NovKV: Efficient Garbage Collection for Key-Value Separated LSM-Stores

Chen Shen, Youyou Lu*, Fei Li, Weidong Liu*, Jiwu Shu
Tsinghua University

***Abstract*—LSM-based key-value stores (LSM-stores) play an important role in many storage systems. However, LSM-stores suffer from high write amplification of their compaction operations. Recently proposed key-value separated LSM-stores reduce the impact, but the garbage collection overheads of the value parts remain high. In this paper, we find that existing key-value separation approaches have to check validity of key-value items by querying the LSM-tree, and update value handles by inserting them back into the LSM-tree during garbage collection. Validity checking and value handle updating introduce heavy overheads to the LSM-tree. To this end, we propose an efficient approach to reduce expensive overheads of garbage collection, by eliminating queries and insertions of the LSM-tree. The approach consists of three key techniques: collaborative compaction, efficient garbage collection, and selective handle updating. We implement this approach atop LevelDB and name it as NovKV. Evaluations show that NovKV outperforms WiscKey by up to 1.98x on random write and 1.85x on random read.**

***Index Terms*—LSM-tree, key-value separation, garbage collection**

I. Introduction

Key-value (KV) stores have become essential storage infrastructures. LSM-stores [1] organize data into different levels. Recently updated data are first kept in the lowest level, and the compaction operation sorts, merges, and moves data from low levels to high levels. In the compaction operations, keys and values are moved for a number of times, and this incurs high overhead.

Recently, key-value separation is proposed to reduce the write amplification overheads in LSM-stores in WiscKey [2]. It stores keys and values into two different components, which we term as **KStore** and **VStore** respectively. The KStore uses an LSM-tree to store keys, and its value keeps the handle of the corresponding value in the VStore. The VStore stores encoded KV items in numbers of value files. While the key-value separation reduces write amplification without moving values in the LSM-tree, an extra garbage collection is required in the VStore, which still incurs high overhead. The garbage collection in the VStore has to query the KStore to determine which values are invalid, so that it can drop obsolete values. For the valid values, it moves them to the new locations in the VStore, and updates their handles in the KStore. The garbage collection in the VStore also incurs high overhead.

In this paper, our key idea is to leverage the dependencies between the KStore and VStore to reduce the garbage collection overhead. We observe that the compaction in the KStore has differentiated the invalid values from the valid ones. We pass this information to the VStore, to avoid the validity checking in the garbage collection of the VStore. To support the validity information passing from the KStore to the VStore, we introduce VTable and SVTable designs to the VStore, so as to append invalid keys that have been identified in the KStore.

We also propose to delay the handle updating till the read, so as to reduce the handle updating overhead. During garbage collection, valid values are moved to new locations in the VStore. Instead of updating these handles in the KStore immediately, we delay the updating until the values are read. As such, the garbage collection overheads can be reduced.

Based on the above ideas, we propose an novel design of key-value separated LSM-stores, NovKV, with an efficient garbage collection approach. Our approach aims to reduce validity checking and handle updating overheads, the two dominated overheads in the garbage collection of the VStore. NovKV achieves the efficient garbage collection using the following key techniques:

- **Collaborative Compaction**. NovKV collects keys that the KStore drops during compaction (termed DropKeys) and appends DropKeys to two newly designed tables (termed VTable and SVTable) (Section III-B).
- **Efficient Garbage Collection**. During the VStore garbage collection, NovKV uses DropKeys to check validity of KV items without querying the KStore, and uses searchable value files (SVTable) to avoid value handle updating and to support fast reads (Section III-C).
- **Selective Handle Updating**. NovKV does not aggressively insert all new handles of valid values back into the KStore. Instead, NovKV will take the value handle and update it in the KStore when a client read request finally searches for the value in the SVTable (Section III-D).

We implement NovKV with our proposed garbage collection approach based on LevelDB. Evaluations show that NovKV achieves up to 1.98x write throughput compared to WiscKey, and nearly the same read throughput. After warming up, NovKV outperforms WiscKey by up to 1.85x on random read.

II. Background and Motivation

A. The LSM-tree Basics

In order to maximize write performance, the LSM-tree buffers incoming KV items in memory and writes the journal

* Youyou Lu (luyouyou@tsinghua.edu.cn) and Weidong Liu (liuwd@tsinghua.edu.cn) are the corresponding authors.

in case of a crash. When the buffer reaches a threshold, the LSM-tree builds this buffer as a searchable table that is internally ordered by key (called SSTable in LevelDB) and writes the table to disk sequentially. In this way, the LSM-tree transfers user-level random writes to disk-level sequential writes, and so that can efficiently use the disk bandwidth.

Since the LSM-tree creates SSTables in write order (but internally in key order), key ranges of these SSTables may overlap each other. As a result, the LSM-tree needs to check all SSTables that possibly contain this key when reading a KV item, but only one of them is likely to contain the KV item, which incurs severe read amplification.

A pivotal process to reduce such useless reads is compaction. The LSM-tree compaction picks some key-overlapped SSTables and compacts these SSTables into several non-key-overlapped SSTables in a merge sort way. During compaction, the LSM-tree drops obsolete values that are updated or deleted previously. However, the compaction process brings in significant overheads and results in many problems such as write amplification [3], write stall [4], and resource contention [5].

B. The LSM-tree Optimizations

Due to the extensive usages of the LSM-tree, both academia and industry have done much research to optimize it. For example, PebblesDB [3] uses a fragmented LSM-tree that allows key overlaps in a level, and avoids rewriting data in the same level to reduce write amplification. TRIAD [5] uses a holistic combination of three techniques, respectively in memory level, storage level, and commit log level to reduce write amplification. LWC-tree [6] introduces a light-weight compaction that only merges and sorts metadata of table files to fasten compaction speed and to decrease write amplification. RocksDB employs dynamic level size adjustment, tiered compression, shared compression dictionary, prefix bloom filters, and different size multipliers to better utilize the SSD storage space [7]. The bLSM [8] proposes a new "spring and gear" merge scheduler to bound write latency and so that satisfying strict latency SLAs.

FlashKV [9] applies the LSM-tree on open-channel SSDs [10], [11], and directly manages the raw flash devices in the LSM-tree layer, to eliminates redundant management and semantic isolation. SineKV [12] adopts the LSM-tree as a decoupled secondary index structure, and proposes a mapping-based lazy index maintenance mechanism to efficiently maintain indexes, and also leverages the CMB in NVMe SSDs to guarantee the crash consistency.

An important optimization method is the key-value separation. WiscKey [2] uses a value file (called $vLog$) to store values while storing keys and value handles in the LSM-tree. During garbage collection, WiscKey first reads a chunk of KV items from the tail of the vLog, then finds those valid values by querying the LSM-tree (validity checking). Then WiscKey appends those valid values back to the head of the vLog and updates value handles in the LSM-tree (handle updating). Finally, WiscKey frees the space occupied by these values. HashKV [13] also adopts the key-value separation approach, but organizes data in value store by hash-based partitioning. Unlike WiscKey, HashKV splits the vLog into several segment groups, and deterministically maps KV items to corresponding segment groups. A garbage collection operation in HashKV first selects a segment group with the largest amount of writes, then sequentially scans the KV items in the segment group without querying the LSM-tree (validity checking), then write all valid KV items back into the segment group, and finally updates the latest value locations in the LSM-tree (handle updating). To avoid querying the LSM-tree in validity checking, HashKV uses a temporary in-memory hash table to store keys and value locations of valid KV items. Since KV items are sequentially stored in the segment group in their write order and all versions of values of a specific key must be hashed into the same segment group, the hash table will hold all the latest versions of valid KV items after this scan is finished. So, HashKV can write all valid KV items back into the segment group.

C. Motivation

Even though the key-value separated approach substantially reduces the compaction overhead, existing approach of garbage collection introduces another query and insert overheads. WiscKey needs to check the validity of KV items, append valid KV items to the head of the vLog, and then insert those new handles of valid KV items back into the LSM-tree. These large amounts of queries and insertions compete with clients for available resources, which degrades performance significantly [2].

We observe that WiscKey runs the KStore compaction and the VStore garbage collection independently. The KStore compaction does not have any collaboration with the VStore garbage collection. However, there are some useful data in the KStore can be used to facilitate the VStore garbage collection. We furthur notice that keys in the KStore are only dropped during compaction. Based on these observations, we record those keys that the KStore drops to avoid querying the KStore during the VStore garbage collection.

Furthermore, the VStore garbage collection may insert valid value handles into the KStore many times during the entire run time. However, most KV items will not be read after garbage collection [14], [15]. As a result, many value handles are inserted into the KStore over and over again, even though they are not read. And even worse, aggressively inserting value handles will increase the compaction frequency and thus increase total compaction overheads. So, we insert those value handles only when they are read.

Based on the above two points, we propose an efficient approach of garbage collection for key-value separated LSM-stores.

III. NovKV Design

A. Architecture

Fig. 1 depicts the architecture of NovKV. NovKV consists of a KStore and a VStore. Inside VStore, there is a current VTable, a VMap, and an SVMap. In NovKV, the value handle

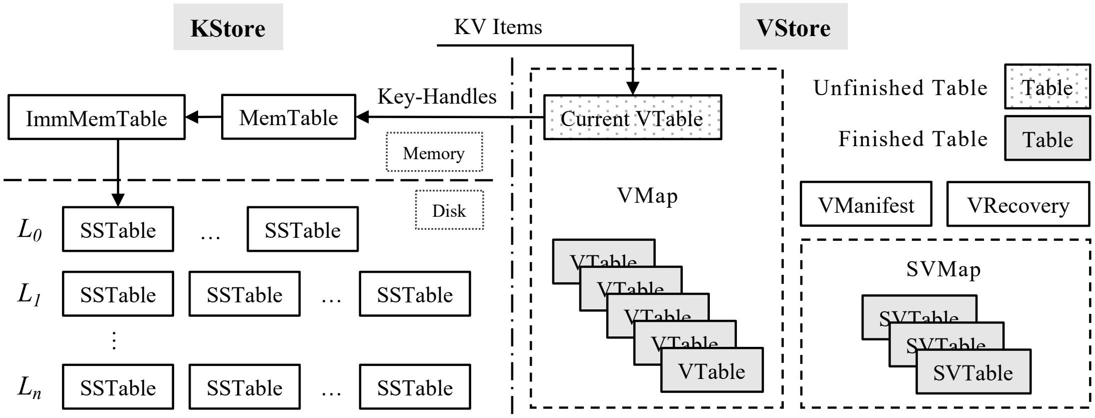

Fig. 1. NovKV Architecture.

consists of a *file number*, an *epoch*, a *file offset*, and an *item size*. The *epoch* is the number of times a table has been rewritten, and also indicates whether a value handle is still valid in the table. Meanwhile, in order to recover from a crash, the VStore saves metadata to the VManifest, and records the last MemTable flush point in the VRecovery.

For a write request, NovKV first encodes the value along with the key as a length prefixed string (termed VItem), as illustrated in Fig. 2, and then appends it to the end of the current VTable. After doing this, NovKV gets the value handle of this VItem, and inserts this value handle to the MemTable in the KStore. When the current VTable reaches its size threshold, NovKV finishes this VTable, creates a new VTable, and inserts the newly created VTable into the VMap.

For a read request, NovKV first queries the KStore to get the value handle. Then NovKV uses the handle's file number to check whether this value file exists in the VMap. If it exists, NovKV directly reads the value from the VTable by the handle. If not exists, it indicates that the VTable has been garbage collected before and been transferred to an SVTable. So, NovKV takes the SVTable from the SVMap. If the epoch of the handle and the epoch of the SVTable matches, NovKV directly reads the value from the SVTable by the handle without searching. If not matches, it indicates that this SVTable is rewritten after this value handle was inserted to the KStore, which means this value handle is also invalid. So, NovKV searches for the KV item in the SSTable part of this SVTable. The search process in SSTable is the same as in the LevelDB.

In order to efficiently store keys that the KStore compaction drops, we introduce two newly designed files, VTable and SVTable.

1) VTable: As illustrated in Fig. 2, VTable is composed of a vLog part and several DropBlocks. The vLog part contains numbers of encoded VItems. Using the offset and size of a value handle, NovKV can directly read the value from the vLog part. A DropBlock contains a couple of DropKeys and a DFooter. The DFooter stores metadata of the DropBlock and the VTable:

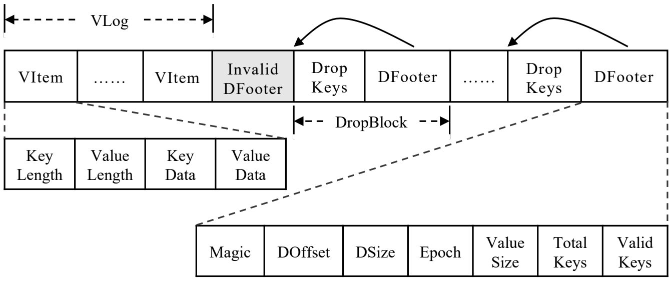

Fig. 2. The VTable Format.

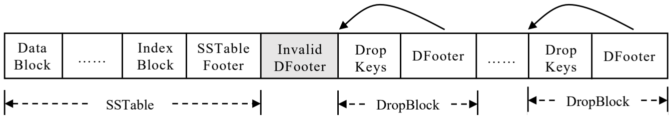

Fig. 3. The SVTable Format.

- Magic. The magic is used to check the file integrity.
- DOffset and DSize. The DOffset and DSize indicates the offset and the size of this DropBlock.
- Epoch. The epoch indicates how many times this table has been rewritten.
- ValueSize. The value size records the size of the vLog part.
- Total and valid keys. The DFooter also records the numbers of total keys and currently valid keys in the VTable. The number of valid keys is equal to the number of total keys minuses the number of all dropped keys.

To avoid an extra read of the DFooter before garbage collection, NovKV keeps the last DFooter in memory, along with the calculated $valid\_rate$ that is equal to $valid\_keys/total\_keys$.

When the current VTable reaches its size threshold, NovKV appends an invalid DFooter to the end of the vLog part to indicate the start of the following DropBlocks, which we term as $finish$. Then NovKV creates a new VTable as the new current VTable, and inserts this VTable to the VMap. The newly created VTable is unfinished and only contains the vLog part. During the KStore compaction, NovKV collects keys that the KStore drops, builds these keys as a DropBlock, and then appends the DropBlock to the end of the VTable.

2) SVTable: Similar to VTable, SVTable is composed of an SSTable part and numbers of DropBlocks, as shown in Fig. 3. As mentioned above, NovKV may need to search for values by keys in SVTable. So, the SVTable replaces the vLog part in VTable with an embedded SSTable. The DFooter keeps the same as VTable except that the ValueSize here represents the SSTable size. Note that once NovKV needs to search values from the SVTable, we can confirm that this key and value must exist in the SSTable. Hence, the bloom filter in the SSTable is not needed.

In addition to supporting fast search, SVTable is also far more cache-friendly than an unordered VTable since values in SVTable is ordered by keys. Thus, the directly read by the value handle in SVTable is faster than the directly read in VTable.

B. KStore: Collaborative Compaction

In LSM-trees, the update operation of a KV item is to directly write the new KV item instead of an in-place update. Thus, there may exist multiple versions of values of a specific key in the LSM-tree, which leads to two issues. On the one hand, a read operation must figure out the right version of values. On the other hand, the LSM-tree needs to release those invalid items that have been updated or deleted. So, there must be a way to determine which item is valid in the LSM-tree.

To address these two issues, the LSM-tree maintains an incremental sequence number and a snapshot list, and drops invalid KV items during compaction.

For each writing key, the LSM-tree appends a sequence number to the tail of this key. The writing key and the composited key is called user key and internal key respectively. When comparing two internal keys, the LSM-tree first compares the user key in ascending order, and then compares the sequence number in descending order if two user keys are identical. Based on the sequence number and internal key mechanism, a snapshot in the LSM-tree is in essence a sequence number.

When the read operation retrieves a specific snapshot of value, the LSM-tree searches the KV item whose sequence number is equal or less than the sequence number of the snapshot.

During compaction, the LSM-tree checks whether the sequence number of a KV item is less than the sequence number of the KV item with the same user key in the oldest snapshot. If it is not, this KV item is still valid. If it is, this KV item is obsolete, the conventional LSM-tree directly drops this item, and thus we term the key of this item as DropKey.

Take Fig. 4 as an example, the oldest snapshot is *Snap@6*, so the oldest version of KV item whose user key is *Key* that can be retrieved is *Key@4*. Thus, *Key@2* is an invalid item.

Based on the implementation of conventional compaction, We add some extra operations to it while the most operations of compaction in NovKV is the same as in conventional LSM-trees, as illustrated in Fig. 5. Before the KStore compaction, NovKV first creates a map whose the key type is file number and the value type is a vector of keys. Then, NovKV reads an item from an SSTable, checks its validity as in conventional compaction. If this item is not valid, NovKV decodes the value to get the file number, inserts this DropKey to the corresponding vector in the map by the file number. After this compaction finishes, NovKV writes these DropKeys to corresponding VTables and SVTables, and then triggers a VStore garbage collection. If there already exists a running garbage collection, this trigger is skipped.

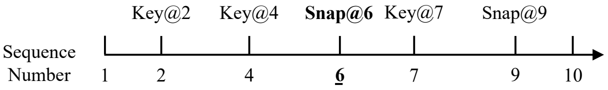

Fig. 4. An Example of Invalid KV Items.

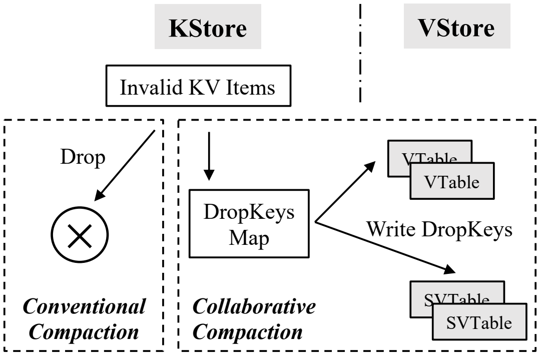

Fig. 5. Collaborative Compaction.

C. VStore: Efficient Garbage Collection

Since the key-value separated LSM-store keeps actual KV items in the VStore while keeps only keys and value handles in the KStore, NovKV also needs a process to free spaces occupied by obsolete KV items in the VStore.

Thus, NovKV introduces a VStore garbage collection process to drop obsolete KV items and then write valid values back as a new value file to the disk. To reduce the overheads of validity checking and handle updating, the VStore persists the validity information that has been differentiated in the KStore before, and uses the SVTable to eliminate updating new value handles.

The VStore garbage collection is illustrated as Fig. 6 from a high-level perspective. VTables or SVTables whose invalid items exceeds a threshold will be garbage collected and be built as new SVTables.

NovKV first collects all of finished VTables and SVTables whose $valid\_rate$ is less than the $gc\_threshold$, and sorts them by $valid\_rate$ in descending order, then picks a number of VTables and SVTables with the smallest $valid\_rate$. Those VTables or SVTables whose $valid\_keys$ is zero are removed from the VMap or SVMap respectively, and are deleted directly. Then, NovKV processes remaining VTables and SVTables one by one, as described below.

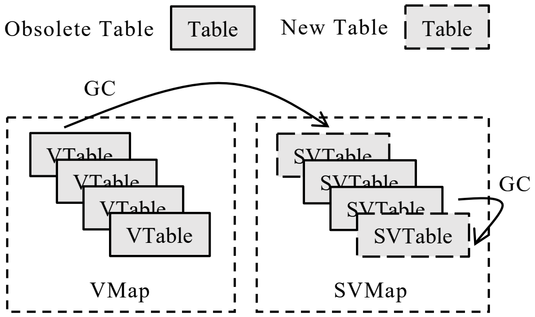

Fig. 6. The VStore Garbage Collection.

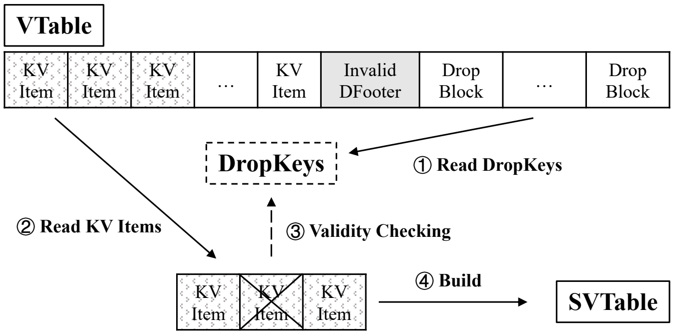

Fig. 7. The VTable Garbage Collection.

Before actually doing the VStore garbage collection, NovKV reads DropKeys from DropBlocks reversely. Through the DOffset and DSize of the in-memory DFooter, NovKV read the last DropBlock along with the previous DFooter. Then, NovKV decodes these DropKeys and inserts them into a hash set. After reading this DropBlock, NovKV checks the previous DFooter. If the previous DFooter is valid, NovKV continues to read the previous DropBlock by this DFooter. Otherwise, NovKV finishes reading.

1) VTable GC: The VTable garbage collection is illustrated as Fig. 7.

In actually garbage collection, NovKV first creates an SVTable whose epoch is 1 to store valid KV items. Then, NovKV reads a fix-sized content from the beginning of the vLog part, decodes a VItem from the content, checks whether the key exists in its DropKeys. If it does not exist, NovKV inserts this key and value to the new SVTable, or drops this item otherwise. After processing this VItem, NovKV reads the next VItem from the content or read another chunk of content if there is no more VItems in the content.

Since VItems in the VTable is out-of-order, SVTable uses a MemTable to hold valid KV items during the construction. When this VTable is processed successfully, NovKV finishes this SVTable. The SVTable builds this MemTable as an SSTable and flushes it to disks. Then NovKV appends an invalid DFooter to the end of the SSTable part.

Next, NovKV removes the obsolete VTable from the VMap, and inserts the newly created SVTable into the SVMap. Note that a lock guards this step.

There is a special issue that needs to be deal with. The KStore may do several times of compaction during this garbage collection, and drop some other keys to the obsolete VTable that is involved in this garbage collection. So, before deleting the obsolete VTable, NovKV needs to check whether there are new DropKeys appended in the obsolete VTable. This run of garbage collection does not read these new DropKeys, so NovKV copies these DropKeys to the newly created SVTable. Finally, NovKV deletes this obsolete VTable.

2) SVTable GC: The SVTable GC is roughly the same as VTable GC. NovKV first collects DropKeys from the obsolete SVTable and creates its iterator to read KV items. The following process is exactly the same as VTable GC. In the end, NovKV opens the newly created SVTable, replace the obsolete SVTable with the newly created SVTable in the SVMap. A lock also guards this step. Note that the epoch of the newly created SVTable is equaled to the epoch of the obsolete SVTable plus one.

D. Selective Handle Updating

NovKV does not insert new valid value handles back into the KStore immediately during garbage collection. So, there is only the file number in the value handle is still valid after garbage collection (GC does not change the file number). To facilitate searching for values in a garbage-collected value file, NovKV uses an embedded SSTable in the SVTable. However, this introduces extra SSTable search overheads to the read process. So, NovKV proposes a mechanism to update the obsolete value handles.

When NovKV searches for a value in an SVTable, NovKV also gets the value handle in the SVTable, and inserts this new value handle back into the MemTable in the KStore to fix the invalid value handle.

The cost of a completely memory write is substantially low, which makes handle updating in read process possible. In this way, NovKV can directly read the value from the SVTable without searching next time.

E. Failure Recovery

In order to recover from crashes, NovKV uses a *manifest* file to record all valid data files (including VTables, SVTables, and the current VTable), and uses a *recovery* file to record the recovery point.

When recovers from a crash, NovKV first reads the *manifest* file and rebuilds the VMap, SVMap, and the current VTable pointer. Then NovKV reads the *recovery* file, reads all KV items after the previous recovery point from VTables, and re-inserts all these keys and value handles to the KStore.

As described in III-A, an incoming KV item in NovKV is first appended to the current VTable before returning the write result, which ensures that a write operation returns successfully only if the KV item is persisted successfully. So, a program crash during writing does not lose any data.

Since the VStore uses DropKeys to discard obsolete KV items instead of querying the KStore, if keys that have been dropped by the KStore are lost, some obsolete KV items will be left and will not be dropped forever.

Thus, NovKV write DropKeys during the KStore compaction. If a crash happens before DropKeys are persisted, the KStore compaction is also unfinished and will be cleaned up after failure recovery. As a result, these DropKeys can be collected again in the next compaction. Furthermore, only if all files in this run of garbage collection are collected successfully and the new version of *manifest* file is persisted successfully, the corresponding file pointers in VMap and SVMap will be replaced. So, a program crash during compaction and garbage collection also does not lose any data or break the consistency.

IV. Evaluation

In this section, we compare some performance indicators of NovKV, WiscKey, and the original LevelDB using the LevelDB's benchmark framework *db_bench*. Our implementation is based on LevelDB 1.22.

Completely sequential workloads are almost rare in general LSM-stores in the real world. Specific methods for time series data will deal with these workloads. So we choose two very representative random benchmarks (random write and random read) to evaluate three LSM-stores. We add a *readrandomsame* benchmark to *db_bench*. The *readrandomsame* benchmark is almost the same as *readrandom* except that it keeps test keys in memory so that the test keys in the next run can be identical with the previous. We run one time of *fillrandom* to evaluate the random write performance, and two times of *readrandomsame* to evaluate the random read performance. During benchmarks, we record the total time of reading the KStore and the VStore respectively, and the write amounts of compaction and garbage collection.

In these benchmarks, We set the max size of VTable and SVTable to 16MiB and the *gc_threshold* to 0.7. To better measure the performance, we disable LevelDB's default snappy compression.

We use an Alibaba Cloud ECS [16] server to evaluate three LSM-stores, which is equipped with 8 vCPUs, 60GiB memory, a 20GiB system cloud disk, a 1788GiB test local SSD and runs Ubuntu 18.04 LTS with the Linux 4.15 kernel. The filesystem running on the test disk is ext4.

All write benchmarks in this section have only one write thread. In these benchmarks, we all vary the value size from 128B to 4KiB while the key size remains 16B unchanged.

A. Randdom Write

This benchmark inserts 100M KV items with values of different sizes. The number of KV items is fixed, so the total data size increases with the value size. Inserting the fixed number of KV items instead of the fixed size of data helps us better understand the performance when storing large amounts of KV items.

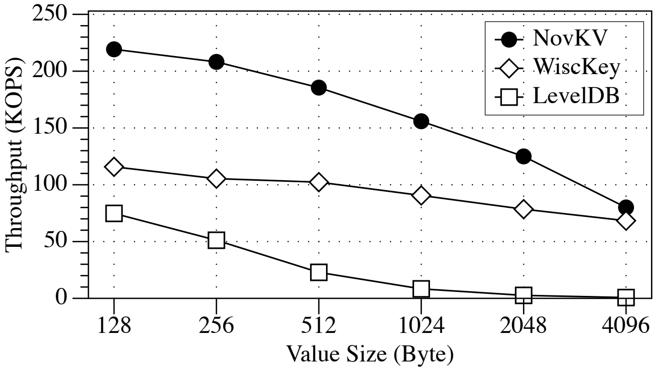

Fig. 8. The Random Write Throughput.

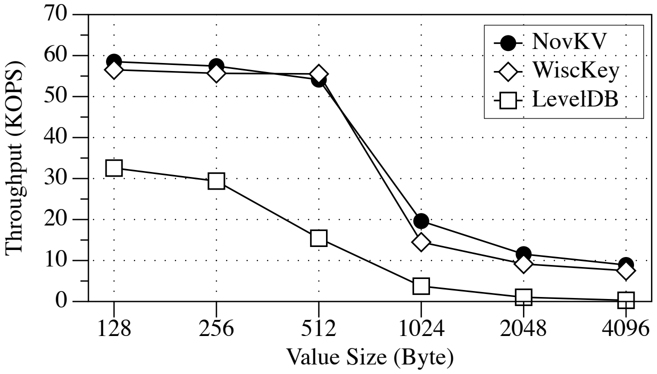

Fig. 9. The Random Read Throughput (The First Run).

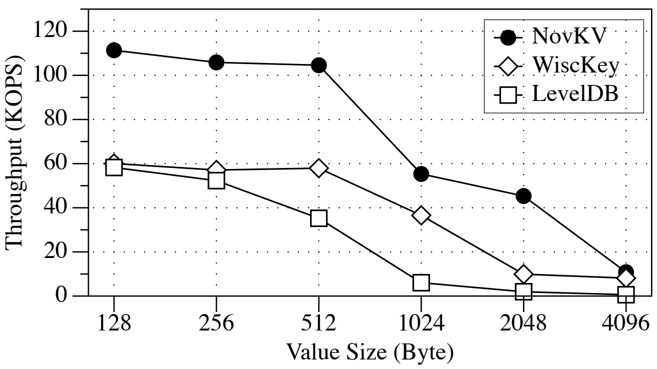

Fig. 10. The Random Read Throughput (The Second Run).

As Fig. 8 shows, the throughput of NovKV is at most roughly 2× over WiscKey. With the value size increases, operations per second (ops) of NovKV starts to get close to WiscKey but always higher than WiscKey. This proves that insertions during garbage collection in WiscKey compete for the write room of the KStore with user-facing insertions. So, benefits from avoiding insertions, NovKV can achieve higher throughput than WiscKey.

B. Random Read

This benchmark reads 10M KV items previously inserted in Section IV-A. When the first time that the client read a specific key, NovKV may need to search for values and update the value handle, which introduces some negative

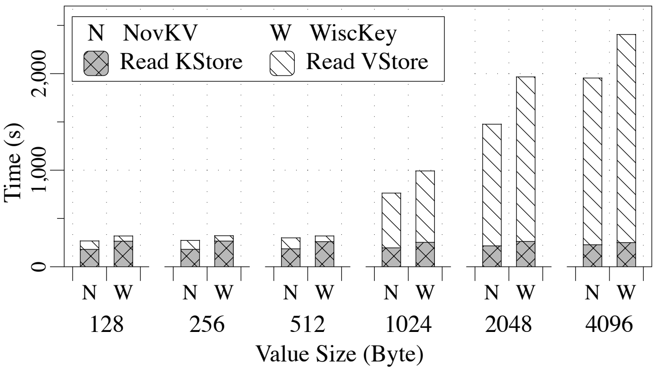

Fig. 11. Read Time Details.

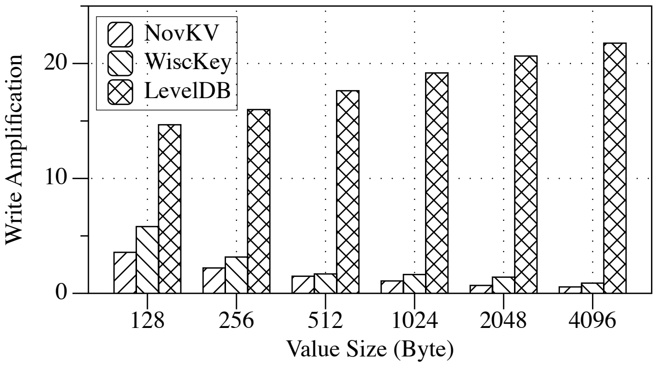

Fig. 12. The Write Amplification.

impacts on this read. However, once the specific key and value handle is updated, latencies of future reads will be much lower than the first time. Based on this, we issue two runs of $readrandomsame$ in this benchmark. The first run of $readrandomsame$ mainly reflects the impacts of searching in SVTables and overheads of value handle updating. The second run of $readrandomsame$ indicates the normal random read performance after updating value handles.

As Fig. 9 shows, the random read throughput of NovKV is much higher than LevelDB and slightly higher than WiscKey even NovKV needs to update handles. This shows that large amounts of queries and insertions of WiscKey during garbage collection have negative impacts on the KStore.

We can see from Fig. 10 that the random read throughput of NovKV on the second run is much higher than WiscKey when the value size is relatively small. According to Fig. 11, when the values size is relatively small, even the time spending in reading the VStore in NovKV is more than in WiscKey, WiscKey suffers more from the slowly read in the KStore. With the value size increases, reading VStore dominates the entire read process. As explained in Section III-A2, benefits from the more efficiently caching, NovKV still performs better than WiscKey. This proves that the SVTable is much more cache-friendly than a naive unordered vLog.

C. Write Amplification

We record the total write amounts of compaction and garbage collection, and define the write amplification as the proportion of this total write amounts to the raw data size.

Fig. 12 shows the write amplification of NovKV, WiscKey, and LevelDB. Both NovKV and WiscKey performs much better than LevelDB, while NovKV is better than WiscKey.

As shown in Fig. 13, large amounts of insertions to the KStore in garbage collection in WiscKey result in much more KStore compaction than NovKV. In contrast, NovKV does not insert value handles during garbage collection so that the write amounts of compaction in NovKV keep almost the same. As a consequence, WiscKey needs to do more data rearrangement than NovKV, which leads to a higher write amplification.

In NovKV, we use a greedy approach of $valid\_rate$ to collect files to do garbage collection, and the overheads of garbage collection in NovKV are less than in WiscKey. So, we can do more garbage collection in NovKV than in WiscKey.

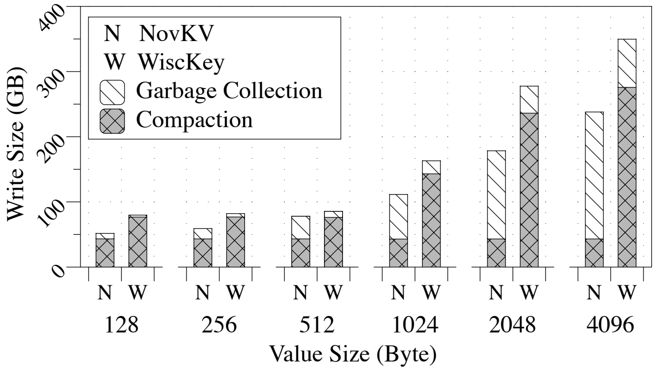

Fig. 13. The Write Amounts of Compaction and GC.

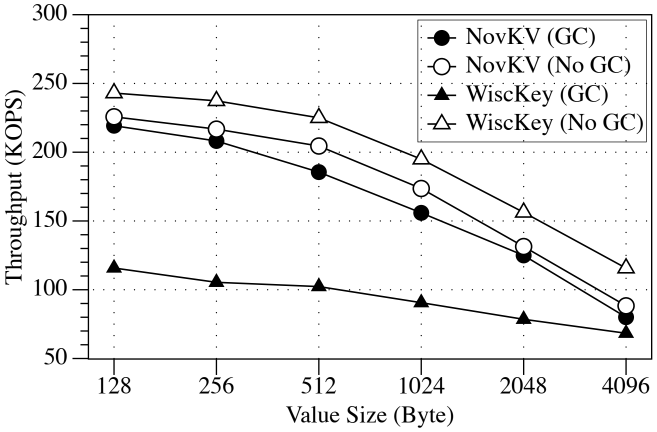

Fig. 14. The Random Write Throughput With and Without GC.

D. The Impact of GC

To better understand how far the garbage collection degrades overall performance, we run an extra $fillrandom$ benchmark without GC.

As shownn in Fig. 14, when GC is disabled, WiscKey can achieve very high performance. Due to some extra operations, NovKV does not perform as good as WiscKey. However, if GC is enabled, the performance of WiscKey is reduced by a factor of 1.3 at most compared to the performance without GC while the factor of NovKV is only 0.1. This proves that the garbage collection of WiscKey has very serious negative impacts on the performance. As a contrast, benefits from the elimination of validity checking and handle updating, the garbage collection of NovKV has only a slight impact on the performance.

V. Conclusion

In this work, we find an efficient approach to reduce the overheads of garbage collection. Three key techniques tactfully eliminate queries and insertions during garbage collection, which also significantly reduce the burden of compaction at the same time. By reducing the overheads of compaction and garbage collection, more available resources can be used to improve user-facing performance. Evaluations show that NovKV indeed achieves higher performance compared to WiscKey. We hope that this work can inspire more work about key-value separated LSM-stores, and the key-value separated approach can be more elegant.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback and constructive suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61772300, 61832011) and the Special Topics of Major Scientific and Technological Projects in Sichuan Province (Grant No. 2018GZDZX0049).

References

[1] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[2] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating Keys from Values in SSD-conscious Storage," in *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, A. D. Brown and F. I. Popovici, Eds. USENIX Association, 2016, pp. 133–148. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[3] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 497–514. [Online]. Available: https://doi.org/10.1145/3132747.3132765

[4] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 753–766. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/balmau

[5] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds. USENIX Association, 2017, pp. 363–375. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[6] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores," in *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017)*, 2017.

[7] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing Space Amplification in RocksDB," in *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[8] R. Sears and R. Ramakrishnan, "bLSM: A General Purpose Log Structured Merge Tree," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 217–228. [Online]. Available: https://doi.org/10.1145/2213836.2213862

[9] J. Zhang, Y. Lu, J. Shu, and X. Qin, "FlashKV: Accelerating KV Performance with Open-Channel SSDs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 139:1–139:19, 2017. [Online]. Available: https://doi.org/10.1145/3126545

[10] Y. Lu, J. Shu, and W. Zheng, "Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems," in *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, K. A. Smith and Y. Zhou, Eds. USENIX, 2013, pp. 257–270. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou

[11] J. Zhang, J. Shu, and Y. Lu, "ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, A. Gulati and H. Weatherspoon, Eds. USENIX Association, 2016, pp. 87–100. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang

[12] F. Li, Y. Lu, Z. Yang, and J. Shu, "SineKV: Decoupled Secondary Indexing for LSM-based Key-Value Stores," in *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, Nov 29-Dec 1, 2020*. IEEE, 2020.

[13] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling Efficient Updates in KV Storage via Hashing," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, H. S. Gunawi and B. Reed, Eds. USENIX Association, 2018, pp. 1007–1019. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/chan

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, P. G. Harrison, M. F. Arlitt, and G. Casale, Eds. ACM, 2012, pp. 53–64. [Online]. Available: https://doi.org/10.1145/2254756.2254766

[15] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 209–223. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[16] Alibaba Cloud, "Elastic Compute Service: An online computing service that offers elastic and secure virtual cloud servers to cater all your cloud hosting needs," https://www.alibabacloud.com/product/ecs.