Received 13 September 2022, accepted 2 November 2022, date of publication 4 November 2022,
date of current version 11 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3219859

RESEARCH ARTICLE

Hybrid Transactional/Analytical Processing Amplifies IO in LSM-Trees

JONGBIN KIM, JAECHAN AHN, KITAEK LEE, MINSOO RYU, AND HYUNGSOO JUNG
Department of Computer Science, Hanyang University, Seoul 04763, South Korea

Corresponding author: Hyungsoo Jung (hyungsoo.jung@hanyang.ac.kr)

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) funded by the Korean Government [Ministry of Science and ICT (MSIT)] through Decentralized High Performance Consensus for Large-Scale Blockchains under Grant 2021-0-00590.

ABSTRACT The log-structured merge tree (LSM-tree) has become an essential component in many key-value systems and expanded its scope to full-fledged database engines (e.g., MyRocks). In the database landscape, vendors face growing customer demands for real-time analytic solutions to handle hybrid transactional/analytical processing (HTAP) workloads that pose significant challenges. Among the challenges is IO amplification that drives system designers to rethink write-optimized engines to survive HTAP loads. This paper follows the same philosophy, reexamines LSM-trees used for database systems, and rethinks IO amplification under HTAP loads to shed some light on practical remedies for upcoming challenges. We propose two practical techniques to alleviate IO amplification: 1) *aligned compaction* for reducing write amplification, 2) *snapshot filters* for reducing read amplification. The two techniques are lightweight (i.e., near-zero resource consumption) and are compatible with state-of-the-art methods. We integrated our techniques into RocksDB and demonstrated that the modified RocksDB exhibits reduced IO amplification under HTAP workloads with negligible resource consumption.

INDEX TERMS LSM-tree, key-value store, MVCC, HTAP, I/O amplification.

I. INTRODUCTION

Log-structured merge trees (LSM-trees) have been mainstream structures for many key-value storage systems. The elegance stands from its scalable architecture for update-intensive workloads and its competitive read performance. The past decade has seen that numerous data management systems [1], [2], [3], [4], [5] have adopted LSM-based architecture for their storage engine; for example, Facebook has released MyRocks [6], MySQL built atop RocksDB. As LSM-trees expand their scope into databases as such, their underlying architecture must address the same challenges that traditional relational database engines strive to survive hybrid transactional/analytical processing (HTAP) workloads.

HTAP combines transactions (i.e., updates) with data analytics (i.e., reads), and there is a growing demand for HTAP systems from users; commercial products use extract-transform-load (ETL) pipelines while striving to maintain data freshness between transactional and analytic databases. ETL-based systems deliver decent performance on data analytics acting on reshaped data ingested from transaction engines but lose their real-time degree of data analytics mainly due to long delays in their ETL workflows. As more organizations invest in real-time data analytics, recent years have seen commercial solutions [7], [8], [9], [10] in industry and research outcomes [11], [12], [13], [14], [15] from academia; all of which evolve towards a single system capable of HTAP. The trend towards a single HTAP system requires the system to combine two types of data processing, despite their heterogeneous nature, into a single system. One can approach this matter by adapting the existing systems specialized for a single type of workload to HTAP [16], e.g., taming write-optimized databases to be suitable for HTAP.

When the LSM-tree-based databases encounter HTAP, IO amplification remains notable and adversely affects the systems; the systems suffer *write amplification* from continuous compaction triggered by transactional loads and *read amplification* from massive scans by analytic queries.

The associate editor coordinating the review of this manuscript and approving it for publication was Byung-Gyu Kim.

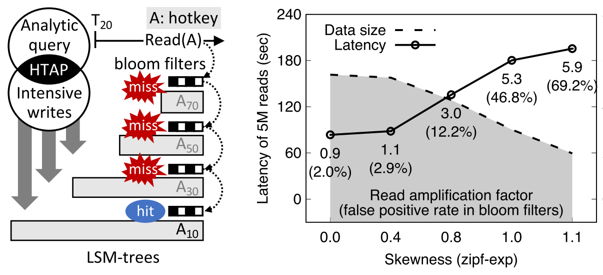


FIGURE 1. **Read amplification under HTAP loads.**

As prior studies [17], [18], [19], [20] revealed valuable insight about the trade-off between query latency and update throughput, compaction strategies and underlying storage layouts considerably influence the degree of IO amplification. The essence of the dilemma under HTAP loads is to pursue the dual goals of reducing all aspects of amplification instead of being lopsided in ripening the current LSM-trees.

What makes situations more demanding and unique under HTAP loads is, as complex analytic queries often run longer, massive updates trigger repeated compaction that would push data versions required by the analytic queries down further to the higher levels. As illustrated in Figure 1, a long-running analytic query (i.e., T_{20}) reading a key A takes longer to find the required version (i.e., A_{10}) when transactions keep updating the key. The query spends more time searching data versions since it traverses all the levels from the root without any chance to skip unnecessary visits. The unexpected phenomenon, in hindsight, results from futile attempts to exploit bloom filters not worth aiding consistent reads having snapshots, leading to the growing false-positive rate and amplifying read IO. This observation motivates our work.

We observe redundant IO in LSM-trees employing a general leveled compaction policy. To address the problem, we propose two optimization techniques: *aligned compaction* and *snapshot filters*. Aligned compaction (§IV) targets to reduce write amplification by excluding an overhang of non-overlapping key ranges from compaction operations, and snapshot filters (§V) aid analytic queries to skip invisible key-values generated by recent update transactions, thus improving read amplification. Our proposed methods take negligible resource consumption in terms of computation and memory, and they are orthogonal to the state-of-the-art techniques sharing the same goal. These optimizations can be applied without concern for significant architectural change and are compatible with prior arts.

We validate the ideas by putting them together in RocksDB, a popular open-source LSM-tree-based key-value store adopting a leveled compaction policy. To execute complex SQL queries of HTAP workloads over the modified RocksDB, we use MyRocks for evaluation, and the experimental results demonstrate that the optimized RocksDB reduces IO amplification without sacrificing system performance.

This paper makes the following contributions:

1) We measure IO amplification in an LSM-tree-based SQL database under HTAP workloads, and confirm the presence of redundant IO (§III).
2) To alleviate IO amplification, we propose two practical optimization techniques that are compatible with the state-of-the-art methods (§IV,§V).
3) We apply the techniques to MyRocks with less than five hundred lines of code and enhance IO amplification under HTAP workloads (§VI).

II. BACKGROUND AND RELATED WORK

A. LSM-TREES

The original insert-optimized design of LSM-trees, *log-structured writing* with *deferred compaction*, deals well with general write-intensive loads; all modifications, including delete operations, are made in an out-of-place way instead of overwriting old data. LSM-trees utilize an in-memory buffer embedding skip lists to enhance performance further; it appends incoming writes to the in-memory buffer and later flushes the buffer to storage as a sorted-string table (SSTable). SSTables are immutable and will be sorted into new SSTable files through compaction. The two goals of compaction are 1) to sort-merge multiple SSTables to maintain a bounded number of sorted runs to prevent read operations from accessing an unbounded number of SSTables, and 2) to physically remove the entries that are (logically) overwritten or deleted. Through repeated compaction, LSM-trees form a hierarchical structure, consisting of multiple levels of an exponentially growing size, and higher levels contain older versions of data than that of a lower level. The size ratio between levels, so-called fanout, is usually configurable, and compaction policy (a.k.a., merge policy) determines the layout of each level.

Two policies are of prevalence, *leveled* and *tiered*; leveled compaction maintains every entry in the same level in a single sorted run, while tiered compaction allows multiple sorted runs to be in the same level, which could overlap with each other in terms of the keyspace. There is no clear winner; leveled compaction favors queries to find the target key by at most one lookup per level, but it requires each entry to be written multiple times to storage until it reaches the highest level, resulting in higher write amplification. Tiered compaction facilitates only one write per level, but queries should traverse multiple sorted runs in the same level, leading to higher read amplification. Because of the trade-off between write and read amplification, popular industrial LSM-tree-based engines take different compaction policies according to their purpose [21]; LevelDB [1] implements leveled compaction while HBase [5] and BigTable [4] adopt tiered compaction. Both policies are supported in RocksDB [2] and Cassandra [3] as a configurable option, although the default policies are leveled and tiered, respectively. Also, prior arts from academia [19], [20], [22], [23] take the hybrid strategy of leveled and tiered compaction to reduce the amplifications, by considering the given workload characteristic.

B. LEVELED COMPACTION IN RocksDB

RocksDB enforces the leveled compaction policy, the focus of this paper, as default. RocksDB's compaction behaves similarly to LevelDB since it originated from a fork of LevelDB. In each level of RocksDB, a single sorted run comprises multiple SSTables, of which each covers a distinct key range, except for the lowest level (L_0), where SSTables flushed from the memory buffer are stacked (i.e., tiered) without being sorted. Once the number (for L_0) or the total size (for L_N, $N > 0$) of SSTables in a level exceeds a certain threshold, compaction is triggered. One or more SSTables in the level, L_N, and the overlapping SSTables in the next level, L_{N+1}, are picked as inputs of the compaction. Compaction first reads the inputs to memory, sort-merges them, and writes the merged SSTables into L_{N+1}. After the output SSTables are written to storage, the obsolete input SSTables are safely deleted from the disk. When the new SSTables in L_{N+1} are merged with existing SSTables in L_{N+2}, it can push the entries in the SSTables to a higher level, or it can merge the entries with newly generated SSTables in L_N so that it rewrites the entries in the same level. However, the entries never move back to lower levels, which is an important invariant facilitating multi-version concurrency control (MVCC) with LSM-trees.

C. SNAPSHOT READ IN RocksDB

Since all modifications are appended out-of-place in LSM-trees, updates create multiple version entries with the same key, making MVCC a natural fit for transaction processing. To distinguish the creation time of each version entry, RocksDB puts a monotonically increasing sequence number next to the key, indicating the commit timestamp of the entry. A read transaction takes the sequence number as a snapshot when it starts and uses it to test the visibility of version entries to guarantee point-in-time consistency; a transaction can see the latest version entry committed before the transaction's snapshot. Searching a key entry navigates LSM-trees from the memory buffer toward the highest level of RocksDB (i.e., new-to-old version search), and it stops navigating once detecting the first entry satisfying the consistent read. In contrast, a range scan starts searching from all levels, including memory buffer, since visible version entries of a given key range could span multiple levels.

There are two commonly used auxiliary structures, also adopted in RocksDB, to reduce the search cost at the expense of relatively small memory space: *fence pointers* and *bloom filters*. Fence pointers represent key ranges covered by each SSTable, allowing binary search on a sorted run in advance in memory to narrow down search space to a few candidate SSTables. Bloom filters provide a membership query of a search key against each SSTable which answers the presence of the given key without traversing the SSTable. We have seen good progress in this area of optimization [18], [24], [25], [26], [27], *all the efforts made to narrow the search space on a key basis*.

III. MOTIVATION

This section describes the hidden costs shaded beneath the well-known IO amplification issue of LSM-trees. We first look into compaction to comprehend the underlying nature of redundant writes. Then, we unveil undesirable read amplification arising from MVCC systems under HTAP workloads.

A. WRITE AMPLIFICATION IN LSM-TREES

Deferred compaction originally intended to expedite the upsurge of incoming writes is alleged to incur some IO in LSM-trees, but its leveled compaction would considerably worsen write amplification. During compaction, LSM-trees partition the sort-merged output into fixed-size SSTables, dividing the key range into non-overlapping groups. The rationale for this policy lies in simplicity and clarity, thus widely accepted as a good design decision across conventional LSM-trees. However, we observe that such a faithful design principle triggers a non-negligible impact on write amplification.

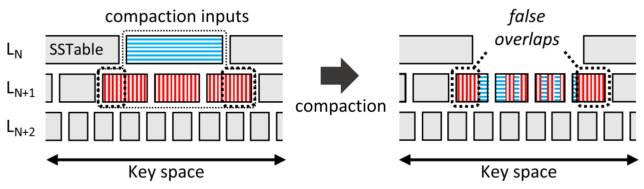


FIGURE 2. Redundant IO in leveled compaction.

Figure 2 shows the phenomenon of the aforementioned write amplification. The compaction first selects input SSTables from the two levels (L_N and L_{N+1}), which may consist of two regions: 1) the overlapping key range that both levels share, and 2) the *overhanging key range* of distinct non-overlapping regions, also denoted as *false overlaps* by Lim et al. [28]. On compaction, both regions are involved in the sort-merge phase, creating an output containing the combination of the two levels' key range. The overlapping key range undergoes a meaningful sort-merge process because the keys may be out-of-order, requiring an arrangement. Unlike the former, false overlaps are a redundant portion of the sort-merge process since the ranges do not interleave between the two levels. Such redundancy during compaction demands unnecessary reads and writes to the same level, amplifying the write IO.

The essence behind the inefficiency is the existence of false overlaps, and eliminating such overhangs, if possible, would reduce a considerable proportion of redundant IO in compaction. Unfortunately, the current fixed-size partition policy does not consider key ranges its decisive factor, making false overlaps inevitable. We measured the ratio of false overlaps among the entire compaction inputs in RocksDB with the default configuration to observe the severity. Under *online transaction processing* (OLTP) workloads, 743 million entries in total have participated in compaction operations, and *12.6% of the entries turned out to be false overlaps*. Hence, **our focus is inventing a lightweight technique to reduce false overlaps**.

B. READ AMPLIFICATION IN LSM-TREES UNDER HTAP

Although LSM-trees run the continual compaction to maintain sorted runs for speeding up reads, its hierarchical structure is less favorable to read amplification due to the search path across multiple levels. Despite considerable efforts to alleviate the problem, less has happened to version searching crucial to consistent reads in MVCC databases. In this regard, we argue that another design dimension is worth investigating when LSM-trees face HTAP loads; its read amplification rises when MVCC systems run long-running analytic queries, as the database community has seen in numerous prior studies [7], [13], [29], [30].

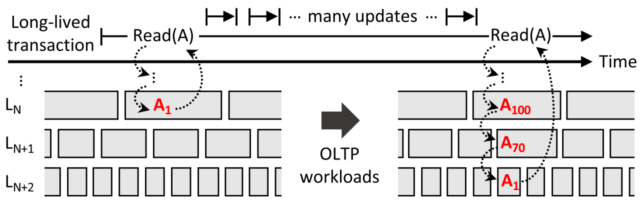


FIGURE 3. Read amplification of data version searching.

Figure 3 illustrates why read amplification gets deteriorated under HTAP workloads. In the early stage, a transaction looking for the entry with key A first finds a visible version A_1 — whose sequence number is 1 —in a relatively low level L_N. As the transaction remains alive, versions pile up from massive updates by OLTP transactions, placing newer versions (A_{70} and A_{100}) on the lower levels (L_{N+1} and L_N). Now aged into a long-lived transaction, the same read on A should fetch the visible version from a higher level (L_{N+2}) than that of its first read (L_N), requiring two more level traversals.

Under HTAP workloads, transactions executing complex *online analytic processing* (OLAP) queries tend to run longer, i.e., long-lived transactions. It spends much time reading a large portion of the database and computing complex operations (e.g., join), while OLTP transactions extensively pile up the new entries onto the lower levels. Thus, the queries demand that more entries be fetched from the higher levels over time, aggravating read amplification. Unfortunately, fence pointers and bloom filters are not much help in this situation since the core matter is not on the key's existence but the visibility of the versions. Hence, **our primary focus is inventing a lightweight technique to aid analytic queries by reducing unnecessary IO**.

IV. ALIGNED COMPACTION

In this section, we describe the design and implementation details of aligned compaction. The proposed solution alleviates write amplification by eliminating the overhang of false overlaps.

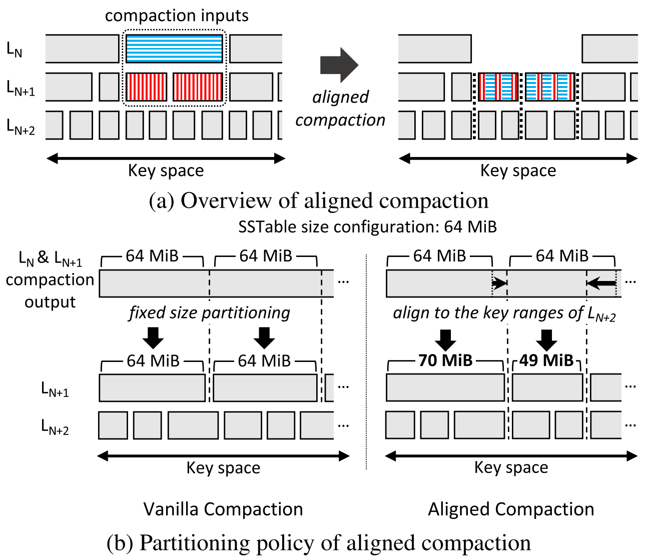


FIGURE 4. Aligned compaction.

A. DESIGN RATIONALE

The legacy compaction algorithm splits sort-merged entries into multiple SSTables depending on the fixed target size of the system configuration. Thus, the SSTables involved in the compaction are likely to create false overlaps, triggering the overhang issue. The central insight of aligned compaction is to relax the constraint of the static size configuration. By allowing dynamic sizes, the SSTables are aligned, preventing false overlaps in the first place. Figure 4a shows the layout of the disk components organized by aligned compaction. SSTables spanning the same key range are aligned with each other instead of containing false overlaps. Hence, the SSTables can be compacted without inducing write amplification for non-overlapping key ranges. Aligned compaction writes the output SSTables while preserving the aligned layout. This repetition keeps all the levels well-arranged since compaction affects the layout of disk components.

Another merit of our proposed solution is practicality. In the context of LSM-trees, the layout of an SSTable is tightly bounded with RocksDB operations, deeply incorporating with operations on a massive scale of data up to tens of petabytes. Hence, the fact that aligned compaction does not involve any changes on the existing layout of an SSTable introduces high compatibility and feasibility to the technique since legacy operations are intact. Such orthogonality allows the implementation of aligned compaction practical, requiring only minor changes to the original compaction algorithm.

B. MAIN DESIGN

To make SSTables aligned with a key range, we changed the partitioning policy used in the output SSTable creation, as shown in Figure 4b. When aligned compaction builds new sort-merged entries into an output level (L_{N+1}), it partitions them into different sizes to align the output SSTables to the ones in the adjacent higher level (L_{N+2}). The new partitioning policy exploits two properties of the LSM-trees:

1) For any pair of overlapping SSTables in the adjacent levels L_N and L_{N+1}, the SSTable in the higher level, L_{N+1}, is always created before the one in the lower level, L_N.
2) Higher-level SSTables usually cover narrow key ranges with high density.

Property 1 guarantees the preexistence of the SSTables in L_{N+2}, and subsequent compaction for the new SSTables and the SSTables in L_{N+2} will be executed without the redundant behavior since they tightly overlap without false overlapping key ranges. One exception is that the corresponding key range of L_{N+2} is empty in the first place, and in this case, the aligned compaction follows the fixed-size partitioning policy.

Allowing SSTables to absorb more or fewer entries than the specified configuration theoretically makes the size of SSTables unbounded. That being said, by Property 2, we can expect that only putting a small number of more or fewer entries into output SSTables is enough to align them with that in the adjacent higher level. Moreover, to systematically bound the size variance of SSTables, we set a configurable threshold (2 by default) to prevent output SSTables from being far from the target size under the exceptional cases. For instance, once the size of a new SSTable being created reaches the target size multiplied by the threshold, the compaction logic cuts the boundary of the SSTable even if it would partially overlap with the higher-level SSTable. Also, until the size of the new SSTable reaches the target size divided by the threshold, the SSTable keeps taking more entries even if it meets the aligned point.

Aligned compaction works regardless of workload characteristics, e.g., key type, range, and distribution. It decides the key range of each partition on the fly according to the prior dataset using the current key comparator of a target system instead of fitting to the pre-defined static partitions. It does not take additional memory and storage space, and it involves negligible extra computation to check the alignment.

C. IMPLEMENTATION ON RocksDB

The compaction process of RocksDB involves three steps: 1) pick the input SSTables, 2) sort-merge them, and 3) pipeline the result to the output level, partitioning the stream into fixed-size SSTable files. Since the essence of the implementation of aligned compaction lies within the partitioning policy, the point-of-interest is the last step, where we choose the partitioning point for output files.

Algorithm 1 describes the general sequence of our aligned compaction. The while loop iterates through the entries of the sort-merge result by fetching an entry from the merging iterator. The key of each entry is compared with the grandparent's range (lines 6 and 10) to seek the partition point. Although Property 2 holds, extreme cases are to be taken care of by systematical restrictions (lines 4 and 7). The maximum size of an SSTable is bound to a product of the configured SSTable size (i.e., `base_sstable_size`) and a configurable variance factor. The minimum size upholds the same logic, using the variance factor as a divisor.

Algorithm 1 Pseudocode of Aligned Compaction

```
Data:
    $G$, overlapping SSTables in output_level+1
    $S_{out}$, output SSTable file
1  $grandparent \leftarrow G.begin()$
2  while $merging\_iterator.Valid()$ do
3      $merging\_iterator.NextAndGetResult(entry)$
4      if $S_{out}.size \geq max\_sstable\_size$ then
5          $S_{out} \leftarrow$ NewSSTable()     // open new SSTable
6      else if $grandparent \neq G.end()$ and $entry.key > grandparent.largest\_key$ then
7          if $S_{out}.size \geq min\_sstable\_size$ then
8              $S_{out} \leftarrow$ NewSSTable()
9          $grandparent \leftarrow grandparent.next()$
10     else if $grandparent = G.end()$ or $entry.key \leq grandparent.smallest\_key$ then
11         if $S_{out}.size \geq base\_sstable\_size$ then
12             $S_{out} \leftarrow$ NewSSTable()
13     $S_{out}.append(entry)$
```

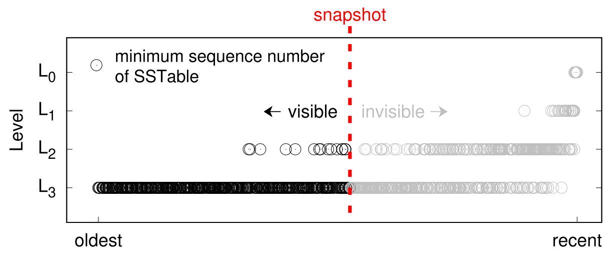


FIGURE 5. Correlation of snapshot age and its visible SSTable.

V. SNAPSHOT FILTERS

This section describes snapshot filters to alleviate read amplification by utilizing the minimum sequence number to filter out unnecessary access to invisible SSTables.

A. DESIGN RATIONALE

An MVCC transaction only sees a key-value entry whose sequence number is lower than its snapshot. Suppose a minimum sequence number of an SSTable representing the oldest entry inside the table is higher than the snapshot. In that case, it implies that all entries of the SSTable are invisible to the transaction. Since key-value entries move toward the highest level in LSM-trees as time passes, entries in the lower level tend to have higher sequence numbers. Likewise, the minimum sequence number of each lower-level SSTables also shows a similar tendency, as depicted in Figure 5. In LSM-tree-based MVCC systems, a transaction with an old snapshot would have skipped traversing lower-level entries or entire

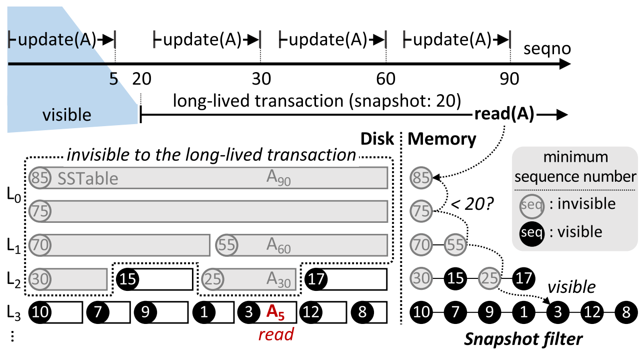


FIGURE 6. Snapshot filters.

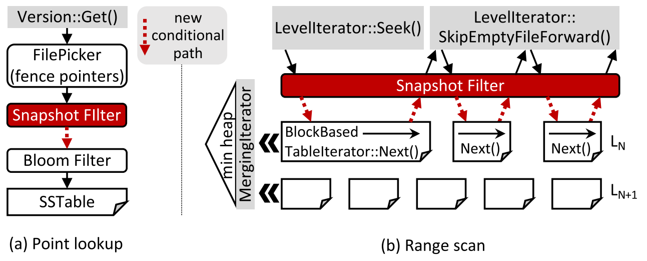


FIGURE 7. Snapshot filter implementation.

SSTables, if the systems had tracked pertinent information. Indeed, a single minimum sequence number can help skip the entire SSTable, which is the essential benefit of snapshot filters.

B. MAIN DESIGN

A snapshot filter is an SSTable-granular in-memory filter. For each SSTable, the snapshot filter keeps track of the minimum sequence number among its entries. When a read transaction finds an SSTable covering the target key, it first compares its snapshot with the corresponding minimum sequence number in the snapshot filter. If the minimum sequence number of the SSTable is higher than the given snapshot, the transaction skips the SSTable since all entries in the SSTable are invisible to the transaction. Figure 6 shows the read path of a long-lived transaction utilizing the snapshot filter. The transaction took a snapshot sequence number of 20 when it started, and the recent version of entry A at that time was 5. As systems process many updates while the read transaction lives a long time, the then-recent version entry A_5 has to go down to L_3 due to compaction. Once the transaction finds the visible version of entry A, it can skip lower-level SSTables filled with newer versions than the snapshot using our snapshot filter.

Snapshot filters show a better effect for lookups on hotkey entries that are frequently updated. Multiple versions of hotkey entries spread across most levels, and in this case, the bloom filters barely filter out unnecessary reads on SSTables during the version search of the key since the filter's domain is based on the presence of the key not visibility. On the contrary, the snapshot filters are effective in this case, preventing the transaction from unnecessarily accessing invisible SSTables, even if they contain the target key entry. Snapshot filters can also benefit range queries that bloom filters cannot, although the impact is less substantial than a point lookup. A point lookup probes one SSTable per level equally (except for L_0) until it finds a visible entry. Thus, the snapshot filters can reduce disk reads directly proportional to the number of skippable lower levels. However, a range query scans more SSTables at a higher level since a higher-level SSTable covers a narrower key range. Filtering out lower-level SSTables does not affect the range query latency as much as the point lookup.

Our lightweight filter uses one metadata, the minimum sequence number, to expedite read operations on HTAP workloads. The opportunity for further advancements in the filter using more complicated techniques is still open, but we forbear the effort for two reasons. First, SSTables distributed on the higher levels tend to show lower minimum sequence numbers than that of lower levels. With each SSTable containing sequence numbers of a somewhat bounded range, the utilization of a single minimum sequence number does its job of filtering out a meaningful load as implied in Figure 5. Second, using a complex strategy may enhance the performance, however, in a trade for extra memory or CPU consumption. The snapshot filters take near-zero resource consumption, using resources already held in memory, which is merit for practicality.

The snapshot filters are not subject to crash recovery. After a system reboots, all the preexisting entries are either visible to or older than new snapshots, so the snapshot filters used before the system shutdown becomes obsolete. Besides, the snapshot filters only occupy 8 bytes of a sequence number for each SSTable, which is a negligible footprint for a database management system. Therefore, it is adequate to manage snapshot filters in memory, assigning a role of an additional filtering layer together with fence pointers and bloom filters.

C. IMPLEMENTATION ON RocksDB

In RocksDB, both point lookup and range scan involve the action of traversing multiple SSTables to search for the desired entries, leaving the critical difference between the two to the direction of their access path: 1) a point lookup invokes a vertical traversal, searching each level sequentially using the target key, whereas 2) a range scan performs a horizontal search with iterators, by collecting the entries while advancing the cursors on each level to the end. The implementations of snapshot filters on both methods share the same logic for the filtering process: by comparing the minimum sequence number of an SSTable to the transaction snapshot. Essential to notice is where to place snapshot filters for each corresponding access path. A point lookup undergoes two filtering layers for each level: fence pointers and a bloom filters. Since fence pointers narrow down search candidates and bloom filters require a higher computational cost (i.e., hashing) than our lightweight filters, we place the snapshot filters between the two as depicted in Figure 7a.

TABLE 1. MyRocks default system configurations.

isolation level	REPEATABLE-READ
memtable size	64 MiB
SSTable file size	64 MiB
block cache size	512 MiB
fanout	10
compression	No
compaction priority	kMinOverlappingRatio

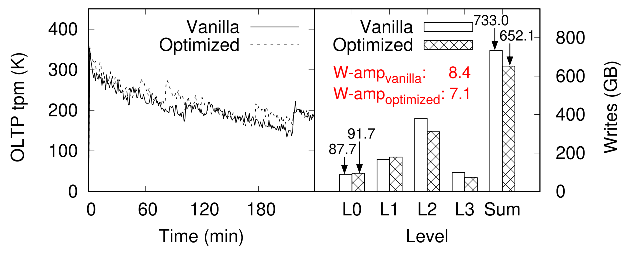


FIGURE 8. OLTP throughput and disk writes of CH-BenCHmark.

Figure 7b shows the process of a range scan with the snapshot filters, making use of iterators for each level (i.e., `LevelIterator`). The iterators are combined and sorted using a min-heap (i.e., `MergingIterator`). The desired key range is extracted by advancing the cursors on each level to the end, traversing multiple SSTables horizontally. The movement within a level is divided into two steps: seek and advance. First, `Seek()` is invoked to put the initial cursor on the first overlapping SSTable, and then `SkipEmptyFileForward()` is used to move the cursor to the next SSTables from thereon. Both invocations induce jumps over SSTables, implying a crucial implementation point for our snapshot filters.

VI. EXPERIMENTAL EVALUATION

We apply our optimizations to RocksDB and evaluate them with various workloads. We first validate the performance gain under HTAP workloads, and then we further investigate them with microbenchmarks of the varying system and workload configurations. We execute all the experiments in the following environment: 96 CPU cores with four Intel Xeon E7-8890, 2 TiB of memory, and Intel NVMe SSD DC P3608.

A. *MyRocks UNDER HTAP WORKLOADS*

To execute complex SQL queries over RocksDB, we use MyRocks-5.6 for the evaluation. We also use the open-source tool of Citusdata [31] implementing CH-BenCHmark [32], a popular HTAP benchmark, with a slight modification for the compatibility with MyRocks. All system configuration parameters of RocksDB remain as defaults (detailed in 1), except for the use of bloom filters. We configure the bloom filters with 10 bits per key in the additional experiment to measure the accurate performance gain beyond the existing optimizations. We dedicate 32 workers for TPC-C loads and six for TPC-H loads and run the benchmark for 4 hours plus extra waiting time for the very last OLAP query to end.

1) THE EFFECT OF ALIGNED COMPACTION UNDER CH-BenCHmark

Figure 8 shows OLTP throughput and the amount of disk writes for each level of the LSM-trees. Vanilla MyRocks and the optimized version show a similar amount in the L_0 disk writes since all modifications should first pass through the lowest level. The amount of disk writes on L_1 is similar because SSTables in L_0 are tiered, which means that most files in L_0 and L_1 are usually compacted together, covering almost the entire key range, so the compaction rarely forms false overlaps in the first place. However, from L_1-L_2 compaction, aligned compaction takes effect, reducing the total disk writes. In terms of the overall write amplification (i.e., total disk writes / L_0 disk writes), MyRocks with aligned compaction shows 14.9% less than vanilla MyRocks.

2) THE EFFECT OF SNAPSHOT FILTERS UNDER CH-BenCHmark

Snapshot filters help reduce read amplification of the queries with the following properties:

- The lifetime of the query is long enough for OLTP transactions to pile up a considerable number of recent versions invisible to the OLAP query.
- Internally queries perform numerous point lookups.

Out of 22 OLAP queries of CH-BenCHmark, we pick out the notable six queries (Q9, Q14, Q15, Q17, Q19, Q21) that show these properties. Each query is assigned to the six dedicated OLAP worker threads, respectively, and each worker thread repeatedly runs the designated query until the benchmark terminates. As shown in Figure 9a, snapshot filters effectively reduce the read amplification, leading to improving the query latencies, of which the degree of improvement depends on the query type. As the query runs long, the latency gap between the vanilla and the optimized system widens since OLTP workers keep increasing the database volume, creating more skippable SSTables during the query lifetime. Note that bloom filters are effective for several long-running queries, but we observe that MyRocks with snapshot filters outperforms vanilla with a wide margin, especially for certain queries (Q15 and Q21) where bloom filters barely impact query latency. In-depth analysis through profiling false-positive rates (i.e., FPR) of bloom filters confirms that queries with high FPR benefit much from snapshot filters, as shown in Figure 9b. The results confirm that long-running queries internally performing numerous point lookup operations are susceptible to HTAP, where our snapshot filters can be beneficial.

B. *MICROBENCHMARKS FOR ALIGNED COMPACTION*

In this section, we evaluate the impact of aligned compaction by varying the configuration. First, we adjust the fanout of LSM-trees to observe the reduction in write amplification. Next, we show the adaptability of aligned compaction depending on different workload distributions.

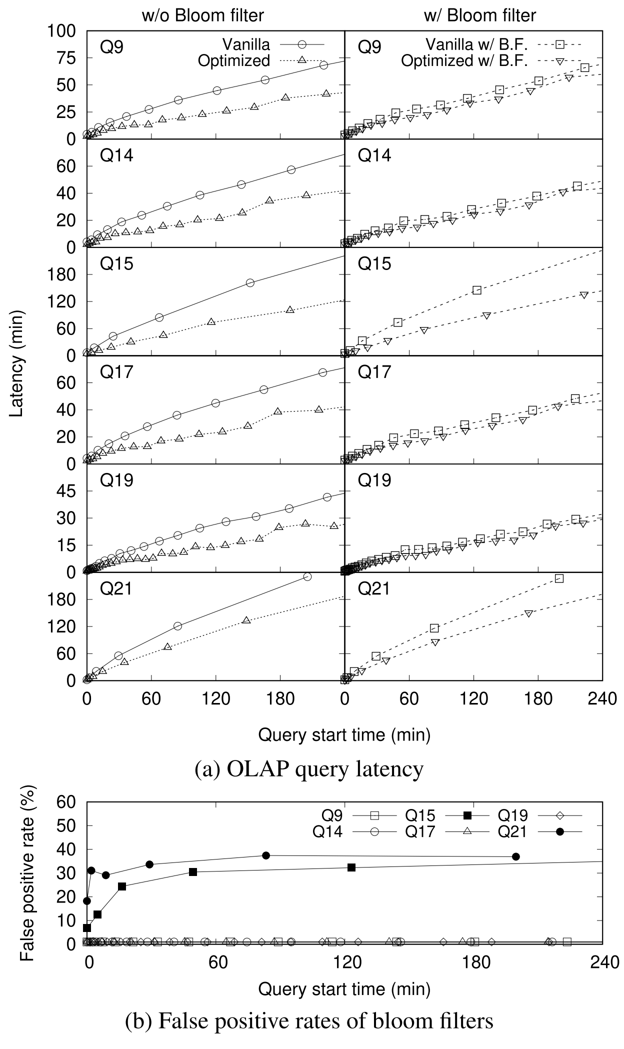


FIGURE 9. OLAP query latency of CH-BenCHmark.

We use RocksDB-6.17 with default configurations for the experiments, except we disable data compression to grow data volume faster. Microbenchmarks tested are from the built-in benchmarks (i.e., db_bench) of RocksDB. We use 64-bit integer keys with the range of 200M, together with 1024 bytes of value each, for write-only workloads putting 200M of random key-values.

1) ALIGNED COMPACTION WITH VARYING FANOUT

To see the effect of aligned compaction on LSM-trees with different fanout, we set the fanout to 2, 6, and 10, respectively. Figure 10a shows the amount of disk writes for each disk-component level. Since the disk writes amount on L_0 represents the volume of user data written to the database, the results of L_0 for all configurations are almost the same. Also, the results of L_1 are similar for the vanilla and the optimized system because our optimization does not benefit L_0-L_1 compaction. However, from L_2 to the highest level, the optimized RocksDB shows fewer disk writes, reducing the total disk writes.

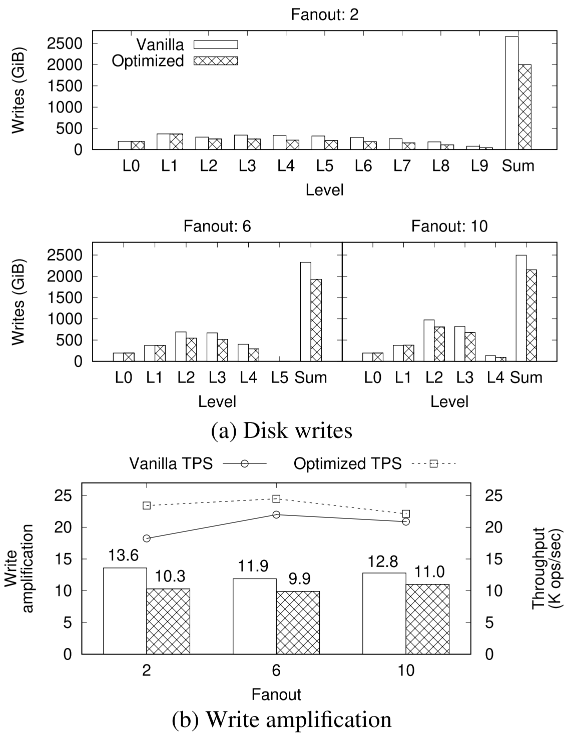


FIGURE 10. Aligned compaction with different fanout.

Figure 10b represents the corresponding write amplification for the evaluations. As shown in the figure, aligned compaction benefits more to the system with lower fanout since the false overlap range tends to be more significant; the write amplification is reduced 24.8% in fanout of 2 and 13.8% in fanout of 10. Reduced write amplification possibly helps throughput increase when compaction threads cannot keep up with the write speed. The result shows 28.4% of throughput improvement in fanout of 2 and 6.0% in fanout of 10.

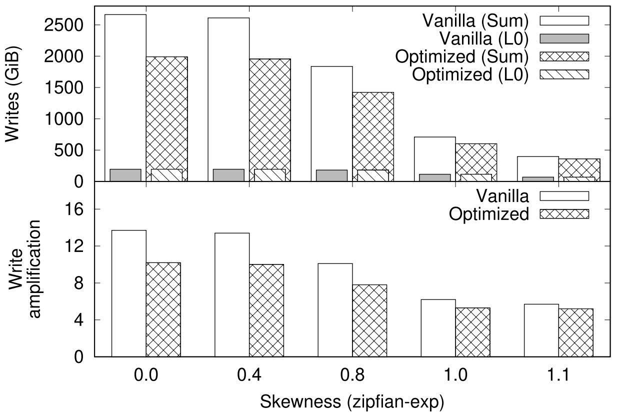


FIGURE 11. Aligned compaction with different workload distribution.

2) ALIGNED COMPACTION UNDER SKEWED WORKLOADS

Aligned compaction dynamically partitions output SSTables to work regardless of the workload distribution. To validate its adaptability, we varied the Zipfian exponent of the key generator from 0.0 (uniform) to 1.1 (highly skewed), and we set the fanout as 2 to expand the LSM-trees with more levels. Figure 11 shows the number of disk writes and write amplification. The optimized RocksDB successfully reduces IO under all distributions, although the improvement diminishes when workloads become skewed. Nevertheless, it does not necessarily mean that the effectiveness of aligned compaction decreases. Under the highly skewed workloads (Zipfian-exp 1.1), the system generates about 17 million different keys during 200 million put operations and incurring disk writes up to L_4, while 126 million different keys are created under the uniform workloads incurring disk writes up to L_9. Considering that the false overlaps appear throughout all levels higher than L_1, there are fewer opportunities for improvement under the skewed workloads in the first place.

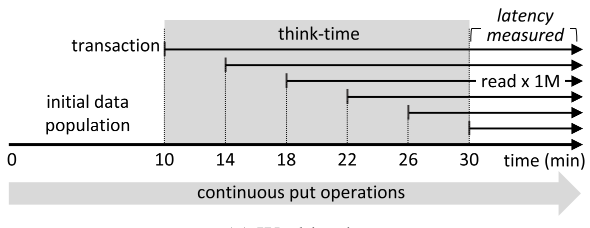
C. MICROBENCHMARKS FOR SNAPSHOT FILTERS

This section evaluates snapshot filters with microbenchmarks running synthetic workloads, using the same configuration as the previous section. We assign a dedicated writer to put new key-value entries continuously, which each key lies in the range of 0 to 100 million. Each reader in every workload spends various think-time to mimic analytic queries, as specified in each figure. We first demonstrate the effect of snapshot filters on point lookups with varying think-time and workload distributions, and then we show the range scan performance by adjusting scan ranges. To see the behavior of the snapshot filters on a larger-than-memory database, for each experiment, we measured results of the low-spec environment where the available memory capacity is limited to 4 GiB by Linux cgroup.

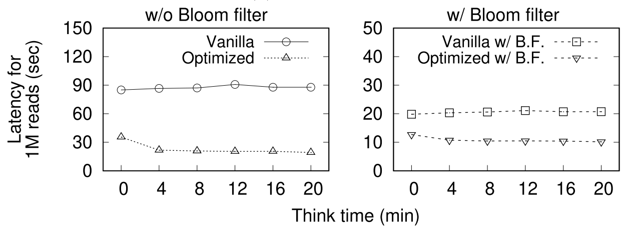
1) POINT LOOKUP WITH VARYING LIFETIME

To see the correlation between transaction lifetime and the effectiveness of snapshot filters, we adjust think-time between the actions of taking a snapshot and reading one million key entries, as illustrated in Figure 12a. We randomly select keys for updates and point lookups with a uniform distribution. As shown in Figure 12b, the optimized RocksDB shows 58.2% improvement in query latency compared to the vanilla system when the think-time is 0. The performance gain reaches up to 78.1%; even with no think-time, the latency is improved notably since many skippable low-level SSTables, especially in L_0, are piled up during one million point lookups.

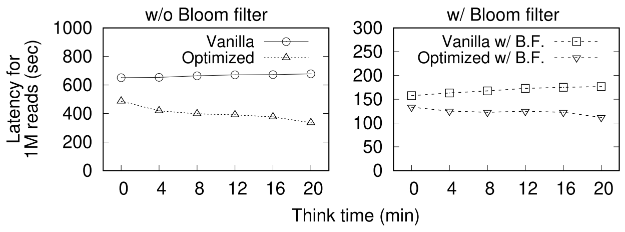
We also measure the latency by enabling the bloom filters in the vanilla and optimized RocksDB. Although the bloom filters significantly reduce the latencies of the vanilla system, the snapshot filters still exhibit meaningful enhancement, from 35.9% to 51.0% latency reduction. The performance gain increases as the transaction lifetime becomes longer, as clearly demonstrated in Figure 12c. When a transaction has a relatively recent snapshot, the snapshot filters only allow skipping lower-level SSTables that are likely to be cached in the page cache. The transaction gradually skips higher-level SSTables that will likely be in storage as the transaction



(a) Workloads



(b) Latency of 1M point lookups



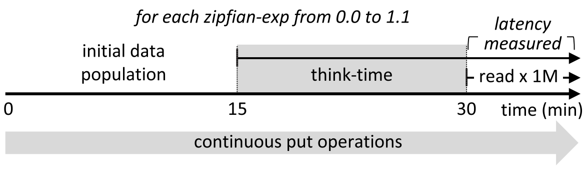
(c) Latency of 1M point lookups (cgroup memory limit 4 GiB)

FIGURE 12. Point lookup latency with different transaction lifetime.

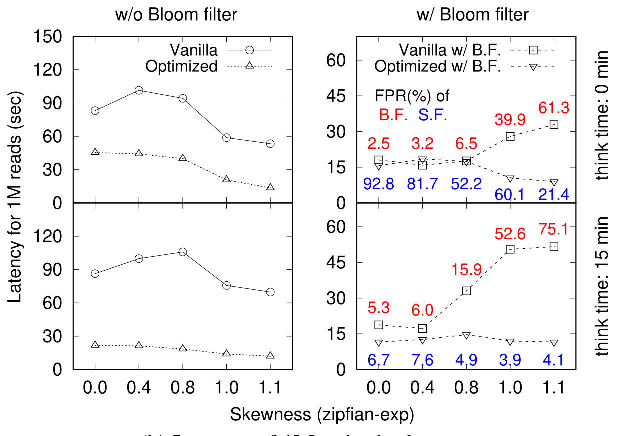
runs long. It meaningfully reduces expensive disk reads and impacts performance more than reducing page access in the page cache. The next section discusses the combined effect of the bloom filters and the snapshot filters.

2) POINT LOOKUP WITH VARYING WORKLOAD DISTRIBUTION

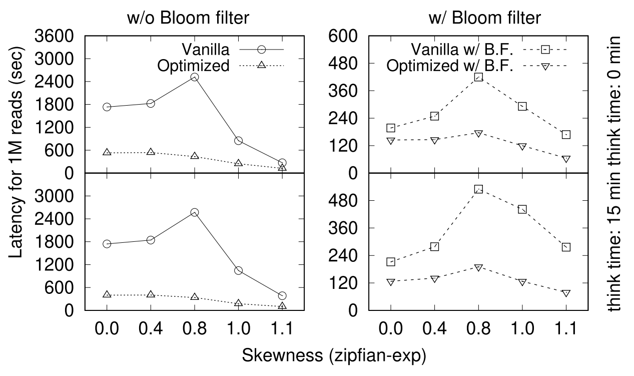
Bloom filters are effective when the search key is not in the target table. However, when a transaction finds a frequently updated key entry that SSTables of multiple levels may include with high probability, the bloom filters are hardly effective in reducing read IO. Figure 13b shows the effectiveness of snapshot filters and bloom filters with different workload distributions from uniform (Zipfian-exp: 0) to highly skewed (Zipfian-exp: 1.1). Snapshot filters enhance the read transactions' performance throughout all workload distributions, with 45.5% latency improvement under the uniform workload and 74.6% under the highly skewed workload. The performance gain increases further with longer transactions that spend 15 minutes of extra think-time since a transaction with an older snapshot has more chances to skip SSTables. Bloom filters also improve the latency of the vanilla system, but the effectiveness decreases as the workload becomes skewed, especially for long-running transactions searching keys across more levels. Combined optimizations show the complementarity between bloom and snapshot filters; the latency remains stable regardless of workload distributions and transaction lifetime.



(a) Workloads



(b) Latency of 1M point lookups



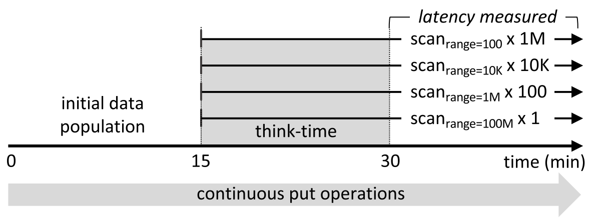
(c) Latency of 1M point lookups (cgroup memory limit 4 GiB)

FIGURE 13. Point lookup latency.

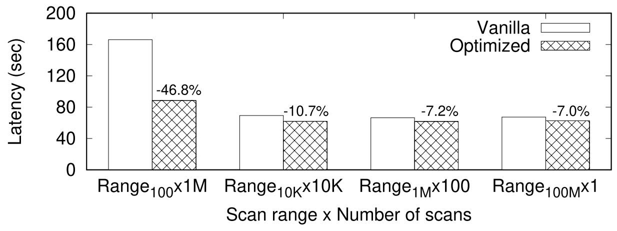
Noticeable is that the false-positive rates of bloom filters and snapshot filters directly correlate to latency behaviors (see the colored FPR numbers on the right-hand side of Figure 13b). Regardless of transaction lifetime, systems with bloom filters only suffer IO amplification as workloads become skewed. On the other hand, systems with both filters substantially reduce read IO, especially under highly skewed loads with long think-time, where snapshot filters would fulfill the job effectively.[1] Although measured in controlled environments, the results confirm that our snapshot filter technique can assist bloom filters in reducing read amplification further under HTAP loads.

Figure 13c shows the results of the same experiments over the memory-constrained environment. Bloom filters successfully reduced query latencies eliminating numerous physical disk reads, especially when the workload is not highly

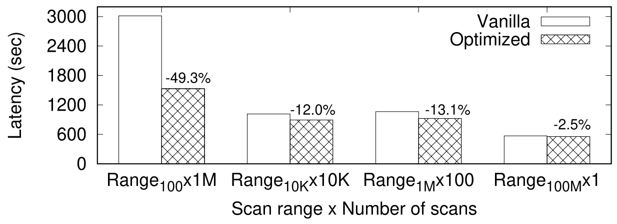
[1]Relatively high FPR numbers observed in systems with snapshot filters and no think-time exhibit high variance depending on query start time, so it needs further investigation.



(a) Workloads



(b) Latency of range scan



(c) Latency of range scan (cgroup memory limit 4 GiB)

FIGURE 14. Range scan latency with different scan range.

skewed; with highly skewed workloads, small numbers of frequently accessed SSTable blocks containing hotkey entries are likely to be cached in the page cache in the first place. The same reason applies to the system without bloom filters, where we can see more dramatic changes in the latency behavior. Nonetheless, the snapshot filters assisting bloom filters still achieve further latency enhancement. It reduces read IO by 26.6% under the uniform workloads and 61.1% under the highly skewed workloads, and the enhancement increases up to 39.9%, 71.4%, respectively, when the transactions take extra think-time.

3) RANGE SCAN WITH DIFFERENT RANGES

In this experiment, we show the effect of the snapshot filters under range scans. As detailed in Figure 14a, we adjust the scan range to 100, 10K, 1M, and 100M, respectively, and each transaction repeats the scan multiple times with the given range after 15 minutes of think-time. Figure 14b and Figure 14c show that the snapshot filters enhance the latency by about 46.8% to 49.3% when the scan range is narrow, but the improvement rapidly drops as the scan range becomes wider. The reason is that when a scan range is narrow, the number of skippable lower-level SSTable blocks accounts for a large proportion of the total blocks that the range scan iterates; in contrast, the broader the range, the greater the ratio of the iterated blocks in the higher levels during the range scan. Overall, the optimized system still outperforms vanilla RocksDB throughout all scan ranges with negligible costs.

VII. DISCUSSION

As practical remedies, aligned compaction and snapshot filters benefit widely used key-value systems; **our prototype implementation in RocksDB only requires less than 100 LOC for snapshot filters and less than 400 LOC for aligned compaction.**[2] Since our focus is on practicality without demanding hidden costs, we have excluded more sophisticated designs for two reasons. First, such designs may require substantial changes from the LSM-tree architecture, which means that deploying the solution may lose practicality. Second, sophisticated designs usually demand complex data structures and more resource requirements, which may cause extra resource contention with storage systems. We, therefore, leave room for improvement and further investigation. In this work, we could not explore design questions, such as how much gain one can attain by deploying sophisticated designs that use more resources and constraints? Would it be beneficial to use finer-grained tracking mechanisms requiring more memory or storage? We also have to think about the question: would approaching HTAP issues by taming write-optimized LSM-tree architecture be worth further investigation? or is it time to rethink the underlying architecture of LSM-trees in the face of upcoming, complex database workloads? We would leave these more fundamental questions to our community.

VIII. CONCLUSION

As we face a growing demand from the data analytics market for HTAP, database systems confront a new challenge of dealing with mixed types of workloads. LSM-trees pervading many key-value storage systems are deemed a suitable alternative for a foundation of HTAP systems due to their superior write-optimized features and acceptable read performance. However, the workload characteristics of HTAP, i.e., transactional reads running together with intensive writes, aggravate IO amplification, which would impact the overall system performance badly. This paper proposed two practical optimizations, aligned compaction and snapshot filters, to alleviate IO amplification under HTAP workloads. The two techniques are lightweight, orthogonal to existing techniques without needing architectural changes. We demonstrated the effectiveness of the proposed techniques by applying them to RocksDB and showed that our proposals generally improve IO amplification under a variety of workloads, proving that the optimizations make LSM-trees more HTAP-friendly.

REFERENCES

[1] (2021). Google Open Source. *Leveldb*. [Online]. Available: https://github.com/google/leveldb

[2] (2021). Facebook Open Source. *Rocksdb: A Persistent Key-Value Store for Fast Storage Environments*. [Online]. Available: http://rocksdb.org/

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[2]Despite being premature implementations, much of what we needed for realizing our techniques is already available in RocksDB, such as minimum sequence number and SSTable metadata.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[5] (2021). Apache Software Foundation. *Apache Hbase*. [Online]. Available: https://hbase.apache.org/

[6] Y. Matsunobu, S. Dong, and H. Lee, "MyRocks: LSM-tree database storage engine serving Facebook's social graph," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, Aug. 2020.

[7] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han, "Hybrid garbage collection for multi-version concurrency control in SAP HANA," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2016, pp. 1307–1318.

[8] (2021). Oracle. *Heatwave User Guide*. [Online]. Available: https://downloads.mysql.com/docs/heatwave-en.pdf

[9] (2021). *SingleStore*. [Online]. Available: https://www.singlestore.com/

[10] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, Jun. 2013.

[11] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 195–206.

[12] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell+: Snapshot isolation without snapshots," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, Nov. 2020, pp. 425–441.

[13] J. Kim, K. Kim, H. Cho, J. Yu, S. Kang, and H. Jung, "Rethink the scan in MVCC databases," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2021, pp. 938–950.

[14] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki, "The case for heterogeneous HTAP," in *Proc. 8th Biennial Conf. Innov. Data Syst. Res.*, 2017, pp. 1–11.

[15] H. Saxena, L. Golab, S. Idreos, and I. F. Ilyas, "Real-time LSM-trees for HTAP workloads," 2021, *arXiv:2101.06801*.

[16] F. Özcan, Y. Tian, and P. Tözün, "Hybrid transactional/analytical processing: A survey," in *Proc. ACM Int. Conf. Manage. Data*, New York, NY, USA, 2017, pp. 1771–1775.

[17] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. D. Callaghan, "Designing access methods: The rum conjecture," in *Proc. EDBT*, 2016, pp. 1–6.

[18] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. ACM Int. Conf. Manage. Data*, New York, NY, USA, May 2017, pp. 79–94.

[19] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, May 2018, pp. 505–520.

[20] N. Dayan and S. Idreos, "The log-structured merge-bush & the wacky continuum," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2019, pp. 449–466.

[21] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, Jan. 2020.

[22] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "SlimDB: A space-efficient key-value storage engine for semi-sorted data," *Proc. VLDB Endowment*, vol. 10, no. 13, pp. 2037–2048, Sep. 2017.

[23] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proc. 26th Symp. Operating Syst. Princ.*, New York, NY, USA, Oct. 2017, pp. 497–514.

[24] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "SuRF: Practical range query filtering with fast succinct tries," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, May 2018, pp. 323–336.

[25] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic Bloom filter with hotness awareness for boosting read performance in large key-value stores," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 739–752.

[26] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos, "Rosetta: A robust space-time optimized range filter for key-value stores," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2020, pp. 2071–2086.

[27] N. Dayan and M. Twitto, "Chucky: A succinct cuckoo filter for LSM-tree," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2021, pp. 365–378.

[28] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, Feb. 2016, pp. 149–166.
[29] J. Böttcher, V. Leis, T. Neumann, and A. Kemper, "Scalable garbage collection for in-memory MVCC systems," *Proc. VLDB Endowment*, vol. 13, no. 2, pp. 128–141, Oct. 2019.
[30] J. Kim, H. Cho, K. Kim, J. Yu, S. Kang, and H. Jung, "Long-lived transactions made less harmful," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*. New York, NY, USA: ACM, 2020, pp. 495–510.
[31] (2021). Citus Data. *Tools for Running CH-benCHmark With HammerDB*. [Online]. Available: https://github.com/citusdata/ch-benchmark
[32] R. Cole, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, F. Waas, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, and T. Neumann, "The mixed workload CH-benchmark," in *Proc. 4th Int. Workshop Test. Database Syst. (DBTest)*, New York, NY, USA, 2011, pp. 1–6.

JONGBIN KIM received the B.S. degree in computer science and engineering from Hanyang University, Seoul, South Korea, in 2017, where he is currently pursuing the Ph.D. degree in computer science. His research interests include the design and implementation of contemporary database systems that can survive upcoming multicore hardware, especially focusing on high-performance transaction processing and efficient multi-version management systems.

JAECHAN AHN received the B.S. degree in computer science and engineering from Hanyang University, South Korea, where he is currently pursuing the M.S. degree with the Department of Computer Science and Engineering. His research interest includes database management systems, specifically focused on MVCC databases for hybrid transactional/analytical processing.

KITAEK LEE received the B.S. degree in computer science and engineering from Hanyang University, Seoul, South Korea, in 2020, where he is currently pursuing the M.S. degree in computer science. His research interest includes high-performance transaction processing systems.

MINSOO RYU received the Ph.D. degree from the School of Electrical Engineering and Computer Engineering, Seoul National University, in 2002. He is currently a Professor at the Department of Computer Science and Engineering, Hanyang University, South Korea. His research interest includes real-time embedded systems. His specific research interests include real-time system design and analysis, real-time operating systems and middleware, and diverse software engineering issues for multicore and/or manycore embedded computing systems.

HYUNGSOO JUNG received the B.S. degree in mechanical engineering from Korea University, Seoul, in 2002, and the M.S. and Ph.D. degrees in computer science from Seoul National University, South Korea, in 2004 and 2009, respectively. From 2010 to 2012, he was a Postdoctoral Research Associate with The University of Sydney, Sydney, Australia. From April to September 2012, he was a Researcher at NICTA. From October 2012 to August 2015, he worked at Amazon Web Services as a Software Development Engineer (Senior). In September 2015, he joined Hanyang University, South Korea, as a Faculty Member, where he is currently an Associate Professor at the Department of Computer Science. His research interests include the areas of distributed systems, database systems, and transaction processing.