레벨KV: 키-밸류 SSD를 위한 B^{+}-트리 기반의 퍼시스턴트 키-밸류 스토어

LevelKV: A Persistent B^{+}-Tree Based Key-Value Store for Key-Value SSD

Abstract

This paper introduces LevelKV, a persistent b^{+}-tree based key-value store with popular key-value store LevelDB as the interface. The architecture of LevelKV was designed to optimize Key-Value SSD. The performance of LevelKV is measured by both microbenchmarks and YCSB workloads. Both results show that LevelKV performs much better than LevelDB in every aspect. Although LevelKV is a bit slower than *KV Stacks*, which is a thin layer of software provided by KV SSD to access KV SSD, this is because some overheads are necessary to support range query operations that simply do not exist in KV Stacks.

1. Introduction

Persistent key-value stores play a critical role 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, KV 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 KV 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 utilizing b^{+}-tree instead of an LSM tree to truly optimize for KV SSD while retaining the interface of LevelDB. The performance of LevelKV is compared with existing key-value store, LevelDB and KV SSD provided software layer, KV Stacks. In all aspects, LevelKV performs significantly better than LevelDB. Meanwhile there is some amount of gap between the performance of LevelKV and that of KV Stacks, but this is inevitable due to required overheads needed to support range query operations.

2. Background and Motivation

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.

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

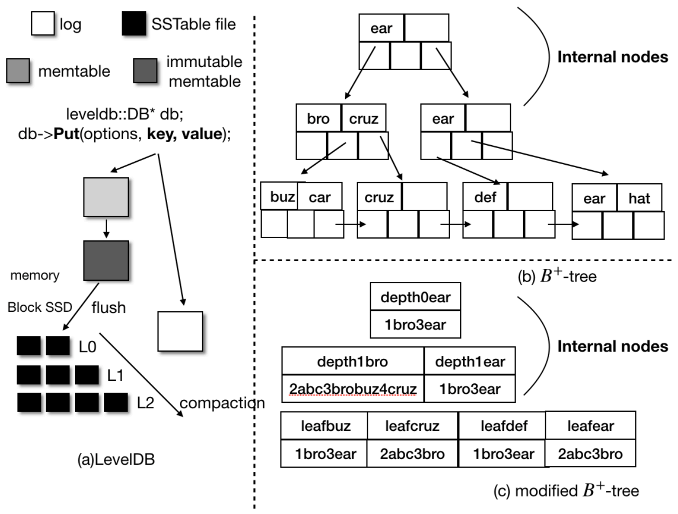
Most of current storage devices are based on a block interface. However, rapid growth of unstructured data has led to the emergence of key-value data format. [13] KV SSD aims to directly support key-value data format at the hardware level.

Host software has been developed for KV SSD called KV Stacks. KV Stacks include standard KV API such as *store, retrieve, delete, exist* commands. However, there are certainly limits when it comes to using KV Stacks itself as a key-value store since KV Stacks does not support additional operations such as range queries. The need to develop a proper host software or a key-value store for KV SSD that supports range queries arises.

2.3 B^+-tree

The easiest form of data structure one can think of when keeping a simple index structure in memory is a sequential list. However, the performance of sequential index data structure degrades rapidly as the size of the list grows for all operations. For this reason, the need for a more sophisticated organization such as a tree data structure arises.

B^+-Tree is one of the most popular index structures for managing a large amount of records [8]. B^+-tree provides efficient insertion, deletion, and retrieval operations with logarithmic access time. The performance of a -Tree is ensured because B^+- Tree takes the form of a balanced tree, where every path from the root of the tree to the leaf of the tree is of the same length.



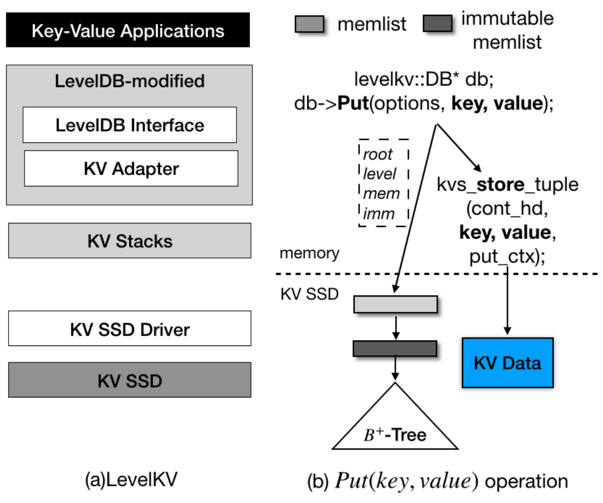
[Figure 1] Architecture of LevelDB and second order B^+-tree

3. LevelKV

3.1 Design Goals

Although current LevelKV does not support snapshots, the final goal of LevelKV is to fully provide the same API as LevelDB. For now, providing support for snapshots will be left for future work.

LevelKV is designed to be a persistent key-value store optimized for KV SSDs. LevelKV aims to induce major performance enhancements not only in range query operations but also in basic operation including *put*, *get* by removing the redundant writes and reads that happens in LevelDB. LevelKV also aims to support crash consistency by assuring the in-order recovery of inserted key-value pairs on a system crash.

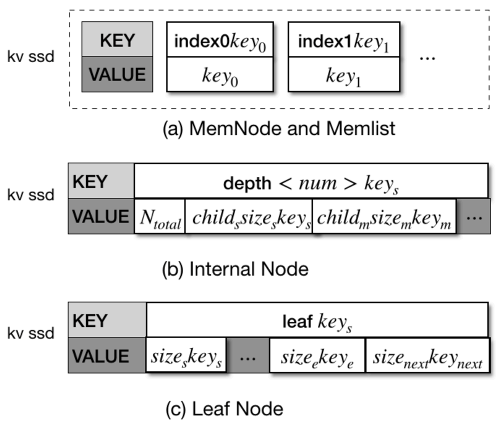


[Figure 2] Architecture of LevelKV

3.2 Architecture

LevelKV' s overall architecture is shown in Figure 2. The main data structures are two in-SSD sorted lists (memlist and immutable memlist), and modified B^+-tree.

For insertion process, LevelKV directly stores inserted key-value pairs in KV SSD while storing only the keys in the memlist. If the memlist reaches a certain limit (preset to 32KB), LevelKV switches to a new memlist to handle additional inserts. In the background, previous memlist is first transformed into an immutable memlist. Then a thread that handles bulk *appending* inserts keys in the B^+-tree. *Bulk appending* is a technique used to make insertion more efficient. Such optimization technique will be explained in the next section. Making memlists and B^+-tree reside on a KV SSD proves to be a quite challenging task. We have to modify the layout of the nodes in lists and trees like the ones shown in Figure



3.

[Figure 3] Layout of Nodes in KV SSD

When the user queries for a key, it should be directly fetched from KV SSD. When the user queries for a range of keys, LevelKV first have to search the lists and tree structure to get all the indexes. From there, values are retrived form KV SSD device by using internal *get* function. We delete the keys from both the lists-tree structure and from the data residing in disk when deletion is called.

3.2 Optimizations

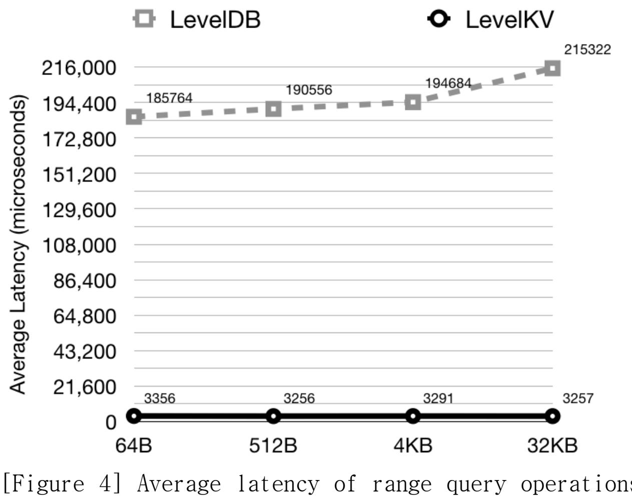
Sorted-based Bulk Appending For each *put* operation, LevelKV needs to append the key to B^{+}-tree. However, issuing a large number of inserts to the tree can add significant overhead. A technique called bulk appending tries to reduce this overhead first by sorting the keys in a memlist. Then the memory buffer that memlist transformed into is inserted in the tree when it reaches a certain size limit. For lookup operation, LevelKV first searches the memlist, then tree is searched.

Tree Node Caching A cache that stores 4KB items is used to store B^{+}-tree nodes in memory for efficient lookup. The cache implemented has internal node synchronization and may be safely accessed concurrently from multiple threads. A least-recently-used eviction policy is implemented.

4. Evaluation

4.1 Microbenchmarks

KVbench (the default microbenchmarks in KV SSD) is used to draw out the performance of LevelKV and LevelDB but not KV Stacks since KV Stacks does not support range queries. Modified *db_bench* (the default microbenchmarks in LevelDB) is attached to KVbench for evaluating the performance of LevelDB and LevelKV for range queries. Data compression is disabled for easier analysis.



[Figure 4] Average latency of range query operations for different value sizes

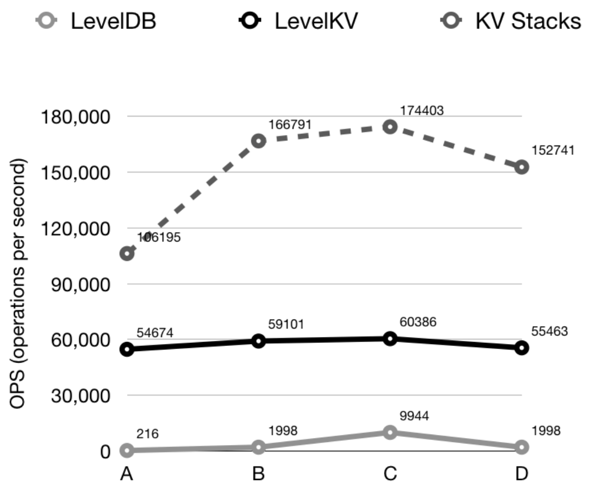
Range Query Performance Figure 4 shows the average latency of range query benchmarks for LevelDB and LevelKV. For all benchmarks, 1,000,000 operations are performed for 16B keys and various sized values from 64B to 32KB. While LevelDB requires multiple files of different levels to be read in the SSTable structure, LevelKV only needs to call range query operation to the tree structure which was optimized for performance. 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.

4.2 YCSB Benchmarks

In this section, the performance of LevelKV is compared with that of KV Stacks and LevelDB through YCSB benchmark. 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.

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

[Table 1] The percentage of operations in each YCSB workload



[Figure 5] Average throughput for YCSB workloads

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. For KV Stacks, key-value data are directly inserted to, retrieved from, and deleted from KV SSD. The overhead for creating and utilizing the tree slows down the performance of LevelKV compared to KV Stacks. However, compared to LevelDB, LevelKV still performs significantly better.

5. Conclusions

This paper aims to develop a proper host software for KV SSD that supports range queries. LevelKV was built to provide compatible API with LevelDB and support all of the operations in LevelDB. However, this paper leaves supporting snapshots for future work.

With the overhead for the data structures to support range queries, LevelKV still outperforms LevelDB by a significant amount. This paper shows the possibility of one day fully replacing block SSD with KV SSDs when utilizing key-value stores.

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. "μ*-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.