L_1,..., L_n) (e.g., 7 levels default in LevelDB) with increasing sizes. Each level contains a number of data files which store the KV items in sorted order, called Sorted String Table (SST) [8]. The size of SST is also ranging from a few MBs to tens of MBs.

When the Memtable becomes full, it turns into the immutable Memtable, then a new Memtable is created. The immutable Memtable will be flushed to the first level (i.e., level L_0) in the disk component. This operation is called flushing. Because of this, the SSTs at level L_0 have overlapping key ranges. When the size of one level in the disk component exceeds its target size, such as level L_i, it will select one or more SSTs from level L_i and merge into SSTs at level L_{i+1} in overlapping key range, discarding stale values and writing new SSTs in sorted order to level L_{i+1}. This operation is called compaction. Thus, SSTs at levels from L_1 to L_n have disjoint key ranges.

Compaction and flushing are indispensable to maintain the LSM structure. Unfortunately, even though they occur outside the critical path of user-oriented operations, they still take up a significant amount of the available resources, thus disturbs the write and read performance of LSMs [14].

We propose two complementary techniques, aiming to mitigate the negative influence on write and read performance caused by compaction and flushing for the random data-intensive workloads. The first technique is to partition the KV items according to their prefix and write them into different SSTs at the first level in the disk component during flushing, aiming to decrease the amount of compaction operations. The second technique uses two disk components to separate the

978-1-5386-8367-5/18/$31.00 ©2018 IEEE

first level in the disk component from the rest levels, aiming to optimize the parallel disk writes caused by concurrent compactions. The two techniques are stand-alone, and can be used independently or in combination for LSMs.

We implement an optimization of HyperLevelDB (OHDB), a LSM structured KV store with the two techniques to reduce the negative influence of concurrent compactions on write and read performance under random data-intensive workloads. We extensively compare OHDB against HyperLevelDB. Experimental results show that with the use of partition technique, the amount of compactions is reduced to 4 to 5 times, thus improving the write and read performance of OHDB. When using the technique of two disk components in a hybrid KV store (HDD + SSD), the performance is close to that of HyperLevelDB run on SSD. When combining the two techniques, OHDB can improve the performance for random data-intensive workloads in many cases.

To summarize, this paper makes the following key contributions: (1) the design of two complementary techniques, (2) the implementation of three OHDBs which are OHDB with each of the technique and OHDB with the combination of both respectively, and (3) an evaluation of the three OHDBs in comparison to HyperLevelDB.

Roadmap. The rest of the paper is structured as follows. Section 2 describes the background and motivation. Section 3 presents our two techniques for reducing the impact of the compaction and flushing operations on performance. Section 4 analyzes its performance. Section 5 discusses related work. Section 6 concludes the paper.

II. Motivation

HyperLevelDB. HyperLevelDB builds upon LevelDB, aiming to improve concurrency by allowing concurrent compactions and thus increases the performance. The architecture of HyperLevelDB can be seen in Fig. 1(a). There are two concurrent compaction threads worked in HyperLevelDB, named *Memtable Compaction* and *Level Compaction*, corresponding to flushing and compaction mentioned above, respectively.

Both of HyperLevelDB and LevelDB are effective for sequential write workloads, with none or rare compactions. However, databases always have a lot of updates in real world, and that will lead to a large number of compactions. HyperLevelDB has better write and read performance than LevelDB for random write-intensive and read-intensive workloads, as shown in Table I. The random workloads denote that the keys are randomly distributed and there are many updates. **WT**, **RT** and **C** stand for write throughput, read throughput and the amount of level compaction operations respectively. The evaluations are run under these parameters: fixed number of KV items (100M), fixed size of MemTable (32MB), and various sizes of SST.

Besides the comparison of HyperLevelDB and LevelDB, we observe that with the increasing of the size of SST, the number of compactions and write throughput are decreased while read throughput is increased. The reason is that the larger the size of SST is, the less the number of SSTs will be, and each read operation will access less SSTs, thus improves the read performance; but meanwhile, each level compaction may need to read and write larger size of KV items which will result in longer period of the compaction process, thus disturbs the write process at front end and lowers the write performance.

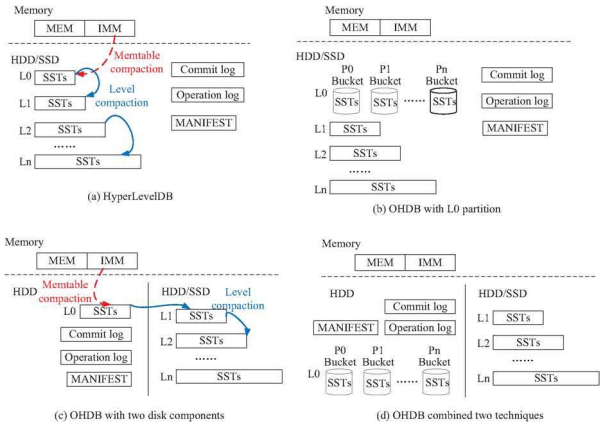

Fig. 1. HyperLevelDB and OHDB Architecture.

TABLE I
HyperLevelDB vs LevelDB.

LSMs	SST size (MB)	WT (MB/s)	RT (ops/s)	C
LevelDB	2	4.9	27837	12513
	4	3.5	30535	6341
HyperLevelDB	4	6.8	42439	9800
	8	6.5	76470	7677
	16	6.0	131337	4896

The effect of compactions. What do the two compaction threads do in HyperLevelDB?

- Memtable compaction is to write the immutable Memtable to the first level (L_0) in the disk component.
- Level compaction is responsible for data merging between levels in the disk component. For example, when the size of level L_i exceeds its target size, it merges SSTs at level L_i with SSTs in overlapping key range at level L_{i+1}, only reserving newer values if duplicated keys exist, and generates new SSTs at level L_{i+1}.

It seems that the two compaction threads are able to concurrently execute, however, actually they use the mutex lock to synchronize and keep consistency. They seize the mutex lock during in-memory processes, and release the mutex lock in the process of writing data to the disk component, thus the start and end of compactions are serial. That means, when the Memtable compaction and level compaction are invoked at the same time, if level compaction acquires the mutex lock first, the Memtable compaction has to wait until the mutex lock released by the level compaction.

It would slow down the data write operations at front end, when the immutable Memtable has not been flushed yet but the Memtable has already been full. That means if the Memtable

compaction costs a long period, it will degrade the write performance. For the random write-intensive workloads which always cause large volumes of level compactions, it may delay the process of Memtable compaction. Thus, we consider a method to reduce the amount of level compactions, that is to partition the KV items according to their prefix in Memtable compaction process, to reduce the frequency of overlapping in key range among SSTs, and then decrease the amount of level compactions.

Because of the concurrent compactions, each compaction releases the mutex lock during the process of writing data to the disk component, it may generate paralleled writes to disk. Since HyperLevelDB builds databases on a single disk, we think about a trick: If we build two disk components for the two compaction threads, keep level L_0 on one disk, and other levels on another disk, will it have a positive impact on write performance? But anyway, if we use a SSD as a disk component for the LSM structured KV stores, to mitigate the wear-out of SSD, it is an effective way to build a hybrid KV store by putting the commit log, operation log and metadata files (i.e., files always generate small writes) on HDD, and put multi-level structured data on SSD, at the cost of sacrificing a little write throughput.

III. Implementation

The overarching goals of OHDB for random data-intensive workloads are (1) to reduce the number of compactions, (2) to optimize concurrent compactions and (3) to improve write and read performance for random workloads. To achieve these goals, we present two techniques: L_0 partition and two disk components, which are stand-alone techniques and generally applicable to LSM-based KV stores. Fig. 1(b-d) shows the architecture of OHDB with each of the technique and the combination of both, respectively. We now provide a detailed description of the two technologies.

A. L_0 Partition

The architecture of OHDB with L_0 partition is shown in Fig. 1(b). The goal of L_0 partition is to reduce the frequency of overlapping in key range among SSTs for random workloads, and hence, reduce the amount of level compactions. L_0 partition changes the process of Memtable compaction, which used to flush the immutable Memtable to the level L_0 in the disk component, replaces with segmenting KV items in the immutable Memtable to different SSTs at level L_0 according to their prefix.

The challenge is that L_0 partition may generate many SSTs at level L_0 (e.g., under uniformly distributed random workloads), resulting in frequent level compactions from level L_0 to L_1. To resolve this issue, we change the data structure of level L_0, from flat organization to partition buckets, as shown in Fig. 1(b). When a partition bucket accumulates a threshold number of SSTs, it starts up a level compaction, and all SSTs in the partition bucket are compacted into level L_1.

With the use of partition buckets, there emerges another gap. All of the partition buckets will accumulate the threshold number of SSTs simultaneously, when the KV items in the workload are uniformly distributed. Thus, it will also generate frequent level compactions from level L_0 to L_1. To remedy this gap, we add a condition variable at the end of the process of the level compaction, to raise the priority of MemTable compaction. The condition variable is used to determine whether the immutable Memtable exists. After completing the total process of a level compaction, if there exists an immutable Memtable, it will release the mutex lock for a second, allowing the MemTable compaction to occupy the mutex lock.

There is a trade-off between the benefits acquired from L_0 partition and the negative impact on write throughput caused by L_0 partition. Because L_0 partition generates a few SSTs (more SSTs under the uniform random workloads) during each Memtable compaction, that results in frequent file creating operations, which have a negative impact on write throughput and will drop the write performance.

B. Two disk components

The architecture of OHDB with two disk components is shown in Fig. 1(c). One disk component is used to store the commit log, operation log, metadata files and SSTs at level L_0. The other disk component is used to store SSTs at the rest of levels. We implement two disk components in the concurrent LSM structured KV store for two reasons. First, we can observe whether using two disk components can accelerate the concurrent processes of writing data to the disk component through parallel compactions, and the effect of this technology on write and read performance for concurrent LSM structured KV stores. Second, it is an effective solution to build a hybrid HDD and SSD architecture for the LSM structured KV stores to improve the endurance of SSD. Using HDD as one disk component to store logs and metadata which take on almost all the small writes, and using SSD to store most of SSTs for better read and write performance.

But for the hybrid architecture, there is also a defect for the write and read performance of OHDB with two disk components. Separating the level L_0 from the rest levels of LSM structure to two disk components may bring about a positive impact on concurrent compactions, but it may also result in negative impacts on write and read performance, which result from lots of write and read operations on HDD, including the writes for logs, metadata and SSTs at level L_0, and reads at level L_0 through each level compaction from level L_0 to L_1.

Fig. 1(d) shows the architecture of OHDB combining the two technologies. We will observe the performance of the three architectures for random data-intensive workloads through extensive experiments.

IV. Evaluation

In this section, we do a set of experiments of HyperLevelDB and OHDB for random data-intensive workloads. The evaluation results demonstrate the benefits acquired from L_0 partition and two disk components. There are two issues

we should consider when evaluating the performance of the LSM structured KV stores:

- What is the relationship between write performance and level compactions?
- How will the Memtable size and SST size influence the performance of LSM structured KV stores?

A. Experimental Conditions

We evaluate the performance of the LSM structured KV stores within different conditions, including the testing machine, the size of the Memtable and the size of SST. We choose two testing machines, which have various profiles to run the experiments with various parameters, as follows:

- Testing machine:
  - High-end commodity: 40 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz processors, 64-GB memory, 64-bit Linux ubuntu, SSD, HDD;
  - Low-end commodity: 4 Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz processors, 32-GB memory, 64-bit Linux 3.10, HDD, HDD;
- Size of Memtable (MB): fixed (32), varied in (16, 32, 48, 64);
- Size of SST (MB): fixed (8), varied in (4, 8, 16, 32).

We compare OHDB against HyperLevelDB, using the default microbenchmarks in HyperLevelDB (*db_bench*), and the default key size and value size, which is of 16 bytes and 100 bytes, respectively. We focus on the performance of HyperLevelDB and OHDB for random data-intensive workloads, so all experiments are run within the two benchmarks: *fillrandom* and *readrandom*. The number of KV items of the workloads is 100M. We use 10 partitions in level L_0.

B. Performance on various testing machines

With the implementation of L_0 partition and two disk components, we evaluate four KV stores, including HyperLevelDB (**HDB**), OHDB with L_0 partition (**OHDB-**L_0P), OHDB with two disk components (**OHDB-**$2DC$) and OHDB combining the two technologies (**OHDB-**C) on two testing machines. On low-end commodity, **HDB** and **OHDB-**L_0P are run on HDD, while **OHDB-**$2DC$ and **OHDB-**C are run on two HDDs. On high-end commodity, **HDB** and **OHDB-**L_0P are run on SSD, while **OHDB-**$2DC$ and **OHDB-**C are run on HDD + SSD.

All experiments are run with these parameters: fixed number of KV items (100M), fixed size of Memtable (32MB) and fixed sizes of SST (8MB). The results are shown in Table II. **SN** stands for the number of SSTs, **TW** stands for the total size writing to the disk components in each experiment. **CR** stands for the total size of reading generated by compactions. **CW** stands for the total size of writing generated by compactions.

Write Performance. Compared to **HDB** and **OHDB-**$2DC$, the volumes of level compactions generated in **OHDB-**L_0P and **OHDB-**C which implement the technique of L_0 partition are reduced by more than 4 times, as shown in Table II (**C**). The lower amount of compactions has a positive impact on write performance. As shown in Table II (**WT**), the write performance of **OHDB-**L_0P is the best on high-end commodity, and a little better than **HDB** and **OHDB-**L_0P on low-end commodity. The write performance of **OHDB-**C is the best on low-end commodity, but the worst on high-end commodity, because of the negative impact caused by lots of operations on HDD, including frequent file creations during each Memtable compaction and more SST reads caused by level compactions from level L_0 to L_1.

Such a significant reduction in the number of compactions is caused by the uniformly distributed random workloads. We found that the KV items in the random write-intensive workload generated by the micro-benchmark is uniformly distributed, thus the key range overlapping always exists, which results in a large number of level compactions. With the use of L_0 partition, OHDB reduces the frequency of overlapping in key range among SSTs, and hence, reduces the amount of level compactions.

The lower amount of compactions did not bring too much improvement on the write amplification, as shown in Table II (**TW**), the write amplification is still over 7 times. Because of the uniformly distributed random workloads, there is always key range overlapping among SSTs, which results in large volume of reads and writes in each level compaction and the total compaction progress, as showed in Table II (**CR**, **CW**). Since we did not optimize the leveled structure, the overlapping in key range leads to the write amplification. And the large volume of reads and writes will result in long duration of each compaction, thus it will also impact the write throughput at the front end.

Read Performance. The read performance of **OHDB-**L_0P and **OHDB-**C are better than the other two KV stores on both low-end commodity and high-end commodity under this parameter setting, as can be seen in Table II (**RT**). The improvement of read performance relies on the less number of SSTs, as shown in Table II (**SN**), and less compaction operations during the intensive read operations. Because of the write-intensive and read-intensive workloads, after the write workload finished, it leaves a lot of potential compactions which will be aroused by read operations, called seek compaction [8] [12]. Reads need to acquire the mutex lock briefly at the beginning and end of the operation, thus plenty of compactions under the read-intensive workload will lead to a negative impact on read performance.

C. Impact of Memory Component Size

To observe the impact of the size of Memtable on performance of the four KV stores for random data-intensive workloads, we run the experiments with fixed number of KV items (100M), fixed size of SST (8MB) and varied sizes of Memtable in the set of (16, 32, 48, 64) MB on low-end commodity and high-end commodity individually.

Performance on low-end commodity. The results of read and write throughput are shown in Fig. 2. The amount of level compaction operations are shown in Table III.

Regardless of the size of Memtable, the amounts of level compactions of the two KV stores with L_0 partition are always near 5 times less than the other two KV stores, as shown in

TABLE II
PERFORMANCE OF 4 KV STORES ON 2 TESTING MACHINES

Testing Machine	LSMs	WT (MB/s)	RT (ops/s)	C	SN	TW (GB)	CR (GB)	CW (GB)
Low-end commodity	**HDB**	6.5	76470	7677	1127	86.7	79.5	75.8
	OHDB-L_0P	6.7	85106	1611	**906**	79.5	72.6	68.6
	OHDB-$2DC$	6.3	71802	7717	1152	87.8	81.0	77.0
	OHDB-C	**7.5**	**88683**	**1601**	909	**79.5**	**72.6**	**68.6**
High-end commodity	**HDB**	12.5	101173	8613	1084	96.3	89.4	85.4
	OHDB-L_0P	**18.8**	117467	**923**	911	**66.8**	**59.9**	**55.9**
	OHDB-$2DC$	12.5	94885	8563	1098	95.8	88.9	84.9
	OHDB-C	11.3	**123061**	1641	**907**	81.1	74.3	70.2

TABLE III
THE VOLUME OF COMPACTIONS ON HDD.

LSMs	Size of Memtable (MB)			
	16	32	48	64
HDB	10099	7677	6405	5410
OHDB-L_0P	2172	1547	1416	1214
OHDB-$2DC$	10074	7717	6511	5480
OHDB-C	2216	1554	1370	1218

TABLE IV
THE VOLUME OF COMPACTIONS ON SSD.

LSMs	Size of Memtable (MB)			
	16	32	48	64
HDB	10979	8613	7082	6080
OHDB-L_0P	1122	923	754	709
OHDB-$2DC$	11067	8563	7227	6076
OHDB-C	2247	1641	1421	1218

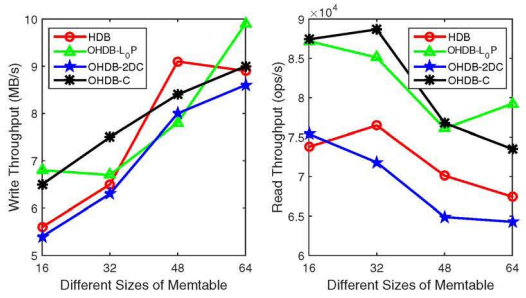

Fig. 2. Performance of 4 KV stores on HDD with different sizes of Memtable.

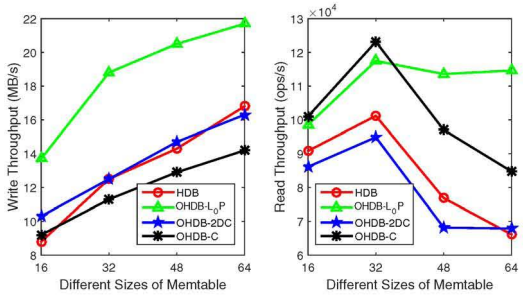

Fig. 3. Performance of 4 KV stores on SSD with different sizes of Memtable.

Table III. With the increasing of the size of Memtable, the number of level compactions decreases.

It is said that as the size of Memtable grows, maintaining the sorted order becomes increasingly costly, and could actually result in throughput losing [16]. However, the write throughputs of the four KV stores in Fig. 2 progressively increase, benefits from the concurrent compactions for data-intensive workloads. But the read performance decreases, because each read has to search the Memtable first, and the larger the Memtable is, the longer the searching will be.

The write performance of **OHDB-**L_0P and **OHDB-**C is better than that of the other two KV stores in most cases, while the read performance of them are always better than the other two KV stores, both benefit from the lower amount of level compactions.

Performance on high-end commodity. The results of read and write throughput are shown in Fig. 3. The amount of level compaction operations are shown in Table IV.

As shown in Table IV, the number of level compactions of **OHDB-**C is also near 5 times less than that of the other two KV stores, while the number of level compactions of **OHDB-**L_0P is even lower, regardless of the size of Memtable. Compared to the results on HDD, **HDB** and **OHDB-**$2DC$ generate more level compactions on SSD.

Fig. 3 shows that **OHDB-**L_0P has the best write and read performance at most cases. **OHDB-**$2DC$ has a better write performance than **HDB** under some settings, which means the technique of two disk components has a positive impact on concurrent compactions when using SSD as a disk component, with a little sacrificing of read performance. **OHDB-**C has the worst write performance, because of the frequent file creations on HDD, but has a better read performance, which benefits from the lower number of level compactions.

D. Impact of Disk Table Size

To observe the impact of the size of SST on performance of the four KV stores for random data-intensive workloads, the experiments are run with fixed number of KV items (100M) and fixed size of MemTable (32MB) on low-end commodity and high-end commodity respectively. The size of SST varies in (4, 8, 16, 32) MB.

Performance on low-end commodity. The results of read and write throughput are shown in Fig. 4. The amount of level compaction operations are shown in Table V.

TABLE V
THE VOLUME OF COMPACTIONS ON HDD.

LSMs	Size of SST (MB)			
	4	8	16	32
HDB	9800	7677	4896	3294
OHDB-L_0P	2858	1547	1036	777
OHDB-$2DC$	9880	7717	7656	3418
OHDB-C	2729	1554	1019	779

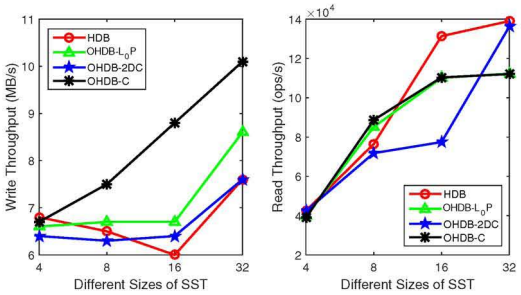

Fig. 4. Performance of 4 KV stores on HDD with different sizes of SST.

Regardless of the size of SST, the amounts of level compactions generated by **OHDB-**L_0P and **OHDB-**C are always near 4 times less than that of the other two KV stores, as shown in Table V.

With the increasing of the size of SST, the number of SSTs decreases, and the volume of level compactions reduces, that brings about the improvement of read throughput, as shown in Fig. 4. The write throughput of the three OHDBs increases progressively, while **OHDB-**C has much better write throughput than the other three KV stores. But the read performance of **OHDB-**C is a little worse than HyperLevelDB at most settings, which may because of more SST reads caused by $L0$ compaction during each read operation and longer period of each seek compaction caused by large volumes of reads and writes during each level compaction.

Performance on high-end commodity. The results of read and write throughput are shown in Fig. 5. The amount of level compactions are shown in Table VI.

TABLE VI
THE VOLUME OF COMPACTIONS ON SSD.

LSMs	Size of SST (MB)			
	4	8	16	32
HDB	10923	8613	5934	3882
OHDB-L_0P	1846	923	515	247
OHDB-$2DC$	10821	8563	5877	3812
OHDB-C	2858	1641	1063	813

As shown in Table VI, the number of level compactions of **OHDB-**C is near 4 to 5 times less than that of the other two KV stores, while the number of level compactions of **OHDB-**L_0P is even lower, regardless of the size of SST. Compared to the results on HDD, **HDB** and **OHDB-**$2DC$ generate more level compactions on SSD at most cases.

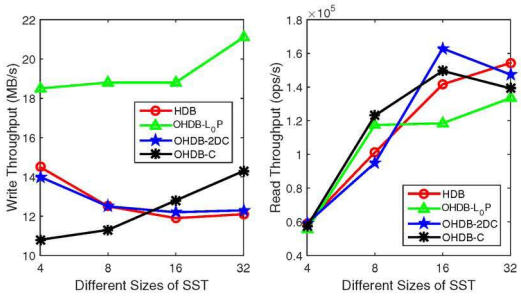

Fig. 5. Performance of 4 KV stores on SSD with different sizes of SST.

With the growing of the size of SST, the write performance of **OHDB-**L_0P and **OHDB-**C increases while **OHDB-**L_0P has the best write throughput; the read performance of the four KV stores increases while **OHDB-**L_0P has the worst read throughput at most settings. The write performance of **OHDB-**L_0P benefits from the lowest amount of level compactions and totally running on SSD, while the read performance is influenced by more SST reads during each read operation and long period of each seek compaction. The write and read throughput of **OHDB-**$2DC$ are close to **HDB**, it means that it is a good choice to use hybrid architecture for concurrent LSM structured KV stores.

Discussion. In all the above experiments, the technique of L_0 partition has an excellent positive impact on the performance of concurrent LSM structured KV stores for random data-intensive workloads at most cases, especially running on SSD. The technique of two disk components has a positive impact on the write performance of hybrid concurrent LSM structured KV stores at the cost of a little sacrificing on read performance, while bringing about the improvement of the endurance of SSD and cost-efficient, in comparison to HyperLevelDB running on SSD.

Combined the two techniques, OHDB has the best performance among the four KV stores on HDDs at most cases; and has a better read performance with a worse write performance on HDD + SSD. There are several reasons for the negative impacts on write performance for hybrid OHDB: lots of write operations on HDD caused by the technique of two disk components; frequent file create operations on HDD generated by the technique of L_0 partition and a lot of read operations on HDD caused by level compactions from level L_0 to L_1.

In this paper, we focus on optimizing the concurrent compactions by the two technologies to improve the performance of LSM structured KV stores for random data-intensive workloads. We find that the KV items in random workloads generated by the micro-benchmarks are uniformly distributed, resulting in the drawbacks as follows: (1) many file creations during each Memtable compaction; (2) frequent demands on level compaction from level L_0 to L_1; (3) reading more SSTs at each read operation. Thus, it may have a negative impact on write and read performance of LSM structured KV stores.

E. Related Work

Many researchers have worked on improving the performance of LSM structured key-value stores, and presented many works to improve the performance or reduce write amplification. OHDB is a stand-alone persistent LSM structured key-value store, and focuses on improving the performance of random data-intensive workloads. OHDB is related to previous designs of LSM-based KV stores. LevelDB [8] is one of the earliest LSM-based KV stores, employs level-style compaction, and the compaction process is single-threaded.

Performance improvement. HyperlevelDB [12] replaces the sequential memory component flushing in LevelDB, makes it concurrent with level compaction, and increases write throughput. However, writes still need to acquire a global mutex lock at the beginning and end of each operation, which is a bottleneck for scalability. RocksDB [2] introduces multi-threaded compaction and tackles other concurrency issues, but still exhibits high write amplification due to its design being fundamentally similar to LevelDB. cLSM [11] is also based on LevelDB and introduces a new algorithm for increasing concurrency with a goal of increasing scalability. bLSM [13] introduces a new merge scheduler to minimize write latency and reduce its negative impact on write performance. LOCS [20] exposes internal flash channels and improves LSM compaction by exploiting the internal parallelism of open-channel SSDs. FloDB [16] inserts an additional fast in-memory buffer on top of the existing in-memory component of the LSM tree to achieve better scalability. OHDB reduces the amount of compaction operations through reducing the overlap frequency between SSTs, and attempts to optimize the efficiency of concurrent writing to disk.

Write amplification improvement. VT-Tree [19] avoids unnecessary data copying for data that is already sorted using an extra level of indirection. LSM-trie [18] uses a trie-tree structure to organize keys, proposes a more efficient compaction to reduce write amplification and builds stronger Bloom filters for fast searches on disk, but gives up the sorted order of the KV items and range queries. WiscKey [17] separates keys from values and only stores keys in a sorted LSM tree, and consequently reduce write amplification. TRIAD [29] uses a holistic combination of three techniques, which respectively optimizes the LSM KV at the memory component level, the storage level and the commit log level, aiming to improve the throughput by reducing the write amplification. PebblesDB [30] proposes a new data structure FLSM, which introduces the notion of guards to organize logs, and avoids rewriting data in the same level to reduce write amplification. The techniques proposed in OHDB are orthogonal to these approaches, and can be leveraged in synergy to them to further enhance I/O efficiency.

Recent studies show that the efficiency of LSM-based KV stores is highly dependent on their proper setting, such as the number and the size of levels, and policies for compaction, as well as workload parameters and requirements like memory budget [21]–[24]. The same goes for OHDB.

Cassandra [10], BigTable [7] and HBase [3] are distributed LSM structured KV stores, employing size-tiered compaction. In addition, Cassandra also supports the leveled compaction strategy, based on LevelDBs compaction scheme [25]. OHDB can be leveraged in synergy to these distributed LSM structured KV stores to further enhance I/O efficiency.

F. Conclusion

This paper presents OHDB, an optimized LSM structured KV store built upon HyperLevelDB, aiming to improve the performance for random data-intensive workloads by optimizing the concurrent compactions. OHDB implements two stand-alone techniques: L_0 partition and two disk components, which also can work together. Thus, we evaluate four LSM structured KV stores, including HyperLevelDB and three OHDBs. With L_0 partition, OHDB improves I/O efficiency by decreasing the amount of compactions by a factor of up to 4x. With two disk components, OHDB has a positive impact on write performance with a little read performance sacrificing for the KV store using the hybrid disks (HDD + SSD) as disk components. Combining the two techniques, OHDB has better read and write performance than the other three KV stores using HDDs as disk components under proper settings, while has a better read performance but a worse write performance using the hybrid disks as disk components.

The two techniques proposed in OHDB are orthogonal to many other LSM structured KV stores, which can be leveraged in synergy to these KV stores to further enhance I/O efficiency. As a future work, we will evaluate OHDB with other types of workloads, such as skewed distributed workload, to observe the impact of the two technologies on performance of LSM structured KV stores and attempt to find out a method to mitigate the write amplification.

REFERENCES

[1] ONeil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351-385.

[2] RocksDB, a persistent key-value store for fast storage environments, 2016. http://rocksdb.org/.

[3] Apache HBase, a distributed, scalable, big data store, 2016. http://hbase.apache.org/.

[4] Harter T, Borthakur D, Dong S, et al. Analysis of HDFS under HBase: a facebook messages case study. FAST 2014.

[5] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store. SOSP 2007.

[6] Debnath B, Sengupta S, Li J. SkimpyStash: RAM space skimpy key-value store on flash-based storage. SIGMOD 2011.

[7] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data. OSDI 2006.

[8] LevelDB, a fast and lightweight key/value database library by Google, 2005. https://github.com/google/leveldb.

[9] Cooper B F, Ramakrishnan R, Srivastava U, et al. PNUTS: Yahoo!'s hosted data serving platform. PVLDB 2008.

[10] Apache Cassandra. http://cassandra.apache.org.

[11] Golan-Gueta G, Bortnikov E, Hillel E, et al. Scaling concurrent log-structured data stores. EuroSys 2015.

[12] HyperLevelDB, a fork of LevelDB intended to meet the needs of HyperDex while remaining compatible with LevelDB, 2014. https://github.com/rescrv/HyperLevelDB.

[13] Sears R, Ramakrishnan R. bLSM: a general purpose log structured merge tree. SIGMOD 2012.

[14] Balmau O M, Didona D, Guerraoui R, et al. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. USENIX ATC 2017.
[15] William Pugh. 1990. A Skip List Cookbook. Technical Report CS-TR-2286.1. University of Maryland.
[16] Balmau O M, Guerraoui R, Trigonakis V, et al. FloDB: Unlocking memory in persistent key-value stores. EuroSys 2017.
[17] Lu L, Pillai T S, Gopalakrishnan H, et al. WiscKey: Separating keys from values in SSD-conscious storage. FAST 2016.
[18] Wu X, Xu Y, Shao Z, et al. LSM-trie: an LSM-tree-based ultra-large key-value store for small data. USENIX ATC 2015.
[19] Shetty P, Spillane R P, Malpani R, et al. Building workload-independent storage with VT-trees. FAST 2013.
[20] Wang P, Sun G, Jiang S, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. EuroSys 2014.
[21] Lim H, Andersen D G, Kaminsky M. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. FAST 2016.
[22] Dayan N, Athanassoulis M, Idreos S. Monkey: Optimal navigable key-value store. SIGMOD 2017.
[23] Dong S, Callaghan M, Galanis L, et al. Optimizing Space Amplification in RocksDB. CIDR 2017.
[24] RocksDB tuning guide, 2016. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.
[25] Leveled Compaction in Apache Cassandra, 2011. http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra.
[26] Redis, an open source, in-memory data structure store. https://redis.io/.
[27] Memcached, an open source, high-performance, distributed memory object caching system. https://memcached.org/.
[28] Liu Y, Song J, Zhou K, et al. Deep Self-Taught Hashing for Image Retrieval. IEEE Transactions on Cybernetics, 2018, PP(99):1-13.
[29] Balmau O, Didona D, Guerraoui R, et al. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. USENIX ATC 2017.
[30] Raju P, Kadekodi R, Chidambaram V, et al. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. SOSP 2017.