RangeReduce: Query-Driven LSM Compactions

Shubham Kaushik
Brandeis Univeristy
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 in the storage layer of modern ingestion-optimized data stores. The high ingestion throughput, however, comes at the cost of sub-optimal range query (RQ) performance. This is because LSM-trees arrange the data as a hierarchical collection of *sorted runs*, which implies that every RQ must (i) probe all sorted runs to locate the qualifying entries, (ii) scan and merge the entries from all qualifying runs, (iii) filter out the logically invalidated entries by updates and deletes on the fly, and (iv) return the most recent version of each qualifying key. 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 the presence of updates and deletes. Additionally, during compactions, the same data is read and written multiple times, further amplifying the read and write 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 and 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%.

***Index Terms*—key-value stores, range queries, compaction.**

I. Introduction

LSM-trees are Everywhere. Log-structured merge (LSM) tree is a storage-based data structure that is highly optimized for data ingestion, and thus, is widely used in the storage layer of modern key-value stores [34, 40, 44]. 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 to maximize write throughput [56]. The data on storage is stored as a hierarchical collection of *immutable sorted runs*. This means updates and deletes are realized *logically*, in an out-of-place manner, which significantly improves the ingestion latency – at the cost of space (SA) and write amplification (WA) [17, 23, 61]. LSM-trees, thus, fuel several commercial key-value stores and relational systems, including LevelDB [28], BigTable [12], RocksDB [26], MyRocks [25], WiredTiger [69], Cassandra [1], HBase [2], CockroachDB [13], ScyllaDB [63], SplinterDB [15], FoundationDB [3], Rockset [55], DynamoDB [21], and Speedb [65].

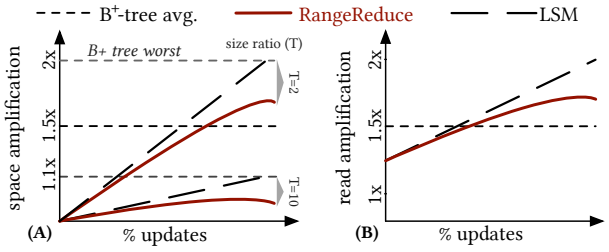

Fig. 1. (A) The average-case space amplification is 1.5× for B$^+$-trees, and it varies between 1.1× and 2× for LSM-trees. (B) *In-place* updates in B$^+$-trees keep read amplification constant; whereas, **out of place** updates in LSM-trees lead to a higher amplifications. RangeReduce improves both.

Merging in LSM-Trees. Data in an LSM-tree is spread across multiple levels on the secondary storage (hard disk or solid-state drives), where each level is exponentially larger than the previous one. A level may have multiple sorted runs ordered by creation time, and each sorted run is a collection of multiple non-overlapping files [57, 61]. A newer run always contains the latest version of an entry [59, 73]. This invariant is critical to ensure the correctness of point queries, where runs are probed from the youngest to the oldest [16, 47].

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. Thus, realizing a RQ entails (i) identifying the qualifying sorted runs, (ii) reading the qualifying entries from the selected runs, and (iii) merging them in memory to add the latest version of each entry to the result set.

Compaction-triggered Merges. To reduce SA and WA, the sorted runs in an LSM-tree are periodically merged during a *compaction* [56, 61]. In a compaction, a subset of files in a level is merged with the overlapping files in the subsequent level. Compacting the sorted runs leads to having fewer, longer sorted runs in the tree, which bounds SA and improves query performance [6, 23]. During a compaction, (i) the files (from adjacent levels) to be compacted are read into memory, (ii) the entries contained are merged to remove all logically invalidated entries, and (iii) the valid entries are written back as new files [57]. This leads to sub-optimal performance.

Challenge 1: High Read and Space Amplification. Updates and deletes in LSM-trees are realized out-of-place, i.e., instead of physically altering the target entries, a new version of the entry is inserted that *logically invalidates* the older target entry.

This leads to considerable SA, which in turn leads to sub-optimal range query performance. Thus, as the proportion of updates and deletes increases in a workload, so does the SA (Fig. 1A). For instance, a leveled LSM-tree with a size ratio $T = 4$ may have up to 25% logically invalidated entries. Thus, 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** (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 $5 \times 10 = 50$ 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 (RA) for RQs, LSM-trees perform recurrent superfluous merges for repeating RQs, even if they are identical. For instance, in a workload where identical or largely overlapping RQs are executed repeatedly, the data is read and processed separately for each query, thus redoing the same work. This means, both the **random I/O cost** of fetching the data from storage and the **CPU cost** to merge, filter, and construct the result must be **paid separately for every RQ** (barring some caching benefits).

Challenge 3: Wasted Work. LSM-trees, by design, create a *compaction debt* [10] during ingestion, which entails reading and writing the same entries repeatedly during future 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 RA and WA** [17, 61]. Thus, an entry that qualifies for C recurring or overlapping RQs is read and written a total of $C + T \cdot L$ times during its lifetime, exacerbating the problem of wasted work.

Why the State of the Art is Not Enough. SuccinctKV [71] is the only academic endeavor that uses RQs as a hint for compactions. However, the benefits in RQ performance come at the cost of increased WA, SA, and compaction debt.

- *High WA*. SuccinctKV fails to estimate the data rewritten within a level during query-driven compaction, causing excessive writes. For example, merging one file from Level i with 100 files from Level $i + 1$ rewrites 99 files' worth of data at Level $i + 1$.
- *Increased SA*. SuccinctKV adds a fixed-size meta-block in each file to track valid entries after a query-driven compaction. In addition, it also retains the terminal RQ-qualifying files data at the same level, increasing SA.
- *Unbounded compaction debt*. Query-driven compactions in SuccinctKV are triggered only when a level becomes saturated. As a result, many invalid entries stay in unsaturated levels, leading to unbounded accumulated compaction debt.

Solution: RangeReduce. We propose RangeReduce, which takes advantage of the merging performed during RQs to trigger *query-driven partial compaction*, and thus, avoid redundant work during future compactions. In the process, RangeReduce eliminates any logically invalid entries and ensures that only the latest version of each entry remains in the tree after compaction. At the cost of writing the data back to secondary storage after the RQ, RangeReduce (i) improves performance for all (identical or overlapping) future RQs and (ii) significantly reduces compaction debt for all (recurring and overlapping) RQs. However, unlike SuccinctKV [71], RangeReduce runs a lightweight algorithm to navigate the tradeoff between the *cost* and *benefits* of eager compactions. This ensures the improved RQ performance never comes at the cost of high WA owing to compactions. RangeReduce, thus, **improve I/O cost** for all subsequent overlapping **RQs** as they read fewer duplicates, thereby **reducing RA and SA**. 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 RA and SA, RQ performance, and offers additional benefits, such as fewer compactions, less data movement, and better CPU utilization.

Contributions. The contributions of our work are as follows.

- 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 triggers compaction on the cue of RQ and, thus, optimizes resource utilization and mean latency for future RQs.
- RangeReduce reduces compaction-debt significantly – by up to 90% – and the overall data movement by 12%. This also ensures that the tree will perform fewer compactions in the future since the shallower levels are now less saturated.
- 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 show that it outperforms the state of the art for RQ-intensive workloads.

II. Background

The LSM Architecture. The ingestion-optimized design of LSM-trees arranges the data on storage as a hierarchical collection of levels [40, 44]. A level may have one (*leveled LSM*) or multiple (*tiered LSM*) sorted runs [56], and each run is typically written as multiple files [24, 56]. As we move deeper, the capacity of the levels increases exponentially, by a factor T, bounding the height of the tree [16, 39]. To facilitate high-throughput writes, incoming data is first batched in a memory buffer. Once the buffer is full, the entries are flushed to slower storage as immutable sorted runs [37, 57].

Compactions. To ensure competitive query performance, LSM-trees compact data across multiple sorted runs periodically [20, 61]. During a compaction, all (*full compaction*) or

part (*partial compaction*) of the data from Level i is merged with the overlapping data in Level $i+1$. This (i) limits the number of sorted runs a query needs to access and (ii) reclaims space by removing logical updates and deletes [18]. An LSM-tree with partial compaction [42, 61] performs up to L cascading compactions every time the 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 [26, 28, 55] . Leveled LSM-trees offer competitive query performance as it has fewer sorted runs overall in the tree, but have a high WA owing to the eager merges [18, 19]. In a tiered LSM-tree, data is organized into multiple *sorted runs* or *tiers* within each level, allowing duplicates within a single level [1, 2, 52]. 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-tree is optimized for writes, but suffers from high query costs [17].

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

Put. A *put* simply inserts a new key into the LSM buffer and considers the ingestion as successful. Out-of-place updates in LSM-trees mean the same *put* API is used to update data.

Get. A *get* returns the value associated with the latest version of the requested key, if it exists. The search begins at the buffer and continues from smaller to larger levels on the storage. Within a level, it probes the sorted runs from youngest to oldest. A search terminates as soon as the target key is located.

Delete. Deletes are realized logically by inserting 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.

Range Query (RQ). A *RQ* retrieves all keys within a specified key range. A RQ is realized by (i) first initializing the iterators for each sorted run (Fig. 2A, Step 1), (ii) performing a seek operation to locate the first key that qualifies for the requested range (Step 2), (iii) reading the qualifying data from each sorted run, and the (iv) performing a k-way merge in memory (Step 3) while (v) filtering out invalid entries and preparing the query result (Step 4) [58]. The scan continues until a key outside the range is found or the iterator reaches the end of a run. This process usually requires $\mathcal{O}(L)$ pages of memory for a leveled LSM-tree and $\mathcal{O}(L \cdot T)$ pages in a tiered tree, as each iterator reads one page per sorted run. 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 [18, 41, 70]. 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 [18]. 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.

III. Problem: Excessive Reads & Merging

Next, we outline the limitations of the state of the art by quantifying the wasted effort during RQs and compactions.

A. 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, including SuccinctKV [71], perform a significant amount of superfluous work when realizing RQs, which 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-engines need to scan and merge data from all qualifying sorted runs in the tree to construct the query result. This leads to superfluous reads, especially in the presence of updates and deletes in a workload (P1, Fig. 2A). This is because updates and deletes in LSM-engines are realized logically and out of place, without physically altering the target data. The logically invalid data retained in the tree leads to SA, and depending on the proportion of updates and point and range deletes in a workload, this can be significantly high [23, 59]. SuccinctKV attempts to address this by compacting data during RQs; however, the benefits offered are limited, as it performs compactions only for levels that are saturated. Moreover, SuccinctKV further increases SA by duplicating RQ-qualifying terminal files' data at multiple levels and adding an extra metadata block to every file. Therefore, LSM-systems suffer from high RA and slower RQ performance for RQ-heavy workloads in the presence of updates and point and range deletes.

(B) Wasted Work for Recurring and Overlapping RQs. The superfluous work during RQs does not benefit any subsequent recurring or overlapping RQs. In fact, every time a RQ recurs, an LSM-engine performs the same amount of superfluous work. SuccinctKV's saturation-based compaction trigger limits its benefits for recurring and overlapping RQs when levels are partially full. This leads to several superfluous 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 [16, 61]. Thus, RQs and compactions

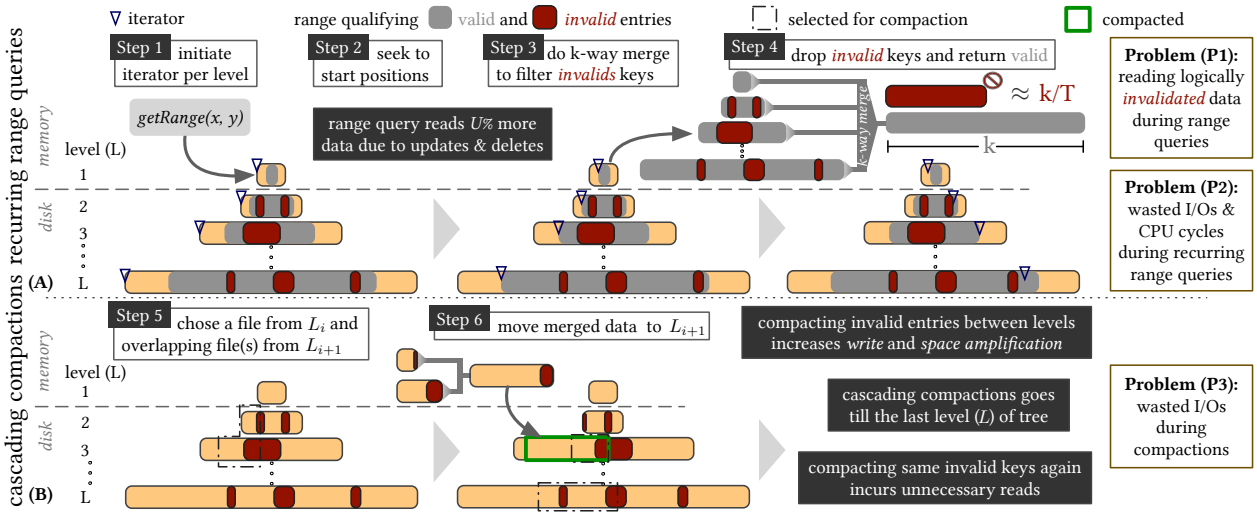

Fig. 2. (A) The execution of the recurring RQs leads to superfluous reads, resulting in higher resource utilization. (B) The compaction rewrites *invalid* keys multiple times on the same level, increasing *read*, *write*, and *space* amplification.

multiply the number of superfluous reads in LSM-engines, propelling the RAs and leading to poor resource utilization and suboptimal overall performance.

B. Modeling & Metrics

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

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

Space Amplification. The 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 SA for a leveled LSM-tree is $\mathcal{O}(\frac{1}{T})$, and that for a tiered LSM-tree is $\mathcal{O}(T)$ [18]. Note that the SA 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 SA is capped at 11.1%. In the presence of deletes, however, the SA is significantly exacerbated, as a smaller-sized tombstone can invalidate a larger-sized entry [7, 60].

TABLE I
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

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 RA 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 RA 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 $N = N^{unq} + \frac{N^{unq.}\ U}{T}$ 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 leveled LSM-tree and performs $\frac{N^{unq.}\ s}{B} \cdot (1 + \frac{U}{T})$ 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 $\frac{N^{unq.}\ s}{B} \cdot (1 + U \cdot\ T)$. However, in the ideal where all updates are realized in place, the tree would have zero SA, 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 $\frac{N^{unq.}\ s}{B}$ in both tiered and leveled LSM-trees. Based on this, we quantify the RA for a RQ in a leveled (RA_{level}) and a tiered (RA_{tier}) as follows.

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

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

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 and constructing the query result. For instance, a tiered LSM-tree with 5 levels and $T = 10$ sorted runs per level would merge up to $5 \times 10 = 50$ 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 $\mathcal{O}(N \cdot\ s \cdot \log(N \cdot\ s) \cdot\ f_{RQ})$ 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.

RA and WA 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 *RA* and *WA*, 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 RA and WA to be $\mathcal{O}(T \cdot\ L)$. For the write-optimized tiered variant, every entry is written once in each level of the tree, resulting in a RA and WA of $\mathcal{O}(L)$.

Compaction Debt. Another artifact of the "***write first, organize later***" principle of LSM-trees is the compaction debt [10, 67]. 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.

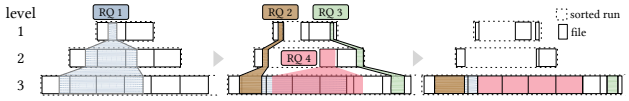

Fig. 3. Merge-on-Scan may rewrite data at the same level, which increases total writes and hence write amplification.

Hidden Cloud Costs. In recent years, there has been a shift toward cloud-native databases, which include BigTable at Google [9], Cassandra at Apache [43], CockroachDB at Cockroach Labs [14], and RocksDB at Meta [54]. 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.

IV. RangeReduce

We propose RangeReduce, a RQ-aware LSM-engine that *piggybacks on the reads performed by RQs to write the valid data back to storage as part of a single sorted run*. RangeReduce improves RQ latency and operational throughput, reduces SA and overall data movement, and significantly diminishes compaction debt. We first propose an ensemble of RQ-driven compaction strategies – File-Size-Aware-Merge-on-Scan and Bounded-Merge – which perform *compactions on the cue of RQs* to improve the latency for future RQs and significantly reduce SA and compaction debt. Next, we introduce Level-Renaming, which reduces the data movement (reads and writes) performed by compactions by moving files to deeper levels *logically* 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.

Blind Merge-on-Scan. Before discussing the proposed solution, we outline the operating principles of a basic query-driven compaction strategy that blindly compacts the data qualifying for a RQ, writing it back as a part of a single sorted run. The core idea of this naïve Merge-on-Scan algorithm is as follows: when realizing a RQ, (i) read data from all qualifying sorted runs, (ii) merge the qualifying entries in memory to prepare the query result, and subsequently, (iii) write back only the valid data to storage as a single sorted run. The data is written back (i) to the deepest qualifying level in a leveled LSM-tree and (ii) to the oldest sorted run in a tiered LSM.

Fig. 3 shows when realizing RQ 1, Merge-on-Scan compacts data from the shallower levels, i.e., Levels 1 and 2, with

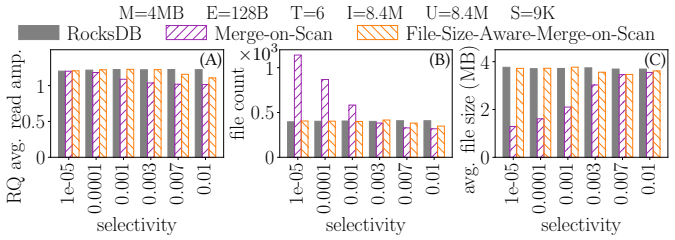

Fig. 4. (A) Merge-on-Scan reads less data compared to RocksDB due to eager compactions; but creates (B) many (C) fragmented files in the process. File-Size-Aware-Merge-on-Scan addresses this using lightweight checks.

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 RQs, reducing RA 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 SA (up to 25%, as shown in Fig. 5D) of the LSM-tree.

Limitations of Blind Merge-on-Scan. The eager compactions of Blind Merge-on-Scan lead to the following pitfalls.

(1) 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 $9K$ RQs with a selectivity of 0.00001, Merge-on-Scan has $2\times$ 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) Slow 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 $1.5\times$ higher (Fig. 5B), owing to the superfluous writes.

(3) Increased WA for short RQs. The problems of blind Merge-on-Scan are further exacerbated in the presence of short RQs that typically access a few files per sorted run. 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 means most of the entries in the non-compacted files in shallower levels are rewritten to the same level without any forward progress, leading to high WA and overall data movement. The eager rewriting increases total writes by up to 5 orders of magnitude in Merge-on-Scan, Fig. 5E. This becomes a major performance bottleneck for workloads with many short RQs.

A. 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, reducing rewriting to the same level.

Avoiding Superfluous Writes to Same Level. Thus, in File-Size-Aware-Merge-on-Scan, we avoid merging levels when the size of qualifying data on a level for a RQ is less than half of the file size ($file\_size/2$). 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-RQ-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 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.

Limitation 1: High Read and Write Amplification. Despite improving the RQ latency, the overall file count, file fragmentation, and compaction debt, File-Size-Aware-Merge-on-Scan still performs $167\times$ more writes, and moves (writes and reads) 31% more data than RocksDB, as shown in Figures 5E and 5F.

Limitation 2: Compaction Trade-off for Short-RQs. Query-driven merging improves latency for future RQs, but its benefits vary considerably with selectivity. For short-RQs, fewer entries are read per level, and such small compactions yield limited long-term gain, while increasing WA significantly.

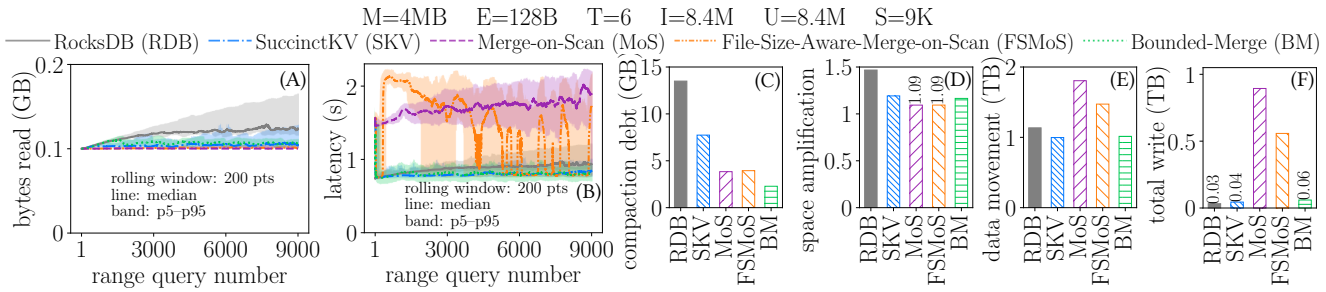

Fig. 5. (A) BM reads fewer bytes than RocksDB, but more bytes than FSMoS during RQs, as it compacts more strategically to reduce total writes and improve RQ performance. (B) Unlike FSMoS, BM initially takes a hit on RQs and improves latency using ρ. (C) FSMoS and BM offer better compaction debt, and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

However, short-RQs that fall within the range of a prior long-RQ can still benefit from query-driven compactions, since the associated write costs have already been paid by the long-RQs. We show this in our experiments, where we issue one long-RQ and a bunch of short-RQs from the same key range.

File-Size-Aware-Merge-on-Scan vs. SuccinctKV. SuccinctKV [71] aims to improve RQ latency in LSM-engines by triggering compactions during RQs. Compactions are triggered (i) on the cue of RQs, but (ii) **only if a level is saturated**. This limits Succinct's benefits only for workloads that insert many unique entries. More specifically, for update and delete-intensive workloads [11], SuccinctKV triggers very few compactions, limiting the benefits, as shown in Fig. 5C. Also, for query-intensive workloads, where compactions are infrequent, the benefits of SuccinctKV are curbed. We lay out the key design differences between RocksDB, SuccinctKV, and RangeReduce in Table II. Moreover, without a cost analysis for compactions, the benefits of SuccinctKV may be obliterated by the overheads. 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 WA, affecting overall performance. We address these by updating the compaction criteria and limiting the superfluous writes during compaction.

TABLE II
ROCKSDB VS SUCCINCTKV VS RANGEREDUCE.

Aspect	**RocksDB**	**SuccinctKV**	**RangeReduce**
RQ-compaction	none	limited	RQ-overlap-aware
Compaction trigger	level fullness	RQs, level fullness	RQs, level fullness
Prune invalid-entries	compactions	RQs, compactions	RQs, compactions
SST immutability	immutable	logically mutable	immutable
Compaction path	standard	complex	standard
RQ path	standard	complex	standard
Metadata overhead	low	high	low

B. 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 amount of data from Level i and y amount of data 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 further restricting compactions during RQs.

RangeReduceRatio (ρ). To this end, we introduce ρ, 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, e.g., if ρ 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

Algo. 1 LevelsSelectionInBounded-Merge

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

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 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 of ρ. To determine the optimal value of ρ, we performed a large-scale empirical analysis, as we measured 9 key performance metrics, as we varied ρ and the size ratio, as shown in Fig. 6. We observe that for a uniform workload, the optimal value of ρ lies around $1/T$, 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 $1/T$) of ρ minimizes SA and compaction debt, but performs significantly more writes overall, leading to slower RQs. On the other hand, larger values of ρ lead to almost no RQ-triggered compactions. A value closer to $1/T$ achieves total data movement comparable to RocksDB, with better performance.

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%, compared to RocksDB, 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 $\frac{N^{uniq.}\ s}{B} \cdot (1 + \frac{U \cdot (1-\beta)}{T}) + \phi \cdot \lambda$ and for a tiered LSM-tree as $\frac{N^{uniq.}\ s}{B} \cdot (1 + U \cdot (1-\beta) \cdot\ T) + \phi \cdot \lambda$.

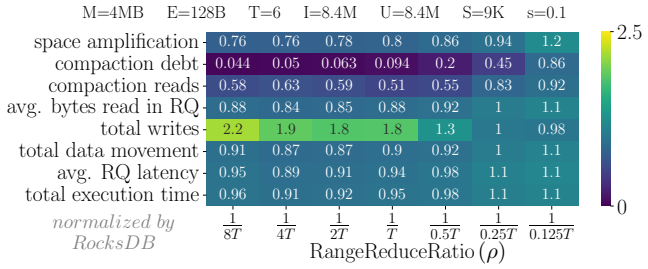

Fig. 6. With $\rho \approx 1/T$, we observe better performance for all metrics, except total writes, which is compensated by fewer reads during RQs, leading to less overall data movement.

(2) Compaction debt. The eager data rewriting frees up shallower levels and offers better compaction debt. Fig. 5C shows a significant reduction in compaction debt – 90% compared to RocksDB and 82% compared to SuccinctKV.

(3) Space amplification. Fig. 5D shows that BM reduces SA by 20% (2%) compared to RocksDB (SuccinctKV).

(4) Total writes. BM offers comparable total writes to that of the state of the art, as it judiciously performs merging based on ρ. In Fig. 5E, it shows only 20% more writes than RocksDB.

(5) Overall data movement. The slightly higher writes performed by Bounded-Merge is offset by reduced reads during RQs and future compactions. This accumulates to 10% less data movement overall than RocksDB (Fig. 5F).

Limitations of Bounded-Merge. In general, query-driven compactions push a large volume of data at the last level of the LSM-tree. As the last level saturates and the tree height increases, data is moved from the saturated levels to the empty ones. While a fraction of the files are moved trivially [53], this still causes a bump in RA and WA. Thus, to further reduce WA, we introduce Level-Renaming, which performs up to $T \cdot L$ fewer cascading compactions when a tree grows in height.

C. Level-Renaming

We propose Level-Renaming as an extension to the LSM compaction design space to reduce WA 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.

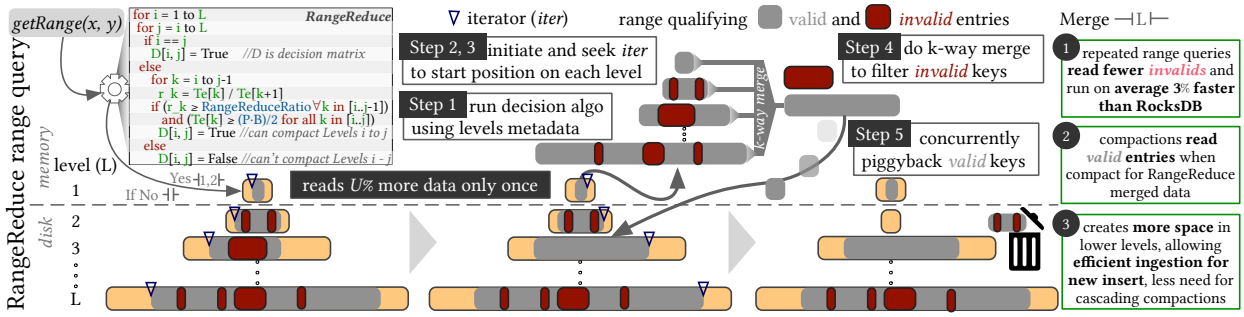

Fig. 7. The flow of a RQ in *RangeReduce*, compacts qualifying data from levels to improve the overall performance.

D. RangeReduce

RangeReduce brings all the proposed optimizations together. We integrate RangeReduce with RocksDB [26], a widely used, commercial key-value store, to evaluate its performance. RangeReduce operates within RocksDB's *main* thread to (i) first decide which levels could be compacted during the RQ (as shown in Fig. 7, Step 1) and then (ii) initializes and positions the iterators to the start position for the requested range in each level, (Fig. 7, Steps 2 and 3). Next, (iii) it reads the data to memory and (iv) performs a k-way merge operation to filter out the valid entries (Fig. 7, Step 4), while simultaneously (v) writing back the merge result using RocksDB's *flush* thread (Fig. 7, Step 5). At the end of each RQ, it marks the files that have been merged into deeper levels for garbage collection. RangeReduce keeps the engineering and operational overhead minimal by introducing only a single knob, the *RangeReduceRatio* (ρ). A default value of $\rho = 1/T$ provides robust performance across a wide range of workloads, see Fig. 6, eliminating the need for workload-specific tuning.

SLO-Bounded Compactions. RangeReduce is easily configurable to comply with application-specific service-level objectives (SLOs). As RQ-driven compactions can increase the tail latency of range queries, RangeReduce first estimates the end-to-end query completion time and suppresses compaction if the estimate violates any SLO. Assuming the storage engine is aware of the effective I/O bandwidth of the system, denoted as $\mathcal{B}^r$ for reads and $\mathcal{B}^w$, before performing a RQ, RangeReduce estimates the number of bytes to be read from the device as $D_{\text{rq}} = N^{unq} \cdot s \cdot E$) using in-memory metadata. Note that this data is read only once and shared between query execution and compaction. Additional I/O may incur when rewriting data during compaction (D_{cmp}); we conservatively bound this overhead by the scan size itself, i.e., $D_{\text{cmp}} \leq D_{\text{rq}}$. Therefore, the estimated latency of a range query that triggers compaction is bounded by $\hat{L}_{\text{rq}} \leq \frac{D_{\text{rq}}+D_{\text{cmp}}}{max(\mathcal{B}^r,\mathcal{B}^w)} \leq \frac{2D_{\text{rq}}}{max(\mathcal{B}^r,\mathcal{B}^w)}$. If $\hat{L}_{\text{rq}}$ exceeds the SLO requirements, RangeReduce disables query-driven compaction for that query.

V. Evaluation

We now compare RangeReduce against (i) RocksDB, which is an open source commercial LSM-engine, and (ii) SuccinctKV, an academic prototype built on top of RocksDB that performs compactions on the cue of RQs. We measure (i) *RQ performance* (latency and bytes processed per query), (ii) *SA*, (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. How much future compaction debt and SA is avoided by RangeReduce? To what extent does this impact the overall data movement and throughput?

(2) Approaching the Ideal LSM Shape. How RangeReduce is shaping the LSM-trees towards the ideal? To what extent does it reduce compaction overhead, SA, and improve RQ latency?

(3) A Better Ingestion-Compaction Tradeoff. How much does RangeReduce improve ingestion? To what extent does the selectivity of RQs play a role in this benefit?

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 [38]. We use KVBench [72] and Tectonic [45] to generate the workloads, and we integrate RangeReduce with RocksDB 8.5.0 [26].

Default Setup. Unless otherwise mentioned, we ingest 1GB data with unique keys, followed by interleaved 1GB of updates and 9K RQs. The average key-value pair is 128B (key is 16B and value is 112B) long. The average range query selectivity is 0.1. The LSM-buffer of 4MB large, and the tree has a size ratio of 6. We compare the performance with both (i) *uniformly-randomly distributed* RQs and (ii) *overlapping* RQs.

A. Holistic Improvements with RangeReduce

We perform two sets of experiments: (i) with fully random RQs where we evaluate RangeReduce for uniform workloads, (ii) with overlapping RQs in which every 100 RQs are identical to test against more adversarial workloads.

RangeReduce Reduces Work of Future Compactions. In Fig. 8A, we compare the compaction debt of RangeReduce with both RocksDB and SuccinctKV as we increase the RQ count. RangeReduce outperforms both baselines, with as few as 32 RQs, as it (i) leverages the merged result of RQs to eliminate logically invalid entries from the LSM-tree and (ii) frees up the shallower levels. In contrast, RocksDB accumulates this debt and performs lazy compaction on a need basis, resulting in high read and write amplifications, increased CPU utilization, and a larger database size. SuccinctKV performs

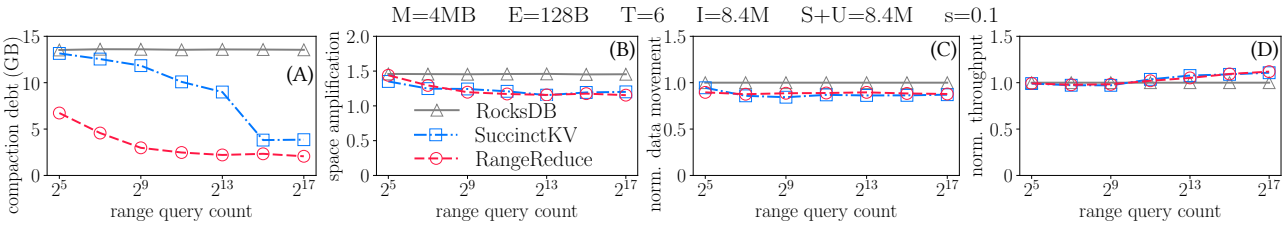

Fig. 8. (A) RangeReduce significantly reduces compaction debt when all RQs are uniformly random. It exhibits (B) less space amplification, (C) better normalized data movement, and higher (D) normalized throughput than RocksDB.

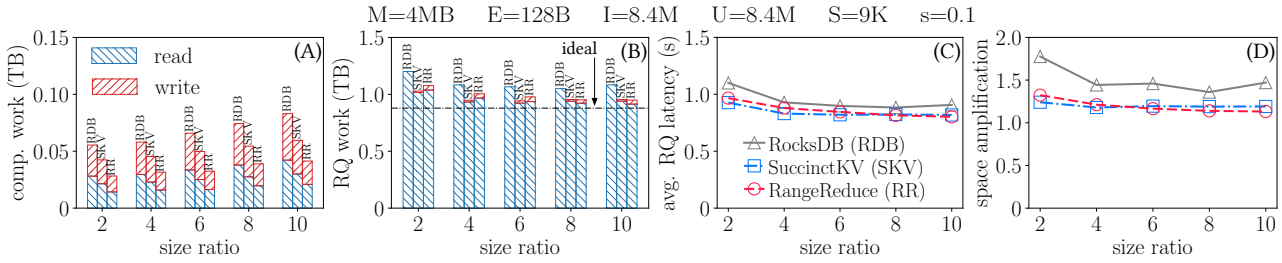

Fig. 9. RangeReduce reduces the need for compactions even when the RQs are random (A). They perform less work during RQ (C), which offers better RQ-latency (B) and near-ideal space amplification (D) than RocksDB.

a few compactions during RQs but does not fully exploit all RQs because of its level saturation-based trigger for RQ-driven compactions. As the RQ count increases, RangeReduce shows up to up to 85% less compaction debt compared to SuccinctKV, eliminating the need for future compactions.

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

Offers Better Throughput with Less Data Movement. In Fig. 8C, we compare overall data movement during compactions and RQs. RangeReduce performs comparable to SuccinctKV but shows 12% less data movement than RocksDB. In Fig. 8D, RangeReduce shows better throughput than RocksDB, whereas SuccinctKV throughput drops for 2^{17} RQs due to compaction triggered based on level saturation.

B. Using RQs to Approach the Ideal LSM Shape

Taming the Need for Compactions. In Fig. 9A, we compare RangeReduce with RocksDB and SuccinctKV as we vary the size ratio from 2 to 10. RangeReduce shows up to 50% less data movement than RocksDB and up to 25% less than SuccinctKV for compactions in merging data between levels. This is because RangeReduce performs compactions (i) when the level reaches saturation and, also, (ii) strategically merges the RQ results without requiring them to be read separately, resulting in less reading. This also allows RangeReduce to reduce the space amplification (Fig. 9D), pushing it toward its theoretical lower bound.

Reducing Latency and Read Overhead of RQs. In Fig. 9B and 9C, we compare the work done by RQs in terms of total bytes read and average latency of RQs. The ideal line – in Fig. 9B – 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 reduce the read overhead during RQs from 10% to 15% for size ratios of 2 to 10, respectively. RangeReduce shows fewer bytes transferred overall than RocksDB when reading from and writing to slow storage, as RocksDB never uses the work done during a RQ. SuccinctKV reads a comparable number of bytes to RangeReduce during RQs but pays the cost in terms of reads and writes during compactions.

C. A Better Ingestion-Compaction Tradeoff

Outperforms SuccinctKV for Update-intensive Workloads. We now ingest 4.2M inserts in the first epoch (E1), then issue 420K updates followed by 100 RQs with selectivity 0.1 in the second epoch (E2). We repeat E2 5 times to emulate an update-intensive workload. We observe that RangeReduce reads on average 16.9% fewer bytes than RocksDB and 12.4% less than SuccinctKV. RangeReduce runs RQs about 5.4% faster than RocksDB and 5.9% faster than SuccinctKV (Fig. 10A and B), while reducing the compaction debt (Fig. 10(C-D)) by 85% and 82% compared to SuccinctKV and RocksDB, respectively. We also see 19% improvement in SA and 9% less overall data movement over SuccinctKV (Fig. 10 (D-E)). RangeReduce outperforms SuccinctKV across the board, as the latter does not trigger compactions unless the levels are saturated.

Operating under Adversarial Workloads. Next, we evaluate RangeReduce under adversarial workloads.

Experimental setup. In this experiment, we vary the entry size between 32B and 1040B. The workload is executed over nine epochs, E1 – E9. E1 preloads the database with 1M unique inserts, and E2 ingests another 1M inserts interleaved with

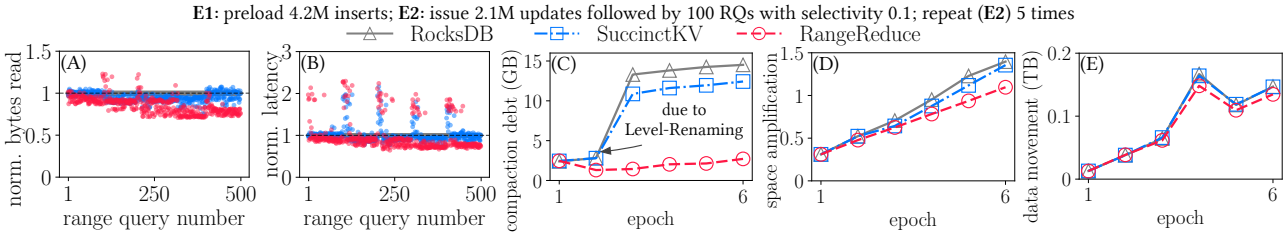

Fig. 10. RangeReduce improves overall system performance over time in terms of RQ read amplification (A), RQ latency (B), significantly reduces compaction debt (C), space amplification (D), and improves overall data movement (E).

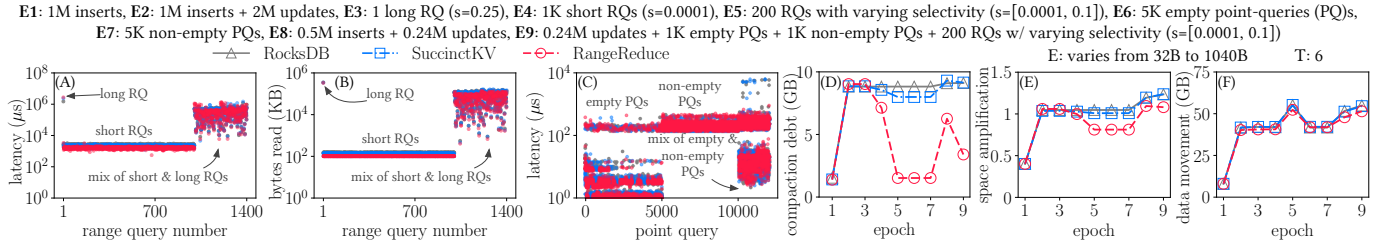

Fig. 11. RangeReduce benefits short RQs that overlap with prior long RQ, improving mix RQs performance (A, B) without adding overhead for operations like empty and non-empty PQs (C). It improves compaction debt (D), reduces SA (E), and shows less data movement (F) compared to RocksDB and SuccinctKV.

2M updates. In E3, we issue a single long RQ with selectivity 0.25, followed by 1K overlapping short RQs in E4. E5 then issues 200 random RQs with mixed selectivity ranging from 0.0001 to 0.1. E6 and E7 issue 5K empty and 5K non-empty point queries, respectively. In E8, we perform another 0.5M inserts interleaved with 0.24M updates. Finally, in E9, we run a heavily interleaved workload with updates (0.24M), empty point queries (1K), non-empty point queries (1K), and RQs (200) with selectivity, ranging from 0.0001 to 0.1.

During this nine-phase workload execution, RangeReduce leverages the RQ-driven compactions to (i) improve latency of future RQs, (ii) reduce SA and (iii) compaction debt, with (iv) comparable overall data movement. Fig. 11A shows that RangeReduce executes short RQs 35% (24%) faster than RocksDB (SuccinctKV), reading 30% (20%) less data compared to RocksDB (SuccinctKV). When running random RQs with varying selectivity, RangeReduce shows 5% latency improvement over SuccinctKV reading 17% less data. Fig. 11C shows that RangeReduce performs comparably to the baselines when realizing point queries (PQs), either sequentially (E6 and E7) or as interleaved with other operations (E9). Overall, RangeReduce reduces compaction debt (Fig. 11D) by up to 62% compared to both systems and improve SA (Fig. 11E) by 12.4% (12%) over RocksDB (SuccinctKV). RangeReduce does this by reducing the data movement (Fig. 11F) by 7% (6%) compared to RocksDB (SuccinctKV).

Operating at Scale. In Fig. 12, we show that as the database size increases, the benefits of RangeReduce hold, and for certain metrics, such as RQ latency and overall data movement, become more pronounced. As the database size grows from 1GB to 8GB, RangeReduce reduces compaction debt (A) by up to an order of magnitude and (B) SA ∼95% compared to both RocksDB and SuccinctKV. Purging the data also improves the mean RQ latency by 23.3% and 17.3% and overall data movement by up to 13.8% and 10.5% compared to RocksDB and SuccinctKV, respectively.

RangeReduce is Better for Long RQs. In Fig. 13(A-C), we compare the compaction debt, SA, and normalized RQ throughput while varying the selectivity of 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. 13C, for selectivity more than 0.03, the normalized throughput of RQs is always higher than that of RocksDB.

Improved Performance for YCSB Workloads. We run an experiment with a RQ selectivity close to the YCSB-E benchmark standard 0.00005% (reads ∼100 entries), recording the latency of each RQ and the total bytes read (Fig. 14A-B). We preload the database with 2.1M unique inserts interleaved with 2.1M updates. Then, we issue 95K range queries with selectivity 0.00005 interleaved with 5K updates. For the initial RQs, RangeReduce shows a spike in latency, but the performance improves for subsequent queries. RangeReduce reads fewer bytes per RQ (6.8% less than RocksDB and 7.4% less than SuccinctKV) and achieves lower latency than RocksDB (up to 18%) while slightly outperforming SuccinctKV (up to 1.1%); the compaction debt (Fig. 14C), SA (Fig. 14D) show 20.2% improvement over both RocksDB and SuccinctKV, respectively. SuccinctKV performs similarly to RangeReduce, but has a significantly higher compaction debt.

RangeReduce Cuts Cloud Costs. We quantify the monetary impact of RangeReduce by quantifying its benefits. Using a 10TB dataset and representative AWS, GCP, and Azure configurations, we project total storage and compaction-related costs over a four-year horizon. In our experiments, Ran-

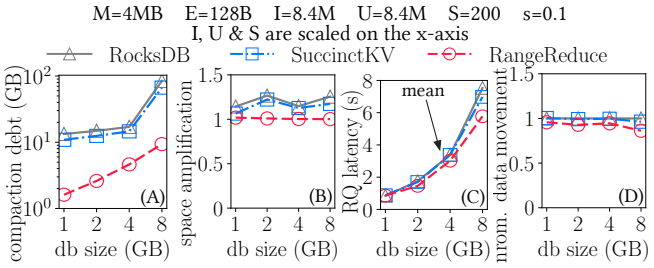

Fig. 12. RangeReduce has more benefits with more data.

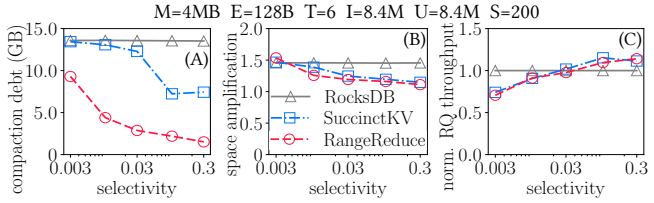

Fig. 13. RangeReduce uses long RQs to improve (A) compaction debt, (B) space amplification, and (C) mean RQ latency.

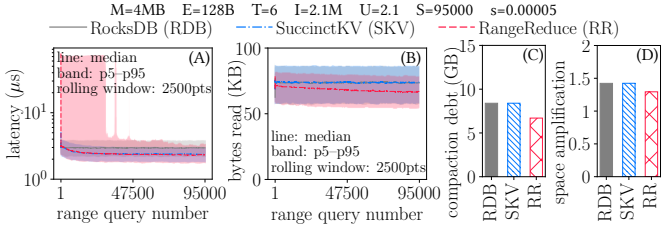

Fig. 14. RangeReduce offers better RQ performance while marginally improving compaction debt and space amplification for YCSB-E.

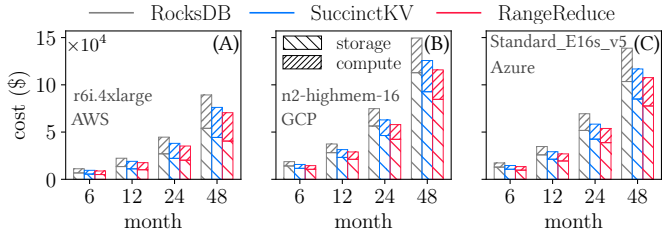

Fig. 15. RangeReduce cuts down cloud costs close to half.

geReduce achieves near-ideal space amplification and reduces overall data movement by 12%, which directly lowers both storage and compute costs. As a result, RangeReduce reduces total cloud expenditure by approximately 45% compared to RocksDB and 18% compared to SuccinctKV by year four (Fig. 15). These savings are consistent across cloud providers, as they stem from reduced hardware and storage requirements.

VI. Related Work

Improving LSM Performance. Past research on LSM-engines has focused on enhancing read and write performance [16, 17, 19, 35, 64], explored various compaction tunings [33, 37, 57, 68], and optimized RA, WA, and SA [18, 20, 56, 59]. While existing literature has focused on workload-aware point and range filters [22, 27, 46, 49, 51], fewer works have focused on optimizing the overall system performance in the presence of RQ-intensive workloads.

Query-Aware Data Reorganization. Past research on database cracking [36], update-aware bitmap indexing (UpBit [8]), hybrid key-value stores [48], out-of-place update management [4, 5], radix-based adaptive indexing (Adaptive Adaptive Indexing [62]), and other adaptive indexes [29, 30, 31, 32, 50, 66] 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 predicate-based cracking to avoid overfitting and uses radix-based partitioning to reorganize, offering a higher partitioning throughput. In the NoSQL realm, TellStore [48] uses scans as hints during garbage collection, and SuccinctKV [71] introduces a scan-based compaction mechanism that aims to reduce CPU overhead during RQs. Files that are fully contained within 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 RA and SA 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.

VII. Conclusion

In this work, we revisited the interplay between RQs and compactions, highlighting how state-of-the-art LSM-engines suffer from redundant reads, repetitive merges, and excessive compaction debt. We point out that LSM-engines fail to utilize the work done by RQs, i.e., merging and filtering invalid entries for each RQ logically, wasting expensive I/Os. We introduce RangeReduce, an RQ-aware LSM-engine that removes invalid entries while reaping from the efforts of the RQ. RangeReduce reduces compaction debt by freeing up shallower levels, which in turn offers richer SA and RA, better CPU resource utilization, and improved overall system performance in the presence of updates and RQs in a workload.

VIII. Acknowledgements

This work is partially funded by the National Science Foundation under Grant No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift.

REFERENCES

[1] Apache, "Cassandra," *http://cassandra.apache.org*, 2023.

[2] Apache, "HBase," *http://hbase.apache.org/*, 2023.

[3] Apple, "FoundationDB," *https://github.com/apple/foundationdb*, 2018.

[4] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica, "MaSM: Efficient Online Updates in Data Warehouses," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2011, pp. 865–876. [Online]. Available: http://dl.acm.org/citation.cfm?id=1989323.1989414

[5] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica, "Online Updates on Data Warehouses via Judicious Use of Solid-State Storage," *ACM Transactions on Database Systems (TODS)*, vol. 40, no. 1, 2015.

[6] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan, "Designing Access Methods: The RUM Conjecture," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2016, pp. 461–466. [Online]. Available: http://dx.doi.org/10.5441/002/edbt.2016.42

[7] M. Athanassoulis, S. Sarkar, T. I. Papon, Z. Zhu, and D. Staratzis, "Building Deletion-Compliant Data Systems," *IEEE Data Engineering Bulletin*, vol. 45, no. 1, pp. 21–36, 2022.

[8] M. Athanassoulis, Z. Yan, and S. Idreos, "UpBit: Scalable In-Memory Updatable Bitmap Indexing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016. [Online]. Available: https://dl.acm.org/citation.cfm?id=2915964

[9] BigTable, "Bigtable pricing," *https://cloud.google.com/bigtable/pricing*, 2025.

[10] M. Callaghan, "RocksDB internals: bytes pending compaction," 2022. [Online]. Available: https://smalldatum.blogspot.com/2022/01/rocksdb-internals-bytes-pending.html

[11] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 209–223.

[12] 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," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 205–218. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323

[13] CockroachDB, "Pebble," 2024. [Online]. Available: https://github.com/cockroachdb/pebble

[14] CockroachDB, "CockroachDB Pricing - The cloud-native, distributed SQL database," *https://www.cockroachlabs.com/pricing/*, 2025.

[15] A. Conway, A. Gupta, V. Chidambaram, M. Farach-Colton, R. P. Spillane, A. Tai, and R. Johnson, "SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020, pp. 49–63. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/conway

[16] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal Navigable Key-Value Store," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2017, pp. 79–94. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064054

[17] N. Dayan, M. Athanassoulis, and S. Idreos, "Optimal Bloom Filters and Adaptive Merging for LSM-Trees," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 4, pp. 16:1–16:48, 2018. [Online]. Available: https://doi.org/10.1145/3276980

[18] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2018, pp. 505–520. [Online]. Available: http://doi.acm.org/10.1145/3183713.3196927

[19] N. Dayan and S. Idreos, "The Log-Structured Merge-Bush & the Wacky Continuum," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2019, pp. 449–466.

[20] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto, "Spooky: Granulating LSM-Tree Compactions Correctly," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 3071–3084, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3071-dayan.pdf

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1323293.1294281

[22] P. C. Dillinger and S. Walzer, "Ribbon filter: practically smaller than Bloom and Xor," *CoRR*, vol. 2103.02515, 2021. [Online]. Available: https://arxiv.org/abs/2103.02515

[23] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing Space Amplification in RocksDB," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[24] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications," *ACM Transactions on Storage (TOS)*, vol. 17, no. 4, pp. 26:1–26:32, 2021. [Online]. Available: https://doi.org/10.1145/3483840

[25] Facebook, "MyRocks," *http://myrocks.io/*, 2023.

[26] Facebook, "RocksDB," *https://github.com/facebook/rocksdb*, 2024.

[27] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*, 2014, pp. 75–88. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674994

[28] Google, "LevelDB," *https://github.com/google/leveldb/*, 2021.

[29] G. Graefe and H. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2010, pp. 371–381. [Online]. Available: http://dl.acm.org/citation.cfm?id=1739041.1739087

[30] G. Graefe and H. A. Kuno, "Adaptive indexing for relational keys," in *Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW)*, 2010, pp. 69–74.

[31] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, "Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores." *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 502–513, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p502_felixhalim_vldb2012.pdf

[32] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt, "Progressive Indexes: Indexing for Interactive Data Analysis," *Proceedings of the VLDB Endowment*, vol. 12, no. 13, pp. 2366–2378, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2366-holanda.pdf

[33] A. Huynh, H. A. Chaudhari, E. Terzi, and M. Athanassoulis, "Towards flexibility and robustness of LSM trees," *The VLDB Journal*, pp. 1–24, 2024. [Online]. Available: https://doi.org/10.1007/s00778-023-00826-9

[34] S. Idreos and M. Callaghan, "Key-Value Storage Engines," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2667–2672.

[35] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu, "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)*, 2019. [Online]. Available: https://www.cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf

[36] S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007. [Online]. Available: https://www.cidrdb.org/cidr2007/papers/cidr07p07.pdf

[37] S. Kaushik and S. Sarkar, "Anatomy of the LSM Memory Buffer: Insights & Implications," in *Proccedings of the International Workshop on Testing Database Systems (DBTest)*, 2024, pp. 23–29. [Online]. Available: https://doi.org/10.1145/3662165.3662766

[38] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the Chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, Jul. 2020.

[39] C. Luo, "Breaking Down Memory Walls in LSM-based Storage Systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, jun 2020, pp. 2817–2819. [Online]. Available: https://dl.acm.org/doi/10.1145/3318464.3384399

[40] C. Luo and M. J. Carey, "LSM-based Storage Techniques: A Survey," *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020. [Online]. Available: https://link.springer.com/article/10.1007%2Fs00778-019-00555-y

[41] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2071–2086.

[42] D. Mo, F. Chen, S. Luo, and C. Shan, "Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads," *CoRR*, vol. abs/2308.0, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.07013

[43] NetApp Instaclustr, "Instaclustr platform pricing," 2025. [Online]. Available: https://www.instaclustr.com/pricing/

[44] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[45] A. H. Ott, S. Kaushik, B. Chen, and S. Sarkar, "Tectonic: Bridging Synthetic and Real-World Workloads for Key-Value Benchmarking," 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:282201482

[46] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson, "Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2021, pp. 1386–1399. [Online]. Available: https://doi.org/10.1145/3448016.3452841

[47] P. Pandey, M. Farach-Colton, N. Dayan, and H. Zhang, "Beyond Bloom: A Tutorial on Future Feature-Rich Filters," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2024, pp. 636–644. [Online]. Available: https://doi.org/10.1145/3626246.3654681

[48] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann, "Fast Scans on Key-Value Stores," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1526–1537, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1526-bocksrocker.pdf

[49] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-, and space-efficient bloom filters," *ACM Journal of Experimental Algorithmics*, vol. 14, 2009.

[50] A. Raman, S. Sarkar, M. Olma, and M. Athanassoulis, "Indexing for Near-Sorted Data," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2023, pp. 1475–1488.

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

[52] RocksDB, "Universal Compaction," *https://github.com/facebook/rocksdb/wiki/Universal-Compaction*, 2020.

[53] RocksDB, "RocksDB Trivial Move," *https://github.com/facebook/rocksdb/wiki/Compaction-Trivial-Move*, 2022.

[54] RocksDB, "RocksDB-Cloud: A Key-Value Store for Cloud Applications," *https://github.com/rockset/rocksdb-cloud*, 2025.

[55] Rockset, "Rockset," *https://rockset.com*, 2024.

[56] S. Sarkar and M. Athanassoulis, "Dissecting, Designing, and Optimizing LSM-based Data Stores," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2022, pp. 2489–2497. [Online]. Available: https://www.youtube.com/watch?v=hkMkBZn2mGs

[57] S. Sarkar, K. Chen, Z. Zhu, and M. Athanassoulis, "Compactionary: A Dictionary for LSM Compactions," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2022, pp. 2429–2432.

[58] S. Sarkar, N. Dayan, and M. Athanassoulis, "The LSM Design Space and its Read Optimizations," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2023, pp. 3578–3684. [Online]. Available: https://doi.org/10.1109/ICDE55515.2023.00273

[59] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis, "Lethe: A Tunable Delete-Aware LSM Engine," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 893–908.

[60] S. Sarkar, T. I. Papon, D. Staratzis, Z. Zhu, and M. Athanassoulis, "Enabling Timely and Persistent Deletion in LSM-Engines," *ACM Transactions on Database Systems (TODS)*, vol. 48, no. 3, pp. 8:1–8:40, 2023. [Online]. Available: https://doi.org/10.1145/3599724

[61] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis, "Constructing and Analyzing the LSM Compaction Design Space," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2216–2229, 2021. [Online]. Available: http://vldb.org/pvldb/vol14/p2216-sarkar.pdf

[62] F. M. Schuhknecht, J. Dittrich, and L. Linden, "Adaptive Adaptive Indexing," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2018, pp. 665–676.

[63] ScyllaDB, "Online reference," 2024. [Online]. Available: https://www.scylladb.com/

[64] R. Sears and R. Ramakrishnan, "bLSM: A General Purpose Log Structured Merge Tree," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213862

[65] Speedb, "Online reference," *https://github.com/speedb-io/speedb*.

[66] E. M. Teixeira, P. R. P. Amora, and J. C. Machado, "MetisIDX - From Adaptive to Predictive Data Indexing," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2018, pp. 485–488. [Online]. Available: https://doi.org/10.5441/002/edbt.2018.53

[67] H. Wang, J. Qiu, F. Yuan, and H. Zhang, "Rethinking the compaction policies in lsm-trees," *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, pp. 1–26, 2025.

[68] R. Wei, Z. Zhu, A. Kryczka, J. Zhuang, and M. Athanassoulis, "Benchmarking, Analyzing, and Optimizing WA of Partial Compaction in RocksDB," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2025, pp. 425–437. [Online]. Available: https://doi.org/10.48786/edbt.2025.34

[69] WiredTiger, "Source Code," *https://github.com/wiredtiger/wiredtiger*, 2021.

[70] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2018, pp. 323–336.

[71] Y. Zhang, S. Yang, H. Hu, C. Yang, P. Cai, and X. Zhou, "SuccinctKV: a CPU-efficient LSM-tree Based KV Store with Scan-based Compaction," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 4, pp. 90:1–90:26, 2024. [Online]. Available: https://doi.org/10.1145/3695873

[72] Z. Zhu, A. Saha, M. Athanassoulis, and S. Sarkar, "KVBench: A Key-Value Benchmarking Suite," in *International Workshop on Testing Database Systems (DBTest)*, 2024, pp. 9–15. [Online]. Available: https://doi.org/10.1145/3662165.3662765

[73] Z. Zhu, S. Sarkar, and M. Athanassoulis, "Acheron: Persisting Tombstones in LSM Engines," in *Companion of the International Conference on Management of Data (SIGMOD)*, 2023, pp. 131–134.