Range Queries and LSM-Trees: From Foes to Friends

Shubham Kaushik
Brandeis University
kaushiks@brandeis.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

Subhadeep Sarkar
Brandeis University
subhadeep@brandeis.edu

ABSTRACT

Log-structured merge (LSM) trees are widely used as the storage layer in modern ingestion-optimized data stores. However, their high ingestion throughput comes at the cost of sub-optimal query performance, specifically for range queries (RQs). This is because LSM-trees arrange the data on storage as a hierarchical collection of *sorted runs*, which implies that every RQ must (i) probe all sorted runs to identify the qualifying entries, (ii) scan and merge the entries from all qualifying run, (iii) filter out the logically invalidated entries by updates and deletes on the fly, and (iv) return the most recent version of the qualifying keys in sorted order. This leads to *high read amplification* and *significant redundant work* in terms of superfluous I/Os to storage and wasted CPU cycles, which is exacerbated in presence of updates and deletes. Further, during compactions the same data is read and written multiple times, leading to high write amplification and an even higher read amplification.

In this paper, we introduce RangeReduce, an RQ-optimized LSM engine that *uses RQs as a hint to compact data that is already read into memory* as part of the query, and thereby, improves the overall performance of the storage engine. The key intuition is to take advantage of the I/Os and CPU cycles spent on reading and merging data from slow storage during RQs and write the RQ-qualifying data back as part of a single sorted run. Such RQ-driven compactions enable RangeReduce to (i) read less data from fewer sorted runs for subsequent RQs, (ii) reduce overall data movement (reads or writes) due to RQs and compactions, and (iii) improve space amplification, while (iv) significantly reducing compaction debt. Experiments show that RangeReduce offers up to 90% lower compaction debt, 12% less data movement, and 20% lower space amplification while improving average RQ latency by up to 18%.

1 INTRODUCTION

LSM-trees are Everywhere. Log-structured merge (LSM) tree is a disk-based data structure that is highly optimized for data ingestion, and thus, is widely used in the storage layer of modern key-value stores [28, 34, 38]. LSM-based storage engines offer high throughput for writes by (i) batching the incoming data in memory and then (ii) writing them opportunistically to slow secondary storage as a hierarchical collection of *immutable sorted runs* [47]. Immutability ensures updates and deletes in LSM-engines are always applied *out of place*, i.e., instead of searching for the target entries and updating or deleting them in place, a new version of the entry is ingested into the tree that logically invalidates the older target entry [50]. This out-of-place paradigm significantly improves latency per operation (at the cost of space and write amplification) [18, 51]. LSM-trees are, thus, fuel several commercial key-value stores and relational systems, including LevelDB [22], BigTable [8], RocksDB [20], MyRocks [19], WiredTiger [58], Cassandra [1], HBase [2], CockroachDB [9], ScyllaDB [53], SplinterDB [10], FoundationDB [3], Rockset [46], DynamoDB [16], and Speedb [55].

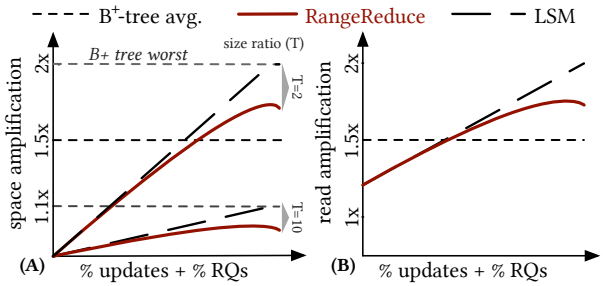

Fig. 1: (A) The average-case space amplification is 1.5× for B$^+$-trees, and it varies between 1.1× and 2× for LSM-trees. (B) *In-place* updates in B$^+$-trees help maintain a constant read amplification; whereas, out of place updates in LSM-trees lead to a higher read amplification as updates in a workload increase. RangeReduce significantly improves both space amplification and read amplification by compacting data on the cue of range queries.

Merging in LSM-Trees. LSM-trees store data as a hierarchical collection of sorted runs across multiple levels on the secondary storage (hard disk or solid state drives), where each level is exponentially larger than the previous one. A newer run always contains the latest version of an entry [49, 63]. This invariant is critical to ensure the correctness of a point query result where runs are probed from the youngest to the oldest [11, 40]. On disk, a sorted run is physically stored as multiple non-overlapping files [48, 51].

Range Query-triggered Merges: As the sorted runs in an LSM-tree may overlap in the key domain, entries qualifying for a range query (RQ) typically span multiple – in the general case, *all* – sorted runs. This means that realizing a RQ requires (i) identifying all qualifying sorted runs, (ii) reading the qualifying entries from the selected runs, and (iii) merging them in memory to add the latest version of every entry to the result set.

Compaction-triggered Merges: Sorted runs are also periodically merged to reduce space and write amplification, with a process termed *compaction* [47, 51]. During compaction, a subset of the files in a sorted run are merged with the overlapping files in the subsequent level to create a longer sorted run. The process involves (i) reading all files to be compacted, (ii) merging their contents in memory while removing all logically invalidated entries, and (iii) writing back the valid entries as new files [48]. Compacting multiple sorted runs periodically to create fewer, longer sorted runs is critical, as it bounds space amplification and improves query performance [4, 18]. Next, we discuss the core challenges of the out-of-place paradigm in conjunction with RQs.

Challenge 1: High Read and Space Amplification. Realizing updates and deletes *logically* and *out of place* leads to considerable space amplification in LSM-based storage engines. This, in turn, leads to higher read amplification and suboptimal RQ performance. Thus, as the proportion of updates and deletes increases in a workload, so does the space amplification, as shown in Fig. 1(A). For instance, a leveled LSM-tree with a size ratio $T = 4$ may have up to 25% logically invalidated entries in the tree; for a tree with $T = 2$, this can be up to 50%. This means for update- (and delete-) intensive workloads, every RQ needs to parse a significant amount of logically invalidated entries (and delete-markers), leading to a considerable **increase in the I/O cost of RQs**, as shown in Fig. 1B. To quote an engineer at Google, "*In presence of deletes, range queries often have to process and skip millions of tombstones and logically deleted data.*" Additionally, the hierarchical data layout in LSM-trees requires RQs to iterate over all qualifying sorted runs and perform a k-way merge in memory. For example, a RQ in a tiered LSM-tree with 5 levels and 10 tiers per level will typically have to read $5 \cdot 10 = 50$ data pages in memory and perform a 50-*way merge* to generate the result. This involves a **high number of random I/Os** for each RQ, as well as **a considerable number of CPU cycles**.

Challenge 2: Repetitive Merging. In addition to the increased read amplification for RQs, LSM-trees lead to a repetition of merging for recurring RQs even if they are identical. Consider a scenario where 20 identical RQs are executed one after the other. The standard read path would initiate a k-way merge for each of those queries, hence repeating work that had just been executed. Overall, the **cost of random I/Os** to fetch the qualifying data blocks from storage and the **cost in terms of CPU cycles** to merge, filter, and construct the result must be **paid separately for every range query** (barring some caching benefits).

Challenge 3: Wasted Work. LSM-trees, by virtue of their design, create a *compaction debt* [7] during ingestion, which entails reading and writing the same entries repeatedly during compactions. On average, in a leveled LSM-tree with L levels and a size ratio of T, every entry is read and written $T \cdot L$ times, contributing to **high read and space amplification** [12, 51]. At the same time, recurring RQs perform similar or identical repetitive merges, oftentimes similar to the ones compactions would do, which further exacerbates the problem of wasted work.

Opportunity: Re-use RQ Merging. The key intuition behind our proposed solution is to *take advantage of the reads and merges performed during RQs and combine this with a compaction.* Such query-driven data reorganization approaches are well-explored in the relational realm [5, 23, 24, 26, 30, 52, 56, 60]; however, they cannot be used out of the box in LSM-like out-of-place systems, as it would lead to extremely high write amplification [42]. SuccinctKV attempts to perform query-driven compactions in LSM-trees, but their high range query throughput comes at the cost of increased write amplification, space amplification, and compaction debt [60]. Our proposed solution, RangeReduce, benefits both (i) future compactions, since it can altogether avoid some compaction in frequently read ranges, and (ii) future range queries, since recently queried ranges are void of stale data and compacted to a single file.

Can RQ Filters Help? RQ filters, such as SuRF [59], Rosetta, [35], and REMIX [61] are auxiliary indexes that help avoid unnecessary disk accesses when processing RQs. During a RQ, the filter quickly identifies which files or data ranges may contain relevant keys using some metadata, allowing the system to bypass the rest and reduce the number of I/Os. RQs filters, thus, improve RQ latency, but still suffer from *all three challenges* discussed. The benefits of RangeReduce are orthogonal to those of RQ filters. In fact, when integrated with systems using some RQ filter, RangeReduce further boosts its performance by further improving the RQ performance and by reducing space amplification and compaction debt.

Our Solution: RangeReduce. We propose RangeReduce, which takes advantage of the merging performed during RQs to trigger *query-driven compaction*, and thus, avoid redundant work during future compactions. In the process, RangeReduce also **eliminates duplicates** (logically invalidated entries) and ensures that a single version of each entry accessed by a RQ remains in the LSM-tree. RangeReduce takes a hit on the initial RQ as triggering a compaction entails writing the data back to secondary storage. However, this (i) improves performance for all (identical or overlapping) future RQs and (ii) significantly reduces compaction debt. Unlike state-of-the-art query-driven compaction algorithms [60], RangeReduce runs a lightweight algorithm to decide whether to compact and reuse merged results. This ensures improved RQ performance never comes at the cost of high write amplification owing to compactions. RangeReduce, thus, **improve I/O cost** for all subsequent overlapping **RQs** as they read fewer duplicates, thereby **reducing read and space amplification**. Future RQs need **fewer CPU cycles** as they fetch and merge fewer data blocks with a higher fraction of valid entries. Overall, our approach improves average read and space amplification, RQ performance, and offers several additional benefits, such as fewer compactions, less data movement, and better utilization of CPU cycles.

Contributions. Our contributions are summarized below.

- We show that state-of-the-art LSM-engines perform RQs suboptimally and do not use resources efficiently, resulting in superfluous reads and redundant merge operations, which slow down RQs and hinder overall system performance.
- We introduce RangeReduce, a RQ-aware LSM-engine that performs compaction on the cue of RQs, and thus, optimizes resource utilization and mean latency for future RQs and compactions.
- Our approach significantly reduces compaction-debt by up to 90%, which decreases overall data movement by 12% in LSM-tree. This indicates that the LSM-tree will need fewer compactions since shallower levels will have free space due to RangeReduce.
- We show that RangeReduce is more space-efficient (reduction up to 12%) and offers better ingestion throughput (up to 6%) compared to existing solutions.
- We implement RangeReduce on RocksDB and demonstrate that it outperforms existing designs for RQ-intensive workloads.

2 BACKGROUND

The LSM Architecture. LSM-tree is an ingestion-optimized, disk-based data structure that organizes data into a hierarchical collection of levels [34, 38]. As we move deeper in the hierarchy, the capacity of the levels increases exponentially, by a constant factor, T, bounding the height of the tree logarithmically [11, 33]. T is

typically referred to as the size ratio of the tree. To facilitate high throughput for writes, LSM-trees first batch the incoming data in memory, and then once the in-memory buffer is saturated, write the entries to slower storage as immutable sorted runs. The newer data is always written to the smallest level and gradually moved to larger levels through compactions [48]. Immutability of the files also means that updates (and deletes) are realized out of place, i.e., instead of updating the target data in place, LSM-trees simply insert the latest version of the data into the tree, which logically invalidates the older target entry [47].

Compactions. Compactions in LSM-trees are performed periodically, typically, once a level reaches its capacity, to ensure competitive point and range query performance [15, 51]. During a compaction, all or part of the data from a Level i is merged with the overlapping data in Level $i+1$. This helps in (i) limiting the number of sorted runs a query needs to access and (ii) reclaiming space by removing logical updates and deletes [13]. As the LSM-tree continues to accumulate data and more levels fill up, compactions become increasingly frequent. When all L levels in the LSM-tree are full, an LSM-tree with partial compaction [36, 51] performs up to L cascading compactions every time the memory buffer is flushed.

Data Layouts and Merging Policies. The classical data layouts supported by commercial LSM-engines are: *leveling* and *tiering*. In a leveled LSM-tree, each level is a single sorted run, and compactions are performed by merging part of a (or an entire) level with overlapping files from the next level [20, 22, 46] . Leveled LSMs offer competitive query performance as it has fewer sorted runs overall in the tree, but has a high write amplification owing to the eager merges [13, 14]. In a tiered LSM-tree, data is organized into multiple *sorted runs* (or *tiers*) within each level, allowing duplicates within a single level but not in the same run [1, 2, 45]. Compactions in a tiered LSM-tree are performed by (i) reading all sorted runs from a level, (ii) performing a multi-way merge, and (iii) writing the merge result as a single sorted run on the next level. A tiered LSM is optimized for writes, but suffers from high query costs [12].

Basic Operations. LSM-based key-value stores support the following key-value operations such as get, put, delete, and RQ.

Put: A *put* simply inserts a new key into an in-memory buffer and immediately responds with success. Due to the out-of-place nature of LSM-trees, the same *put* API is used to update data as well.

Get: A *get* retrieves the most recent value of the requested key, if it exists. The search begins from an in-memory buffer and continues from smaller to larger levels on disk. For a tiered LSM-tree, within a level, it probes the sorted runs from youngest to oldest. The search stops as soon as the target key is located.

Delete: To realize a delete, LSM-trees simply insert a special marker called a *tombstone* that logically invalidates the target entry. During a compaction, a tombstone physically purges any entries with a matching key, and the tombstone itself gets dropped when it reaches the last sorted run in the LSM-tree.

Range Query (RQ): A *RQ* operation retrieves all keys within a specified key range. LSM-engine begins this process by (i) initiating iterators on every level, (ii) performing a seek operation to locate the first key that qualifies for the requested range, (iii) executing a merge operation in memory to produce an ordered set of entries while (iv) simultaneously discarding any logically invalid entries. The scan continues until a key outside the range is found or all relevant keys from a level are processed, with the iterator reaching the end. Once complete, the result is returned, and iterators are reset. This process usually requires $O(L)$ pages of memory for a leveled LSM-tree, as each iterator reads one page per level. The system then performs a k-way merge, and once a particular page is fully read, the iterator fetches the next page from the same level.

Short and Long RQ. A short RQ typically issues at least one I/O per level, and the number of I/Os scales directly with the size of the targeted range [13, 35, 59]. The size of a RQ is determined by $s \cdot N$ where s is the selectivity and N represents the total keys stored across L levels. In a leveled LSM, a short RQ issues $O(L)$ I/Os, while a long RQ performs $O(\frac{N \cdot s}{B})$ I/Os, depending on the page size, where B is the number of entries per page. In a tiered LSM, a short RQ requires $O(T \cdot L)$ I/Os, and a long RQ involves $O(\frac{N \cdot s \cdot T}{B})$ I/Os, based on the page size [13]. We point out that **LSM-engines perform suboptimally in the presence of RQs and require more CPU cycles and memory resources**. They end up performing superfluous reads and merges during RQs and compactions.

3 PROBLEM: EXCESSIVE READS & MERGING

Next, we outline the limitations of state-of-the-art LSM-engines by quantifying the amount of superfluous reads/writes from/to disk and in-memory operations performed during RQs and compactions.

3.1 Limitations of the State of the Art

LSM-based storage engines, by design, are unable to support RQs efficiently. All state-of-the-art LSM-engines perform a significant amount of superfluous work when realizing RQs, and this is further exacerbated in the presence of updates (and deletes). To this end, we point out the three key limitations of modern LSM-based systems.

(A) Reading Logically Invalidated Data during RQs. When realizing RQs, LSM-based engines need to scan and merge the qualifying data from every sorted run in the tree to construct the query result. This leads to reading superfluous data during RQs, especially in the presence of updates and deletes in a workload (P1, Fig. 2A). This is because, like any out-of-place data structure, LSM-trees realize updates (and deletes) logically, i.e., instead of physically updating (or deleting) the target data object, LSM-trees insert a new version of the entry that logically invalidates the target data. This logically invalid data causes high space amplification, and depending on the proportion of updates, deletes, and range-deletes in a workload, which can be significantly higher [18, 49].

When realizing a RQ, a state-of-the-art LSM engine will (i) first initialize iterators for each sorted run (Fig. 2, Step 1), (ii) seek to the first entry that qualifies for the RQ (Step 2), and then (iii) read the qualifying data from each sorted run, (iv) performing a K-way merge in memory (Step 3) while (v) filtering out invalid entries and preparing the query result. In the presence of updates, deletes, and range-deletes, reading and merging logically invalidated data increases the system's read amplification and slows down the RQ.

(B) Wasted Work for Recurring and Overlapping RQs. The superfluous work during RQs also does not translate to any subsequent overlapping RQs. In fact, every time a RQ is recurring, an LSM-based storage engine would perform the same amount of superfluous work for each query. This leads to several superfluous

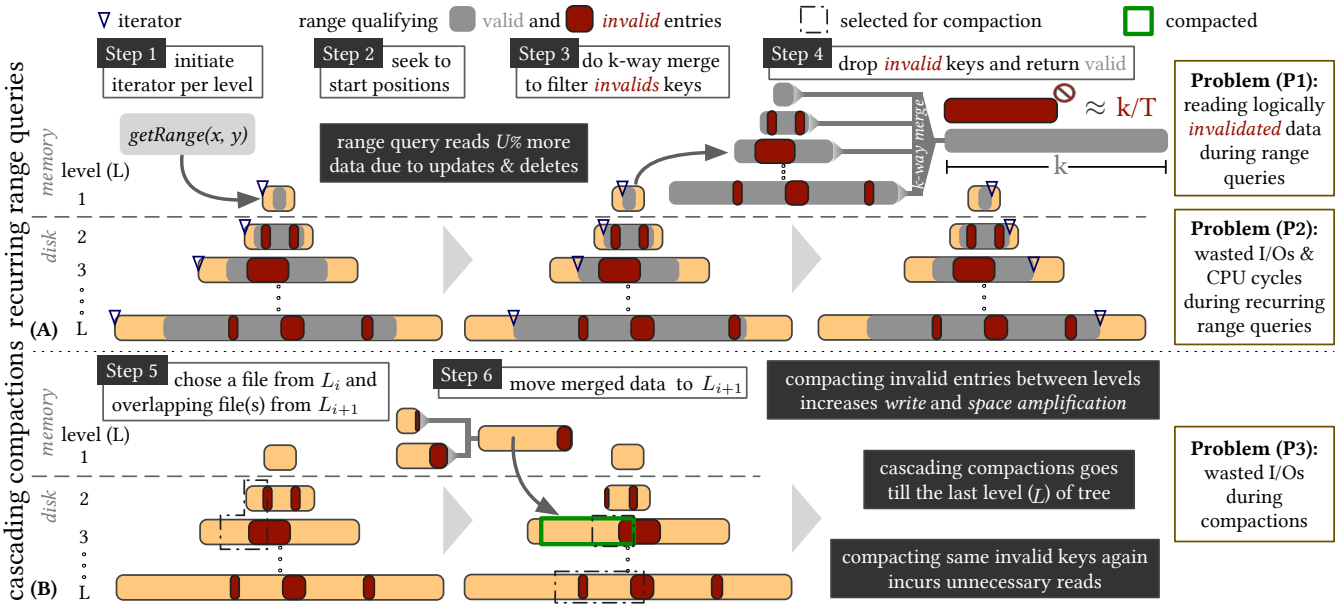

Fig. 2: (A) The execution of the same or overlapping RQs leads to superfluous reads and results in higher resource utilization. (B) The compaction merges data between two adjacent levels, which leads to rewriting *invalid* keys multiple times on the same level, which increases *read*, *write*, and *space* amplification.

I/Os to the storage, as well as wasted in-memory data filtering, P2 in Fig. 2A. RQ-intensive workloads lead to increased query latency and poor CPU utilization, as shown in Fig. 2A.

(C) Wasted I/Os during Compactions. Compactions in LSM-trees reorganize the data layout by periodically merging data from adjacent levels that overlap in the key domain. This means, for a workload with RQs, the data that was read during RQs will once again be read during compactions, P3 in Fig. 2B. In fact, in a leveled LSM-tree with L levels and a size ratio of T, compactions alone would read every entry $T \cdot L$ times on average [11, 51]. Thus, RQs and compactions multiply the number of superfluous reads in LSM-engines, propelling the read amplifications and leading to poor resource utilization and suboptimal overall performance.

3.2 Modeling & Metrics

The suboptimal RQ performance of state-of-the-art LSM-engines adversely impacts the overall performance of the storage engine. We quantify the performance implications in terms of (i) *read amplification and wasted CPU cycles during RQs*, (ii) *write amplification during compactions*, and (iii) the overall *compaction and write amplification debt* of the storage engine.

Model details: We assume an LSM-tree of L levels with a size ratio of T. We populate the tree starting from an empty database and by inserting entries with N^{unq} unique keys with U% updates. The resulting tree has a total of N (valid and logically invalid) entries. The size of the buffer in memory is given by M=P·B·E, where P represents the number of pages that can fit in a buffer, B is the average number of entries that fit in a page, and E is the average size of a key-value entry. The mean selectivity of a RQ is s, i.e., each RQ reads an average of s% of the database. Table 1 lists the parameters used for modeling.

Space Amplification. The space amplification (SA) in an LSM-tree is defined as the ratio of the total number of *valid + logically invalid* entries (N) to the number of *valid* entries (i.e., entries with a unique key) N^{unq} in the tree: $SA = (\frac{N}{N^{unq}})$. For a workload with updates, all entries from Level 1 to Level $L-1$ in an LSM-tree could be updated to the data stored in the last level (Level L). Thus, the worst-case space amplification for a leveled LSM-tree is $O(\frac{1}{T})$, and that for a tiered LSM-tree is $O(T)$ [13]. Note that the space amplification of a leveled and a tiered LSM-tree is inversely and directly proportional to the size ratio (T) of the tree, respectively. For instance, for a leveled LSM-tree $T = 2$, up to 50% of the entries in the tree can be logically invalid; whereas, for a tree with $T = 10$, the space amplification is capped at 11.1%. In presence of deletes, however, the space amplification is significantly exacerbated, as a smaller-sized tombstone can invalidate a larger-sized entry [50].

Read Amplification during RQs. RQs in LSM-trees are realized by merging data from all qualifying sorted runs in a tree. For workloads with updates (and deletes), a RQ performs proportionally as many superfluous I/Os to slower storage as it reads the logically invalid entries (and tombstones) from all qualifying sorted runs. We define read amplification during a RQ as the ratio between the *total bytes read from the LSM-tree* to realize the RQ and the *total bytes returned as part of the query result*. For a RQ of selectivity s, we quantify the read amplification as follows. Assuming the tree has N^{unq} unique inserts and U% updates, we estimate the total number of (*valid + logically invalid*) entries in the tree as $N = N^{unq} + \frac{N^{unq} \cdot U}{T}$ for a leveled LSM-tree, and as $N = N^{unq} + N^{unq} \cdot U \cdot T$ for a tiered LSM-tree. Thus, a RQ with selectivity s reads $N \cdot s$ entries from a

Table 1: Parameters that are used in this paper

Symbol	Description
N	total number of entries in the LSM-tree
N^{unq}	number of unique entries in the LSM-tree
U	percent of updates
S	total number of RQs
s	average selectivity of RQs
T	size ratio of the tree
L	levels on disk holding N + (updates/deletes)
P	size of memory buffer in pages (relative to disk)
B	average number of entries in a page
E	average size of key-value pair in bytes
M	size of memory buffer in bytes
ϕ	fraction of RQs that trigger additional writes
λ	average I/Os a RQ performs while writing
β	reduction factor for update overhead after merging

leveled LSM-tree and performs $\frac{N^{unq}\cdot s}{B}\cdot(1+\frac{U}{T})$ I/Os to read the qualifying data, where B is the average number of key-value entries in a page. For a tiered LSM-tree, the I/O cost of realizing a RQ of selectivity s is given by $\frac{N^{unq}\cdot s}{B}\cdot(1+U\cdot T)$. However, in the ideal where all updates are realized in place, the tree would have zero space amplification, i.e., with exactly N^{unq} entries stored across N^{unq}/B pages. Under such a scenario, realizing a RQ of selectivity s would require only $\frac{N^{unq}\cdot s}{B}$ in both tiered and leveled LSM-trees. Based on this, we quantify the read amplification for a RQ in a leveled (RA_{level}) and a tiered (RA_{tier}) as follows.

$$\mathrm{RA}_{level}=\frac{\frac{N^{unq}\cdot s}{B}\cdot(1+\frac{U}{T})\cdot B}{N^{unq}\cdot s}\cdot S\approx(1+\frac{U}{T})\cdot S \quad (1)$$

$$\mathrm{RA}_{tier}=\frac{\frac{N^{unq}\cdot s}{B}\cdot(1+U\cdot T)\cdot B}{N^{unq}\cdot s}\cdot S\approx(1+U\cdot T)\cdot S \quad (2)$$

CPU Overhead during RQs. Realizing RQs in LSM-engines is a CPU-intensive task. This is because for an LSM-tree with K sorted runs, the data qualifying for a RQ may be scattered across all K runs, in the general case (i.e., if data is uniformly distributed). Thus, constructing the query result entails merging up to K sorted runs in memory. During this K-way merge, any logically invalid data (and tombstones) are filtered out, and as the number of invalid entries in the tree increases, so does the CPU cost for processing the data and constructing the query result. For instance, a tiered LSM-tree with 5 levels and T sorted runs per level would merge up to $5\times10=50$ sorted runs for each RQ. This requires at least 50 buffer pages in memory, and in the presence of updates (and deletes), spends proportionally as many superfluous CPU cycles to filter out the logically invalid data. We quantify the CPU cost of merging as $O(N\cdot s\cdot\log(N\cdot s)\cdot f_{RQ})$ where f_{RQ} is the frequency of RQs. Merging of data from multiple sorted runs makes RQs even slower. Further, for recurring RQs or for RQs that overlap in the key domain, this superfluous work in the CPU is repeated for every query.

Read and Write Amplification due to Compactions. The write-optimized design of LSM-trees thrives on the principle of "***write first, organize later***". LSM-based storage engines achieve this by quickly writing the data to the in-memory buffer and eventually writing it to storage in the form of a hierarchical collection of sorted runs. As more entries are ingested into the tree, the smaller sorted runs are merged into longer runs through *repeated merging*. We classify the superfluous reads and writes involved in this process as *read amplification* and *write amplification*, respectively, due to compactions. For a leveled LSM-tree with L levels, every entry is rewritten $T-1$ times on average in each level, leading to an average-case read and write amplification to be $O(T\cdot L)$. For the write-optimized tiered variant, every entry is written once in each level of the tree, resulting in a read and write amplification of $O(L)$.

Compaction Debt. Another artifact of the "***write first, organize later***" principle of LSM-trees is the compaction debt. We define compaction debt as the *amount of pending effort required to compact all data in the intermediate levels of an LSM-tree with that in the last level*, creating a single sorted collection of data. Compaction debt allows us to quantify the amount of superfluous writes needed to compact all data in an LSM-tree to a single sorted run. For instance, in a leveled LSM-tree with L levels and a size ratio of T, a data page in Level i ($i<L$) is expected to be rewritten T times in each of the $L-i$ subsequent levels. Based on this, we quantify the compaction debt in a leveled LSM-tree as $\sum_{i=1}^{L-1}(P\cdot B\cdot E)\cdot T^i\cdot T\cdot(L-i+1)$, where $P\cdot B\cdot E$ represents the file size (equals to buffer size) on slow storage, $(P\cdot B\cdot E)\cdot T^i$ is the size of Level i, and, since every entry is written $T\times$ on each level, it will be written $T\cdot(L-i+1)$ times to get a single sorted run of the current LSM-tree. For tiered, we quantify it as $\sum_{i=1}^{L-1}(P\cdot B\cdot E)\cdot T^{i-1}\cdot T\cdot(L-i+1)$, where $(P\cdot B\cdot E)\cdot T^{i-1}$ represents the size of tier at Level i, and as we have T tiers in every level, written once, hence, will be written $(L-i+1)$ times for achieving a single sorted run.

Hidden Cloud Costs. In recent years, there has been a significant shift toward cloud-native databases, which include BigTable at Google [6], Cassandra at Apache [37], CockroachDB at Cockroach Labs [32], and RocksDB at Meta [43]. These systems are deployed under consumption-based pricing models, where users are billed for (i) the number of I/Os performed per unit time, (ii) the amount of storage used, and (iii) the number of CPU cycles consumed. The suboptimal performance and resource utilization of state-of-the-art LSM-engines under RQ-heavy workloads, thus, lead to significant cost escalations.

4 RangeReduce

We propose RangeReduce, a RQ-aware LSM engine that *piggybacks on the reads performed by RQs and writes the qualifying data back to storage as part of a single sorted run.* This allows for improved RQ latency and operational throughput, reduced space amplification and overall data movement, and significantly reduced compaction debt. To this end, we first propose an ensemble of RQ-driven compaction strategies, namely, Merge-on-Scan, File-Size-Aware-Merge-on-Scan, and Bounded-Merge, which perform *compactions on the cue of RQs* and, thereby, improve the latency for future RQs and significantly reduce space amplification and compaction debt. Next, we introduce Level-Renaming, which significantly reduces the data movement (reads and writes) performed by compactions by trivially moving files toward deeper levels every time the LSM-tree grows in height. Finally, we present RangeReduce, which combines these ideas, improving the overall performance of an LSM engine in the presence of RQs in a workload.

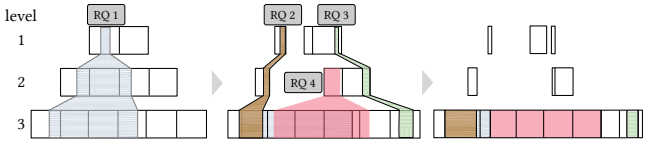

Fig. 3: Merge-on-Scan may rewrite data at the same level, which increases total writes and hence write amplification.

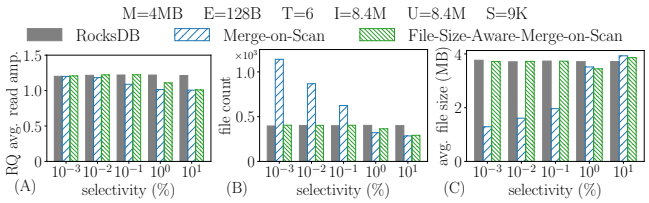

Fig. 4: (A) The average bytes read by RQs is fewer than RocksDB, as Merge-on-Scan removes invalid entries eagerly during RQ. However, Merge-on-Scan creates many (B) smaller files (of small size) (C) for short RQs, which is addressed by File-Size-Aware-Merge-on-Scan.

4.1 Merge-on-Scan

Merge-on-Scan eagerly writes back the data qualifying for a RQ as part of a single sorted run. The core idea is as follows: when realizing a RQ, Merge-on-Scan (i) reads the necessary data from all qualifying sorted runs, (ii) merges the qualifying entries in memory to prepare the query result, and subsequently, (iii) writes back the merged data to storage as a single sorted run. Any logically invalidated data within the range is purged during the merge. The remaining entries are written back (i) to the deepest qualifying level in case of a leveled LSM-tree and (ii) to the oldest sorted run in a tiered LSM-tree. Fig. 3 shows when realizing *RQ* 1, Merge-on-Scan compacts data from the shallower levels, i.e., Levels 1 and 2, with overlapping data in the last qualifying level, i.e., Level 3. The subsequent RQs, *RQ* 2 - *RQ* 3, also compact some data from the shallower levels to the last level, rendering the shallower levels partially full. This allows future ingestion to be absorbed in the "gaps" without necessarily triggering cascading compactions every time the buffer is flushed.

Compacting the data in shallower levels eagerly also means that subsequent RQs that precede inserts and fall within the range of the initial query can be answered simply by reading data from a single target sorted run and do not require a k-way merge in memory. Fig. 4A shows that Merge-on-Scan reduces the overall data read during range queries, reducing read amplification by up to 17%. The eager merges performed by Merge-on-Scan also significantly reduce the compaction debt (up to 78%, as shown in Fig. 5C) and space amplification (up to 25%, as shown in Fig. 5D) of the LSM-tree. This, however, comes at the cost of the eager writes during RQs.

Limitations of Merge-on-Scan. The eager writes after every RQ by Merge-on-Scan lead to the following performance pitfalls.

(1) Creates many small files: During a RQ, moving only the qualifying data to the target level leads to the creation of up to two small fragmented files per sorted run, as shown in Fig. 3. We observe in Fig. 4B, that for a workload with 9000 RQs with a selectivity of 0.001%, Merge-on-Scan has 2× more files than the state of the art, i.e., RocksDB. We also observe in Fig. 4C, as each RQ in Merge-on-Scan compacts some data from shallower levels to a target level, leaving fragmented files in the shallower levels, the average size in the tree drops to only 30% of the file size set. This significantly adds to metadata and file management overhead.

(2) Slows down RQs: Merge-on-Scan writes back the RQ-qualifying data for every RQ regardless of the amount of superfluous writes involved. The superfluous writes correspond to the data that are rewritten to the same level after Merge-on-Scan. Note that *immutability of files* in LSM-trees does not allow for in-place modifications, leading to repeated rewriting of data at the same level. Fig. 5 shows that while the average bytes read during Merge-on-Scan is 16% less than that in RocksDB (Fig. 5A), the average RQ latency is still 1.5× higher (Fig. 5B), owing to the superfluous writes.

(3) Increased write amplification for short RQs: The eager rewriting increases total writes by 3 orders of magnitude in Merge-on-Scan, Fig. 5E, and it can be even higher, up to 5 orders of magnitude more, leading to high write amplification. For range queries with smaller selectivity, i.e., for short RQs, the problems of Merge-on-Scan are further exacerbated. This is because, for short RQs, Merge-on-Scan often writes at most one file in the target level, moving only a few entries from the remaining qualifying levels. This becomes a major performance bottleneck for workloads with many short RQs, as in the shallower levels, a lot of the data is simply rewritten without any forward progress, while increasing write amplification and overall data movement as every RQ comes with a cost of rewriting s% data back to the LSM-tree, Fig. 5E.

4.2 File-Size-Aware-Merge-on-Scan

To avoid generating too many small files and reduce the amount of superfluous writes, specifically, during short RQs, we propose File-Size-Aware-Merge-on-Scan. The key intuition is to increase forward progress by moving as much data as possible during a RQ while reducing the amount of data rewritten to the same level.

Avoiding Superfluous Writes to Same Level. Thus, in File-Size-Aware-Merge-on-Scan, we avoid merging levels when the number of qualifying entries for a RQ is less than half of the file size ($M/2$). This means if the amount of data that qualifies for a RQ from a given level is less than half of a file, File-Size-Aware-Merge-on-Scan will prevent this file from being compacted. Note that to trigger a RQ-driven compaction between Level i and Level $i+1$, File-Size-Aware-Merge-on-Scan needs both levels to pass the compaction criteria. For instance, in a leveled LSM-tree with 5 levels, say Levels 1-4 qualify for a RQ, but only Level 1, Level 3, and Level 4 pass the compaction criterion of File-Size-Aware-Merge-on-Scan. This means Level 1 cannot be compacted with Level 2 as the target level has to satisfy the compaction criterion. But the entries qualifying from Level 3 will be combined with those in Level 4, as both levels satisfy the compaction criterion.

Writing Fewer Small Files. File-Size-Aware-Merge-on-Scan combines the non-range query-qualifying data from the terminal files of the same sorted runs into a single file before writing. This way, instead of creating two small fragmented files in every qualifying sorted run, File-Size-Aware-Merge-on-Scan writes only a single file

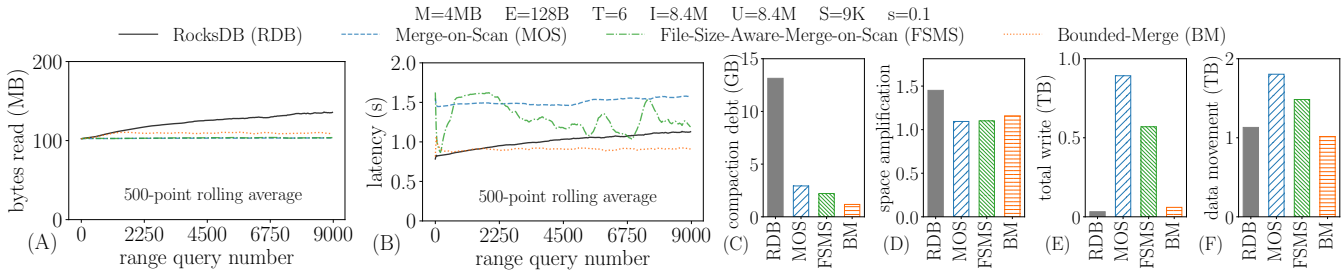

Fig. 5: (A) BM reads fewer bytes than RDB, but more bytes than FSMS during RQs, as it compacts more strategically to reduce total writes and improve RQ performance. (B) Unlike FSMS, BM initially takes a hit on RQs and improves latency using RangeReduceRatio. (C) FSMS and BM offer better compaction debt and (D) reduced space amplification. (E) BM further optimizes total writes and overall data movement (F), i.e., better than RDB.

per sorted run, (i) reducing the count of fragmented files by half and (ii) increasing the average file saturation. In Fig. 4, we observe that, File-Size-Aware-Merge-on-Scan, in exchange of a nominal increase (4% on average) in the RQ cost (Fig. 4A), reduces the overall file count (Fig. 4B) and file fragmentation (Fig. 4C) significantly, matching or bettering the state of the art. In Fig. 5B, we also observe that the RQ latency in File-Size-Aware-Merge-on-Scan is 10% lower than that of Merge-on-Scan on average. This is because File-Size-Aware-Merge-on-Scan pays this compaction cost only when a RQ reads more than $M/2$ size data from every qualifying level, which improves the average RQ latency.

File-Size-Aware-Merge-on-Scan vs. SuccinctKV: SuccinctKV [60] aims to improve RQ latency and CPU utilization in LSM-trees by triggering compactions during RQs. Compactions are triggered (i) on the cue of RQs, but (ii) only if a level is saturated. However, blindly triggering compactions without estimating the cost of compactions leads to higher space amplification, read amplification, and compaction debt. For instance, if one file from Level i and 100 files from Level $i+1$ qualify for a RQ, SuccinctKV would compact all 101 qualifying files and write them back to Level $i+1$. This leads to extremely high write amplification, affecting overall performance. SuccinctKV, thus, suffers from the same limitations as File-Size-Aware-Merge-on-Scan, as discussed below.

High Cost for Writes. Despite improving the RQ latency, the overall file count, file fragmentation, and compaction debt, File-Size-Aware-Merge-on-Scan still performs 167× more writes and moving (writing and reading) 31% more data than the state of the art, as shown in Fig. 5E and Fig. 5F, respectively. We address this by updating the compaction criteria and by further limiting the amount of superfluous writes during a RQ-triggered compaction.

4.3 Bounded-Merge

Avoiding Superfluous Writes in Target Level. Bounded-Merge picks levels for compaction judiciously from the RQ-qualifying sorted runs with the goal of minimizing superfluous rewriting of data to the same level. The key intuition here is that if a RQ reads x entries from Level i and y entries from Level $i+1$, the likelihood of overlapping entries increases when $x \geq y$, which means that Level $i+1$ contains more invalid entries, especially in the workload with updates (and deletes). On the other hand, $x << y$ indicates that Level i has very few entries qualifying for a RQ and may invalidate fewer entries from Level $i+1$. For instance, in Fig. 3, *RQ* 4 reads about 10% of data from Level 2 and the remaining 90% from the last level (Level 3) and would compact all qualifying data under File-Size-Aware-Merge-on-Scan, writing the compacted data in Level 3. However, this means 90% of the data written to Level 3 post-compaction was already in the same level. This leads to the spike in total data written under File-Size-Aware-Merge-on-Scan, as shown in Fig. 5E. In practice, File-Size-Aware-Merge-on-Scan frequently encounters this scenario, because after the first RQ-triggered compaction, subsequent (overlapping) RQs are expected to read more data from the deeper level and fewer from the shallower levels. Frequently rewriting compacted data during RQs increases overall writes significantly and unnecessarily slows down RQs. Bounded-Merge addresses this issue by introducing a second criterion for triggering compactions during RQs.

RangeReduceRatio. To this end, we introduce RangeReduceRatio, a configurable knob that defines the minimum amount of data that must qualify for a RQ from Level i, with respect to the qualifying data from Level $i+1$ for a compaction to be triggered. For instance, if RangeReduceRatio is set to 0.5, then Bounded-Merge will trigger a compaction between two consecutive levels, Level i and $i+1$, only if the RQ-qualifying data in Level i is at least half of that in $i+1$.

Levels Selection in Bounded-Merge. Bounded-Merge uses a light-weight algorithm (Algo. 1) to distill the levels from which the RQ-qualifying data will be merged and written back to storage. Most commercial LSM-engines maintain some file metadata, including the total number of entries (n), the smallest key (k_{min}), and the largest key (k_{max}) for every file. Bounded-Merge uses this metadata to find the count for qualifying entries from each level for a RQ. Each file falls into one of the following four categories: (i) *complete overlap*, where all entries in the file qualify for a RQ; (ii) *no overlap*, where no entries qualify; (iii) *partial overlap*, where either the first or last half of the file qualifies; and (iv) *contained overlap*, where only a subset of entries from the middle of the file qualifies for a RQ. In case of partial and contained overlap, Bounded-Merge performs approximately $\frac{file\_size}{P \cdot B}$ I/Os per level to *Compute* the qualifying entries count; these pages are cached and later used while executing RQ. Lastly, it selects the consecutive levels and merges the qualifying data from those levels during the RQ. In the

Algo. 1 LevelsSelectionInBounded-Merge

Input: Meta Data of L levels, RQ $R = [r_{start}, r_{end}]$
Output: Best levels set to compact (empty if none)
▷ **Phase 1: Compute RQ qualifying entries per level**
Initialize $T_e \leftarrow$ array of size L ▷ Entries count per level
for each $level \in \{1, \ldots, L\}$ **do**
  Initialize $T_{level} \leftarrow 0$
  for each file f in $level$ **do**
    Get file keys $[k_{min}, k_{max}]$ and number of entries n
    if $r_{start} \leq k_{max}$ **and** $r_{end} \geq k_{min}$ **then**
      $e \leftarrow n$ ▷ Complete overlap
    else if $r_{end} \geq k_{min}$ **and** $r_{end} \leq k_{max}$ **then**
      $e \leftarrow Compute(r_{start}, k_{max}, n)$ ▷ Partial overlap
    else if $r_{start} \leq k_{max}$ **and** $r_{start} \geq k_{min}$ **then**
      $e \leftarrow Compute(k_{min}, r_{end}, n)$ ▷ Partial overlap
    else if $r_{start} > k_{min}$ **and** $r_{end} < k_{max}$ **then**
      $e \leftarrow Compute(r_{start}, r_{end}, n)$ ▷ Contained
    else
      $e \leftarrow 0$ ▷ No overlap
    end if
    $T_{level} \leftarrow T_{level} + e$
  end for
  $T_e[level] \leftarrow T_{level}$
end for
▷ **Phase 2: Decide on levels to compact**
Initialize matrix D of size $L \times L$
for each Level i from 1 to L **do**
  for each Level j from i to L **do**
    if $i = j$ **then**
      Set $D[i, j] \leftarrow True$
    else
      Calculate ratios $r_k = \frac{T_e[k]}{T_e[k+1]}$ $\forall\, k = i$ to $j-1$
      if $\begin{cases} r_k \geq \text{RangeReduceRatio } \forall\, k = i \text{ to } j-1 \textbf{ and} \\ T_e[k] \cdot E \geq M/2 \;\forall\, k = i \text{ to } j \end{cases}$ **then**
        Set $D[i, j] \leftarrow True$
      else
        Set $D[i, j] \leftarrow False$
      end if
    end if
  end for
end for

Let $d \leftarrow$ *Select deepest levels from D where* $D[i, j]$ *is True*
return d ▷ Best levels set to compact, empty if none

case of a tie, where two sets of levels (e.g., Levels 1–4 and Levels 6–8) pass all the checks and qualify for a merge, Bounded-Merge picks the deepest levels set (Levels 6 – 8), as they have proportionally more logically invalidated entries due to their larger capacity.

Optimal Value for RangeReduceRatio. We performed a large-scale empirical analysis to identify the optimal value(s) of RangeReduceRatio, by varying its values for all realistic size ratios, i.e., 2 through 16, while tracking 9 performance metrics for every experiment, as shown in Fig. 6. We observe that for a uniform workload, the optimal value of RangeReduceRatio lies around $\frac{1}{T}$, which means for Bounded-Merge to trigger a compaction between Levels i and $i+1$, Level $i+1$ must contain $T\times$ more qualifying data than that in Level i. The smaller value (than $\frac{1}{T}$) of RangeReduceRatio minimizes space amplification and compaction debt, but performs significantly more writes overall, leading to slower RQs. On the other hand, larger values of RangeReduceRatio lead to almost no RQ-triggered compactions and mimic the state of the art. A value closer to $\frac{1}{T}$ achieves total data movement comparable to the state of the art, with overall better performance.

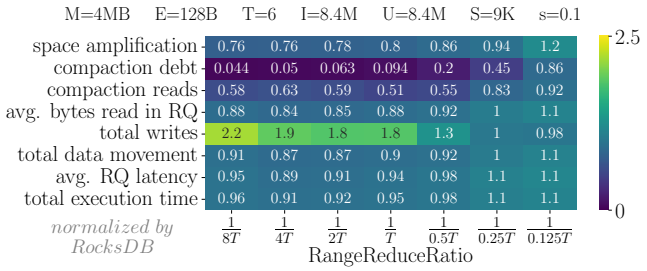

Fig. 6: A value of RangeReduceRatio around $1/T$ shows better performance across all metrics, except total writes, which is compensated by the fewer reads performed during range queries, leading to less overall data movement.

Bounded-Merge Offers Benefits Across the Board. Bounded-Merge compacts data strategically and improves the overall performance of the LSM-based system.

(1) RQ latency and bytes read during RQs: Bounded-Merge improves the average RQ latency by 3% (as shown in Fig. 5A) and reduces the average bytes read during RQs by 10%-15% for uniform workloads. This improvement is achieved by removing invalid entries, i.e., by reducing their count by a factor of β over ϕ RQs at the cost of λ additional I/Os. This is achieved by purging logically invalid entries and writing back the RQ-qualifying data as part of a single sorted run. As a result, less data is read overall, which speeds up subsequent queries, as shown in Fig. 5B. We model the average I/O cost, in Bounded-Merge, for a leveled LSM-tree as $\frac{N^{uniq.}\cdot s}{B} \cdot (1 + \frac{U\cdot(1-\beta)}{T}) + \phi \cdot \lambda$ and for a tiered LSM-tree as $\frac{N^{uniq.}\cdot s}{B} \cdot (1 + U \cdot (1-\beta) \cdot T) + \phi \cdot \lambda$.

(2) Read amplification: Bounded-Merge provides overall better read amplification than RocksDB, as, unlike normal compactions, it uses RQs to read data and piggybacks valid entries. The RQ read amplification is more pronounced and shows up to 15% improvement in Fig. 5A.

(3) Compaction debt: The eager data rewriting frees up shallower levels and offers better compaction debt. Fig. 5C shows a 90% reduction in compaction debt, allowing shallower levels to ingest more with fewer compactions.

(4) Space amplification: The space amplification is improved compared to RocksDB and can be minimized at the cost of more writes by decreasing the value of RangeReduceRatio. In Fig. 5D, Bounded-Merge shows an improvement in space amplification for 20% of RocksDB, which is even better in both Merge-on-Scan and File-Size-Aware-Merge-on-Scan. We model RQ read amplification for Bounded-Merge as follows.

$$\text{BM-RA}_{level} \approx (1 + \frac{U \cdot (1-\beta)}{T}) \cdot S \quad (3)$$

$$\text{BM-RA}_{tier} \approx (1 + U \cdot (1-\beta) \cdot T) \cdot S \quad (4)$$

(5) Total writes: Bounded-Merge offers comparable total writes to that of the state of the art as it judiciously performs merging based on RangeReduceRatio. In Fig. 5E it shows only 20% more writes than RocksDB.

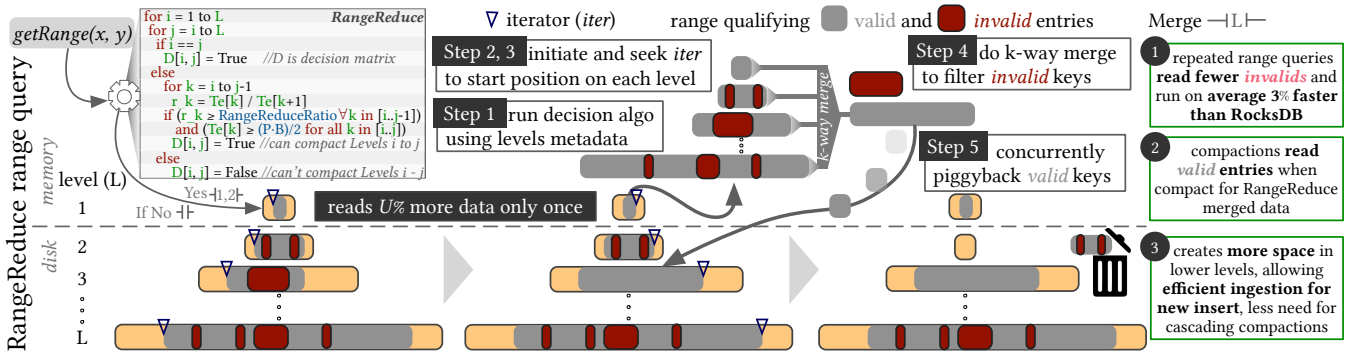

Fig. 7: The flow of a RQ in *RangeReduce*, compacts qualifying data from levels to improve the overall performance.

(6) Overall data movement: The slightly higher total writes performed by Bounded-Merge, however, is compensated by the reduced data read during RQs and future compactions. This leads to 10% less data movement overall, as shown in Fig. 5F.

To further reduce total writes in Bounded-Merge, next, we introduce Level-Renaming, which performs up to $T \cdot L$ fewer cascading compactions whenever the LSM-tree grows in height.

4.4 Level-Renaming

We propose Level-Renaming as an extension to the LSM compaction design space to reduce write amplification owing to compactions. With Level-Renaming every time the last level of an LSM-tree (Level L) reaches capacity, we rename all levels in the tree by incrementing their level ID by one, which updates the capacity of every level by a multiplicative factor of the size ratio of the tree. This means that Level L becomes Level $L+1$, Level $L-1$ becomes Level L, and so on, and the capacity of the levels is incremented by a factor of T. We then add an empty new level at the top of the tree and label it as Level 1. Level 1, at this point, can absorb T buffer flushes before it is saturated, avoiding the need for (cascading) compactions after every flush. In state-of-the-art LSM-engines, files are moved lazily – one (or more) file(s) at a time, from Level L to Level $L+1$ – once Level L is saturated, and then wait until Level L reaches its capacity again to move more files. This leads to fewer files moved *trivially*, i.e., through pointer manipulation, and more bytes compacted overall. Level-Renaming reduces the overall need for compactions, which optimizes for total writes.

4.5 RangeReduce

RangeReduce brings all the proposed optimizations together. We integrate RangeReduce with RocksDB [20], a widely used, state-of-the-art, commercial key-value store, to evaluate its performance. We use RocksDB's *main* thread to (i) first decide which levels can be merged during a RQ (as shown in Fig. 7 Step 1), before (ii) initiating iterators on every level and (iii) fast-forwarding them to the starting position for the requested range, (Fig. 7 Step 2, 3). Next, we (iii) perform a k-way merge operation to filter the invalid entries (Fig. 7 Step 4), while simultaneously (iv) pushing the merged result using RocksDB's *flush* thread from the chosen levels (Fig. 7 Step 5). At the end of each RQ, we (v) mark the files that have been merged into deeper levels for garbage collection. This way, future compactions will also read less data with a higher fraction of valid entries, improving the overall read and write amplification. Next, we discuss our experimental results and findings.

5 EVALUATION

In this section, we evaluate RangeReduce against the state of the art in terms of (i) *RQ performance* (latency and bytes processed per query), (ii) *space amplification*, (iii) *compaction debt*, (iv) *throughput*, and (v)*overall data movement*. Our analysis shows that RangeReduce both enhances the performance of RQs and brings the overall state of an LSM-engine *closer to ideal*. The core questions we set out to answer in our evaluation are the following.

(1) Holistic Improvements with RangeReduce: How much future work is reduced by RangeReduce in terms of compaction debt and space implication? To what extent does this impact the overall data moment and throughput of LSM-trees?

(2) Using RQs to Approach the Ideal LSM Shape: How RangeReduce is shaping the LSM-trees towards the ideal? To what extent does it reduce the compaction overhead, worst-case space amplification and improve average RQ latency?

(3) A Better Ingestion-Compaction Tradeoff: How much RangeReduce improves on ingestion performance while performing eager compactions during RQs? To what extent does the selectivity of RQs play a role in optimizing this performance?

Experimental Infrastructure. We use a server with an Intel(R) Xeon(R) Gold 6240R CPU with 2.40GHz cores, 192GB of RAM, 1TB SSD, running Ubuntu 20.04 LTS. We use KVBench [62] to generate the experimental workloads, and we integrate RangeReduce with RocksDB 8.5.0 [20].

Default Setup. The default experimental setup involves ingesting 1GB data with unique keys, each 128B (key is 16B and value is 112B), followed by interleaved 1GB of updates and 9K RQs. The average range query selectivity is 0.1. Unless mentioned otherwise, the LSM-tree has an in-memory buffer of 4MB and has a size ratio of 6. We compare the performance with both (i) *uniform-randomly distributed* range queries and (ii) *overlapping* range queries.

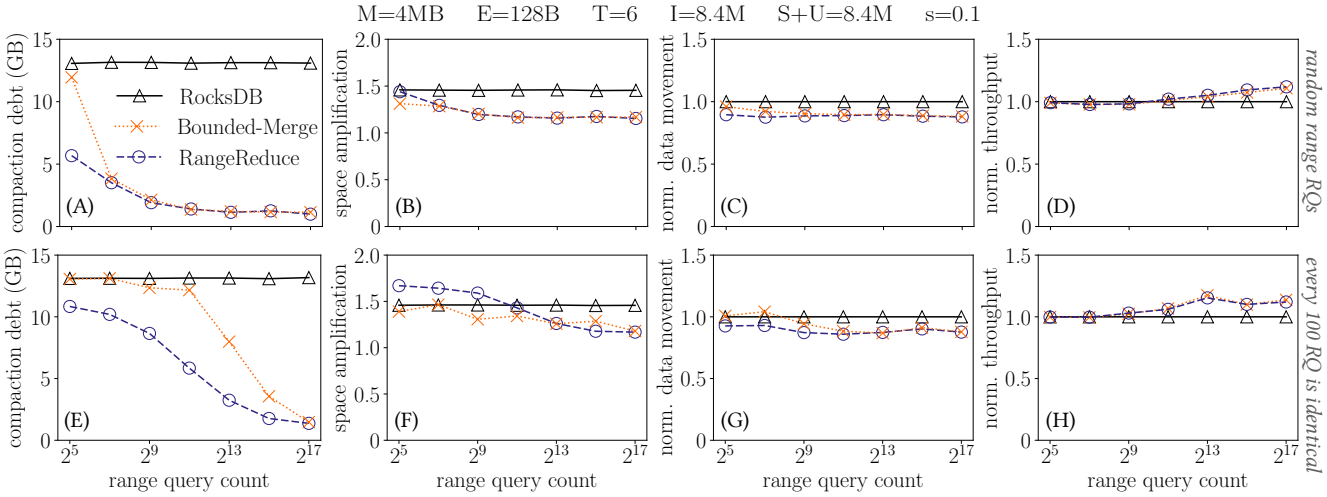

Fig. 8: Bounded-Merge and RangeReduce significantly reduces compaction debt in the worst case (all RQs are distributed uniformly randomly, (A-D)) and under a practical workload (each set of 100 RQs is identical, (E-H)). They show less space amplification (B, F), better normalized data movement (C, G), and normalized throughput (D, H) than the state of the art.

5.1 Holistic Improvements with RangeReduce

RangeReduce Reduces Work of Future Compactions. In Fig. 8A and 8E, we compare the compaction debt of RangeReduce and Bounded-Merge to that of RocksDB as we increase the number of RQs. Both RangeReduce and Bounded-Merge outperform RocksDB even with as few as 32 RQs, as they (i) leverage the merged result of RQs to eliminate logically invalid entries from the LSM-tree and (ii) free up the shallower levels. In contrast, RocksDB accumulates this debt and performs lazy compaction on a need-basis, which results in high read and write amplifications, more CPU resource utilization, and larger database size on slow storage. As the amount of RQs increases, RangeReduce and Bounded-Merge show up to 90% less compaction debt than RocksDB, reducing the future work. While Bounded-Merge and RangeReduce behave similarly in many scenarios, the latter reduces future work even for workloads without RQs by performing Level-Renaming whenever the height of an LSM-tree increases, which reduces compaction debt for fewer RQs (Fig. 8A and 8E).

Optimizes Space Amplification. In Fig. 8B and 8F, we use the same experiment to compare space amplification. As we increase the proportion of RQs to updates in the workload, the space amplification approaches the ideal. This is because RangeReduce and Bounded-Merge remove duplicate entries from the LSM tree and write more packed files (only valid) at larger levels, aiming to convert the LSM-tree into a single sorted run, which is the ideal. Space amplification (y-axis) improves up to 20% with the increase in the proportion of RQs along the x-axis.

Performs Less Data Movement. In Fig. 8C and 8G, we compare overall data movement in terms of compaction read, compaction write, RQ read, and RQ write. Both approaches show up to 12% less overall data movement from/to the slow storage during RQs and compactions. Both RangeReduce and Bounded-Merge outperform RocksDB in overall data movement while merging valid entries during an RQ, as they both reduce the number of I/Os required for future similar or overlapping RQs and the need for compactions.

Offers Better Throughput. In Fig. 8D and 8H, we compare RangeReduce and Bounded-Merge throughput with RockDB for the same experiment. Each RQ is an expensive task in terms of I/Os and CPU cycles, and the invalid entries make them even more costly. RangeReduce and Bounded-Merge capitalize on a few RQs and improve the performance of others by reducing the number of I/Os and CPU cycles needed and improving throughput and reducing space amplification by up to 20%.

5.2 Using RQs to Approach the Ideal LSM Shape

Taming the Need for Compactions. In Fig. 9A and 9E, we compare RangeReduce and Bounded-Merge with RocksDB as we vary the size ratio (T) from 2 to 10 along the x-axis. RangeReduce and Bounded-Merge show 45% to 50% less data movement than RocksDB for compactions in merging data between levels. This is because both approaches perform compactions (i) when the level reaches saturation (state-of-the-art way) and, also, (ii) strategically merge the RQ results without requiring them to be read separately, resulting in less reading.

Reducing Latency and Read Overhead of RQs. In Fig. 9B, 9C, 9F, and 9G, we compare the work done by RQs in terms of total bytes read and average latency of RQs for the same experiment. The ideal line – in Fig. 9B, 9F – shows the lower bound for total bytes that must be read from the LSM-tree, based on selectivity, for workloads with no updates or deletes. RangeReduce and Bounded-Merge reduce the read overhead during RQs from 10% to 15% for size ratios of 2 to 10, respectively. Both approaches show fewer bytes transferred overall than RocksDB when reading from and writing to slow storage, as RocksDB never uses the work done during a RQ.

Taming Worst-case Space Amplification. In Fig. 9D and 9H, we compare the worst-case space amplification as we vary the size ratio

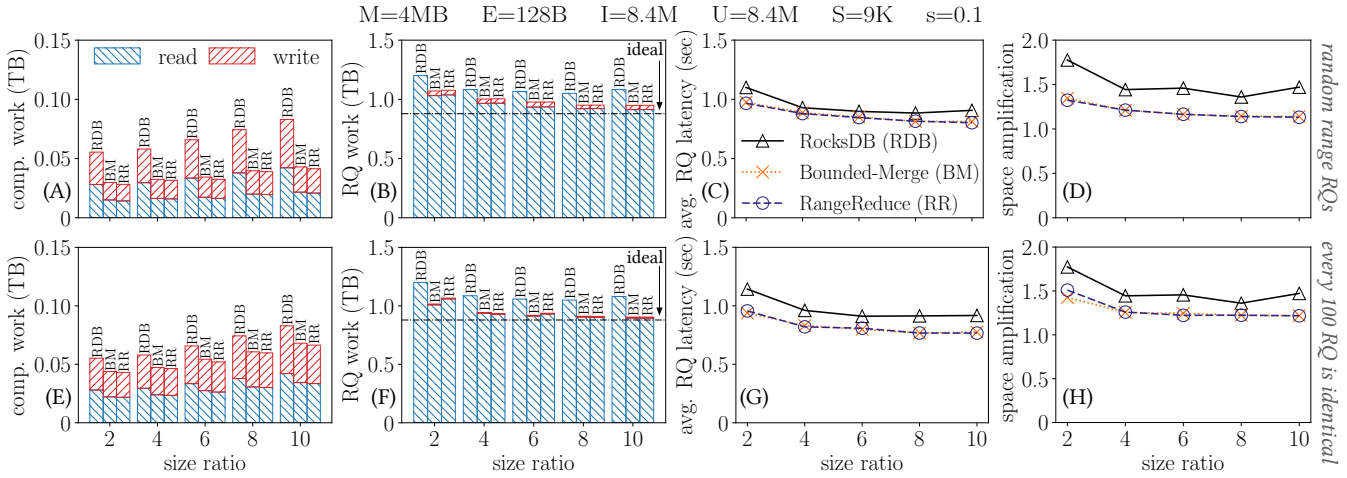

Fig. 9: Bounded-Merge and RangeReduce reduce the overall need for compactions in the worst case (all RQs are distributed uniformly randomly, (A-D)) and under a practical workload (each set of 100 RQs is identical, (E-H)). They perform less work during RQ (C, G), which offers better RQ-latency (B, F) and near-ideal space amplification (D, H) than RocksDB.

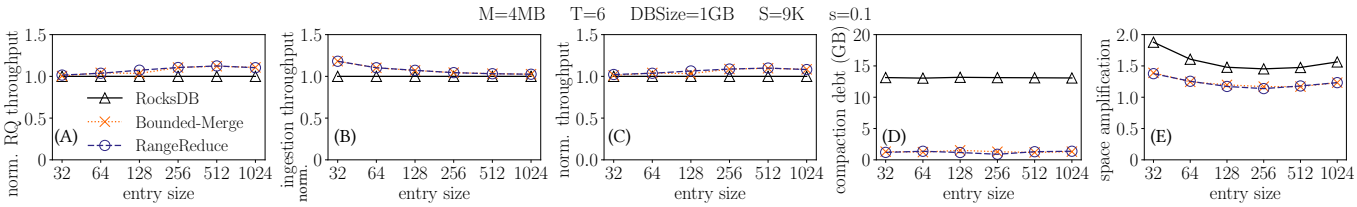

Fig. 10: For varying entry sizes, RangeReduce and Bounded-Merge provide better RQs (A), faster ingestion (B), and improved throughput (C) while leaving significantly less compaction debt (D) and close to ideal space amplification (E) than RocksDB.

from 2 to 10 along the x-axis. The state-of-the-art implementation of LSM-trees, such as RocksDB, stores nearly $\frac{1}{T}$ worth of duplicates on slow storage for worst-case workload scenarios, where Levels 0 to $L-1$ are all updates against entries stored in the last level (Level L). RangeReduce and Bounded-Merge push this boundary for all size ratios and improve the space implication up to 25% for workloads in the presence of RQs.

5.3 A Better Ingestion-Compaction Tradeoff

Better Ingestion and Nearly Ideal Database Size. In Fig. 10A-10E, we compare the RQs, ingestion (inserts and updates), and overall throughput of the LSM-trees, along with compaction debt and space amplification, to show that RangeReduce can maintain performance benefits while varying entry sizes from 32B to 1024B. As we increase the entry size, we proportionally decrease the number of unique inserts and updates to maintain a consistent database size. The count of RQs remains unchanged as well. Both RangeReduce and Bounded-Merge improve RQs throughput (Fig. 10A) up to 12%, ingestion throughput (Fig. 10B) by 2 – 3%, and overall throughput (Fig. 10C) by 9%. They also maintain their benefits for compaction debt (90%), Fig. 10D, and space amplification (25%), Fig. 10E, across different entry sizes.

In Fig. 11A and 11B, we compare the ingestion throughput and database size in an epoch-based experiment, where during the first

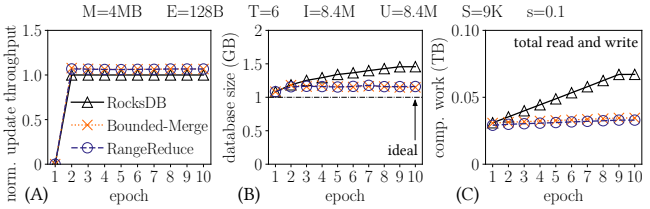

Fig. 11: RangeReduce offer better ingestion (A), nearly ideal DB size (B), and fewer compaction (C) than RocksDB.

epoch we load the database with 1GB of unique entries, followed by interleaved updates and RQs. RangeReduce and Bounded-Merge show an improved ingestion throughput, up to 6%, (Fig. 11A) and nearly ideal database size (12% reduction) in Fig. 11B. This is because both approaches remove duplicates from the LSM-tree during the execution of a RQ, which optimizes the database size, bringing it closer to ideal and freeing up the shallower levels. This reduces the need for compaction (and cascading compaction) and provides better ingestion performance. RangeReduce and Bounded-Merge show up to 50% less bytes movement (read and write) during compactions than RocksDB, Fig. 11C.

RangeReduce is Better for Long RQs. In Fig. 12A, 12B, and 12C, we compare the compaction debt, space amplification, and normalized RQ throughput, respectively, while varying the selectivity of

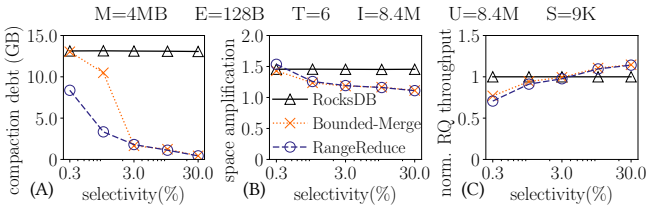

Fig. 12: RangeReduce uses long RQs to improve (A) compaction debt, (B) space amp., and (C) average RQ latency.

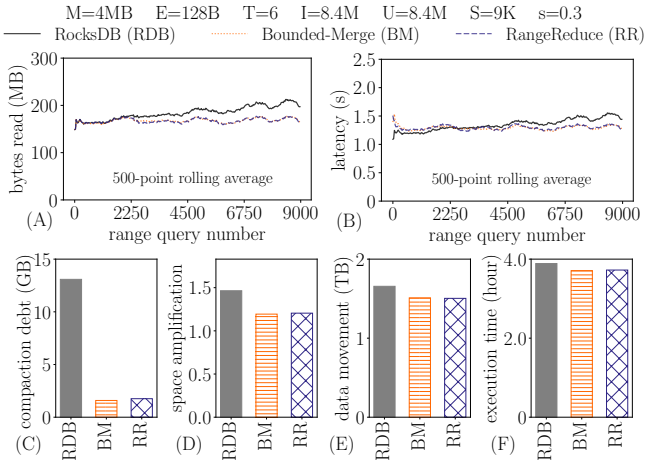

Fig. 13: RangeReduce and Bounded-Merge read fewer bytes (A), show better RQ latency (B), reduce compaction debt (C), space amplification (D), overall data movement (E), and improve total execution time for YCSB-E benchmark.

RQs on the x-axis. RangeReduce capitalizes on long RQs as they read more data, which creates an opportunity to merge and remove more invalid entries from the LSM-tree. When a RQ reads more data, the likelihood of encountering invalid entries increases. The short RQs read fewer entries and tend to remove fewer invalid entries, which does not help reduce the RQ latency. In Fig. 12C, for selectivity more than 0.3%, the normalized throughput of RQs is always higher than that of RocksDB.

Improved Performance for YCSB Workloads. We compare the performance of the RQ in terms of the number of bytes read (Fig. 13A) and latency (Fig. 13B) during query execution. We run a YCSB-E benchmark with a RQ selectivity cap of 30%, recording the latency of each RQ and the total bytes read. For the initial few RQs, RangeReduce shows a spike in latency, but performance improves for subsequent queries. RangeReduce shows on average 5% faster RQs with 10% fewer reads; the compaction debt (Fig. 13C), space amplification (Fig. 13D), and overall data movement (Fig. 13E) show 86%, 18%, and 9% improvement over RocksDB, respectively. The overall workload execution time (Fig. 13F) also shows an improvement of 4% over RocksDB.

6 RELATED WORK

Improving LSM Performance. LSM-trees are widely used in modern data stores as part of their storage layer due to their efficient ingestion performance. Past research has focused on enhancing read and write performance [11, 12, 14, 29, 54], explored various compaction tunings [27, 48, 57], and optimized read, write and space amplifications [13, 15, 47, 49]. While existing literature has focused on configurable compaction styles [48], workload-aware point and range filters [17, 21, 39, 41, 44], and write buffer optimization [31] to balance throughput and latency, none of them focus on optimizing for range query-intensive workloads. Instead, many of them added either a knob or an overhead of additional metadata to improve the performance of the LSM-tree in some way, whereas RangeReduce makes use of RQ results to enhance the overall performance.

Query-Aware Data Reorganization. Past research on adaptive indexing techniques, such as database cracking [30], UpBit [5], Adaptive Adaptive Indexing [52], and other adaptive indexes [23–26, 56] has shown that queries can be used as hints to update the data layout. Database cracking was introduced as a query-driven gradual reorganization of a column of a relational table, where data is partitioned based on the query predicates. UpBit provides efficient updates on bitmap indexing by performing out-of-place updates and merging them to the base bitvectors opportunistically at query time. Adaptive Adaptive Indexing steps away from classical query predicates-based cracking to avoid overfitting and uses radix-based partitioning to reorganize, offering a higher partitioning throughput. In the NoSQL realm, SuccinctKV [60] introduces a scan-based compaction mechanism that aims to reduce CPU overhead during RQs. Files that are completely subsumed by the query range are compacted and written back to storage. For partially overlapping files, the valid data within the file is marked using a key-range block (Meta cut), and such files are retained at the same level. While this metadata-based technique helps avoid unnecessary rewrites, it leads to two main issues: (i) increased read and space amplification due to the presence of multiple versions of the data, and (ii) it results in files becoming mutable, which contradicts the fundamental principles of LSM. Other approaches focus on automatically creating indexes, enhancing RQ performance, and optimizing for non-key lookups. Our approach, RangeReduce, is inspired by the core idea of reusing query predicates to physically reorganize data on storage. To our knowledge, RangeReduce is the first approach that performs RQ-driven compactions in LSM-trees to reorganize data on slow storage opportunistically and optimize performance specific to workloads.

7 CONCLUSION

We show that logically invalid entries in LSM-based key-value stores negatively impact the system's overall performance and accumulate a significant compaction debt, leading to high space amplification and poor resource utilization, particularly during RQ execution. The state-of-the-art LSM-engines fail to utilize the work done by RQs, i.e., merging and filtering logically invalid entries for each RQ, and thereby, wasting CPU resources and performing expensive I/Os. RangeReduce, an RQ-conscious LSM engine that removes invalid entries while reaping from the efforts of the RQ, reducing compaction debt by freeing up shallower levels, which in turn offers richer space and read amplification, better CPU resource utilization, and improved overall system performance in the presence of updates and RQs in a workload.

REFERENCES

[1] Apache. 2023. Cassandra. *http://cassandra.apache.org* (2023).

[2] Apache. 2023. HBase. *http://hbase.apache.org/* (2023).

[3] Apple. 2018. FoundationDB. *https://github.com/apple/foundationdb* (2018).

[4] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461—-466. http://dx.doi.org/10.5441/002/edbt.2016.42

[5] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. 2016. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://dl.acm.org/citation.cfm?id=2915964

[6] BigTable. 2024. Bigtable pricing. *https://cloud.google.com/bigtable/pricing* (2024).

[7] Mark Callaghan. 2022. RocksDB internals: bytes pending compaction. *https://smalldatum.blogspot.com/2022/01/rocksdb-internals-bytes-pending.html* (2022).

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 205–218. https://doi.org/10.5555/1267308.1267323

[9] CockroachDB. 2024. Pebble. (2024). https://github.com/cockroachdb/pebble

[10] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[11] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94. https://doi.org/10.1145/3035918.3064054

[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

[13] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520. https://doi.org/10.1145/3183713.3196927

[14] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 449–466. https://doi.org/10.1145/3299869.3319903

[15] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084. https://www.vldb.org/pvldb/vol15/p3071-dayan.pdf

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220. https://doi.org/10.1145/1323293.1294281

[17] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *CoRR* 2103.02515 (2021). https://arxiv.org/abs/2103.02515

[18] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[19] Facebook. 2023. MyRocks. *http://myrocks.io/* (2023).

[20] Facebook. 2024. RocksDB. *https://github.com/facebook/rocksdb* (2024).

[21] Bin Fan, David G Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 75–88. https://doi.org/10.1145/2674005.2674994

[22] Google. 2021. LevelDB. *https://github.com/google/leveldb/* (2021).

[23] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 371–381. https://doi.org/10.1145/1739041.1739087

[24] Goetz Graefe and Harumi A. Kuno. 2010. Adaptive indexing for relational keys. In *Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW)*. 69–74.

[25] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment* 5, 6 (2012), 502–513. http://vldb.org/pvldb/vol5/p502{_}felixhalim{_}vldb2012.pdf

[26] Pedro Holanda, Stefan Manegold, Hannes Mühleisen, and Mark Raasveldt. 2019. Progressive Indexes: Indexing for Interactive Data Analysis. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2366–2378. http://www.vldb.org/pvldb/vol12/p2366-holanda.pdf

[27] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24. https://doi.org/10.1007/s00778-023-00826-9

[28] Stratos Idreos and Mark Callaghan. 2020. Key-Value Storage Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2667–2672. https://doi.org/10.1145/3318464.3383133

[29] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf

[30] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2007/papers/cidr07p07.pdf

[31] Shubham Kaushik and Subhadeep Sarkar. 2024. Anatomy of the LSM Memory Buffer: Insights & Implications. In *Proccedings of the International Workshop on Testing Database Systems (DBTest)*. 23—-29. https://doi.org/10.1145/3662165.3662766

[32] Cockroach Labs. [n. d.]. CockroachDB Pricing - The cloud-native, distributed SQL database. *https://www.cockroachlabs.com/pricing/* ([n. d.]).

[33] Chen Luo. 2020. Breaking Down Memory Walls in LSM-based Storage Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2817–2819. https://doi.org/10.1145/3318464.3384399

[34] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[35] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2071–2086. https://doi.org/10.1145/3318464.3389731

[36] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *CoRR* abs/2308.0 (2023). https://doi.org/10.48550/ARXIV.2308.07013

[37] NetAppInstaclutr. [n. d.]. InstaClustr Platform Pricing. *https://www.instaclustr.com/pricing/* ([n. d.]).

[38] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[39] Prashant Pandey, Alex Conway, Joe Durie, Michael A Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1386–1399. https://doi.org/10.1145/3448016.3452841

[40] Prashant Pandey, Mart\'\in Farach-Colton, Niv Dayan, and Huanchen Zhang. 2024. Beyond Bloom: A Tutorial on Future Feature-Rich Filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 636–644. https://doi.org/10.1145/3626246.3654681

[41] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics* 14 (2009). https://doi.org/10.1145/1498698.1594230

[42] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 497–514. https://doi.org/10.1145/3132747.3132765

[43] RocksDB. [n. d.]. RocksDB-Cloud: A Key-Value Store for Cloud Applications. *https://github.com/rockset/rocksdb-cloud* ([n. d.]).

[44] RocksDB. 2020. Prefix Bloom Filter. *https://github.com/facebook/rocksdb/wiki/Prefix-Seek#configure-prefix-bloom-filter* (2020).

[45] RocksDB. 2020. Universal Compaction. *https://github.com/facebook/rocksdb/wiki/Universal-Compaction* (2020).

[46] Rockset. 2024. Rockset. *https://rockset.com* (2024).

[47] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2489–2497. https://doi.org/10.1145/3514221.3522563

[48] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2429–2432. https://doi.org/10.1145/3514221.3520169

[49] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. https://doi.org/10.1145/3318464.3389757

[50] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2023. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Transactions on Database Systems (TODS)* 48, 3 (2023), 8:1—-8:40. https://doi.org/10.1145/3599724

[51] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. https://doi.org/10.14778/3476249.3476274

[52] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. 2018. Adaptive Adaptive Indexing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 665–676. https://doi.org/10.1109/ICDE.2018.00066

[53] ScyllaDB. 2024. Online reference. (2024). https://www.scylladb.com/

[54] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 217–228. https://doi.org/10.1145/2213836.2213862

[55] Speedb. [n. d.]. Online reference. *https://github.com/speedb-io/speedb* ([n. d.]).

[56] Elvis Marques Teixeira, Paulo Roberto Pessoa Amora, and Javam C Machado. 2018. MetisIDX - From Adaptive to Predictive Data Indexing. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 485–488. https://doi.org/10.5441/002/edbt.2018.53

[57] Ran Wei, Zichen Zhu, Andrew Kryczka, Jay Zhuang, and Manos Athanassoulis. 2024. Codebase for Benchmark, Analyze, Optimize Partial Compaction in RocksDB. (2024). https://github.com/BU-DiSC/Benchmark-Analyze-Optimize-Partial-Compaction-in-RocksDB-Codebase

[58] WiredTiger. 2021. Source Code. *https://github.com/wiredtiger/wiredtiger* (2021).

[59] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 323–336. https://doi.org/10.1145/3183713.3196931

[60] Yinan Zhang, Shun Yang, Huiqi Hu, Chengcheng Yang, Peng Cai, and Xuan Zhou. 2024. SuccinctKV: a CPU-efficient LSM-tree Based KV Store with Scan-based Compaction. *ACM Trans. Archit. Code Optim.* 21, 4 (2024), 90:1—-90:26. https://doi.org/10.1145/3695873

[61] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 51–64. https://www.usenix.org/conference/fast21/presentation/zhong

[62] Zichen Zhu, Arpita Saha, Manos Athanassoulis, and Subhadeep Sarkar. 2024. KVBench: A Key-Value Benchmarking Suite. In *International Workshop on Testing Database Systems (DBTest)*. 9—-15. https://doi.org/10.1145/3662165.3662765

[63] Zichen Zhu, Subhadeep Sarkar, and Manos Athanassoulis. 2023. Acheron: Persisting Tombstones in LSM Engines. In *Companion of the International Conference on Management of Data (SIGMOD)*. 131–134. https://doi.org/10.1145/3555041.3589719