Improved Key-Value Store

2019-2nd Semester

Interdisciplinary Project

Student ID	20141440
Name	SeungWon Lee
School	ECE
1 track	CSE
2 track	EE
Advisor 1	Young-ri Choi
Advisor 2	Jongwon Lee

FIRST IN CHANGE

I. Introduction

1) Topic and Purpose of Research

In recent days, key-value (KV) store is used at various computer science fields such as SNS, web indexing and cloud services. Thus, it is important to make KVs efficient to modern devices as much as possible. There are many cases that existing KV store system does not fully exploit state-of-the-art devices such as Persistent Memory (PM) or Solid-State Drive (SSD).

The purpose of this research is to optimize KV store to modern computer systems using state-of-the-art devices. There are many important parts in KV store, but I will explain KV store in three large sections; memory, storage (disk) and KV architecture itself. Single-Level Merge DB (SLM-DB) will be described in memory part. In structure part, Scalable Low-Latency Indexes for a Key-Value Store (SLIK) that utilizes secondary indexing will be described. Lastly, in storage part, KVell which exploits Non-Volatile Memory Express (NVMe) Solid-State Drive (SSD) will be introduced. All of three parts have common goals such as durability, low latency or consistency. There are some trade-offs for achieving all goals but I will introduce how to minimize such trade-offs.

First part is memory. Single-Level Merge DB (SLM-DB) is designed for leveraging byte-addressable persistent memory (PM) to improve the performance of KV stores. It is made of the advantages of both B+-tree index and the Log-Structured Merge Trees (LSM-tree). B+-tree index is a structure optimized for reading items since leaf nodes of B+-tree structure are linked each other by linked lists. On the other hand, LSM-tree is optimized for writing items as a batch to a disk using buffer. Thus, SLM-DB exploits a good read ability from B+-tree and write ability from LSM-tree. SLM-DB, literally, has a single-level disk. It doesn't have to sort KV pairs on disk because it can utilize B+-tree for searching. However, SLM-DB should provide some degree of sequentiality for range queries. Therefore, SLM-DB also performs restricted merge for KV pairs called the selective compaction scheme.

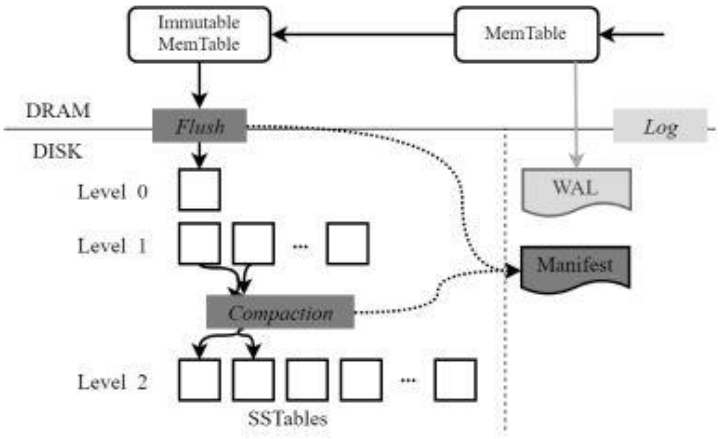
Second is an architecture of KV store, especially secondary indexing. Existing large-scale KV storage systems sacrifice secondary indexing or consistency for achieving other performance such as scalability or performance. Thus, Scalable Low-Latency Indexes for a Key-Value Store (SLIK) is designed for achieving low latency, high scalability, consistency and high availability while using secondary indexing. It provides higher performance even with indexes that span hundreds of servers while also providing consistency. Furthermore, it gives low-latency by using simple mechanisms for KV store. It also provides fast crash recovery live index split and migration with high level of availability.

Last is a storage part. Although, Solid State Drive (SSD) is commercialized to the public, many persistent KV stores were designed targeting old-generation storage devices. They avoid random accesses, keep data sorted on disk and synchronization which induces CPU to be bottlenecks. However, KVell presents a new paradigm for KV design. It crashes the stereotype designs of KV store by flipping them. No attempt is made at sequential access, and data is not sorted when stored on disk. Shared-nothing scheme is adopted to avoid synchronization overhead. Also, no commit log is made in KVell. It shows how such design can help KV store get better performance.

2) Background

1) Single-Level Merge DB (SLM-DB)

SLM-DB is implemented based on LevelDB and exploits the properties of B+-tree, LSM-tree Persistent Memory (PM). Thus, I will explain each of them.



(Figure 1) Level DB structure

-Level DB

LevelDB is a popular KV store that uses the LSM-tree for internal data structures. The LSM-tree has two main parts: MemTable and Immutable MemTable which are located at DRAM and multiple levels of Sorted String Table (SSTable) which is located in persistent storage such as disk. (Figure 1)

MemTable simply receives inserted KV pairs and keep them until it becomes full. When MemTable becomes full, LevelDB changes MemTable to an Immutable MemTable and makes a new MemTable. Then a background thread flushes KV pairs which is recently inserted Immutable MemTable to the disk. In disk, KV pairs are stored in sorted order. To provide durability, KV pairs are appended to the write ahead log (WAL) first, and then added to MemTable. After the writing is finished, the log is deleted.

LevelDB has multiple levels of SSTables on the storage. Size of level gets larger as the level goes up. Mechanism is similar with the mechanism between MemTable and Immutable MemTable. Once the size of level x becomes full, a background compaction thread selects one of the SSTable file in level x. Then the KV pairs in the SSTable flushes into level x+1 while the thread also performs merge sort with other SSTable files in that level. After the compaction is done, the result if written in the MANIFEST file which resides on disk, and then old SSTable files are removed. Thus, it provides durability and consistency even when there is crash during compaction.

LevelDB has some limitations. First, the performance of read operations is slow. To read something, LevelDB first searches a key in MemTable and then Immutable Table. If there is not a key in both of them, then it searches the key through the multiple level of the system until the key is found. Second, even LevelDB has a good write performance using buffer, but it has high write and read amplification. It's because an inserted KV needs to be merge-sorted and written to the disk. These limitations are what SLM-DB tried to overcome.

-B+-tree

B+ tree is one type of the B-tree. It has similar properties. However, in B+ tree, only leaf nodes have records. Internal nodes just have keys. In addition, leaf nodes are linked each other by linked list, which means they can easily handle the sequential process. B+ tree is suitable for read-intensive workloads but have high write amplification.

-Log Structured Merge Tree (LSM-tree)

LSM-tree is a data structure that is optimized to provide indexed access to files with high insert volume, such as transactional data. Unlike B+ tree, it has a good performance for writing because it buffers keys and values in memory and then writes them as a batch to a disk.

-Persistent Memory (PM)

Persistent memory is any method or apparatus for efficiently storing data structures such that they can continue to be accessed using memory instructions or memory APIs even after the end of the process that created or last modified them.

2) KVell

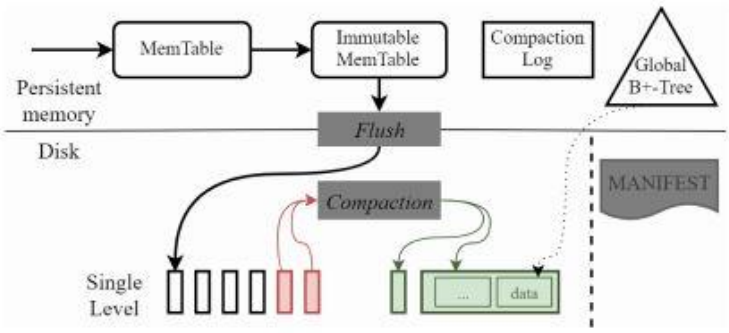
-Non-Volatile Memory Express (NVMe)

NVMe is a logical device interface specification for accessing non-volatile storage media attached via a PCI Express (PCIe) bus.

II. Main Subject

1) Research

1) Single-Level Merge DB (SLM-DB)



(Figure 2) SLM-DB structure

As described on introduction part, SLM-DB is designed to improve the performance of KV store by leveraging persistent memory (PM) to store MemTable and Immutable MemTable, while adopting B+-tree and LSM tree also. Single-Level Merge DB (SLM-DB) is implemented based on LevelDB but its structure is little different. Figure 2 shows the structure of SLM-DB.

Unlike LevelDB, SLM-DB has no write ahead log (WAL) because persistent memory is persistent literally. Thus, it provides durability and consistency upon system failures. SLM-DB only have a single level for SSTables, which means there is no need to multiple write in the disks resulting in decreasing write amplification. SLM-DB has a global B+-tree in PM to maximize the performance of read operations on a single-level organization of SSTable files. That is, data don't have to be sorted on the disk because it finds KV pairs from B+-tree.

SLM-DB performs a selective compaction scheme to provide reasonable range query performance. Selective compaction scheme selectively merges SSTables. The information of on-going compaction is backed up by the compaction log in PM, thus it has consistency against system failure.

-Persistent MemTable

MemTable is a persistent skiplist in SLM-DB. Skiplist operations are performed by 8-byte write operation. To keep consistency of KV store in MemTable, first, it makes a new node where its next pointer is set by calling memory fence and cacheline flush instructions. The entire algorithm is shown Figure 3. Since MemTable resides on PM, it provides durability without using WAL. Also, since skiplists can be reconstructed easily from the lowest level, there is no need for mechanism for guaranteeing consistency.

Algorithm 1 Insert(*key*, *value*, *prevNode*)

```
1: curNode := NewNode(key, value);
2: curNode.next := prevNode.next;
3: mfence();
4: clflush(curNode);
5: mfence();
6: prevNode.next := curNode;
7: mfence();
8: clflush(prevNode.next);
9: mfence();
```

(Figure 3) Algorithm for skiplist insert operation

-B+-tree Index in PM

SLM-DB adopts a B+-tree index to make a good performance for searching a KV pair in SSTables. When flushing a KV pair in Immutable MemTable to the disk (SSTable), the key is stored at the leaf node of the B+-tree. It includes a pointer that points a PM object which contains the information about where the KV pair is located on the disk. SLM-DB makes two background threads when performing a flush operation, one for file creation and the other for B+-tree insertion. File creation thread creates a new SSTable file and flushes KV pairs from Immutable MemTable to the file. Then, the KV pairs are added to a queue which is created by a B+-tree insertion thread. It moves KV pairs to B+-tree one by

one. After the all works are done, the change of the organization is added to MANIFEST files as a log, and then SLM-DB deletes Immutable MemTable.

To keep a key in fresh value, if a key already exists in the B+-tree, a fresh key is written to a new SSTable and an obsolete location objects will be garbage collected by a persistent memory manager.

SLM-DB provides a B+-tree iterator to help users scan keys stored in SSTable files in a sequential manner.

-Selective Compaction

To improve sequentiality and collect garbage of obsolete KV store, SLM-DB provides a selective compaction. To support the selective compaction, it maintains a compaction candidate list of SSTables. A compaction thread selects a subset of SSTables from the candidate list.

Similar with flush operation in B+tree index, the compaction process is done by two threads, one for file creation and the other for B+-insertion. However, unlike flushing Immutable MemTable to the disk, the file creation threads should create multiple SSTable files. It's because the file creation thread does not stop until the compaction is completed. It creates a new file continuously while the insertion thread concurrently stores KV pairs in B+-tree using queue. Once the compaction is done, the change of SSTable metadata is written to the MANIFEST file and the old SSTable files are deleted.

For selecting candidate of SSTables for compaction, SLM-DB uses three criteria: the live-key ratio, the leaf node scans and the degree of sequentiality per range query. First, for live-key ratio criterion, SLM-DB maintain the ratio of valid KV pairs to all KV pairs. If the ratio for some SSTable is lower than the threshold, then SLM-DB decides that the SSTable contains lots of garbage.

Second, for leaf node scan criterion, SLM-DB scans B+-tree leaf nodes for a certain fixed number of keys in a round-robin manner. As a result of scan, if the number of unique files is larger than the threshold, those files are selected to the compaction candidate list.

Third, for the degree of sequentiality per range query, SLM-DB divides a queried key range into multiple subranges. Once the range query operation is done, SLM-DB find which sub-range has the maximum number of unique files. If the number of unique files is larger than the threshold, those files are added to the compaction candidate list.

-Crash Recovery

SLM-DB guarantees consistency from a crash for in-memory data in PM, on-disk data and metatdata in SSTables. SLM-DB implements the linked list of the lowest level of the skiplist to be guaranteed to be consistent with 8 bytes atomic manner. Unlike LevelDB, no effort is needed to log like Write Ahead Log (WAL). SLM-DB uses MANIFEST file when it performs the recovery process.

2) Scalable Low-Latency Indexes for a Key-Value Store (SLIK)

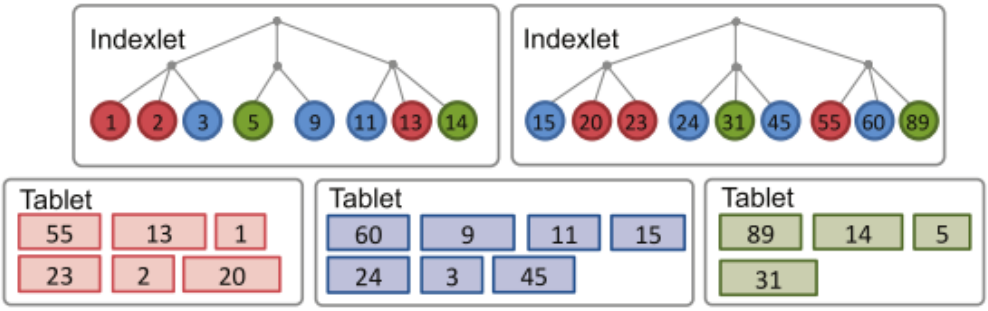
SLIK is a large-scale key-value storage system which provides secondary indexes while also achieving low latency, high scalability, consistency and high availability. SLIK exploits B+ tree structure to store indexes. The data model of SLIK is a multi-key-value store which allows each object to have multiple secondary keys. More precisely, indexes and data are stored in separated servers in SLIK. It adopts ordered write approach and provide garbage collection and multi-object update for consistency. For durability, SLIK uses backup approach which is the backup data of nodes of B+ tree. It also provides novel method for index creation, live index split and migration for scalability.

-Data Model

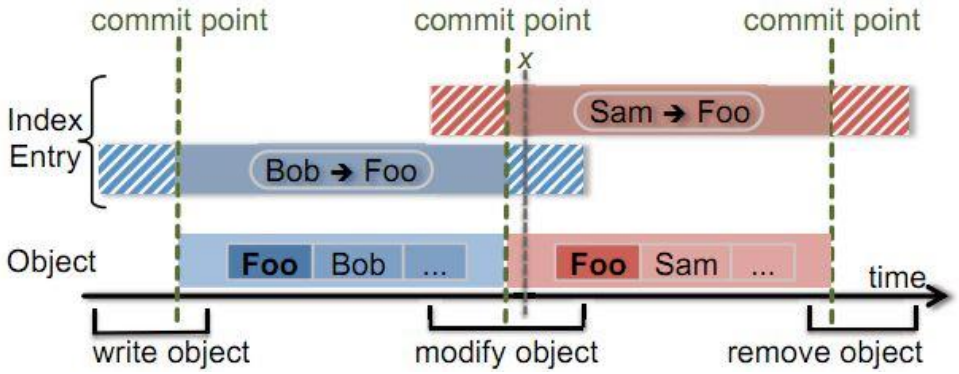
To adopt secondary index architecture, the system should decide where the secondary keys are located in objects. One common method is to store the keys as part of the object's value. However, SLIK does not use this method since this approach induces additional overhead when the server parses objects. Since an object structure can directly identify all the secondary keys, there is no need to parsing when performing index operations. This format is called multi-key-value format. In this structure, an object consists of one or more variable-length primary keys and variable-length uninterpreted value that is secondary indexes which need not be unique within the table.

-Index Partitioning

Since SLIK targets for large-scale storage systems, tables should be so large that neither the objects nor the indexes fit on a single server. Thus, index set should be able to be split and stored on a different server. It is called indexlet. There are several ways how to compose a table partition (tablet) and an indexlet. Among them, SLIK choose independent partitioning approach. The approach is shown in Figure 4. In this approach, index entries and objects are stored in independent servers. It enables one indexlet to have all keys which are in specific range. Thus, it can handle small range index query in single server. By this method, SLIK provides nearly constant and low latency irrespective of the number of servers it spans.



(Figure 4) Independent Partitioning



(Figure 5) Ordered Write Approach

-Consistency During Normal Operations

Even if indexes and objects are not on the same server in reality, SLIK gives clients a view as if they are on the same server. That is, SLIK guarantees two consistency properties. One is if an object contains a given secondary key, then an index lookup using the key will return the object. The other is if an object is returned by an index lookup, then it means the object must contains a secondary key for that index within the specified range. SLIK satisfies first property by ordered write approach and second property by treating indexlets as hints and objects as ground truth to decide the liveness of indexlets.

To satisfies first property, ordered write approach performs operations like following method. When a client issues a write request, a server, first of all, makes index entries which is corresponding to each of the secondary keys in the new object's data. Next, it writes the object and replicates it durably. Finally, if the index entry was an overwrite, then it removes old index entries. Therefore, if an object exists, then it means there should be corresponding index entries. In this way, SLIK ensures that the lifespan of each index entry spans that of the corresponding object.

Second property is guaranteed by viewing index entries as hints and an object by the truth. There can be case that SLIK issues request operation and gets key but there is no corresponding object. In this kind of situation, the SLIK client library recognizes this inconsistency by rechecking the actual index key present in each object. Only object with valid keys in the given range will be returned to the application. For example, at point x in Figure 3, the old entry (Bob) will be removed during lookups.

-Consistency after Crashes

SLIK also guarantees consistency after crashes. There are two cases for crash, one is the case for between after inserting an index and before updating the object, and the other case is while inserting or removing an entry.

For the first case, if a crash occurs between the after inserting an index and before updating the object, a background process performs garbage collect for entries which are affected by crash. Occasionally, the process scans this information and reconstructs items. There could be some overhead for storing garbage but, actually, it is so small that it is usually less than 3 KB per server per year, which means its size is negligible.

The second case is case that a crash occurs while inserting or removing an entry. To prevent such situation, SLIK makes multi-object updates occur atomically. That is, there can be only two cases, either all updates are completed or no updates are completed.

-Backup Approach

SLIK keeps B+ tree entirely in DRAM to provide low-latency, but it is dangerous that important data is on RAM. Thus, to guarantee durability, SLIK adopts backup approach. It uses backup table which is a sort of backup space for data of B+ tree nodes. The backing table is invisible to clients and has a single tablet. Each node of B+ tree is represented by one object in the backing table. Once the table becomes available, the indexlet is fully functional.

The backup approach has another advantage as well as durability. It enables nodes in B+ tree to have variable-size, which results in eliminating internal fragmentation and simplifying allocation.

-Large Scale Operation

Large-scale long-running operations must not block other operations to maximize scalability. That is, the time for using lock should be minimized. SLIK considers two large-scale operations.

First is index creation. When a new index is created, the entire table needs to be scanned which requires lock. SLIK uses lock only during creating an empty indexlet. If there is a crash, it can be restarted from the beginning. Similar with index creation, index deletion works similarly to index creation.

Second large-scale operations are live index split and migration. The ability for splitting or migrating indexlets when it is too large is required for SLIK. During migration, SLIK keeps track changes since the copying started and transfers those as well. Thus, lock is only required for a short duration for the last mutation.

Through the above two methods, SLIK provides high scalability.

3) KVell

In the past, storage devices had different performance in between random and sequential access. Thus, it is so natural that KV store exploits those properties. It usually avoids random access, keep data sorted and synchronize the items. However, technology improved so much and so do storage devices. KVell utilizes state-of-the-art storage devices called NVMe SSDs. It provides much higher bandwidth than old-generation devices and similar performance for random and sequential access.

Even using new-generation storage devices, many KV stores are optimized to old-generation ones. In such mismatch situation between modern storage devices and KV stores, too much optimization for KV store is not helpful rather harmful. In fact, researches found that CPU is the bottleneck of performance, not the storage device. Thus, there is need to change the design of storage device in terms of storage devices.

KVell tries to flip a fixed idea for designing KV stores and proposes a new paradigm. Four ideas will be described from the following contents.

-Share Nothing

To eliminate the overhead of synchronization, KVell makes worker threads handle requests for a given subset of the keys and maintain a thread-private set of data structures.

The key data structures consist of four main parts. First is a lightweight in-memory B tree index. It keeps track of information about keys' location in persistent storage. Second part is I/O queues. I/O queues are responsible for storing and retrieving information from persistent storage. Third, there is a free list which contains free spaces to store items. Fourth part is a page cache. KVell does not depend on OS-level constructs. It uses its own internal page cache.

However, this approach has a disadvantage. It can lead to load imbalance since worker threads do not know each other's loads, but researcher found that the effects are low with proper partitioning.

-Do Not Sort Data on Disk

Keeping data sorted on disk is good for sequential read. However, it has a high write amplification and maintaining those data is also expensive operation. Since modern SSD has similar performance for random and sequential access, there is no need to keep data sorted.

Thus, Kvell do not sort data on disk. It enables the items on the disk to be written at their final location. It reduces the overhead for writing items to the disk and eliminates CPU overhead for maintaining data to be keep sorted.

-Aim for Fewer Syscalls, Not for Sequential I/O

As described previously, there is no need to stick to sequential I/O at all. That's why all operations in KVell are random access. KVell batches I/O requests to reduce syscalls, which results in reducing CPU-overhead.

Batching comes with a trade-off. Disks should be kept busy at all times to reach maximum IPOS. A good system should issue enough requests to keep disks busy, but has to sure that it does not overwhelm them with large queues of requests which lead to high latency.

Due to 'share nothing' scheme, a worker does not know how many requests other workers have. Thus, it is important to limit the number of pending requests per disk. To limit it, each worker of KVell only stores files on one disk.

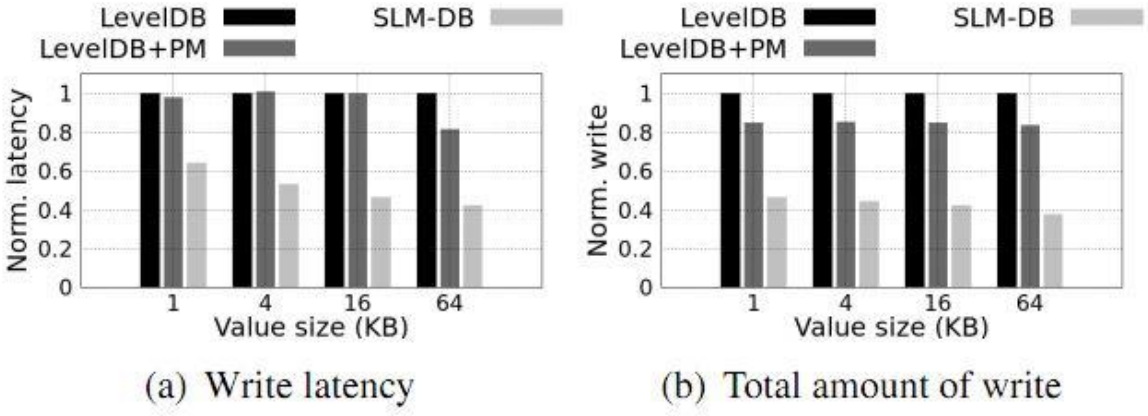
-No Commit Log

KVell admits updates only after items are persisted to disk at their final location. An update is persisted to the disk in the next I/O batch after it is submitted to a worker thread. Since KVell does not use commit log, disk can have higher bandwidth than before.

III. Conclusion and Discussion

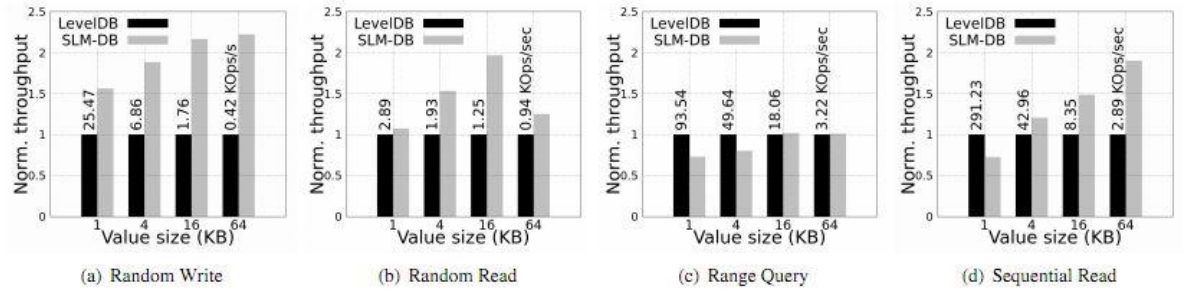
1) Results

1) Single-Level Merge DB (SLM-DB)



(Figure 6) Random Write Performance Comparison

Figure 6 shows the performance for random write of LevelDB, LevelDB+PM and SLM-DB. LevelDB + PM shows slightly good performance than LevelDB, but SLM-DB shows best performance almost more than 2 times than LevelDB. As the size of value gets larger, the performance gap also gets larger.



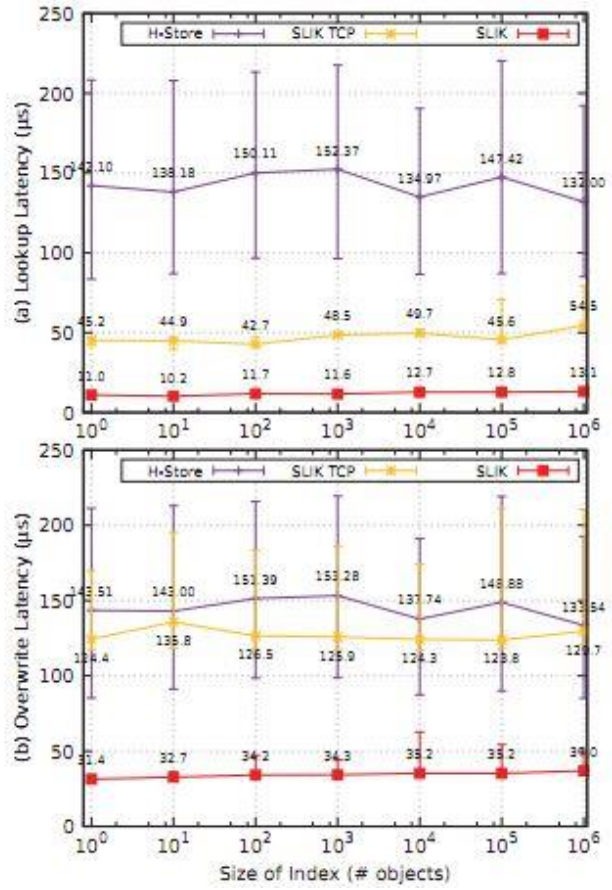
(Figure 7) Throughput of SLM-DB Normalized to LevelDB

Figure 7 shows the throughput for four cases: random write/read, range query and sequential read. All four cases show the same trend that as the size of value becomes larger, the performance of SLM-DB dramatically grows too, except 16 KB in random read case.

In random case, no matter how the size of value is, SLM-DB always shows better performance. It's because SLM-DB consists of only a single level of SSTables, which reduces the write amplification.

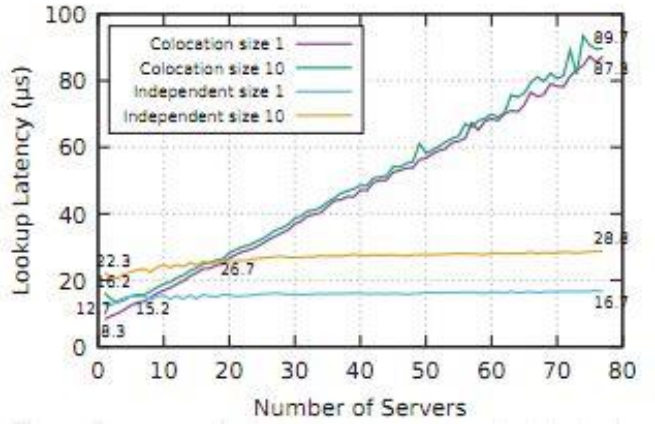
In range query and sequential cases, it shows somewhat different pattern compared to random case. When the size of value is too small, LevelDB shows good performance than SLM-DB. However, SLM-DB gets better performance as the value size grows.

2) Scalable Low-Latency Indexes for a Key-Value Store (SLIK)



(Figure 8) Overwrite & Lookup Latency with the size of index

Figure 8 compares H-Store, SLIK TCP and SLIK, which all three systems use in-memory method for KV stores. The upper graph is the case when reading a single object using a secondary key. The below one is the case when overwriting an existing object. Compared to other systems, SLIK shows significantly lower latency.



(Figure 9) Lookup Latency with The Number of Servers

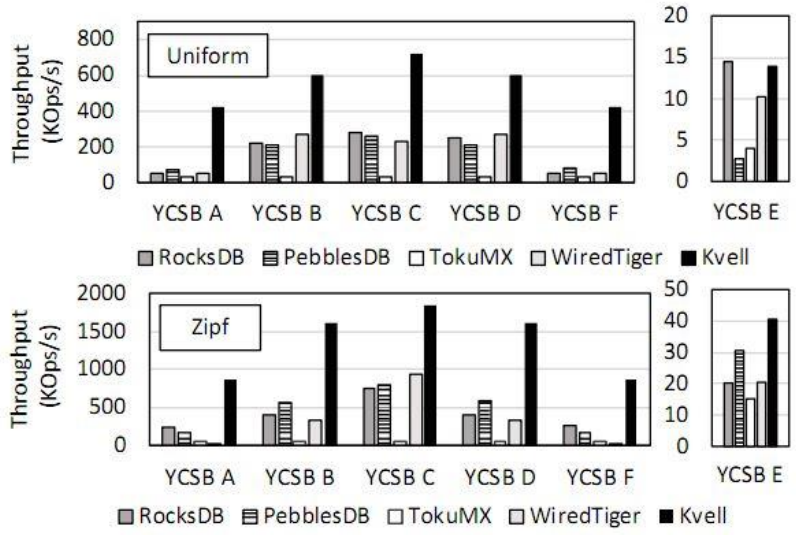
Figure 9 shows the lookup latency according to the number of servers. When the number of servers is small, colocation approach shows better performance, but as the number of servers grows up, its performance dramatically gets worse. Through the graph, we can find that index partitioning has much higher scalability than the colocation approach. Thus, for large-scale KV store systems, index partitioning approach is better choice to improve scalability.

3) KVell

YCSB A, F : write-intensive workload

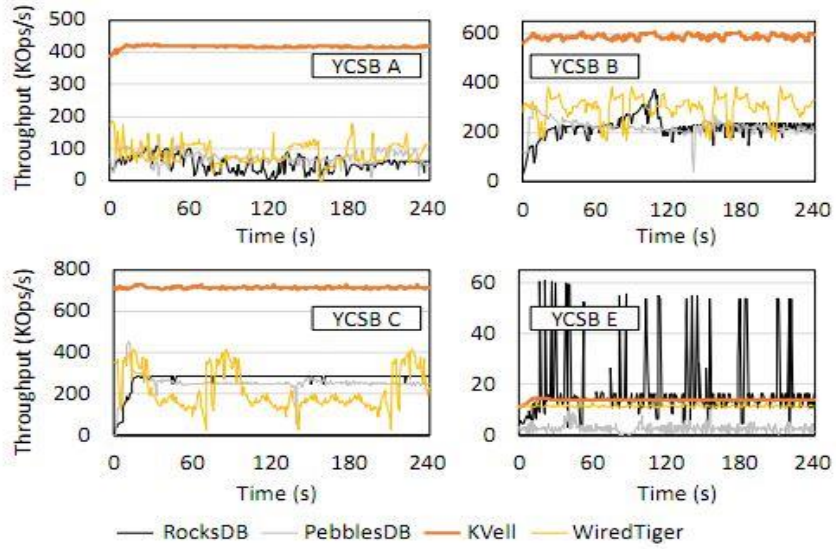
YCSB B, C, D : read-intensive workload

YCSB E : scan workload



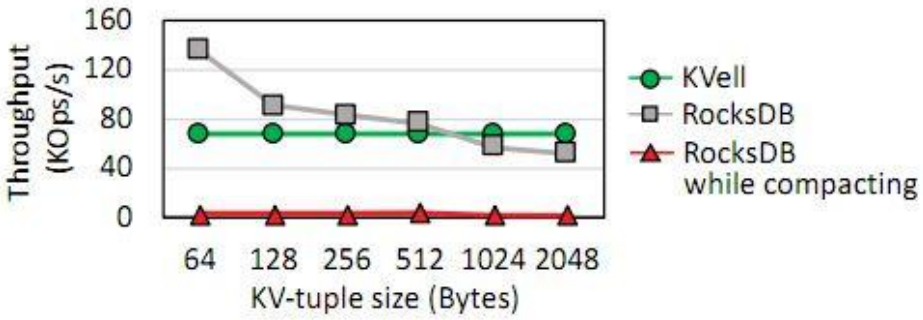
(Figure 10) Average Throughput for YCSB workloads.

Figure 10 shows the average throughput according to various type of workloads. As expected, KVell performed well on write-intensive workloads because it does not sort the items on disk. Kvell even shows better performance for read-intensive workload than other systems. It's because the random access performance of modern SSDs have improved to a similar level as sequential access. In addition, amazingly, KVell also shows a good performance with scan workload.



(Figure 11) Throughput Timelines

Figure 11 shows the throughput of each workloads according to time. As shown by the orange line on the graph, KVell shows the best performance except for scan workload. Even in case of scan workload, compared with RocksDB, the maximum throughput value is small but shows very stable throughput.



(Figure 12) YCSB E (scan workload) throughput with KV-tuple size

Figure 12 shows the throughput for scan workload according to KV-tuple size. If the Item size is small, RocksDB performed better than KVell. This was because the RocksDB's items were aligned and read fewer pages than KVell. However, as the item size grew, the advantages of saving items in alignment gradually disappeared, so KVell performed better.

2) Conclusion

So far, we have seen three ways to improve the performance of the KV stores. All three methods show a good performance than original systems.

SLM-DB constructs a single-level KV store structure exploiting the benefit of high write throughput from the LSM-tree and integrate it with a persistent B+-tree for indexing KV pairs while eliminating write ahead log (WAL). SLM-DB devises three compaction candidate selection schemes for selective compaction. It also provides good consistency and durability performance from crashes.

SLIK is designed for achieving low latency, high scalability, consistency and high availability using secondary indexes. It provides low-latency by index partitioning, which enables small range of queries to be handled in a single server. This approach also gives a stable environment for large-scale KV store system, which means high scalability. SLIK improves consistency by ordered write approach, garbage collect and atomic updates of multi-objects.

KVell analyzes the conventional LSM and B tree KV store, and it proves out that CPU is the bottleneck of the current system. Thus, it proposes a new paradigm for designing the architecture of KV stores. It tries four schemes for a new system, ‘share nothing’, ‘do not sort data on disk’, ‘no force to sequential access’ and ‘no commit log’.

Although I have explored the above three methods independently, it may also possible to create a new system combining these methods. For example, there can be a new KV store structure whose architecture is from SLM-DB and SLIK while adopting KVell’s design paradigm.

※ References

1. Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S. H., & Choi, Y. R. (2019). **SLM-DB: single-level key-value store with persistent memory**. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)* (pp. 191-205).

2. Kejriwal, A., Gopalan, A., Gupta, A., Jia, Z., Yang, S., & Ousterhout, J. (2016). **{SLIK}: Scalable Low-Latency Indexes for a Key-Value Store**. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)* (pp. 57-70).

3. Lepers, B., Balmau, O., Gupta, K., & Zwaenepoel, W. (2019, October). **KVell: the design and implementation of a fast persistent key-value store**. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (pp. 447-461). ACM.