LevelKV*: Checkpoint Based Persistent Key Value Store for Key-Value SSD

ABSTRACT

This paper introduces *LevelKV**, a persistent checkpoint based key-value store with popular key-value store LevelDB as the interface. The architecture of *LevelKV** is Key-Value SSD optimized. The performance of *LevelKV** is measured by both microbenchmarks and YCSB workloads. Both results show that *LevelKV** is faster than LevelDB in every aspect. Although *LevelKV** is a bit slower than KV Stacks, a thin layer of software API provided by KV SSD, due to necessary overheads, *LevelKV** supports range queries and crash consistencies while KV Stacks does not.

KEYWORDS

key-value store, KV SSD, LevelDB, YCSB

1 INTRODUCTION

Persistent key-value stores play a critical roles in a variety of modern data-intensive applications by enabling efficient insertions, point lookups, and range queries [1]. Although Log-Structured Merge-Trees (LSM-trees) have become the state of the art for key-value stores, the success of LSM-based key-value stores are tied closely to its usage upon classic hard-disk drives (HDDs) [1]. In order for key-value stores to optimize for solid state devices (SSDs), key-value stores such as WiscKey[1] have been proposed.

Recently, another big change has been introduced to storage landscape. Key-Value SSD aims to store key value pairs at a hardware level [2]. Compared to existing block SSDs, Key-Value SSDs are fundamentally different in their performance and characteristics. Therefore just replacing a block SSD with a KV SSD underneath an LSM-tree structured key-value store will result in poor optimization as the true potential of Key-Value SSD will go hugely unrealized.

This paper presents *LevelKV**, an Key-Value SSD-conscious persistent key-value store derived from LevelDB. The central idea behind *LevelKV** is to utilize B^+-Tree instead of an LSM tree based on checkpoint techniques to truly optimize for Key-Value SSD while retaining the interface of LevelDB. The performance of *LevelKV** is compared with existing key-value store, LevelDB and Key-Value SSD provided thin layer of software interface, KV Stacks. In all aspects, *LevelKV** performs significantly better than LevelDB. Meanwhile there is an gap between the performance of *LevelKV** and that of KV Stacks due to the required overhead in *LevelKV** to support range query and crash consistency.

The rest of the paper is organized as follows. Section 2 explains the background and motivation for this paper. The design of *LevelKV** is explained in Section 3 and the result of performance evaluation is described in Section 4. Finally, the paper is concluded in Section 5.

2 BACKGROUND AND MOTIVATION

In this section, the design of LevelDB, a popular key-value store based on LSM-tree technology, is explained. Then we describe the characteristics of Key-Value SSD hardware in detail.

2.1 LevelDB

LevelDB is a widely used key-value store based on LSM-trees. LevelDB supports range queries, snapshots, and other features that are useful in modern applications. [1]

LevelDB consists of mainly 4 parts. On-disk log file, in-memory memtable and immutable memtable, seven levels (L_0 to L_6) of on-disk Sorted String Table (SSTable) files. The architecture of LevelDB is shown in Figure 1(a).

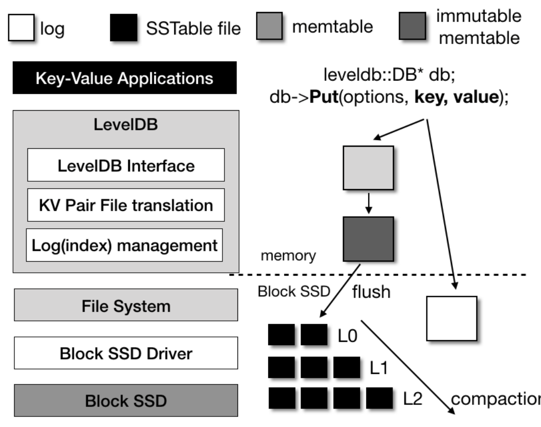

Figure 1: LevelDB Architecture and the Insertion Process

The insertion process of a key-value tuple can be described as follows: LevelDB stores key-value pair in a log file while storing in the memtable. Once the memtable is full, LevelDB switches to a new memtable and log file [1]. In the background, the previous memtable is converted into an immutable memtable, and a compaction thread then flushes it to the disk, generating a new SSTable file (about 2MB usually) at level 0 (L_0); the previous log file is discarded [1]. As seen above, LevelDB writes more data than necessary.

For point query, LevelDB searches the memtable, immutable memtable, then files then files L_0 to L_6 in order [1]. To lookup a key-value pair, LevelDB may need to check multiple levels. In the worst case, LevelDB needs to check eight files in L_0, and one file for each of the remaining six levels: a total of 14 files. Moreover, to find a key-value pair within a SSTable file, LevelDB needs to read multiple metadata blocks within the file [1].

For range queries, LevelDB supports iterator-based interface. Since keys and values are stored together and sorted, a range query

,

can sequentially read key-value pairs from SSTable files [1]. However, note that for efficiency, it is only necessary to keep the keys sorted.

Excessive reads and writes that are unnecessary has been one of the major drawbacks of the LSM-tree structured key-value store. When it comes to utilizing KV SSD, minimizing these avoidable read and write behaviors becomes one of the main goals in designing a new data structure for KV SSD.

2.2 Key-Value SSD

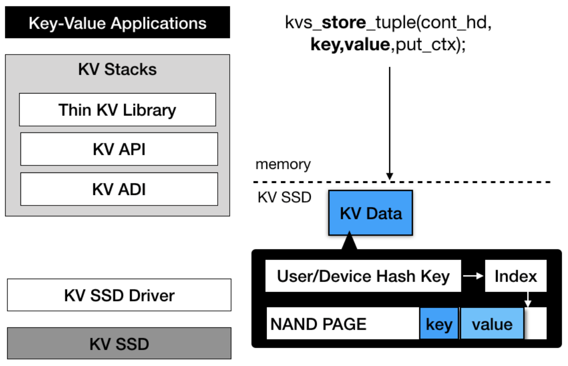

Figure 2: KV Stacks Architecture and the Insertion Process

Most of current storage devices are based on a block interface. However, rapid growth of unstructured data has led to the emergence of key-value data format.[13] When key-value data utilizes storages that support block interface, an overhead is created in the software that requires the conversion of key-value data into block data. The addition of the transformation layer leads to degradation of performance due to an increase of Write Amplification Factor (WAF). It also increases the complexity of host software. [13] KV SSDs aims to directly support key-value data format at the hardware level. Host software including KV SSD device driver, KV API (or KV Stacks) have been developed for KV SSD [10]. This standard interface includes *Store*, *Retrieve*, *Delete*, *Exist* commands. In this paper, the term 'KV Stacks' will be used to refer to the overall host software developed for KV SSD mentioned above in order to match term used in SNIA document[14]. Key-value data is directly stored through KV Stacks provided by the host. Although KV Stacks do support basic operations, there are certainly limits when it comes to using KV Stacks itself as a key-value store. As of now, KV Stacks does not support additional operations such as range queries or snapshots. Therefore the need to develop a proper host software that could substitute the state-of-the-art key-value stores arises.

3 *LevelKV**

In this section, we present the design and layout of *LevelKV**, a key-value store that is optimized for KV SSD.

3.1 Design Goals

*LevelKV** is a persistent key-value store, derived from LevelDB which can be deployed as the storage engine for a distributed key-value store. *LevelKV** aims to provide the same API as LevelDB. The architecture of *LevelKV** follows these main goals.

KV SSD optimized *LevelKV** is optimized for KV SSD devices instead of block SSDs which leads to significant increase in performance.

Major performance enhancements Internal structure of LevelDB allows redundant writes. On the other hand, LevelKV targets small read and write amplification by eliminating these redundancies resulting in drastic performance improvement. Moreover, the major performance cost of LevelDB is the compaction process, which constantly sorts SSTable files in the background [1]. Several files are read into memory, sorted, and written back during compaction, which introduces more redundancies. By utilizing Key Value SSDs, these background compactions are not needed.

LevelDB Compatible API LevelKV aims to support modern features that have made LSM-trees popular, such as range queries and snapshots [1]. Snapshots allow database status to be captured at a specific time.

3.2 Architecture and Layout

LevelKV's overall architecture is shown in Figure 3.

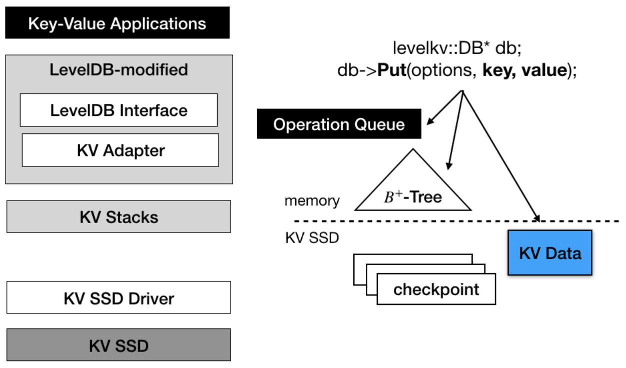

Figure 3: *LevelKV Architecture and the Insertion Process**

The main data structures are in-memory B^+-Tree, OpQueue (Operation Queue), and on-SSD CheckNodes (Checkpoint Nodes). For insertion process, *LevelKV** directly stores inserted key-value pairs in KV SSD while storing only the keys in the in-memory B^+-Tree. Insertion operation is stored on OpQueue. Only insertion operations or deletion operations are stored in OpQueue because other operations does not effect the structure of B^+-Tree that needs to be crash consistent. In the background, *LevelKV** handles checkpointing or logging the operations from OpQueue. An operation in OpQueue is first transformed into an CheckNode. Then CheckNode is flushed to KV SSD. An operations number NUM(default: 10,000 operations) is fixed for checkpointing the B^+-Tree. For every NUM operations, the keys in the B^+-Tree are checkpointed into an CheckNode and then flushed. The process of background thread is illustrated in Figure 5(d). Making CheckNodes reside on KV SSD proves to be quite a challenging task. (1)Insertion or deletion operation in OpQueue or (2)*Slice* (data type in KV SSD interface that is similar to type *std* :: *string* in c^++) type keys that were sorted in the B^+-Tree have to be transformed into a single key-value pair in order to be stored as a single CheckNode in KV SSD.

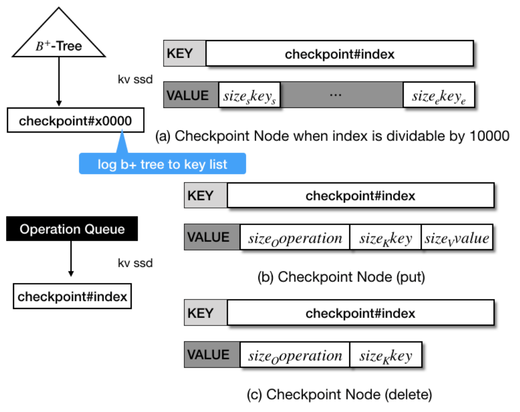

Figure 4: ***LevelKV** Transformation Process**

The transformation process is shown in Figure 4(a)(b) and (c). Keys in internal node and leaf node are sorted as well just like a normal B^+-tree for efficient range query operation.

3.3 Operation

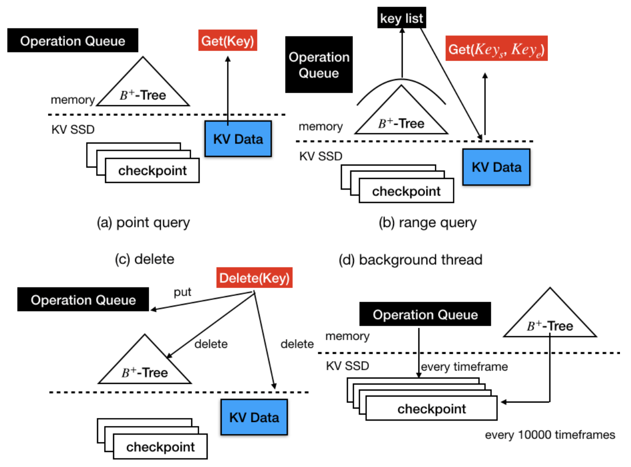

Figure 5: ***LevelKV** Insertion, Retrieval, Deletion and Background Procedure**

Insertion The insertion process of *LevelKV** first stores the key-value tuple in KV SSD. Then the key is inserted in the B^+-Tree as an index and the operation is queued in OpQueue.

Retrieval When the user queries for a key (point query), it should be directly fetched from KV SSD by using KV SSD's internal get function. It is unnecessary to look through the B^+−-tree since the key already acts as an index to the value.

When the user queries for a range of keys (range query), the range of keys first have to be searched in the B^+-Tree to get all the indexes. From there, values are retrieved (efficiently) from KV SSD device by using the internal get function. Queueing the operation in the OpQueue for retrival is unnecessary because it does not change the B^+−-tree structure that needs to be crash-consistent.

Deletion For deletion, the key has to be deleted from both the tree and SSD device. Then the delete operation is queued in OpQueue. In KV SSD, key-value tuple is deleted by using KV SSD's internal delete function.

3.4 Challenges

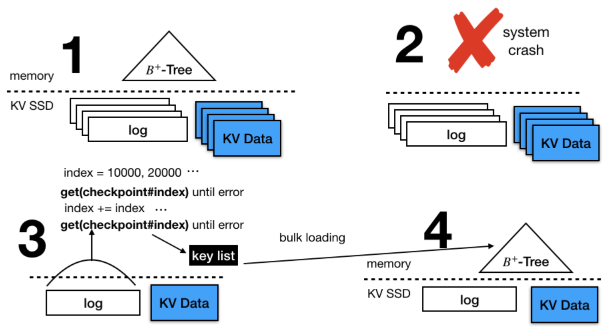

Figure 6: ***LevelKV** Recover Operation**

Crash Consistency LevelDB assures the in-order recovery of inserted key-value pairs on a system crash [12]. *LevelKV** offers similar or more crash guarantees by checkpointing B^+−-tree every fixed operation and logging insert, delete operations in the background thread.

Insert and delete operations are queued sequentially in OpQueue. Therefore, if key_s regarding operation s is lost in a crash, all keys inserted or deleted after key_s is not appended in the queue, thus not logged to KV SSD which means they also become lost. Moreover, because key-value pairs are stored or deleted in KV SSD asynchronously and sequentially, if $< key_s, value_s >$ pair is lost in a crash, all key-value pairs put into or deleted after $< key_s, value_s >$ are also lost.

When the system crashes and B^+−-tree and OpQueue structure are lost in memory, *LevelKV** recovers the structure by first calling the last indexed CheckNode that is dividable by NUM. This ChecKNode contains all the keys that the B^+−-tree structure was made up of. B^+−-tree structure is constructed again in memory using these ordered keys. Then, *LevelKV** loops through the CheckNodes until the last indexed CheckNode and adds or deletes the key to or from the B^+−-tree that was operated after checkpointing the tree.

4 EVALUATION

This section presents the evaluation results of *LevelKV**. All experiments are run on a testing machine with two Intel(R) Xeon(R)

CPU @ 2.10GHz (24 cores per CPU processors and 64-GB of memory. The operating system is 64-bit Ubuntu 18.04.3 LTS and the file system used is ext4. The storage device used is 3.84-TB SAMSUNG PM983 SSD, which has both block interface and KV interface. All workloads are carried out in a single PM983 and KV-PM983.

4.1 Microbenchmarks

KVbench (the default microbenchmarks in KV SSD) is used to draw out the performance of LevelKV and LevelDB. Modified db_bench (the default microbenchmarks in LevelDB) is attached to KVbench for evaluating the performance of LevelDB and $LevelKV^*$. A fixed size of 16B is used for key size while a combination of 512B, 2KB, 4KB in the ratio of 1:5:4 is used for value size in order not to elicit performance on a particular value size. Data compression is disabled for easier analysis.

Insertion Performance

The benchmark performs 1,000,000 operations first by inserting keys in a sequential order and then finally inserting keys in a uniformly distributed random order.

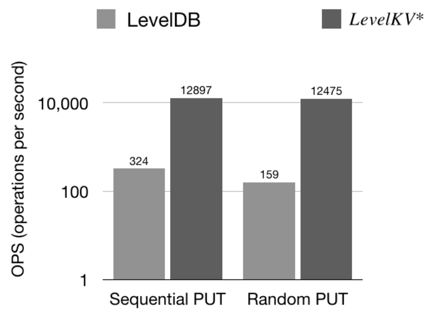

Figure 7: LevelDB and $LevelKV^*$ Insertion Performance

Figure 7 shows the throughput(ops/s) of LevelDB and $LevelKV^*$. It is shown that LevelDB's throughput is far worse than $LevelKV^*$ in both sequential and random inserts. This is due to the fact that LevelDB has to write to the log file in disk while memtable is flushed in the background. In $LevelKV^*$, usually only OpQueue is flushed in the background. It is worth noting that the larger the value size is, the more the LevelDB's per-operation efficiency gets swamped because of extra copies of large values [15]. $LevelKV^*$ shows significant improvement in performance compared to LevelDB.

Query Performance

Query benchmarks are divided into two parts, point query benchmarks and range query benchmarks. Figure 5 shows the result of the sequential and random point query benchmarks.

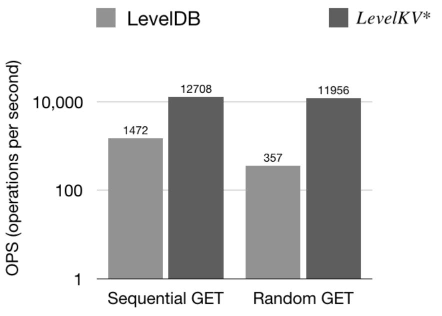

Figure 8: LevelDB and $LevelKV^*$ Point Query Performance

For point query, 1,000,000 operations are performed for each workload. As expected, the throughput of LevelDB is much worse than $LevelKV^*$. This is because of LevelDB's high read amplification mentioned in earlier sections. $LevelKV^*$ avoids any compaction process in LevelDB by removing much bigger LSM-tree and reading data efficiently through B^+-Tree that only indexes keys. However, query performance of LevelDB depends a lot on the size of DRAM [15], so the results of the query performance can vary a great deal. It is more accurate to focus on the performance of insertions.

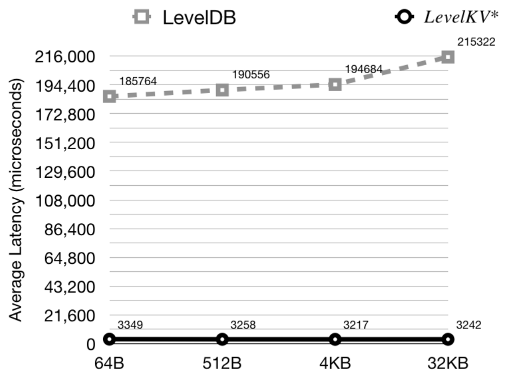

Figure 9: LevelDB and $LevelKV^*$ Range Query Performance

Figure 8 shows the average latency of range query benchmarks for LevelDB and $LevelKV^*$. For all benchmarks, 1,000,000 operations are performed for 16B keys and various sized values from 64B to 32KB. While LevelDB requires multiple files of different levels to be read in the SSTable structure, $LevelKV^*$ only needs to call range query operation which is known to be efficient to the B^+-Tree. One thing to notice is that latency increases a lot in LevelDB between 4KB and 32KB. Beyond a value size of 4KB, since SSTable file can store only a small number of key-value pairs, LevelDB's overhead is dominated by opening many SSTable files and reading the index blocks and bloom filters in each file [1]. This is why for larger values, $LevelKV^*$ performs even better.

	read	update	insert	modify	scan
A	0.5	0.5	0	0	0
B	0.95	0.05	0	0	0
C	1	0	0	0	0
D	0.95	0	0.05	0	0

Table 1: The percentage of operations in each YCSB workload.

4.2 YCSB benchmarks

In this section, the performance of $LevelKV^*$ is compared with that of KV Stacks and LevelDB through YCSB benchmark [9]. The YCSB benchmark provides a framework for evaluating the performance of key-value stores. Table 1 shows properties of each workload within YCSB. Although original YCSB benchmark includes a set of six workloads, this paper only utilizes workloads A, B, C, D since workload E, F includes scan operations that cannot be performed on KV Stacks. KV Stacks, LevelDB, $LevelKV^*$ are evaluated with 16B-4KB sized key-value pairs, and data compression is disabled. Zipf distribution is used for all of the workloads.

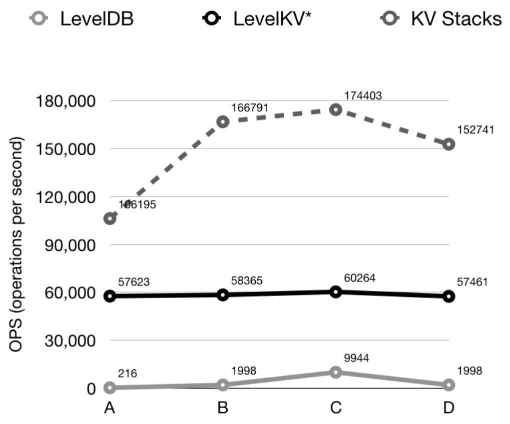

Figure 10: LevelDB, KV Stack and $LevelKV^*$ YCSB Performance

KV Stacks performs significantly better than the other two key-value stores. This is due to the fact that KV Stacks is a lightweight interface for KV SSD that does not support complicated operations such as range queries. For KV Stacks, key-value data are directly inserted to, retrieved from, and deleted from KV SSD. The overhead for creating and utilizing B^+-Tree and OpQueue structure slows down the performance of $LevelKV^*$ compared to KV Stacks. However, compared to LevelDB, $LevelKV^*$ still performs significantly better.

5 CONCLUSION

This paper aims to develop a proper host software for KV SSD that could substitute state-of-the-art key-value stores such as LevelDB. For this reason, the developed persistent key-value store $LevelKV^*$ provides compatible API with LevelDB and supports all of the important functions in LevelDB including not only insertions, deletions but also range queries. By utilizing KV SSD instead of a block SSD, the performance of $LevelKV^*$ increased drastically compared to state-of-the-art key-value store LevelDB. For future work, snapshots can be developed in $LevelKV^*$ using checkpointing methods in this paper and making B^+-Tree structure into a copy-on -write B^+-Tree structure. This paper leaves this as a future work.

REFERENCES

[1] Lu, Lanyue, et al. "Wisckey: Separating keys from values in ssd-conscious storage." ACM Transactions on Storage (TOS) 13.1 (2017): 1-28.
[2] Kang, Yangwook, et al. "Towards building a high-performance, scale-in key-value storage system." Proceedings of the 12th ACM International Conference on Systems and Storage. 2019.
[3] Li, Yinan, et al. "Tree indexing on solid state drives." Proceedings of the VLDB Endowment 3.1-2 (2010): 1195-1206.
[4] Roh, Hongchan, et al. "AS B-tree: A Study of an Efficient B+-tree for SSDs." J. Inf. Sci. Eng. 30.1 (2014): 85-106.
[5] Ghemawat, Sanjay, and Jeff Dean. "LevelDB, A fast and lightweight key/value database library by Google." (2014).
[6] Dent, Andy. Getting started with LevelDB. Packt Publishing Ltd, 2013.
[7] Lim, Hak-Su, and Jin-Soo Kim. "Leveldb-raw: Eliminating file system overhead for optimizing performance of leveldb engine." 2017 19th International Conference on Advanced Communication Technology (ICACT). IEEE, 2017.
[8] Ahn, Jung-Sang, et al. "*-Tree: An ordered index structure for NAND flash memory with adaptive page layout scheme." IEEE Transactions on Computers 62.4 (2012): 784-797.
[9] Cooper, Brian F., et al. "Benchmarking cloud serving systems with YCSB." Proceedings of the 1st ACM symposium on Cloud computing. 2010.
[10] KVSSD. https://github.com/OpenMPDK/KVSSD.
[11] YCSB. https://github.com/brianfrankcooper/YCSB.
[12] LevelDB. https://github.com/google/leveldb.
[13] KVSSD wiki. https://github.com/OpenMPDK/KVSSD/wiki.
[14] Samsung key value SSD enables high performance scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.
[15] LevelDB Benchmarks. http://www.lmdb.tech/bench/microbench/benchmark.html.