REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong\textsuperscript{1}, Chen Chen\textsuperscript{1}, Xingbo Wu\textsuperscript{1}, and Song Jiang\textsuperscript{2}

\textsuperscript{1}University of Illinois at Chicago
\textsuperscript{2}University of Texas at Arlington

Abstract

LSM-tree based key-value (KV) stores organize data in a multi-level structure for high-speed writes. A range query on this structure has to seek and sort-merge data from multiple table files on the fly, which is expensive and often leads to mediocre read performance. To improve range query efficiency on LSM-trees, we introduce a space-efficient KV index data structure, named REMIX, that records a global sorted view of KV data spanning multiple table files. A range query with REMIX on multiple data files can quickly locate the target key using a binary search, and retrieve the subsequent keys in the sorted order without key comparisons. We build RemixDB, an LSM-tree based KV-store that adopts a write-efficient compaction strategy and employs REMIX for fast point and range queries. Experimental results show that REMIX can substantially improve range query performance in a write-optimized LSM-tree based KV-store.

1 Introduction

Key-value stores (KV-stores) are the backbone of many cloud and datacenter services, including social media \cite{1,2,6}, real-time analytics \cite{5,8,20}, e-commerce \cite{13}, and cryptocurrency \cite{36}. Log-Structured Merge-tree (LSM-tree) \cite{33} is the core data structure of many KV-stores \cite{7,13,15,21,26,37}. In contrast to traditional storage structures (e.g., B+-tree) that require in-place updates on disk, LSM-tree follows an out-of-place update scheme which enables high-speed sequential write I/O. It buffers updates in a log and periodically flushes them to persistent storage to generate immutable table files. However, it comes with penalties on search efficiency that keys in a range may reside in different tables, slowing down queries because of the high computation and I/O cost. The LSM-tree based designs are trade-offs between update cost and search cost \cite{12}, maintaining a lower update cost and a much higher search cost compared with B+-tree.

Attempts have been made to improve query performance. To speed up point queries, every table is usually associated with memory-resident Bloom filters \cite{4} so that a query can skip the tables that do not contain the target key. However, Bloom filters cannot handle range queries. Range filters such as SuRF \cite{40} and Rosetta \cite{30} are proposed to accelerate range queries by filtering out tables not containing any keys in the requested range. However, when the target keys of a query reside in most of the candidate tables, the filtering approach can hardly improve query performance, especially for long range queries. Furthermore, the computation cost on accessing the filters can lead to mediocre performance if the queries can be answered from the cache, which is often the case in real-world workloads \cite{2,6,9}.

To bound the number of tables that a search request needs to access, LSM-tree keeps a background compaction thread to consistently sort-merge tables. The table selection is determined by a compaction strategy. The \textit{leveled} compaction strategy has been adopted by a number of KV-stores, including LevelDB \cite{21} and RocksDB \cite{15}. Leveled compaction sort-merges smaller sorted runs into larger ones to limit the number of overlapping tables under a threshold. In practice, leveled compaction provides the best read efficiency but has a high write amplification due to its aggressive sort-merging policy. Alternatively, the \textit{tiered} compaction strategy waits for multiple sorted runs of a similar size and merges them into a larger run. Tiered compaction provides lower write amplification and higher update throughput. It has been adopted by many KV-stores, such as Cassandra \cite{26} and ScyllaDB \cite{37}. Since tiered compaction cannot effectively limit the number of overlapping tables, it leads to much higher search cost compared with leveled compaction. Other compaction strategies can better balance the read and write efficiency \cite{11,12}, but none can achieve the best read and write efficiency at the same time.

The problem lies in the fact that, to limit the number of sorted runs, a store has to sort-merge and rewrite existing data. Today’s storage technologies have shown much improved random access efficiency. For example, random read on commodity Flash SSDs can reach over 50% of their sequential read throughput. New technologies such as 3D-XPoint (e.g.,
Intel’s Optane SSD) offer near-equal performance for random and sequential I/O [38]. As a result, KV-pairs do not have to be physically sorted for fast access. Instead, a KV-store could keep its data logically sorted for efficient point and range queries while avoiding excessive rewrites.

To this end, we design REMIX, short for Range-query-Efficient Multi-table Index. Unlike existing solutions to improve range-queries that struggle between physically re-writing data and performing expensive sort-merging on the fly, REMIX employs a space-efficient data structure to record a global sorted view of KV data spanning multiple table files. With REMIX, a KV-store can take advantage of a write-efficient compaction strategy without sacrificing search performance.

We build RemixDB, a REMIX-indexed LSM-tree based KV-store. Integrated with the write-efficient tiered compaction strategy and a partitioned LSM-tree layout, RemixDB achieves low write amplification and fast searches at the same time. Experimental results show that REMIX can effectively improve range query performance when searching on two or more overlapping tables. Performance evaluation demonstrates that RemiXDB outperforms the state-of-the-art LSM-tree based KV-stores on both read and write operations simultaneously.

2 Background

LSM-tree is optimized for better write efficiency on persistent storage devices. It achieves high speed writes by buffering all updates in a log file. When the log file fills up, the buffered keys will be sorted and flushed to persistent storage as a sorted run by a process called minor compaction. Minor compaction is write-efficient because all the updates are batched and written sequentially at once, without merging with existing data in the store. Since the sorted runs have overlapping key ranges, a lookup request will query all the possible runs, leading to a high lookup cost. To limit the number of overlapping runs, LSM-tree uses a major compaction process to sort-merge several overlapping runs into fewer ones.

A compaction strategy determines how tables are selected for major compaction. The two most commonly used strategies are leveled compaction and tiered compaction. A store using leveled compaction has a multi-level structure where each level maintains a sorted run consisting of one or more tables. The capacity of a level ($L_n$) is a multiple (usually 10 [15]) of the smaller one ($L_{n-1}$), which allows a huge KV-store to be managed by a few levels (usually 5 to 7). Leveled compaction makes lookup relatively efficient, but it leads to inferior write efficiency. Leveled compaction selects overlapping tables from adjacent levels ($L_n$ and $L_{n+1}$) for sort-merging and generates new tables in the larger level ($L_{n+1}$). Because of the exponentially increasing capacity, one table from the smaller level often overlaps several tables in the larger level. As a result, the majority of the writes are for rewriting existing data in $L_{n+1}$, leading to high write amplification (WA) ratio\(^1\) of up to 40 in practice [35]. Figure 1 shows an example of leveled compaction where each table contains two or three keys. If the first table in $L_1$ (containing keys (4, 21, 38)) is selected for sort-merging with the first two tables in $L_2$ ((6, 26) and (31, 40, 46)), five keys in $L_2$ will be rewritten.

With tiered compaction, multiple overlapping sorted runs can be buffered in a level, as shown in Figure 2. The number of runs in a level is bounded by a threshold denoted by $T$, where $T > 1$. When the number of sorted runs in a level ($L_n$) reaches the threshold, all sorted runs in $L_n$ will be sort-merged into a new sorted run in the next level ($L_{n+1}$), without rewriting any existing data in $L_{n+1}$. Accordingly, an LSM-tree’s WA ratio is $O(L)$ using tiered compaction, where $L$ is the number of levels. With a relatively large $T$, tiered compaction provides much lower write amplification than leveled compaction does with a similar $L$. However, since there can be multiple overlapping sorted runs in each level, a lookup operation will need to check up to $T \times L$ tables, leading to a much slower search.

Range query in LevelDB/RocksDB is realized by using an iterator structure to navigate across multiple tables as if all the keys are in one sorted run. A range search first initializes an iterator using a seek operation with a target key. The seek operation positions the iterator so that it points to the smallest key in the store that is equal to or greater than the target key (in lexical order for string keys). The next operation advances the iterator such that it points to the next key in the sorted order. A sequence of next operations can be used to retrieve the subsequent keys in the target range until a certain condition is met (e.g., number of keys, end of a range). Since the sorted runs are generated chronologically, a target key can reside in any of the runs. Accordingly, an iterator must keep track of all the sorted runs.

Figure 1 shows an example of seek on an LSM-tree using leveled compaction. To seek key 67, a binary search is used on each run to identify a key satisfying the comparison criteria ($cursor \_key \geq target \_key$). Each identified key is marked

\(^1\)WA ratio: ratio of the amount of write I/O to the amount of user write.
by a cursor. Then the keys are sort-merged using a min-heap structure [18], and thus the key 67 in \( L_2 \) is selected. Subsequently, each next operation will compare the keys under the cursors, return the smallest key, and advance the corresponding cursor. This process presents a global sorted view of the keys, as shown in the upper right corner of Figure 1. In this example, all three levels must be accessed for the sort-merging. Figure 2 shows a similar example with tiered compaction. Having six overlapping sorted runs, a seek operation is more expensive than the previous example. In practice, the threshold \( T \) in tiered compaction is often set to a small value, such as \( T = 4 \) in ScyllaDB [37], to avoid having too many overlapping sorted runs in a store.

3 REMIX

A range-query operation on multiple sorted runs constructs a sorted view of the underlying tables on the fly so that the keys can be retrieved in sorted order. In fact, a sorted view inherits the immutability of the table files and remains valid until any of the tables are deleted or replaced. However, existing KV-stores have not been able to take advantage of the inherited immutability. Instead, sorted views are repeatedly reconstructed at search time and immediately discarded afterward, which leads to poor search performance due to excessive computation and I/O. The motivation of REMIX is to exploit the immutability of table files by persisting and reusing their sorted views for future searches.

For I/O efficiency, the LSM-tree based KV-stores employ memory-efficient metadata formats, including sparse indexes [34] and Bloom filters [4]. If we record every key and its location to persist the sorted views in a store, the store’s metadata could be significantly inflated, leading to compromised performance for both reads and writes. To avoid this issue, the REMIX data structure must be space-efficient.

3.1 The REMIX Data Structure

The top of Figure 3 shows an example of a sorted view containing three sorted runs, \( R_0, R_1, \) and \( R_2 \). The sorted view of the three runs is illustrated by the arrows, forming a sequence of 15 keys. To construct a REMIX, we first divide the keys of a sorted view into groups, each containing a fixed number of keys. Each group is attached with three kinds of metadata: anchor key, cursor offsets, and run selectors. The anchor key stores the smallest key in a group. The anchor keys of all the groups collectively form a sparse index on the sorted view. The cursor offsets of a group record the cursor positions corresponding to a seek to the anchor key. The run selectors are a sequence of numbers, representing the corresponding sorted runs of each key. The run selectors encode the sequential access path of the keys on the sorted view, starting from the anchor key of a group.

An iterator for REMIX does not use min-heap. Instead, an iterator contains a current pointer and a set of cursors. The current pointer points to a run selector, and each cursor points to the location of a key on a sorted run. The run selector under the current pointer selects a sorted run, and the cursor on the sorted run determines the current key.

It takes three steps to seek a key using an iterator on REMIX. First, a binary search is performed on the anchor keys to find the target group whose range covers the target key, satisfying anchor_key \( \leq \) target_key. Second, the iterator is initialized to point to the anchor key of the group. Specifically, the cursors are positioned using the cursor offsets of the group, and the current pointer is set to point to the first run selector of the group. Finally, the target key can be found by scanning linearly in the group. To advance the iterator, the cursor of the current key is advanced to skip the current key. Meanwhile, the current pointer is also advanced to point to the next run selector. After a seek operation, the subsequent keys on the sorted view (within and beyond the target group) can be retrieved by advancing the iterator in the same manner.

The following presents an example of a seek operation. As shown in Figure 3, the four boxes on the bottom contain the metadata of the sorted view in REMIX. It is worth noting that the gray keys in parentheses are not part of the metadata. To seek key 17, the second group that contains keys (11, 17, 23, 29) is selected using binary search. Then the cursors are placed on keys 11, 17, and 31 in \( R_0, R_1, \) and \( R_2 \), respectively, according to the recorded cursor offsets ((1, 2, 1)). Meanwhile, the current pointer is set to point to the first run selector of the group (0, the fifth run selector in the figure), indicating that the current key (11) is under the cursor of \( R_0 \). Since \( 11 < 17 \), the iterator needs to be advanced to find the smallest key \( k \) satisfying \( k \geq 17 \). To advance the iterator, the cursor on \( R_0 \) is first advanced so that it skips key 11 and is now on key 23. The cursor offsets of the three sorted runs now become 2, 2, and 1. Then, the current pointer is advanced to the second run selector of the group (1, the sixth selector). The updated iterator selects run \( R_1 \), and the current key 17 can be found under \( R_1 \)’s cursor. This concludes the seek operation.

To retrieve the following keys (23, 29, 31, ... ) on the sorted view, one could repeatedly advance the iterator by updating the current pointer and the corresponding cursors.
3.2 Binary Search in a Target Group

A seek operation initializes the iterator using binary search on the anchor keys to find the target group and scans forward on the sorted view to find the target key. Having more keys in a group speeds up the binary search for having fewer anchor keys and a smaller size of metadata. However, it can slow down seek operations because scanning in a large target group needs to access more keys on average. To avoid the performance penalty, we also use binary search in a target group to minimize the search cost.

To perform binary search in a group, we must be able to randomly access every key in the group. A key in a group belongs to a sorted run, as indicated by the corresponding run selector. To access a key, we need to place the cursor of the run on the correct position. This can be done by counting the number of occurrences of the same run selector in the group prior to the key and advancing the corresponding cursor the same number of times. The number of occurrences can be quickly calculated on the fly using SIMD instructions on modern CPUs. By repeating this process, binary search can be performed until the target key is found. To conclude the seek operation, we initialize all the cursors using the occurrences of each run selector prior to the target key.

Figure 4 shows an example of a group having 16 run selectors. The number under each run selector represents the occurrences of the same run selector prior to its position. For example, 41 is the third key in R3 in this group, so the corresponding number of occurrences is 2 (under the third “3”). To access key 41, we initialize the cursor of R3 and advance it twice to skip 5 and 23.

To seek key 41 in the group in Figure 4, keys 43, 17, 31, and 41 will be accessed successively during the binary search, as shown by the arrows and the circled numbers. Key 43 is the eighth key in the group and the fourth key of R3 in the group. To access key 43, we initialize the cursor of R3 and advance it three times to skip keys 5, 23, and 41. Then, key 17 can be accessed by reading the first key on R2 in this group. Similarly, 31 and 41 are the second key and the third key on R1 and R3, respectively. To conclude the search, the uninitialized cursors will be updated using the values of the corresponding occurrences prior to key 41. In this example, the cursors will be placed on keys 61, 53, 89, and 41, where 41 is the current key of the iterator.

3.3 Search Efficiency

REMIX improves range queries in three aspects.

**REMIX finds the target key using one binary search.** REMIX provides a sorted view of multiple sorted runs. Only one binary search on a REMIX is required to position the cursors on the target keys for multiple sorted runs. Whereas in a traditional LSM-tree based KV-store, a seek operation requires a number of binary searches on each individual sorted run. For example, suppose a store with four equally-sized sorted runs has N keys in each run. A seek operation without REMIX requires 4 × log₂N key comparisons, while it only takes log₂4N, or 2 + log₂N key comparisons with a REMIX.

**REMIX moves the iterator without key comparisons.** An iterator in a REMIX directly switches to the next (or the previous) KV-pair by using the prerecorded run selectors to update the current pointer and the cursors. This process does not require any key comparisons. Reading a KV-pair can also be avoided if the iterator skips the key. In contrast, an iterator in a traditional LSM-tree based KV-store maintains a min-heap to sort-merge the keys from multiple overlapping sorted runs. In this scenario, a next operation requires reading keys from multiple sorted runs for logN comparisons.

**REMIX skips the sorted runs that are not on the search path.** A seek operation with REMIX requires a binary search in the target group. Only those sorted runs containing the keys on the search path will be accessed at the search time. In the best case, if a range of target keys reside in one run, such as the group (31, 43, 52, 67) in Figure 3, only one sorted run (R2 in the example) will be accessed. However, a merging iterator must access every sorted run in a seek operation.

Furthermore, the substantially reduced seek cost also allows for efficient point queries (e.g., GET) on multiple sorted runs indexed by a REMIX without Bloom filters. We will comprehensively evaluate the point query efficiency in § 5.1.

3.4 Storage Cost of REMIX

A REMIX consists of three components: anchor keys, cursor offsets, and run selectors. We define D to be the maximum number of keys in a group. REMIX stores one anchor key every D keys, requiring 1/D of the total key size in a level on average. Assuming the size of a cursor offset is S bytes, REMIX requires S × R bytes to store the cursor offsets every D keys, where R denotes the number of sorted runs indexed by REMIX. A run selector requires ⌈log₂(R)⌉ bits. Adding all the three parts together, REMIX is expected to store (⌈L + R S + ⌈log₂(R)⌉/8) bytes per key, where L is the average anchor key size.

We estimate the storage cost of REMIX using the average KV sizes of publicly reported Facebook’s production KV workloads [2, 6]. In practice, S is implementation-defined, and R depends on the number of tables being indexed. In the estimation, we use cursor offsets in 4 bytes (S = 4) so that
Table 1: Storage cost of REMIX with real-world KV sizes. BI is short for Block Index. BF is short for Bloom Filter. The last column shows the size ratio of REMIX to KV data.

| Store | Name | Average | Average | Bytes/Key | REMIX data |
|-------|------|---------|---------|-----------|------------|
| | Key Value | Size | BI | BI+BF | R = 8 | R = 8 |
| | | | D=16 | D=32 | D=32 | (D=32) |
| UDB | 27.1 | 126.7 | 1.2 | 2.4 | 4.1 | 2.2 | 1.3 | 1.44% |
| Zippy | 47.9 | 42.9 | 1.2 | 2.4 | 5.4 | 2.9 | 1.6 | 3.16% |
| UP2X | 10.45 | 46.8 | 0.2 | 1.5 | 3.0 | 1.7 | 1.0 | 2.97% |
| USR | 19 | 2 | 0.1 | 1.4 | 3.6 | 2.0 | 1.2 | 9.38% |
| APP | 38 | 245 | 2.9 | 4.2 | 4.8 | 2.6 | 1.5 | 0.91% |
| ETC | 41 | 358 | 4.4 | 5.6 | 4.9 | 2.7 | 1.5 | 0.67% |
| VAR | 35 | 115 | 1.4 | 2.7 | 4.6 | 2.5 | 1.4 | 1.65% |
| SYS | 28 | 396 | 3.3 | 4.6 | 4.1 | 2.3 | 1.3 | 0.53% |

The last column shows the size ratio of REMIX to KV data. REMIX stores ((L + 32)/D + 3/8) bytes/key.

Table 1 shows the storage cost of REMIX for each workload with different D (D = 16, 32, and 64). For comparison, it also shows the storage cost of block index (BI) and Bloom filter (BF) of the SSTable format in LevelDB and RocksDB. An SSTable stores one key and a block handle for each 4 KB data block. The storage cost of block index is estimated by dividing the sum of average key-value size and an approximate block handle size (4 B) by the estimated number of KV-pairs in a 4 KB block. Bloom filters are estimated as 10 bits/key. The storage cost of REMIX varies from 1.0 to 5.4 bytes per key for different D and L. For every key size, increasing D can substantially reduce the storage cost of REMIX. When D ≥ 32, the storage cost of REMIX is comparable to SSTable with and without Bloom filters. The last column (REMIX/BF) shows the size ratio of REMIX to its indexed KV data. In the worst case (the USR store), the REMIX’s size is still less than 1/10 of the KV data’s size.

4 RemixDB

To evaluate the performance of REMIX, we implement an LSM-tree-based KV-store named RemixDB. RemixDB employs the tiered compaction strategy to achieve the best write efficiency [11]. The conventional tiered compaction strategy used in Bigtable [7] and Cassandra [26] has two known issues—long write stalls and high storage cost during compaction [29, 31]. To avoid these issues, recent studies propose to use a partitioned store layout to reduce the time and space requirement of compaction processes [19, 24]. RemixDB adopts this approach by dividing the key space into partitions of non-overlapping key ranges. The table files in each partition are indexed by a REMIX, which provides the best read efficiency in the partition. In this way, RemixDB not only inherits the write efficiency of tiered compaction but also achieves efficient reads with the help of REMIX. The

point query operation (GET) of RemixDB performs a seek operation and returns the key under the iterator if it matches the search key. RemixDB does not use Bloom filters.

Figure 5 shows the system components of RemixDB. Similar to LevelDB and RocksDB, RemixDB buffers updates in a MemTable in the in-memory structure. Meanwhile, the updates are also appended to a write-ahead log (WAL) for persistency. When the size of the buffered updates reaches a threshold, the MemTable is converted into an immutable MemTable for compaction, and a new MemTable is created to receive updates. A compaction in a partition creates a new version of the partition that includes a mix of new and old table files and a new REMIX file. The old version is garbage-collected after the compaction. The following introduces the file structures (§ 4.1), the compaction process (§ 4.2), and the write-optimized WAL (§ 4.2).

4.1 The Structures of RemixDB Files

Table Files Figure 6 shows the table file format in RemixDB. Similar to the SSTable format, RemixDB’s table file consists of multiple data blocks and a metadata block. A data block is 4 KB by default. A large KV-pair that does not fit in a 4 KB block exclusively occupies a jumbo block that is a multiple of 4 KB. Each data block contains a small array of its KV-pairs’ block offsets at the beginning of the block for random access to individual KV-pairs.

The metadata block is an array of 8-bit values, each recording the number of keys of a 4 KB block. Accordingly, a block can contain up to 255 KV-pairs. In a jumbo block, except for the first 4 KB, the remaining ones have their corresponding numbers set to 0 so that a non-zero number always corresponds to a block’s head. With the offset arrays and the metadata block, a search can quickly move to any adjacent block and skip an arbitrary number of keys without accessing the data blocks. Since the KV-pairs are indexed by a REMIX, table files do not contain indexes or filters.

Figure 6: Structure of a table file in RemixDB
REMIX Files  Figure 7 shows the REMIX file format in RemixDB. The anchor keys in REMIX are organized in an immutable B+-tree like index (similar to LevelDB/RocksDB’s block index) that facilitates binary searches on the anchor keys. Each anchor key is associated with a group ID that identifies the target cursor offsets and run selectors of the search. A cursor offset includes a 16-bit block index and an 8-bit key index, shown as blk-id and key-id in Figure 7. The 16-bit block index can index a 256 MB table file (64k 4 KB blocks). Each block can manage 256 KV-pairs using the 8-bit key index.

Multiple versions of a key could exist in different table files of a partition. A range query operation on a REMIX must skip the old versions and return only the newest version of each target key. To simplify searches, multiple versions of a key are ordered from the newest to the oldest on the sorted view. In addition, the newest version of each key will have its run selector’s highest bit set. A forward scan operation will always encounter the newest versions of a key, and then the old versions can be skipped by checking the highest bit of each run selector without comparing any keys.

If a sequence of a key’s multiple versions spans two groups, the anchor key of the second group will correspond to an old version of the key. A search has to move backward to the previous group to retrieve the newest version of the key. To avoid this situation, we move the sequence forward to the second group by appending a few special run selectors as place holders to the end of the first group when constructing a REMIX. We also make sure that the group size is large enough to hold all the versions of a key ($D \geq R$). In this way, an anchor key always points to its newest version. Accordingly, a binary search in a target group needs to check and skip the old versions.

To accommodate the special values mentioned above, each run selector in RemixDB occupies a byte. The highest bit is used to distinguish between new and old keys. A special value 127 ($0x7f$) represents a placeholder. In this way, RemixDB’s run selector can manage up to 127 sorted runs (0 to 126), which is ample in practice.

4.2 Compaction

Real-world workloads often exhibit strong access locality for both read and write operations [2, 6]. There can be multiple instances of a frequently updated key in the persistent tables. This can cause repeated writes and increased storage usage due to the immutability of the tables. To address this issue, TRIAD [3] proposes to exclude the frequently updated keys from a compaction. To this end, the excluded keys must be saved in a file, such as in the write-ahead log, for persistency.

RemixDB takes a similar approach for reducing writes to the tables files. In RemixDB, every key in a MemTable is associated with an 8-bit counter recording the number of updates to the key, including deletions that generate tombstones. A key’s counter increments on each update unless the maximum value (255) has reached. A compaction process skips the keys having an update count greater than a threshold. After a compaction, the excluded keys will be inserted to the current MemTable with their counters halved. If the current MemTable already contains an update or tombstone of an excluded key, the old version’s update count is halved and added to the new version’s counter without replacing the key. Meanwhile, these inserted keys remain valid in the WAL after garbage collection (See § 4.3).

In each partition, the compaction process estimates the compaction cost based on the size of new data entering the partition and the layout of existing tables. Based on the estimation, one of the following procedures is executed:

- Abort: cancel the partition’s compaction and keep new data in the MemTables and the WAL.
- Minor Compaction: write new data to one or multiple new tables without rewriting existing tables.
- Major Compaction: merge new data with some or all of the existing tables.
- Split Compaction: split the partition into a few ones; merge new data with all tables and write to new partitions.

Abort  After a compaction, a partition that sees any new table file will have its REMIX rebuilt. When a small table file is created in a partition after a minor compaction, rebuilding the REMIX can lead to high write amplification (WA). For example, the USR store in Table 1 corresponds to the highest size ratio of REMIX to KV data (9.38%). With this workload, writing 100 MB of new data to a partition with 1 GB old table files will create a 100 MB REMIX. In this scenario, the WA ratio is 2, which is low for LSM-trees. Because the size of a REMIX is dependent on the number of keys in a partition, the WA ratio can be higher if the size of new data is small. To address the issue, RemixDB can abort a partition’s compaction if the estimated WA ratio of the minor compaction is above a threshold. In this scenario, the new KV data should stay in the MemTables and the WAL until the next compaction. The WA ratio threshold is set to 5 in our implementation.

However, in an extreme case, the compaction process cannot effectively move data into the partitions if most of the partitions have their compactions aborted. To prevent this problem, we further limit the size of new data that can stay in the MemTables and the WAL to be no more than 15% of the total size of the new data. The compaction process aborts the compactions in partitions of the highest WA ratios until the WA ratio is below 5 or the size limit has been reached.
Minor Compaction A minor compaction writes new KV data from the immutable MemTable into a partition without rewriting existing table files then rebuilds the REMIX of the partition. Depending on the new data’s size, a minor compaction creates one or a few new table files. Minor compaction is used when the expected number of table files after the compaction (number of existing table files plus the estimated number of new table files) is below a threshold $T$, which is set to 10 in our implementation. Figure 8 shows an example of minor compaction that creates one new table file.

Major Compaction A major (or split) compaction is required when the expected number of table files in a partition exceeds the threshold $T$, as described above. A major compaction sort-merges existing table files into fewer ones. With the reduced number of table files, minor compactions can be performed in the future. The efficiency of major compaction can be estimated by the ratio of the number of input table files to the number of output table files. Figure 9 shows an example of major compaction. In this example, the new data is merged with three small table files, and the partition has three table files in total after the compaction (ratio=3/1). If the entire partition is sort-merged, the compaction needs to rewrite more data but still produces three tables (ratio=5/3) because of table file’s size limit. Accordingly, major compaction chooses the number of input files that corresponds to the highest ratio.

Split Compaction Major compaction may not effectively reduce the number of tables in a partition filled with large tables, which can be predicted by a low estimated input/output ratio, such as 10/9, as described above. In this case, the partition should be split into multiple partitions so that the number of tables can be substantially reduced. Split compaction sort-merges new data with all existing table files in the partition and produces new table files into several new partitions. Figure 10 shows an example of split compaction. To avoid creating many small partitions in a split compaction, the compaction process creates $M$ ($M = 2$ by default) new table files in a partition before switching to the next partition. In this way, a split compaction creates $\lceil E / M \rceil$ new partitions, where $E$ is the expected number of new table files.

4.3 Write-Ahead Log

In an LSM-tree based KV-store, a log file’s size is often only tens of MBs. However, RemixDB’s log needs to store KV-pairs that are updated frequently or returned from aborted compactions. Accordingly, RemixDB requires a relatively large log. In addition, a larger log can also accumulate more updates before a compaction is triggered, which helps to reduce write amplification. On the other hand, a large log file requires a MemTable of a comparable size for indexing and caching the KV-pairs. That being said, the log file and the MemTables have near-constant cost, which is practical given the large memory and storage capacity in today’s datacenters. In RemixDB, the maximum log size is set to 4 GB.

LSM-tree based KV-stores discard the current log file after a minor compaction and replace it with a new log file. To keep some keys in the log, RemixDB would have to rewrite the KV-pairs in the new log file. To minimize rewrites in this scenario, RemixDB creates virtual logs using a layer of indirection (a mapping table) in a log file. A virtual log consists of a sequence of 4 KB blocks. A garbage collection process creates a new virtual log in the same file by reusing blocks in the old virtual log. During this process, when at least 1/4 of the data in a block remains valid, the block will be mapped to the new virtual log as a valid block without rewriting. Otherwise, it is mapped to the new virtual log as an unwritten block with the valid KV-pairs rewritten elsewhere.

Since a remapped valid block can contain a mix of valid and invalid keys, a bitmap is attached to each valid block for masking the unwanted keys. To distinguish unwritten blocks from valid blocks, each block’s first byte contains a 1-bit value, which is flipped every time the block is overwritten. The virtual log records the inverted 1-bit value of an unwritten block in its mapping table. When scanning a virtual log for recovery, a valid block has its 1-bit value equal to the corresponding bit in the mapping table. Otherwise, the block is unwritten. Each virtual log is assigned a unique timestamp so that the newest one can be identified at the start time.

5 Evaluation

In this section, we first evaluate the performance characteristics of REMIX (§ 5.1), and then benchmark RemixDB with a set of micro-benchmarks and Yahoo’s YCSB benchmark tool that emulates real-world workloads [9] (§ 5.2).

The experiments are run on a Dell PowerEdge T440 server with two 2.2 GHz 10-core Intel Xeon Silver 4210 processors and 64 GB of DDR4-2666 DRAM. The server runs a 64-bit Linux (v5.8.7). The experiments run on an Ext4 file system on a 960 GB Intel 905P Optane PCIe SSD.
5.1 Performance of REMIX-indexed Tables

We first evaluate the effectiveness of indexing multiple overlapping table files with REMIX. The baseline is individually-indexed SSTable files that use Bloom filters and merging iterators to facilitate queries.

**Experimental Setup** In each experiment, we first create a set of \( R \) table files (\( 1 \leq R \leq 16 \)), which resembles a partition in a RemixDB or a level in an LSM-tree using tiered compaction. Each table file contains 64 MB of KV-pairs, where the key and value sizes are 16 B and 100 B, respectively. When \( R \geq 2 \), the KV-pairs can be assigned to the tables using two different patterns:

- **Weak locality**: each key is assigned to a randomly selected table, which provides weak access locality since logically consecutive keys often reside in different tables.
- **Strong locality**: every 64 logically consecutive keys are assigned to a randomly selected table, which provides strong access locality since a range query can retrieve a number of consecutive keys from few tables.

Each SSTable contains Bloom filters of 10 bits/key. A 64 MB user-space block cache\(^2\) is maintained for accessing the files.

We measure the single-threaded throughput of three range and point query operations, namely Seek, Seek+Next50, and Get, using different sets of tables created with the above configurations. A Seek+Next50 operation performs a Seek and retrieves the next 50 KV-pairs. In these experiments, the search keys are randomly selected following a uniform distribution. For REMIX, we set the group size to 32 (\( D = 32 \)), and measure the throughput with its in-group binary search turned on and off, denoted by *full* and *partial* binary search, respectively (see §3.2). For point queries (Get), we measure the throughput of SSTables with Bloom filters turned on and off. We run each experiment until the throughput reading is stable. Figures 11 and 12 show the throughput results with the tables of weak and strong access locality, respectively.

**Seek on Tables of Weak Locality** Figure 11a shows the throughput of seek operations using a REMIX and a merging iterator. We observe that the throughput with the merging iterator is roughly 20% higher than that of REMIX with full binary search when there is only one table file. In this scenario, both the mechanisms perform the same number of key comparisons during the binary search. However, when searching in a group, REMIX needs to count the number of occurrences on the fly and move the iterator from the beginning of the group to reach a key for comparison, which is more expensive than a regular iterator.

The throughput of merging iterator quickly drops as the number of table files increases. Specifically, the throughput of two tables is 50% lower than that of one table; a seek on eight tables is more than \( 11 \times \) slower than a seek on one table. The seek time of a merging iterator is proportional to the number of table files. This is because the merging iterator requires a full binary search on every table file. REMIX’s throughput also decreases with more tables files. The slowdown is mainly due to the growing dataset that requires more key comparisons and memory accesses during a search. However, the REMIX with full binary search achieves increasingly high speedups compared with the merging iterator. Specifically, the speedups are \( 5.1 \times \) and \( 9.3 \times \) with 8 and 16 table files, respectively.

The throughput of REMIX decreases by 20% to 33% when the in-group binary search is turned off (with partial binary search). In this scenario, a seek has to linearly scan the target group to find the target key. With \( D = 32 \), the average number of key comparisons in a target group is \( 5 \log D \) with full binary search and 16 (\( D/2 \)) with partial binary search. However, the search cost is still substantially lower than that of a merging iterator. The REMIX with partial binary search outperforms the merging iterator by \( 3.5 \times \) and \( 6.1 \times \), with 8 and 16 table files, respectively.

**Seek+Next50** Figure 11b shows the throughput of range queries that seek and copy 50 KV-pairs to a user-provided buffer. The overall throughput results are much lower than that in the Seek experiments because the data copying is expensive. However, REMIX still outperforms the merging iterator when there are two or more tables. The speedup is \( 1.4 \times, 2.3 \times \), and \( 3.1 \times \) with 2, 8, and 16 table files, respectively. The suboptimal scan performance of the merging iterator is due to the expensive next operation that requires multiple key comparisons to find the next key on the sorted view. For each KV-pair copied to the buffer, multiple KV-pairs must be read and compared to find the global minimum. In contrast, REMIX does not require any key comparisons in the next operations.

In contrast to the substantial performance gap between the two REMIX curves in Figure 11a, the two curves in Figure 11b are very close to each other. This phenomenon is the result of two effects: (1) the next operations dominate the execution time and (2) the linear scanning of a seek operation in a group warms up the block cache, which makes the future next operations faster.

**Point Query** Figure 11c shows the results of the point query experiments. The REMIX’s curve is slightly lower than its counterpart in Figure 11a because a get operation needs to copy the KV-pair after a seek using the REMIX. The searches on the individual tables files (SSTables with Bloom filters) outperform that of REMIX when there are fewer than 14 tables. The reasons for the differences are two-fold. First, a search can be effectively narrowed down to only one table file at a small cost on checking the Bloom filters. Second, searching in one SSTable is faster than that on a REMIX managing many more keys. In the worst case, the REMIX’s throughput is 20% lower than that of Bloom filters (with 3 tables), but their performance are comparable. Unsurprisingly, the searches on SSTables without Bloom filters are much slower when there are more than two tables.

\(^2\)LevelDB’s LRU Cache implementation in util/cache.cc.
Performance with Tables of Strong Locality  Figure 12 shows the range and point query performance on tables with strong access locality. The results in Figures 12a and 12b follow a similar trend of their counterparts in Figure 11. In general, the improved locality allows for faster binary searches since in this scenario the last a few key comparisons can often use keys in the same data block. However, the throughput of the merging iterator remains low because of the intensive key comparisons that dominate the search time. The REMIX with partial binary search improves more than that with full binary search. This is because improved locality reduces penalty on the scanning in a target group, where fewer cache misses are incurred in each seek operation.

The point query performance of REMIX also improves due to the strong locality that speeds up the underlying seek operations, as shown in Figure 12c. Meanwhile, the results of Bloom filters stay unchanged because the search cost is mainly determined by the false-positive rate and the search cost on individual tables. As a result, REMIX is able to outperform Bloom filters when there are more than 9 tables.

Group Size (D) We further evaluate REMIX’s range query performance using different group sizes (D \in \{16, 32, 64\}) on eight table files. The other configurations are the same as the previous experiments. Figure 13 shows the performance results. The throughput of seek-only operations exhibits the largest variations with different Ds when the in-group binary search is turned off. This is because the linear scanning in a seek operation adds a significant cost with a large D. On the other hand, the differences become much smaller with full binary search. In the meantime, a larger group size still leads to higher overhead because of the slower random access speed within a group. In the Seek+Next50 experiments, the data copying dominates the execution time and there is no significant difference between using different Ds.

5.2 Performance of RemixDB

The following evaluates the performance of RemixDB, a REMIX-indexed KV-store based on LSM-tree.

Experimental Setup  We compare RemixDB with state-of-the-art LSM-tree based KV-stores, including Google’s LevelDB [21], Facebook’s RocksDB [15], and PebblesDB [35]. LevelDB and RocksDB adopt the leveled compaction strategy for balanced read and write efficiency. PebblesDB adopts the tiered compaction strategy for improved write efficiency at the cost of having more overlapping sorted runs.

LevelDB (v1.22) supports only one compaction thread. For RocksDB (v6.10.2), we use the configurations provided by its official Tuning Guide\(^3\) [16]. Specifically, RocksDB can have at most three MemTables (one more immutable MemTable than LevelDB). Both RocksDB and RemixDB are configured with four compaction threads. RemixDB, LevelDB, and RocksDB are all configured to use 64-MB

---

\(^3\)The configuration for “Total ordered database, flash storage.”
table files. For PebblesDB (#703bd01), we use the default configurations in its db_bench benchmark program. For fair comparisons, we disable compression and use 4 GB block cache in every KV-store. All the KV-stores are built with optimizations turned on (release build).

In our experiments, we choose three value sizes—40, 120, and 400 bytes. They roughly represent the small (ZippyDB, UP2X, USR), medium (UDB, VAR), and large (APP, ETC, SYS) KV sizes in Facebook’s production systems [2, 6]. We use 16-byte fixed-length keys, each containing a 64-bit integer using hexadecimal encoding.

**Range Query** The first set of experiments focuses on how different KV sizes and access patterns affect the search efficiency of the KV-stores. In each experiment, we first sequentially load 100 million KV-pairs into a store using one of the three value sizes. After the loading, we measure the throughput of seek operations using four threads with three access patterns, namely sequential, Zipfian, and uniform.

As shown in Figure 14, each set of results shows a similar trend. While RemixDB exhibits the highest throughput, LevelDB is also at least $2 \times$ faster than RocksDB and PebblesDB. The sequential loading produces non-overlapping table files in every store, which suggests that a seek operation needs to access only one table file. However, a merging iterator must check every sorted run in the store even though they are non-overlapping, which dominates the execution time of a seek operation if the store has multiple sorted runs. Specifically, each $L_0$ table in LevelDB and RocksDB is an individual sorted run, but each $L_i$ ($i > 0$) contains only one sorted run; PebblesDB allows multiple sorted runs in every level. However, LevelDB outperforms RocksDB by at least $2 \times$ even though they both use leveled compaction. We observe that RocksDB keeps several tables (eight in total) at $L_0$ without moving them into a deeper level during the sequential loading. In contrast, LevelDB directly puts a table to a deep level ($L_2$ or $L_3$) if it does not overlap with other tables, which leaves LevelDB’s $L_0$ always empty. Consequently, a seek operation in RocksDB needs to sort-merge at least 12 sorted runs on the fly, while the number is only 3 to 4 in LevelDB.

The seek performance is sensitive to access locality. The weaker access locality leads to increased CPU and I/O cost on the search path. In each experiment of a particular value size, the throughput with uniform access pattern is about 50% lower than that of sequential access. Meanwhile, the performance with sequential access is less sensitive to value size because the memory copying cost is insignificant.

The second set of experiments evaluates the range-scan performance with different store sizes and query lengths. Each experiment loads a fixed-size KV dataset with 120 B value size into a store in random order, then performs range-scans with four threads using the Zipfian access pattern.

As shown in Figure 15, RemixDB outperforms the other stores in all the experiments. However, the performance differences between the stores become smaller with longer scans. The reason is that a long range-scan exhibits sequential access pattern on each sorted run where multiple consecutive KV-pairs can be retrieved during the scan. In the meantime, the memory-copying adds a constant overhead to every store.

As the store size increases to 256 GB, the throughput of LevelDB quickly drops to the same level as RocksDB. Since the stores in the experiments are configured with a 4 GB block cache, the cache misses lead to intensive I/Os that dominate the query time. While RocksDB exhibits high computation cost for having too many $L_0$ tables with a small store size, that cost is overshadowed by the excessive I/Os in large stores. Meanwhile, RemixDB maintains the best access locality because it incurs a minimal amount of random accesses and cache misses by searching on a REMIX-indexed sorted run.

**Write** We evaluate the write performance of each store by inserting a 64 GB KV dataset to an empty store in random order using one thread. The dataset has 500 million KV-pairs, and the value size is 120 B. We measure the throughput and the total write I/O to the SSD. As shown in Figure 16, Both RemixDB and PebblesDB show high throughput because they employ the write-efficient tiered compaction strategy. Their total writes to the SSD are 272 GB and 196 GB, corresponding to WA ratios of 4.25 and 3.06, respectively. Comparatively, LevelDB and RocksDB adopts the leveled compaction strategy, which leads to high WA ratios of 12.3 and 13.5, respectively. On the other hand, RocksDB employs four compaction threads to exploit the SSD’s I/O bandwidth. LevelDB only supports one compaction thread, and it shows a much lower throughput than RocksDB.
Table 2: YCSB Workloads.

| Workloads | A | B | C | D | E | F |
|-----------|---|---|---|---|---|---|
| Operations | R: 50% | R: 95% | C | D | E | F |
|            | U: 50% | U: 5% | R: 95% | S: 95% | I: 5% | M: 50% |
| Req. Dist. | Zipfian | Zipfian | Latest | Zipfian | Zipfian | Zipfian |

The YCSB Benchmark The Yahoo Cloud Serving Benchmark (YCSB) [9] is commonly used for evaluating KV-store performance under realistic workloads. We use the stores constructed in the write experiments and run the YCSB workloads from A to F with four threads. The details of the workloads are described in Table 2. In workload E, a Scan operation performs a seek and retrieves the next 50 KV-pairs.

As shown in Figure 17, RemixDB outperforms the other stores by more than 2× except in workload D, where the read requests (95%) query the most recent updates produced by the insertions (5%). This access pattern exhibits strong access locality, and most of the requests are directly served from the MemTable(s) in every store. Meanwhile, LevelDB’s performance (1.1 MOPS) is hindered by slow insertions caused by the single-threaded compaction.

Even though REMIX does not show an advantage over Bloom filters in the micro-benchmarks (see Figure 11c), RemixDB outperforms the other stores in workloads B and C, where point queries are the dominant operation. The reason is that a point query in the multi-level LSM-tree has a high cost selecting the candidate tables on the search path. Specifically, for each $L_0$ table, about two key comparisons are used to check if the search key is covered by the table. If the key is not found at $L_0$, a binary search is used to select a table at each deeper level $L_i$ ($i \geq 1$) until the target key is found. Furthermore, a Bloom filter is about 600 KB for a 64 MB table in this setup. Each accessed Bloom filter can perform up to seven random memory accesses, which leads to excessive cache misses in a large store [17]. The REMIX-indexed partitions in RemixDB form a global sorted view, on which a point query can be quickly answered with a binary search.

6 Related Work

Improving Search with Filters Bloom filters [4] are indispensable for LSM-tree based KV-stores in reducing computation and I/O cost of point queries on a multi-leveled store layout [10]. However, range queries cannot be handled by Bloom filters because the search targets are implicitly specified by range boundaries. Prefix Bloom filters [14] can accelerate range queries [15,21], but it can only handle closed-range queries on common-prefix keys (with an upper bound). Succinct Range Filters (SuRF) [40] support both open-range and closed-range queries. The effectiveness of SuRF is highly dependent on the distribution of keys and query patterns. Rosetta [30] uses multiple layers of Bloom filters to achieve lower false positive rates than SuRF. However, it does not support open-range queries and has prohibitively high CPU and memory cost with long range queries. A fundamental limitation of the filtering approach is that it cannot reduce search cost on tables whose respective filters produce positive results. When the keys in the target range are in most of the overlapping tables, range filters do not speed up queries but cost more CPU cycles in the search path. In contrast, REMIX directly attacks the problem of having excessive table accesses and key comparisons when using merging iterators in range queries. By searching on a global sorted view, REMIX improves range query performance with consistently low computation and I/O cost.

Improving Search with Efficient Indexing KV-stores based on B-tree or B+-tree [22, 32] achieve optimal search efficiency by maintaining a global sorted view of all the KV data. These systems require in-place updates to disk, which leads to high write amplification and low write throughput. KVell [27] achieves very fast reads and writes by employing a volatile full index to manage unordered KV data on disk. However, the performance benefits come at costs, including high memory demand and slow recovery. Similarly, SLMD-DB [24] stores a B+-tree [23] in non-volatile memory (NVM) to index the KV data on disk. This approach does not have the above limitations, but it requires special hardware support and increased software complexity. These limitations are also found in NVM-enabled LSM-trees [25, 39]. Wisckey [28] stores long KV-pairs in a separate log to reduce index size for search efficiency. However, the approach requires an extra layer of indirection and does not improve performance with small KV-pairs that are commonly seen in real-world workloads [6]. REMIX is not subject to these limitations. It accelerates range queries in write-optimized LSM-tree based KV stores by creating a space-efficient persistent sorted view of the KV data.

Read and Write Trade-offs in LSM-trees Dostoevsky and Wacky [11, 12] navigate LSM-tree based KV-store designs with different merging policies to achieve the optimal trade-off between read and write. Tiered compaction has been widely adopted for minimizing write amplification in LSM-tree based KV-stores [19, 26, 35, 37]. The improvements on write performance often come with mediocre read performance. Instead of making trade-offs, REMIX addresses the issue of slow read in tiered compaction. It achieves fast range query and low write amplification simultaneously.

7 Conclusion

We introduce REMIX, a compact multi-table index data structure for fast range queries in LSM-trees. The core idea is to record a global sorted view of multiple table files for efficient search and scan. Based on REMIX, RemixDB effectively improves range query performance while preserving low write amplification using tiered compaction.
References

[1] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: a database benchmark based on the facebook social graph. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013 (2013), K. A. Ross, D. Srivastava, and D. Papadias, Eds., ACM, pp. 1185–1196.

[2] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. 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 (2012), P. G. Harrison, M. F. Arlitt, and G. Casale, Eds., ACM, pp. 53–64.

[3] Balmau, O., Didona, D., Guerraouï, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. 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 (2017), D. D. Silva and B. Ford, Eds., USENIX Association, pp. 363–375.

[4] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (July 1970), 422–426.

[5] Braun, L., Etter, T., Gasparis, G., Kaufmann, M., Kossmann, D., Widmer, D., Avitzur, A., Iliopoulos, A., Levy, E., and Liang, N. Analytics in motion: High performance event-processing and real-time analytics in the same database. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (New York, NY, USA, 2015), SIGMOD ’15, Association for Computing Machinery, pp. 251–264.

[6] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. C. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In 18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020 (2020), S. H. Noh and B. Welch, Eds., USENIX Association, pp. 209–223.

[7] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26, 2 (2008), 1–26.

[8] Chen, G. J., Wiener, J. L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., and Yilmaz, S. Realtime data processing at facebook. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016 (2016), F. Özcan, G. Koutrika, and S. Madden, Eds., ACM, pp. 1087–1098.

[9] Cooper, B. F., Silverstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing (New York, NY, USA, 2010), SoCC ’10, Association for Computing Machinery, pp. 143–154.

[10] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal navigable key-value store. In Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017 (2017), S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds., ACM, pp. 79–94.

[11] Dayan, N., and Idreos, S. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018 (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 505–520.

[12] Dayan, N., and Idreos, S. The log-structured merge-bush & the wacky continuum. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019 (2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 449–466.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon’s highly available key-value store. In Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007 (2007), T. C. Bressoud and M. F. Kaashoek, Eds., ACM, pp. 205–220.

[14] Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. E. Longest prefix matching using bloom filters. In Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August
[34] Pilman, M., Bocksrocker, K., Braun, L., Marroquin, R., and Kossmann, D. Fast scans on key-value stores. *Proc. VLDB Endow.* 10, 11 (2017), 1526–1537.

[35] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. 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* (2017), ACM, pp. 497–514.

[36] Raju, P., Ponnapalli, S., Kaminsky, E., Oved, G., Keener, Z., Chidambaram, V., and Abraham, I. mLSTM: Making authenticated storage faster in Ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (Boston, MA, July 2018), USENIX Association.

[37] Scylladb. scylladb/scylla. [https://github.com/scylladb/scylla](https://github.com/scylladb/scylla).

[38] Wu, K., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)* (Renton, WA, July 2019), USENIX Association.

[39] Yao, T., Zhang, Y., Wan, J., Cui, Q., Tang, L., Jiang, H., Xie, C., and He, X. Matrixkv: Reducing write stalls and write amplification in lsm-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020 (2020)*, A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 17–31.

[40] Zhang, H., Lim, H., Leis, V., Andersen, D. G., Kaminsky, M., Keeton, K., and Pavlo, A. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 323–336.