Rethinking The Compaction Policies in LSM-trees

HENGRUI WANG, Tsinghua University, China
JIANSHENG QIU, Tsinghua University, China
FANGZHOU YUAN, Tsinghua University, China
HUANCHEN ZHANG*, Tsinghua University, China

Log-structured merge-trees (LSM-trees) are widely used to construct key-value stores. They periodically compact overlapping sorted runs to reduce the read amplification. Prior research on compaction policies has focused on the trade-off between write amplification (WA) and read amplification (RA). In this paper, we propose to treat the compaction operation in LSM-trees as a computational and I/O-bandwidth investment for improving the system's future query throughput, and thus rethink the compaction policy designs. A typical LSM-tree application handles a steady but moderate write stream and prioritizes resources for top-level flushes of small sorted runs to avoid data loss due to write stalls. The goal of the compaction policy, therefore, is to maintain an optimal number of sorted runs to maximize average query throughput. Because compaction and read operations compete for the CPU and I/O resources from the same pool, we must perform a joint optimization to determine the appropriate timing and aggressiveness of the compaction. We introduce a three-level model of an LSM-tree and propose EcoTune, an algorithm based on dynamic programming to find the optimal compaction policy according to workload characterizations. Our evaluation on RocksDB shows that EcoTune improves the average query throughput by 1.5× to 3× over the leveling policy and by up to 1.8× over the lazy-leveling policy on workloads with range/point query ratios.

CCS Concepts: • **Information systems → Data structures**.

Additional Key Words and Phrases: LSM-Trees, Concurrency Control, Optimization, Dynamic Programming

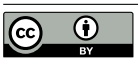
ACM Reference Format:
Hengrui Wang, Jiansheng Qiu, Fangzhou Yuan, and Huanchen Zhang. 2025. Rethinking The Compaction Policies in LSM-trees. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 207 (June 2025), 26 pages. https://doi.org/10.1145/3725344

1 Introduction

The Log-structured merge-tree (LSM-tree) is the foundation of many modern key-value stores [1–4, 6, 9, 11, 41, 43, 73]. An LSM-tree buffers inserted keys in memory and then flushes them to disk in batches as sorted runs. Each sorted run can contain multiple files, called SSTables. Because the key ranges of these sorted runs overlap, a query must probe each sorted run to find a particular key-value pair, causing significant read amplification. As more runs accumulate on disk, they are sort-merged to create fewer but larger sorted runs. This process is known as compaction. To further improve read performance, LSM-trees typically equip each SSTable with a filter data structure (i.e., point filter [20, 30, 34, 36, 39, 40, 42, 67, 89], range filter [29, 51, 61, 76, 78, 81, 87]) to reduce unnecessary I/Os. LSM-trees are used in a wide range of real-world applications [16, 31, 37, 44, 52, 62, 68, 70].

*Huanchen Zhang is also affiliated with the Shanghai Qi Zhi Institute. Corresponding author.

Authors' Contact Information: Hengrui Wang, wang-hr21@mails.tsinghua.edu.cn, Tsinghua University, China; Jiansheng Qiu, qjc21@mails.tsinghua.edu.cn, Tsinghua University, China; Fangzhou Yuan, yfz23@mails.tsinghua.edu.cn, Tsinghua University, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Tsinghua University, China.



ACM 2836-6573/2025/6-ART207
https://doi.org/10.1145/3725344

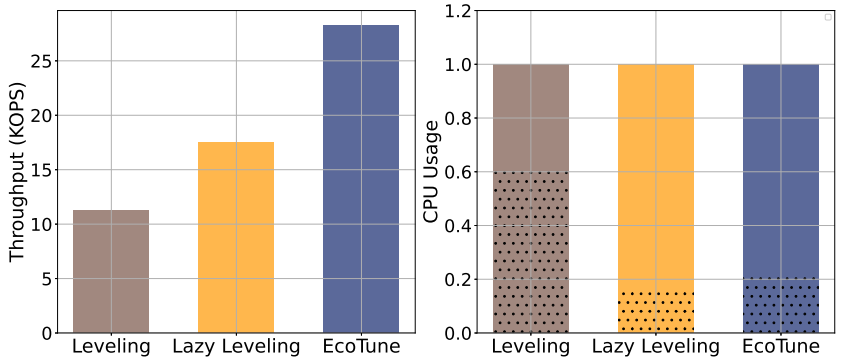


Fig. 1. Average throughput and compaction overhead of different compaction policies. The shadow part corresponds to the compaction's CPU usage.

Compaction is the key operation of the LSM-tree. Compacting aggressively leads to higher write amplification while reducing read amplification. Compacting lazily reduces write amplification but can hurt query performance. The classic compaction policy is the leveling policy. In the leveling policy, when a sorted run reaches a certain size capacity, it is merged with a larger sorted run. As a result, the sizes of different sorted runs increase exponentially. Each sorted run is referred to as a level, and the number of levels is determined by the size ratio (T) between adjacent levels. As the proportion of writes increases, the tiering policy was introduced to reduce write amplification. The tiering policy allows each level to contain $T - 1$ sorted runs. Researchers have explored more flexible compaction policies. Examples include hybrid leveling and tiering policies [32], different size ratios between levels [33], and allowing an arbitrary number of sorted runs in each level [56, 65]. Regardless of the compaction policy, the LSM-tree periodically undergoes a global compaction that merges all sorted runs into one. We define the time between two global compactions as a *compaction round*.

Prior works have primarily focused on the trade-off between write amplification (WA) and read amplification (RA), assuming that higher WA would lead to lower write performance. However, an LSM-tree application typically undergoes a steady and moderate write stream, and it only prioritizes the top-level flushes to avoid data loss due to write stalls. For example, Meta has demonstrated that the highest write speed in a real-world workload is around 45 MB/s [38]. Given that modern NVMe SSDs provide more than 2 GB/s of write bandwidth [5, 10], the bandwidth consumption of the prioritized I/Os is relatively small. We show experiments that the distribution of the remaining CPU and I/Os resources between compaction and query does not affect the LSM-tree's write performance. Therefore, the goal of a compaction policy is to optimize the query (read) performance, and it requires joint optimization because the compaction and read operations compete for the CPU and I/O resources from the same remaining pool.

Meanwhile, previous studies model query performance using the worst-case read amplification *right after* the compaction operations (we denote this as instantaneous query performance) [30, 32, 33, 56, 65]. However, the read amplification varies over time because the number of sorted runs keeps changing within a compaction round. The average query throughput is more important than instantaneous RA metrics for a long-running key-value service. Using the complexity upper bound for instantaneous RA to model read performance leads to sub-optimal compaction policy designs. While a compaction operation could immediately speed up certain queries, it may hurt average query performance because compactions are not free: they compete for resources against

user queries. Compactions could temporarily occupy CPU threads or saturate SSD bandwidth, thus preventing these resources from being used for queries. To show this, we evaluated different compaction policies on Optane SSD with a fixed write speed. The query workload consists of 35% Get, 35% Seek, and 30% long range scans. Figure 1 shows the average throughput and compaction overhead for each policy in a compaction round. Although the classic leveling policy leads to a smaller read amplification than lazy leveling [32, 33, 56, 65] in the traditional instantaneous query performance analysis, the query throughput of leveling is only 64% of that of lazy leveling in the experiment. This is because the compaction operations in leveling consumed more than half (62%) of the CPU resources that could have been used for queries.

We propose to rethink the compaction policy design from the perspective of optimizing the system's average query throughput. We model the problem as follows: given a steady stream of LSM-tree writes (at a constant flush speed), how can we design a compaction policy within a compaction round to maximize the average query throughput? The essence of compaction is investing computations and I/Os to reduce the number of sorted runs, thereby enabling future queries to probe fewer sorted runs. However, this effect is temporal, as new sorted runs are continuously generated. The future gain of a compaction depends on both its impact on instantaneous query throughput and the duration of that impact.

We find that the timing of compaction is crucial in determining the duration of its effects. Our insight is that the earlier a compaction is performed before the next global compaction, the greater the subsequent gain in average query throughput. This is because an earlier compaction can reduce query overhead over a longer period, substantially increasing the number of future queries benefiting from the compaction. Therefore, an ideal design should employ different compaction policies at different times. This implies that sorted runs at the same physical level should have different sizes because they are created at different times. Consequently, the concept of the physical level becomes vague, as the LSM-tree no longer has groups of equal-sized sorted runs.

We propose to view an LSM-tree using a three-level model when pursuing an optimal compaction policy. We then introduce EcoTune, an algorithm based on dynamic programming that can find the optimal compaction policy quickly for a given query workload and write speed. We evaluate the performance of EcoTune in RocksDB. We compare EcoTune to the Leveling and Lazy Leveling compaction policies. Compared with the Leveling policy, our EcoTune achieves 1.5x to 3x average query throughput under different workloads and up to five orders of magnitude lower latency. Compared with the Lazy Leveling policy, our EcoTune achieves up to 2.5x average query throughput under different workloads and up to four orders of magnitude lower latency.

EcoTune is the first work that focuses on designing compaction policies to optimize LSM-trees' average query throughput. We make three primary contributions in this paper. First, we propose to treat compaction in LSM-trees as a resource investment for improving the system's average query throughput. Second, we introduce a three-level model for LSM-trees conceptually and design a dynamic programming algorithm EcoTune to find optimal compaction policies for different LSM-tree instances. Finally, we integrate our EcoTune compaction policies into RocksDB. Experiments show that EcoTune outperforms other solutions in improving RocksDB's average query performance across different workloads.

2 Background & Related Work

LSM-tree Overview. The LSM-tree maintains multiple sorted runs to store data, organized into multiple levels with exponentially increasing capacities. The capacity of these levels is controlled by a size ratio, often set to a fixed integer T, which represents the capacity ratio between two consecutive levels. The LSM-tree also maintains in-memory tables as write buffers. When an in-memory table becomes full, a background thread flushes it to level 0 in the storage. Each

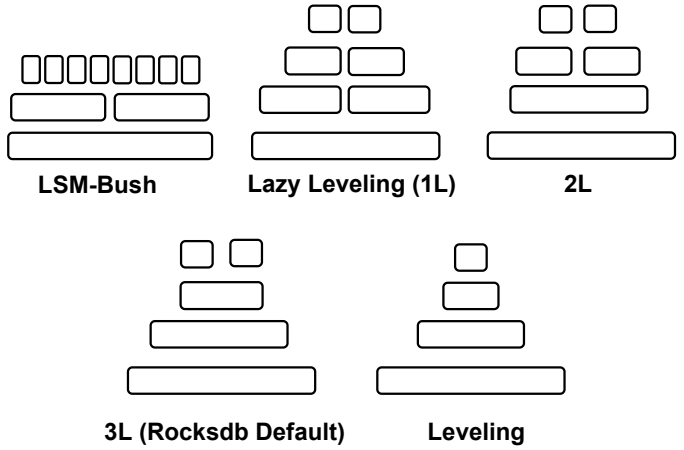


Fig. 2. An overview of different compaction policies.

Term	Definition
R	the number of *TM* compactions in a compaction round
T	LSM-tree size ratio (if fixed)
K	Long Range Scan length
s	the number of small sorted runs
S	the size of the small level
N	total data size quantified by the number of entries
M	the size of the whole LSM-tree
L	total level number
C	the capped size ratio between the main and the large levels
T_w	time between two consecutive *TM* compactions
T_c	time to rewrite S data on SSD with *MLC* threads
c	# incoming large sorted runs before global compaction
e	existing number of large sorted runs

Table 1. Terms in this paper.

level is compacted to the next level when it reaches its capacity. The compaction process can be triggered by different conditions and has varying compaction granularity. According to [72], different compaction methods have different trade-offs. The LSM-tree has numerous variants that optimize its various aspects [17, 23, 24, 32, 33, 45, 57, 66, 75]. Examples include optimizing compaction policies [24, 25, 32, 33, 45, 46, 69], developing update-friendly compaction schemes [15, 35, 71, 82, 84, 85], adopting unconventional hardware [14, 75, 77, 79, 88], narrowing the gap between LSM-trees and update-in-place designs [86], separating keys and values [23, 58], keeping hot entries in the buffer with selective flushing [17], reducing tail latency [18, 59], improving memory management mechanisms [21, 50, 60], and exploiting data characteristics [13].

Compaction Policies. When a level reaches its compaction conditions, the LSM-tree uses background threads to compact the data in that level into the next one. There are mainly two types of compaction policies for the LSM-tree: the leveling policy, which greedily sorts all data in a level, and the tiering policy, which allows each level to have multiple sorted runs. These two policies balance write amplification and read amplification. The leveling policy allows each level to contain only a single sorted run, with each level's capacity being T times that of the level above it. When a level reaches its capacity, it is merged with the sorted run in the next level. During this process, all data in these two levels is rewritten, resulting in high write amplification. Modern LSM-trees often use a hybrid policy, employing the tiering policy in upper levels and the leveling policy in lower levels [4, 32, 33]. A representative hybrid policy is the lazy leveling policy, which allows only one sorted run in the last level and up to $T - 1$ sorted runs in non-last levels. When a non-last level reaches T sorted runs, all sorted runs in that level are compacted together to create a new sorted run in the next level. If the next level is not the last level, the compaction process does not access the data in the next level. As a result, the compaction overhead is lower compared to the leveling policy. We illustrate these hybrid policies in Figure 2.

Filters. LSM-trees leverage filters to reduce I/O operations for queries. Various filter data structures [20, 36, 39, 40, 42, 51, 53, 61, 76, 87] are widely employed. Filters can quickly determine the presence of keys within a dataset, though they introduce a false positive rate (FPR). In an LSM-tree, each SSTable builds a Bloom filter to avoid unnecessary I/O. To further optimize range queries, range filters have been proposed [27, 29, 51, 61, 76, 81, 87]. To reduce the overhead of probing filters, some filters [28, 34, 78] are built globally to avoid probing each sorted run's filter individually.

LSM-tree Structure Tuning. Recent studies on LSM-tree structure tuning focus on tuning the compaction policy. Dostoevsky [32] explores the trade-off between tiering and leveling compaction. Later, LSM-bush [33] introduces a more flexible structure by using different size ratios between adjacent levels, significantly enhancing write performance. MOOSE [56] explores a more flexible configuration space in LSM-tree designs, allowing distinct configurations to be assigned to individual levels, including size ratio, number of runs, and Bloom filter bits. RUSKEY [65] makes the first attempt at designing reinforcement learning models to optimize the performance of LSM-tree-based key-value systems. Endure [45] attempts to find a robust structure for LSM-trees when the workload is uncertain. Although these works use different methods, their high-level designs are similar. They all try to minimize the average cost of the mixed read-write workload. Such designs imply the assumption that read and write operations are serialized and equally treat compaction and flush. However, on current high-bandwidth SSDs, read and write operations can be parallelized. We also need to consider the higher priority of flush. Finally, since these designs assume serialized read and write operations, they do not account for how query speed changes during the write process.

3 Rethinking the Compaction Design

In this section, we propose a new perspective on the compaction policy. For higher query performance, we hope the LSM-tree to have as few sorted runs as possible. Unfortunately, keeping the number of sorted runs moderate is not free. Compacting multiple sorted runs together consumes a lot of CPU and I/O resources. Traditionally, compaction has been considered to degrade the write performance and improve query performance. We argue that modern SSDs have changed this situation. We should rethink the impact of compaction on both write and query performance.

3.1 Impact on Write Performance

There was a common belief that higher write amplification degrades write performance. However, this argument was based on earlier HDDs. Flush and compactions could not be parallelized on HDDs, meaning that compaction would inevitably block flush. Therefore, it was important to finish

Type	Optane SSD			NVMe SSD		
Latency	Average / 99th (μs)			Average / 99th (μs)		
# Cores	4	8	16	4	8	16
1L	2.8/4.1	2.8/4.2	2.9/4.4	3.3/4.3	3.4/4.3	3.4/4.5
2L	2.8/4.2	2.8/3.9	2.9/4.3	3.4/4.2	3.4/4.5	3.5/4.8
3L	2.9/4.3	2.8/4.3	2.9/4.3	3.4/4.1	3.5/4.4	3.5/4.4

Table 2. Write Latency (μs) on Different SSDs.

compactions as soon as possible, before the in-memory table became full. Otherwise, writes would be stalled, and some incoming data might be lost (especially if the write workload is a one-pass data stream). To reduce such write stalls on HDDs, the write amplification caused by compaction needed to be as low as possible.

Modern NVMe SSDs, with much higher write bandwidth [26], have changed this situation. Flush and compactions can now be easily parallelized on modern SSDs. Meta has shown [38] that the write throughput of MyRocks [64], ZippyDB [12], UDB [16], and UP2X [22] is typically not high. The highest write speed in a real workload is around 45 MB/s [38]. Similarly, IoTDB reported that its real workload's highest write throughput is only about 500 KB/s [48]. These moderate write streams are far from saturating modern NVMe SSDs' bandwidth (which exceeds 2 GB/s). As discussed above, flush should have the highest priority [18, 55]. *We should always reserve enough bandwidth and CPUs for flush, using the remaining resources for both compactions and queries. On the other hand, once flush stalls are avoided, we can freely design the compaction policy for higher query performance without worrying about write performance.* Modern SSDs provide a very large design space for compaction, allowing approximately 50 WA for the highest write speed reported by Meta.

We conducted simple experiments to support our argument. We evaluated the average write performance of a 4-level LSM-tree with different compaction policies. In our experiments, we varied the number of leveling levels, as shown in Figure 2. 1L means only the last level uses the leveling policy, and the other levels use the tiering policy, corresponding to the lazy leveling compaction policy. 2L means the last two levels use the leveling policy, and other levels use the tiering policy. 3L means only the first level uses the tiering policy, while other levels use the leveling policy, which is the default compaction policy of RocksDB and LevelDB. A compaction is triggered when the number of sorted runs at level 0 reaches 10. We set the write throughput to 100 MB/s, which is much higher than Meta's reported maximum throughput. We measured the average and P99 tail write latency of different compaction policies on two different SSDs (800GB Intel Optane SSD DC P5800X and 3TB D7-P5620 NVMe SSD).

We aim to support two arguments: (1) The write speed (flush) can be guaranteed by reserving enough bandwidth and CPUs. (2) The remaining bandwidth and CPUs can be freely used to support a wide range of compaction policies without affecting the write speed. We used different numbers of CPUs (4C, 8C, 16C) to conduct compactions under different policies. The number of CPUs controls the bandwidth and parallelism available for compaction. The results are shown in Table 2, we observe that on both SSDs, the average and tail latencies are not impacted by either the compaction policy or the number of CPUs. These results support our arguments. *In the rest of our paper, we assume that the LSM-tree has a constant write speed that is not influenced by the compaction policy.* As a result, the design of the compaction policy should focus on how to use the remaining bandwidth and CPUs to achieve the highest query performance. This simplifies the design compared to the traditional WA-RA perspective.

3.2 Impact on Query Performance

As discussed above, the compaction policy should focus on how to use the remaining bandwidth and CPUs to achieve the highest query performance. We should rethink how compaction influences query performance. Previous works [30, 32, 33, 45, 56] typically measure query performance by looking at the worst-case RA complexity (i.e., when the LSM-tree is just prior to a global compaction). In these traditional measurements, compaction always appears to benefit query performance. However, *this measurement is unsuitable because the number of sorted runs in each level is always dynamically changing*. Although the upper bound of RA complexity always holds, the concrete query performance is dynamically changing. We refer to this kind of temporal query performance as instantaneous query performance. These states do not last for long. We argue that the average query performance provides a better representation of the LSM-tree's overall query performance. *Unlike instantaneous performance, compaction's impact on average query performance becomes double-edged.* In the following sections, we first formulate the average query performance. Then, we explain why compaction could also negatively impact the average query performance.

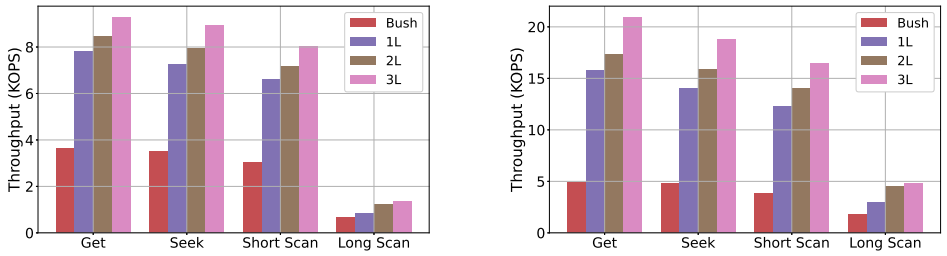
Average Query Performance. The average query performance is the aggregation of each instantaneous query performance over a long period. A natural question is how to define such a long period. We note that, regardless of the compaction policy, the LSM-tree always undergoes a global compaction that merges all sorted runs into one. We define the time between two global compactions as a *compaction round*. Each global compaction can be viewed as a *checkpoint* that ensures future queries are not affected by the compaction policy used in the previous compaction round. We use the average query throughput over a compaction round as the measure of average query performance.

Compaction's Double-Edged Effect on Queries. A more aggressive compaction policy, which frequently merges different sorted runs, would benefit instantaneous query performance more, as a query needs to probe fewer sorted runs immediately after a compaction. However, for average query performance, the situation is different because compaction and queries are always competing for CPU and I/O resources. In a compaction round, the CPU and I/O bandwidth might occasionally be occupied by background threads for compaction, temporarily reducing the query speed or even blocking queries (if no more threads or bandwidth are available). Aggressive compaction occupies both CPU and I/O bandwidth for a longer period. If we avoid certain compactions, a single query might incur slightly higher I/O costs, but we will have more time and resources available for queries throughout the compaction round. This can potentially increase the average query throughput.

The Investment View. *We view compaction as a form of investment. Compaction's write amplification is the cost of the investment, and the increased instantaneous query throughput is the immediate return.* Maximizing average query performance means maximizing the accumulated return from each investment. A compaction's accumulated return might be less than the opportunity cost.

4 The EcoTune Algorithm

In this section, we propose a method to find a compaction policy that optimizes the average query throughput in a compaction round, given a client-determined constant write speed. The average query performance should be modeled by both the costs and the accumulated returns of compactions. The cost of a compaction can be easily modeled based on the amount of involved data, and such cost for each compaction is independent. However, modeling the return of a compaction is more complicated because we need to consider both its immediate improvement in query speed and how long such improvement will last. We first analyze the compaction's impact on instantaneous query speed. Then, we note that the timing of a compaction bridges the gap between instantaneous and average performance. It is the key to evaluating how long a compaction's impact will last.



(a) Query Speed on NVMe SSD.

(b) Query Speed on Optane.

Fig. 3. Speeds of Different Queries. xL means the last x levels are leveling levels.

	Bush	1L	2L	3L
Get	1.06	1.009	1.007	1.003
Seek	1.08	1.03	1.02	1.01
Short Scan	1.11	1.07	1.07	1.07

Table 3. I/O Performance of Different Compaction Policies.

When incorporating timing information into the compaction policy, the concept of a *physical level* becomes vague. Therefore, we describe the LSM-tree structure using a three-logical-level model. We present a dynamic programming algorithm, namely EcoTune, to quickly determine the optimal compaction policy for a given query workload and machine configuration.

4.1 Query Performance Analysis

We first analyze the impact of compaction on instantaneous query speed. The instantaneous speed refers to the query speed that can be achieved immediately after compaction. In LSM-bush [33], the author points out that merging small sorted runs is less impactful for point reads. We are interested in whether this conclusion holds true for other types of queries. Therefore, we conducted a simple experiment. We used the books dataset from SOSD [63] to build a 4-level LSM-tree. The block cache size was set to 8MB. For each sorted run, we built a SNARF [76] to accelerate both point and range queries. In our experiments, the target key for the *Get* operation exists in the last level of the LSM-tree to simulate worst-case performance. The short scan operation scans a closed key range of length 100. The short scan usually contains only 1 target key and always contains fewer than 5 keys. The long scan operation is an open range scan ($Seek + 100Next$). The workloads are read-only to keep the LSM-tree static and test the instantaneous throughput after each compaction.

We show the single-thread query throughput of different compaction policies (Figure 2) in Figure 3. We also present the corresponding I/O numbers in Table 3. The performance difference between Bush and $1L$, $2L$, and $3L$ represents the impact of merging small sorted runs. The performance difference between $1L$ and $2L$ represents the impact of merging large sorted runs. We found that neither merging small sorted runs nor large sorted runs significantly improves I/O performance for *Get*, *Seek*, and short closed scans. The reason is that SNARF has already reduced the unnecessary I/O operations to nearly zero. LSM-bush has a much lower throughput due to its significant CPU overhead. It has too many small sorted runs and needs to probe too many filters for each query. Merging large sorted runs (from $1L$ to $2L$) greatly improves the throughput of long range scans (Figure 3a), whereas merging small sorted runs (from $2L$ to $3L$) provides only marginal improvements

(Figure 3a and Figure 3b). This is because most of the target keys for long range scans are located in large sorted runs. The read amplification arises from the SSD's block access granularity, and compactions can reduce this read amplification. In contrast, few small sorted runs have target keys. Therefore, range filters make merging small sorted runs less beneficial. Let d and S represent the total number and total size of small sorted runs, respectively. M denotes the size of the entire LSM-tree, and p_i is the size proportion of the i_{th} small sorted run. We omit the range filter's false positive rate here due to its low value. As p_i is low, the total number of I/Os for small sorted runs during a long-range scan (*Seek* + *KNext*) is independent of d:

$$\sum_{i=1}^{d}\left(1-(1-p_i)^K\right) \approx \sum_{i=1}^{d}(1-(1-Kp_i)) = \sum_{i=1}^{d} Kp_i = K \cdot \frac{S}{M}$$

We summarize our analysis as following: (1) The CPU overhead prevents the compaction policy for small sorted runs from being overly lazy (LSM-bush). (2) The I/O overhead of long range scans is highly affected by the number of *large* sorted runs. (3) The I/O overhead for other types of queries is robust to the number of sorted runs, including both small and large sorted runs.

4.2 Three-Level Design

From Immediate Return to Accumulated Return. An optimal compaction policy aims to maximize its accumulated return while minimizing its cost. We have analyzed the relationship between a compaction's immediate effect on query speed and its cost. In a compaction round, the LSM-tree's data layout and query throughput are dynamically changing, so a compaction's impact gradually diminishes over time. To analyze a compaction's accumulated return, we must also consider how long its impact will last.

The Importance of Compaction Timing. We propose an insight: **The earlier a compaction is conducted, the greater the cumulative future returns will be.** Specifically, when the LSM-tree is far from a global compaction, compacting multiple sorted runs into one improves query speed for a longer period. When the LSM-tree is close to a global compaction, compacting sorted runs yields less benefit since the newly created sorted runs will not be queried for long. Due to the effect of timing, two compactions with the same cost could have very different cumulative returns. Therefore, at the beginning of a compaction round, we should merge sorted runs more aggressively. As the compaction round approaches its end, the compaction policy should become lazier.

Limitation of Previous Compaction Policies. All previous compaction policies [32, 33, 56, 65] restricted sorted runs to certain allowed sizes and grouped equal-sized sorted runs together as physical levels. For example, in a lazy leveling [32] policy, level l contains sorted runs with size $T^l \cdot F$ and F is the size of the in-memory write buffer. *Each physical level corresponds to an allowed size of sorted runs. There are only L allowed sizes.* Previous works tuned the size ratio between adjacent levels and the allowed number of sorted runs in each level. In other words, they focus on how to choose these L allowed sizes and the allowed number of sorted runs for each allowed size. The allowed size and the allowed number determine the compaction aggressiveness of a level. Therefore, *all previous compaction policies have fixed aggressiveness at all times (although different physical levels could have different aggressiveness).* For example (Figure 4), in a traditional multi-level LSM-tree, all sorted runs at the same level have the same size. When optimizing the average query throughput, an ideal compaction policy that considers compaction timing is shown in Figure 4. This is because different sorted runs in a level are created at different times (*i.e.* different distances to the next global compaction). At the beginning of a compaction round, an aggressive compaction policy will make those early-created sorted runs larger than those created later.

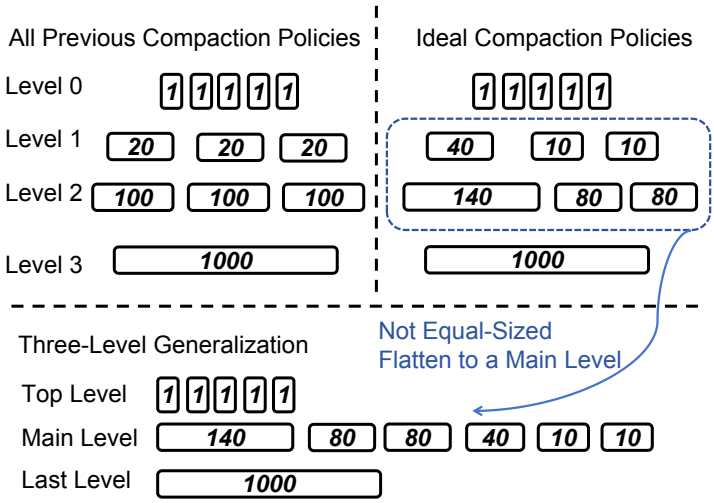


Fig. 4. Multi-Level LSM-tree vs Three-Level LSM-tree.

Three-Level Generalization. The aggressiveness of an optimal compaction policy should change over time. In other words, the size of each sorted run depends on its creation time rather than being restricted to L allowed sizes. Therefore, there are no groups of equal-sized sorted runs. As the concept of *physical level* comes from the L allowed sizes, there isn't a definition for such physical levels any more, and there is also no need to have. Instead, we view the LSM-tree as a set of sorted runs, similar to RocksDB's universal compaction [8]. Since merging small sorted runs offers little improvement, we redivide the sorted runs into three logical levels based on their impact on query speed: the top level, the main level, and the last level (Figure 4). With the three-level model, there are three kinds of background jobs: flush from memory to the top level, compaction from the top level to the main level (*TMC*), and compaction within the main level (MLC). These three kinds of background jobs could be conducted in parallel.

Top Level. The top level acts as a write buffer on the SSD, temporarily storing newly flushed sorted runs. As the analysis in Section 4.1 suggests, the I/O count for each query is not affected by the data layout (*i.e.* the number of sorted runs) in the top level. *Therefore, we do not conduct any compaction in the top level to save resources.* This raises two problems: (1) How should the capacity of the top level be set? (2) How can we handle the significant CPU overhead when probing the top level? For the first problem, we set the top level's capacity S to M/K, where K is the average number of keys involved in a long range scan. We will explain why we set S to M/K later. Since no compaction happens in the top level, there will be many sorted runs, leading to significant CPU overhead. Inspired by global filters [34, 78], we build a full index for keys in the top level. In this full index, we record the full keys and their corresponding sorted run IDs. To limit memory overhead, we set S to $M/\max(100, K)$. With this setting, the full index for all keys in the top level increases the average bits per key by approximately 0.8.

Main Level. When the top level reaches its capacity, it is compacted into a sorted run in the main level. The number of sorted runs in the main level affects the I/O performance of long range scans. If the workload contains fewer long range scans, we can use a lazier policy, allowing more large sorted runs on the SSD. Therefore, the main level compaction policy is central to the LSM-tree's

compaction strategy [33, 65]. We propose a dynamic programming algorithm, EcoTune, to find the optimal compaction policy for the main level. In the main level, we build a range filter for each sorted run to accelerate all types of queries. The size limit of the main level is controlled by a capped size ratio C, which represents the size ratio between the main and the last level.

Last Level. We allow only one sorted run in the last level to limit space amplification, a design that is widely used [32, 33, 56, 65]. When the main level reaches its capacity limit, the LSM-tree triggers a global compaction, merging all sorted runs into the last level. The capacity of the main level is determined by the size of the last level and the allowed space amplification (controlled by C).

Definition of *Long Range Scan.* Our definition of a long range scan involves K keys. However, as K decreases, the long range scan gradually becomes a short range scan, and the compaction policy at the main level no longer significantly impacts its performance. A natural question is: what is the boundary between *long range* and *short range*? We define a range scan as long if the number of involved keys is larger than $(C+1)$. This is because the main level's size proportion is $\frac{1}{C+1}$. When the main level is expected to contain more than one target key of a range scan, merging sorted runs in the main level can be considered beneficial. Otherwise, the expected I/O count at the main level is smaller than 1 and is not affected by the data layout in the main level. This is because at least one I/O is required if a target key exists in the main level, and the existence of that key is independent of the data layout.

The Choice of S. We show why we set the top-level capacity S to $\frac{M}{K}$. In the top level, there are d sorted runs. The total number of I/Os for a long-range scan is: $\sum_{i=1}^{d}\left[1-\left(1-\frac{S}{d\cdot M}\right)^K\right] = d\cdot(1-e^{-\frac{S\cdot K}{M\cdot d}})$. After being compacted into the main level, the long-range scan I/O count for the newly created large sorted run is: $1-\left(1-\frac{S}{M}\right)^K = 1-e^{-\frac{S\cdot K}{M}}$. When $S < \frac{M}{K}$, merging these sorted runs in the top level together cannot reduce the long-range scan I/O count. The expected long-range scan I/O count in the top level is less than one. If there exists a key in the target range, at least one I/O is required. The existence of such a key is independent of the data layout. On the other hand, when $S > \frac{M}{K}$, merging the first several sorted runs in the top level could always improve the query speed during the time of writing these S amounts of data. Therefore, when $S < \frac{M}{K}$, we could delay merging sorted runs in the top level. When $S > \frac{M}{K}$, it would be better to reduce the number of sorted runs. Thus, we set $S = \frac{M}{K}$.

Data Size Scalability. As the data size grows, some designs [32, 33] typically result in more levels and higher write amplification within a compaction round. This is because these designs use a fixed size for the in-memory table. We believe that the size of the in-memory table should scale proportionally with the size of the data. This is because hardware configurations for running key-value stores are typically set with a fixed memory-to-SSD ratio, as reported by Facebook [49]. With this design, the LSM-tree will maintain a fixed number of physical levels. In our three-level model, the number of sorted runs in both the top level and the main level are fixed, allowing performance to scale effectively as the data size grows.

Threads Allocation. In LSM-tree, there are usually background threads for flush/compaction and foreground threads for write/query. As the write speed is fixed, we also use a fixed number of threads for writing the in-memory table and flush (1 thread according to RocksDB tuning guide [7]). Within a compaction round, there are two types of compactions: compactions from the top level to the main level (***TM* compaction**) and compactions within the main level (***ML* compaction**). These two types of compactions can be conducted in parallel. An *TM* compaction creates a new large sorted run in the main level and limits the size of the top level. The *ML* compaction manages the large sorted runs in the main level according to a specific compaction policy. We allocate two threads for the *TM* compaction (*TMC* threads) to keep up with the flush speed. When no

TM compaction is being processed, we keep *TMC* threads free in case a sudden write speed burst happens. We use all other available threads for the *ML* compaction (*MLC* threads) and query (query threads). Different from traditional LSM-tree that totally separate the foreground and background threads, we also use *MLC* threads for queries when they are free. We argue that given a fixed number of threads, using background threads for queries when they are free could directly improve the query throughput. So we use the terms *MLC* threads and query threads instead of the original foreground threads and background threads. Such allocation is set by the user as RocksDB does.

Example. We use a simple example to illustrate the compaction process. We do not show the flush and write process in the example because they are conducted in parallel with the compactions. As shown in Figure 5, a global compaction will happen after 7 *TM* compactions. Table 4 shows the compaction policy. The first two *TM* compactions create two sorted runs ($f2$ and $f3$) in the main level. An *ML* compaction aggressively merges $f2$ and $f3$ into $f5$ to reduce the number of large sorted runs. Meanwhile, a new sorted run $f4$ is created by *TM* compaction. *MLC* threads conduct queries during the time between $f5$ and $f4$'s creation. Then, an *ML* compaction merges $f4$ and $f5$ together into $f8$ according to the policy. Meanwhile, a new sorted run $f6$ is created as the time for this *ML* compaction is relatively long. *MLC* threads also conduct queries before the creation of $f7$. After that, an *ML* compaction merges $f6$ and $f7$ together. The compaction policy becomes lazier as the LSM-tree approaches the next global compaction. For the last two sorted runs, no *ML* compaction will be involved.

4.3 Dynamic Programming Algorithm

We aim to determine the optimal compaction policy between two global compactions. For convenience, we refer to the sorted run created by a *TM* compaction as a *unit sorted run*, whose size is S. In each compaction round, the main level starts as empty. A new unit sorted run is created every T_w. The time required to use *MLC* threads to rewrite S amount of data is T_c, a measured value based on the hardware configuration. We first reserve sufficient bandwidth and CPU resources for flush and *TM* compactions to ensure that write requests are not stalled. We then utilize the remaining bandwidth and *MLC* threads to measure T_c. After R unit sorted runs are created ($R \cdot T_w$ time), a global compaction is triggered.

There are e sorted runs in the main level and one sorted run in the last level. For a long-range scan, the expected I/O count in the top level is 1. Assuming the proportion of long-range queries is r, the cost of long-range queries is proportional to $(e+2) \cdot r$ [1]. Assuming all point queries are positive and the false positive rate is f, the cost of point queries is proportional to $(1-r) \cdot (1+f)$. Therefore, the total query cost is $(e+2) \cdot r + (1-r) \cdot (1+f)$. We define the query speed as:

$$q(e) = \frac{1}{(e+2) \cdot r + (1-r) \cdot (1+f)}.$$

We define the **score** as query speed $\times$ time. The score during a period of time is proportional to the number of queries the system can handle during that time.

In the following section, we first propose a simplified algorithm based on **two assumptions**: (1) We assume that no query can be processed during a *ML* compaction. (2) We assume that an *ML* compaction always takes less than T_w time. This simplified algorithm helps readers understand the key idea of our EcoTune. In reality, queries can be processed in parallel with *MLC*, and if an *ML* compaction involves more than $\frac{T_w}{T_c} \cdot S$ data, it will take more than T_w time. We will later introduce two additional parameters to modify the algorithm to handle this more complex scenario.

[1] Although we have range filters for large sorted runs, a long-range scan is still likely (at least $1 - \left(1 - \frac{1}{K}\right)^K \approx 1 - \frac{1}{e}$) to access a large sorted run. Incorporating this exact probability into the algorithm would result in high complexity. However, our experiments demonstrate that modeling the cost of long-range scans as $(e+2) \cdot r$ is sufficiently accurate.

# *TM* Compactions	1	2	3	4	5	6	7
Main Level Layout	1	2	3	3 1	3 2	3 2 1	3 2 1 1

Table 4. Compaction Policy Example.

We start by considering a general case: given e existing sorted runs in the main level, how do we determine the optimal compaction policy for c incoming unit sorted runs? These e sorted runs will not be involved in the compaction process of the c incoming unit sorted runs. We call such a problem an (e, c) problem. Our root problem is a $(0, R)$ problem. Some of the c incoming unit sorted runs will be merged together. We define a sorted run as a ***final sorted run*** of the (e, c) problem if its lifetime lasts until all c unit sorted runs have been created. In Figure 5, f_8 is the first final sorted run of the root $(0, 7)$ problem. Finding the compaction policy for f_6, f_7, f_9, f_{11} is a $(1, 4)$ problem, and f_{10}, f_9, f_{11} are final sorted runs for this problem. Since our $q(e)$ does not consider the concrete size of each main level sorted run, the optimal compaction policy for all (e, c) problems is the same.

4.3.1 Dynamic Programming Overview. For an (e, c) problem, once its first final sorted run (with size $x \cdot S$) has been created, the remaining problem becomes an $(e + 1, c - x)$ problem. On the other hand, the optimal policy to organize the first x unit sorted runs, which form the first final sorted run, is an (e, x) problem. This optimal substructure property inspires us to use a dynamic programming algorithm to solve an (e, c) problem. As with standard dynamic programming, the key idea of our EcoTune is to iterate over all possible sizes for the first final sorted run to reduce the size of the problem.

The Score Function and The Recurrence Relation. We define $f(e, c)$ as the best score of compaction policies for an (e, c) problem. An (e, c) problem spans $c \cdot T_w$ time. However, $f(e, c)$ accounts for the number of queries conducted during the last $(c - 1) \cdot T_w$ time and excludes the first T_w time in the (e, c) problem. The reason is that all compaction policies have the same behavior during the first T_w time. The first unit sorted run of the problem cannot be directly merged with any other sorted runs. The recurrence formula is:

$$f(e, c) = \max_x \Big(f(e, x) + (T_w - x \cdot T_c) \cdot q(e + 1) + f(e + 1, c - x) \Big).$$

The term $(T_w - x \cdot T_c) \cdot q(e + 1)$ represents the number of queries conducted during the first T_w time of the $(e + 1, c - x)$ problem. After managing the first x sorted runs, the LSM-Tree requires an *MLC* to merge them into a final sorted run. This process takes $x \cdot T_c$ time and overlaps with the first T_w time of the $(e + 1, c - x)$ problem. Based on our assumptions above, $T_w - x \cdot T_c > 0$, and no query is conducted during the $x \cdot T_c$ time. The complexity of this algorithm is $O(R^3)$. The root problem is a $(0, R)$ problem.

4.3.2 Allowing Query During MLC. In this section, we relax assumption (1) because, in reality, we cannot completely block queries during *MLC*. We define $q'(e) = \beta \cdot q(e)$ as the query speed during *MLC*, where β is a measured parameter. Here, q represents the speed of queries when both *MLC* threads and query threads are used, while q' represents the speed when only query threads are used, as *MLC* threads are conducting *MLC*. Compared to the simplified EcoTune above, during each $x \cdot T_c$ period and the time of global compaction, the LSM-Tree can still process queries. Unfortunately, the query speed during $x \cdot T_c$ is difficult to determine because it depends on the layout of these $x \cdot S$ data right before they are merged into a single final sorted run.

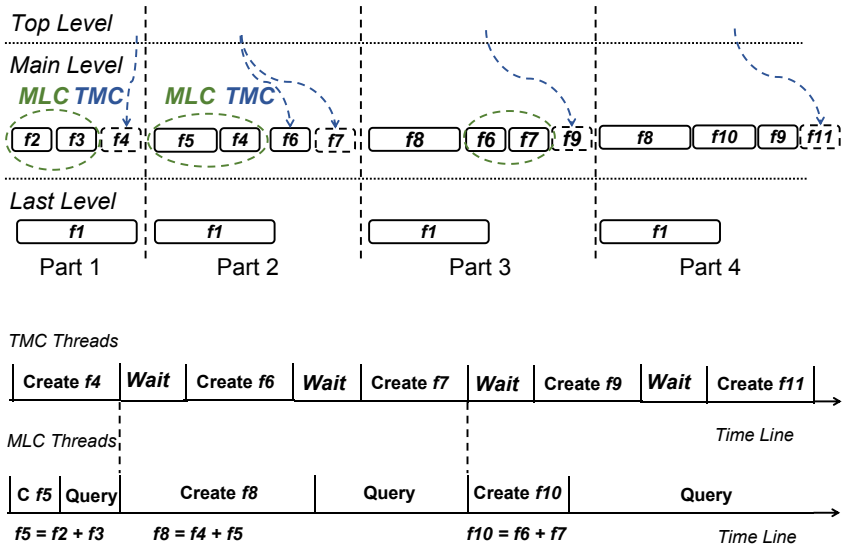


Fig. 5. An overview of our insight. *C f5* is the short for *Create f5*.

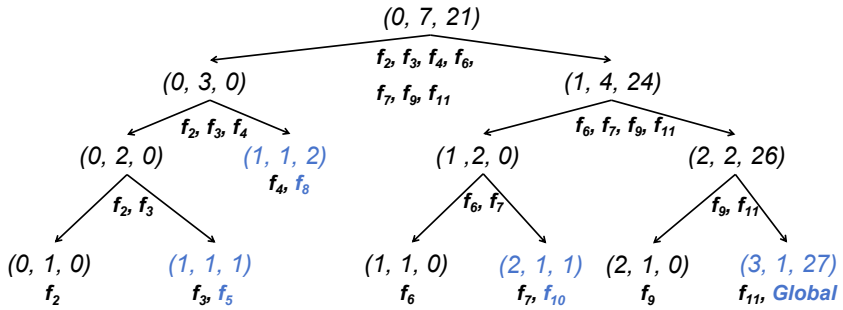


Fig. 6. Problem Partition Example. Sub-problems marked blue are the right most sub-problems. The sorted runs corresponding to each sub-problem are shown near the problem. For example, problem $(1, 1, 1)$ calculates the score during the creation of f_3 and f_5. The creation of f_5 is an *MLC*. In this sub-problem, $e = 1$ and the existing sorted run is f_2.

To address this challenge, we introduce an additional parameter m. The complexity of this algorithm is $O(R^4)$. An (e, c, m) problem is defined as finding the optimal compaction policy for c incoming unit sorted runs, where e sorted runs currently exist, and an *ML* compaction involving $(m + c) \cdot S$ data will occur after the c unit sorted runs are created. The root problem becomes a $(0, R, C \cdot R)$ problem because a global compaction involves $C \cdot R \cdot S$ data in the last level and $R \cdot S$ data in non-last levels.

We divide an (e, c, m) problem into two sub-problems: (1) **The left sub-problem**: creating the first final sorted run, and (2) **The right sub-problem**: managing the remaining incoming unit sorted runs. Each sub-problem can be recursively divided. The *MLC* involving $(m + c) \cdot S$ data happens when the right sub-problem is finished. The score during such compaction is calculated

Algorithm 1: Find the best score and compaction policy

```
1  ans = Empty map whose value is (score, x)
2  Procedure f(e, c, m)
3      if c == 1 then
4          ans[(e, c, m)] = {score: (m + c) · T_c · q'(e), x: 1}
5          return ans[(e, c, m)]
6      best = {score: T_w · q(e + 1) + f(e + 1, c − 1, m + 1), x: 1}
7      for x from 2 to (c − 1) do
8          score = f(e, x, 0) + (T_w − xT_c) · q(e + 1)
9          score = score + f(e + 1, c − x, m + x)
10         if best.score < score then
11             best = {score, x}
12     ans[(e, c, m)] = best
13     return best.score
14 score = T_w · q(0) + f(0, R, C · R)
15 return (score, ans)
```

in its *right-most* sub-problem. As shown in Algorithm 1, we pass the parameter m to the right sub-problem so that the algorithm can calculate the score during the *MLC* at the end of the right sub-problem (Lines 3–5). In Figure 6, we illustrate the problem partition in the example from Figure 5 ($R = 7$ and $C = 3$). The left sub-problems are represented as left child nodes, while the right sub-problems are represented as right child nodes. The *right-most* sub-problems are marked in blue. For instance, when solving the $(1, 1, 1)$ problem, we calculate the score during the *MLC* creating f_5. The $(1, 1, 2)$ and $(3, 1, 27)$ problems contain the scores during *MLC* creating f_8 and the global compaction, respectively.

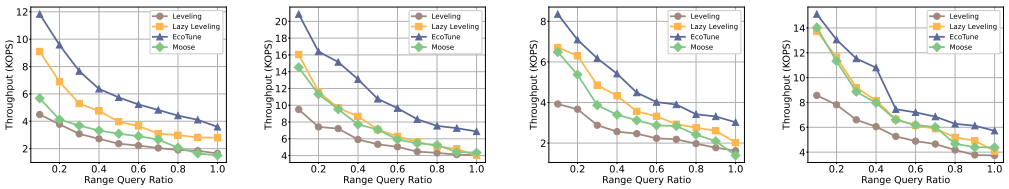
4.3.3 Allowing Pending Sorted Runs. In this section, we further relax assumption (2). An *MLC* may involve a large amount of data, $x \cdot S$, causing $x \cdot T_c > T_w$. Under assumption (2), a left sub-problem does not influence the compaction policy of the corresponding right sub-problem. Without this assumption, during the creation of the first final sorted run, multiple new unit sorted runs may be created (denoted as *pending sorted runs*). In part 2 of Figure 5, before an *MLC* completes the creation of f_8, f_6 and f_7 have already been created and are pending sorted runs. Therefore, at the start of the corresponding right sub-problem, there will be b pending unit sorted runs and $c - x - b$ incoming sorted runs. Both the incoming sorted runs and the pending sorted runs will be involved in the compaction process of the right sub-problem. These pending sorted runs not only affect the compaction policy of the right sub-problem but also influence the score during the creation of the first final sorted run. Consequently, we need to describe a problem using four parameters (e, b, c, m), and the complexity of the algorithm becomes $O(R^5)$. Due to space limitations, we do not delve deeply into this final version of EcoTune. The idea remains the same as the previous two versions. The algorithm is shown in Algorithm 2.

The Additional Overhead: Our EcoTune algorithm requires tracking workload information and calculating the optimal compaction policy. In practice, the workload information can either be derived from prior knowledge of the workload or gathered from statistics based on previous queries, similar to other workload-aware algorithms such as Cosine [24] and Limousine [25]. Specifically,

Algorithm 2: Find the best score and compaction policy

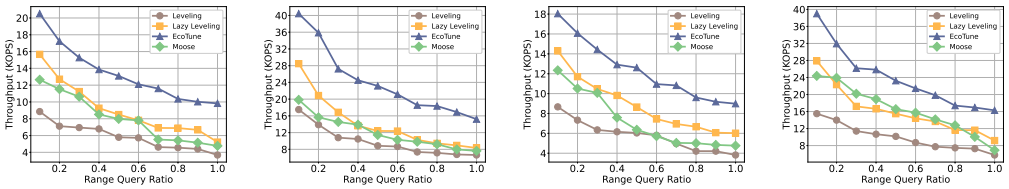
```
ans = Empty map whose value is (score, x)
Procedure MergeScore(e, m)
    Δb = ⌈mT_c / T_w⌉ − 1
    score = Σ_{i=0}^{Δb−1} T_w q'(e + i)
    score = score + q'(e + Δb)(mT_c − ΔbT_w)
    return score
Procedure f(e, b, c, m)
    if c == 0 then
        ans[(e, b, c, m)] = {score: MergeScore(e + b, m + b), x: 1}
        return ans[(e, b, c, m)]
    if b > 0 then
        score = f(e + 1, b − 1, c, m + 1)
    else
        score = T_w · q(e) + f(e + 1, 0, c − 1, m + 1)
    best = {score, x: 1}
    for x from 2 to (b + c − 1) do
        Δb = ⌈xT_c / T_w⌉
        if x ≤ b then
            score = MergeScore(e + b, x)
            b' = b − x + Δb
        else
            score = f(e, b, x − b, 0)
            b' = Δb
        score = score + (ΔbT_w − xT_c) · q(e + b')
        available = b + c − x
        if b' > available then
            continue
        score = score + f(e + 1, b', available − b', m + x)
        if best.score < score then
            best = {score, x}
    ans[(e, b, c, m)] = best
    return best.score
score = f(0, 0, R, C · R)
return (score, ans)
```

the algorithm requires three parameters: the long-range scan ratio r, the write speed T_w, and the compaction speed T_c. We only need a single *if* statement for each query to track the long-range scan ratio and only need to measure the time of a previous compaction round to determine the write speed. The other key parameter, the compaction speed, is workload-independent and can be measured offline. Such statistics are straightforward to obtain. We solve the optimal policy during the time of each global compaction. The dynamic programming algorithm takes less than 1 second,



(a) books + 6 cores. (b) books + 10 cores. (c) osm + 6 cores. (d) osm + 10 cores.

Fig. 7. Throughput vs Long Range Scan Ratio on NVMe SSD.



(a) books + 6 cores. (b) books + 10 cores. (c) osm + 6 cores. (d) osm + 10 cores.

Fig. 8. Throughput vs Long Range Scan Ratio on Optane SSD.

while a global compaction takes approximately 160 seconds and a compaction round takes about 350 seconds in our experiments. As a result, the policy-solving overhead is negligible.

5 Evaluation

5.1 Experiments Setup

Baselines: We compare our EcoTune with three baselines: Leveling (RocksDB's default compaction policy), Lazy Leveling [32], and Moose [56]. Moose is a recently proposed compaction policy that allows each level to have a completely different size ratio and a varying number of sorted runs. Since we do not conduct any compactions at the top level, the data layout of the Lazy Leveling policy closely resembles that of LSM-bush [33]. We implement our EcoTune and Moose using Dostoevsky, a RocksDB-based LSM-tree that already integrates the leveling and lazy leveling policies. We use a two-dimensional vector to represent the data layout after each *TMC*. The compaction picker in Dostoevsky will determine how to merge sorted runs according to the vector.

Setup: Our experiments were conducted on a machine with the following specifications: an AMD EPYC 7742 64-Core Processor, 512MB of L3 cache, and two SSDs (800GB Intel Optane SSD DC P5800X and 3TB D7-P5620 NVMe SSD). The memory write buffer was set to 8MB. A SNARF range filter was constructed for every file with a default memory allocation of 14 bpk. An 8MB block cache was allocated for data blocks. We used 64-bit integer keys from the SOSD dataset [63] and 256 bytes of value for each key. We enabled direct I/Os for both read and write operations. The size ratio for leveling and lazy leveling policy is set to 10. The LSM-tree for leveling and lazy leveling consists of four levels. For Moose[2], its LSM-tree consists of five levels. Moose allows 3 sorted runs in each non-last level and one sorted run in the last level. The size ratio between level 3 and level 4 (the last level) is 6, while the size ratio between the other levels is 7. In all experiments, there is one client thread to write the in-memory write buffer, one thread for flush, and two threads for *TMC*.

[2]Moose's configuration was provided by its authors after we shared our experimental settings with them.

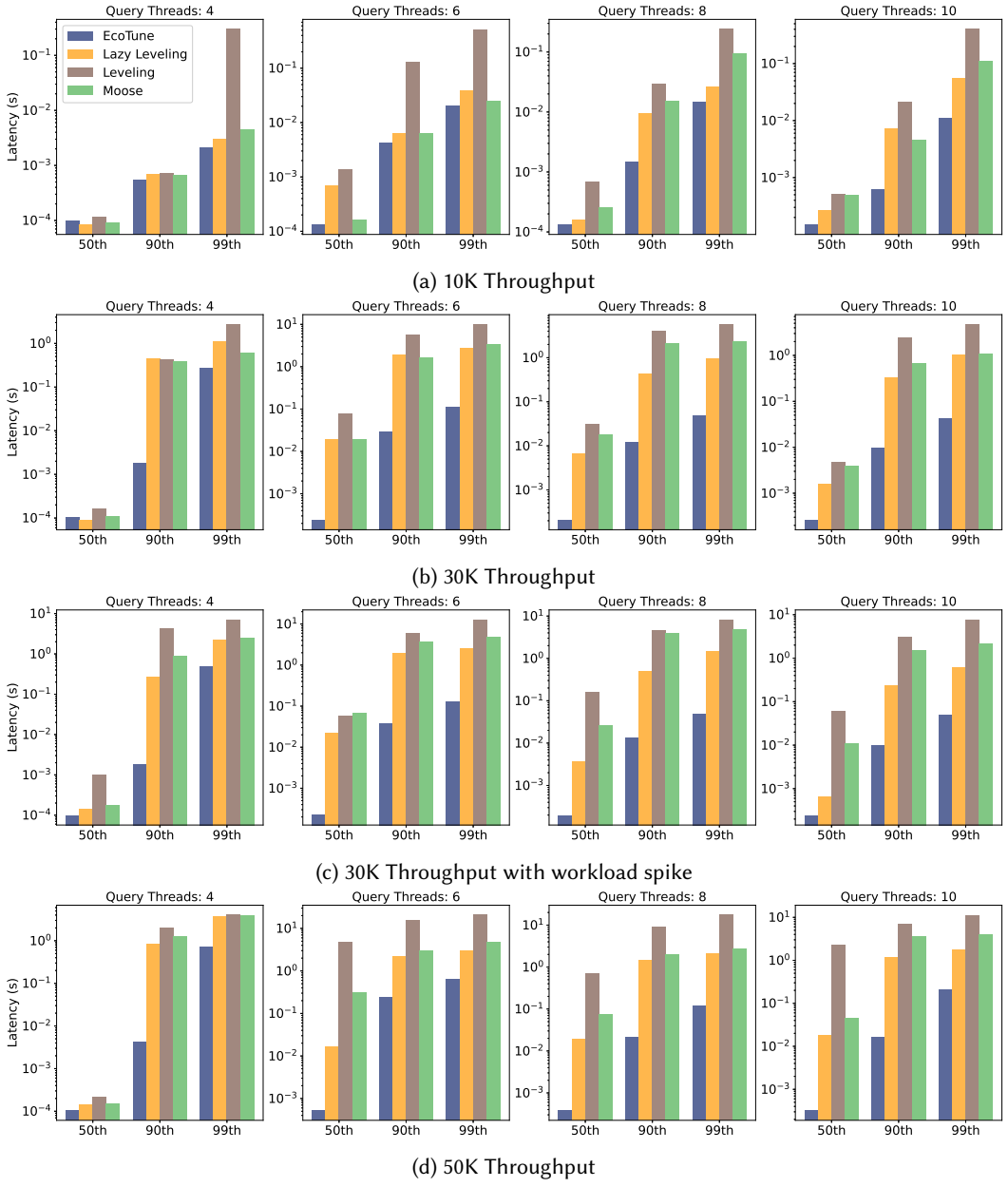
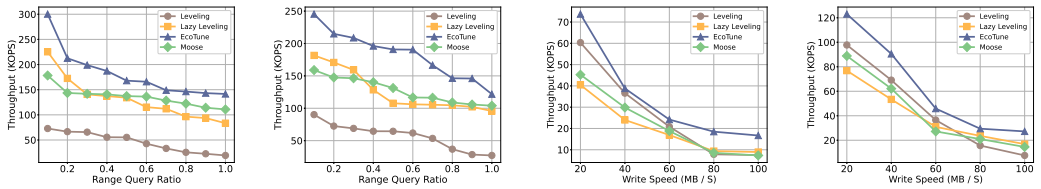


Fig. 9. Latency under different query arrival speeds.

We provide two CPU cores for these threads and only vary the number of cores for *MLC* and query.

Workload: In our experiments, we varied the long range scan ratio in the workload from 0.1 to 1. The long range scan contains *Seek* + 100*Next*. The remaining part of the workload contains the same number of *Get* and *Seek* operations. Assuming the long range scan ratio is r, each operation



(a) books + Optane. (b) osm + Optane. (c) books + NVMe. (d) books + Optane.

Fig. 10. (a) and (b) are results of 64 threads (16 *MLC* threads and 48 query threads) experiments on Optane SSD. (c) and (d) shown query throughput under different write speed. We use 10 cores and the workload contains 30% long range scan.

# *TMC*	Main Level Layout
7	7[1] -> 7
14	7 7[1] -> 7 7
21	7 7 7[1] -> 7 7 7
24	7 7 7 3[1] -> 7 7 7 3
29	7 7 7 3 5[1]

Table 5. 10 cores + 10% Long Range Scan

# *TMC*	Main Level Layout
3	3[1] -> 3
8	3 5[1] -> 8
11	8 3[1] -> 8 3
15	8 3 4[1] -> 8 7
18	8 7 3[1] -> 8 7 3
23	8 7 3 4[1] -> 8 7 7
25	8 7 7 4[1] -> 8 7 7 3
29	8 7 7 3 4[1]

Table 6. 10 cores + 20% Long Range Scan

# *TMC*	Main Level Layout
3	3[1] -> 3
7	3 4[1] -> 7
10	7 3[1] -> 7 3
14	7 3 4[1] -> 14
17	14 3[1] -> 14 3
21	14 3 4[1] -> 14 7
25	14 7 4[1] -> 14 7 4
29	14 7 4[1]

Table 7. 10 cores + 30% - 80% Long Range Scan

# *TMC*	Main Level Layout
3	3[1] -> 3
7	3 4[1] -> 7
10	7 3[1] -> 7 3
13	7 3 3[1] -> 13
16	13 3[1] -> 13 3
20	13 3 4[1] -> 13 7
23	13 7 3[1] -> 13 7 3
26	13 7 3 3[1] -> 13 7 3 3
29	13 7 3 3 3[1]

Table 8. 10 cores + 90% - 100% Long Range Scan

in the workload have r, $\frac{1-r}{2}$, $\frac{1-r}{2}$ probabilities to be a long range scan, *Seek* operation and *Get* operation respectively. The targe key is chosen randomly. Without further specification, we fix the write speed to 100 MB/s, which is much higher than the real workload's speed. This setting is to verify our argument that LSM-tree's performance is independent of the compaction policy. We use half threads as *MLC* threads and the other half as query threads by default. A global compaction is triggered when the large level's size reaches 1/3 of the main level's size. We use the average throughput of a compaction round to measure the performance of each compaction policy.

5.2 Experiments on Average Query Throughput

As discussed above, considering the overhead, there are two types of queries: long-range scans and others. We vary the long-range scan ratio from 0.1 to 1. We find that our EcoTune consistently achieves the highest throughput. EcoTune outperforms the Leveling policy by 30% to 150% because EcoTune spends more time on queries rather than compaction while maintaining a moderate number of sorted runs. EcoTune also outperforms the Lazy Leveling policy by 15% to 80% due to having fewer sorted runs for workloads with a high ratio of long-range scans. Our solved compaction policies are shown in Table 5 - Table 8. We found that our EcoTune is also robust to workload shift. The solved policy is not sensitive to different workloads. For long range scan ratio varies from 30% to 80%, the optimal compaction policy remains unchanged. Therefore, workload shift can not easily violate the performance of EcoTune.

Impact of Long Range Scan Ratio. From Figure 7 to Figure 8, we measure the average throughput of different compaction policies with varying long range scan ratios. Our EcoTune achieves up to 1.8x the throughput compared to the second-best algorithm. As the long range scan ratio increases, the throughput of all policies decreases. Previous research indicated that the leveling compaction policy offers the best query performance. In our experiment, we find that the lazy leveling compaction policy and Moose compaction policy outperform the leveling compaction policy in most cases. This is because, during a compaction round, the leveling policy spends too much time on compaction, leaving less time for queries. In contrast, the lazy leveling policy has a slower query speed due to more large sorted runs. Our EcoTune achieves the best performance because it can better balance the time for queries and compactions.

Impact of CPU Cores. We use 6 cores and 10 cores (including 2 cores for writing in-memory buffer, flushing, and *TMC*) to conduct our experiments, respectively. The available cores are used to run both query threads and compaction threads (*MLC* threads). The query threads are dedicated to queries only. When there are no compaction tasks, the compaction threads also perform queries to achieve higher throughput, as previously illustrated. With 6 cores, there are 5 query threads and 5 *MLC* threads. With 10 cores, there are 7 query threads and 7 *MLC* threads. In our experiments, allocating more threads in each case degrades write speed, as excessive threads negatively impact the flush thread. We observe that in all cases, Moose and the lazy leveling compaction policy exhibit similar performance. Our EcoTune consistently achieves the best performance. Furthermore, we find that EcoTune shows a greater advantage with 10 cores compared to 6 cores. This is because EcoTune optimizes the assignment of compactions and queries to threads. However, CPU cores have their own scheduling algorithms to execute different threads concurrently rather than in parallel. This leads to additional CPU contention and context-switching overhead, which is not accounted for in our EcoTune. Such unaccounted overhead is relatively significant with 6 cores.

Impact of SSD Bandwidth and Latency. We use an NVMe SSD and an Optane SSD in our experiment. We find that our EcoTune has a greater advantage on the Optane SSD. The reasons are: (1) The Optane SSD has a larger write bandwidth and similar sequential write latency compared to the NVMe SSD. This enables the write speed to scale better, making compaction time relatively low, while the flush time remains similar to that of the NVMe SSD. Since the time of a compaction round is determined by flush time, the Optane SSD allows more time for queries during a compaction round. (2) The random read latency is lower, allowing for more sorted runs on the SSD. This enlarges the design space for the compaction policy.

5.3 Experiments on Latency

When discussing query latency, all previous works [19, 47, 54, 55, 74, 80] focused solely on execution latency. However, we found that the primary contributor to query latency is **queuing latency**. In

real applications, the query request arrival rate is also determined by clients. When the system is temporarily unable to provide sufficient query throughput, incoming query requests will queue up, leading to increased latency. Unfortunately, previous research has not considered this queuing latency. This queuing phenomenon is common in all LSM-trees because compaction consumes significant CPU and I/O resources, which greatly slows down (or even halts) queries during compaction, causing requests to queue.

In this section, we demonstrate how EcoTune reduces query latency (including queuing latency) compared to other compaction policies by achieving higher average query throughput. Under a 100 MB/s write speed, we vary the query arrival speed from 10K/s to 50K/s. As shown in Figure 9, our EcoTune reduces latency by up to 4 and 3 orders of magnitude compared to Leveling and Lazy Leveling, respectively. This is because we spend much less time on compactions compared to the leveling policy, so the average number of accumulated query requests during compactions is relatively low. Our policy also maintains fewer sorted runs, resulting in faster query execution compared to Lazy Leveling. As a result, EcoTune allows most of the accumulated query requests during compaction to be processed quickly. With a query arrival rate of 30K/s, we conducted two experiments. In Figure 9b, queries arrive uniformly, while in Figure 9c, the query arrival rate is set to 15K/s, with each query having a 5% probability of triggering 20 queries simultaneously. Our results show that all algorithms experience increased latency under the workload spike. However, the latency of our EcoTune remains relatively stable, increasing by less than 2 times compared to the uniform case.

We also find that the allocation of threads between *MLC* threads and query threads significantly impacts query latency. Although allocating all available threads to *MLC* threads results in higher average query throughput, it is suboptimal for query latency. With more *MLC* threads, the LSM-tree completes compaction faster, but the query speed during compaction decreases. Conversely, reserving more query threads leads to longer compaction times but higher query speed during compaction. This trade-off is complex, and we are the first to experimentally study this issue.

Given 14 available threads, we vary the number of query threads from 4 to 10. The results are shown in Figure 9. Although our EcoTune is not particularly designed for latency, we find that EcoTune consistently exhibits the lowest latency regardless of thread allocation as it has higher average query throughput[3]. Each policy experiences relatively high tail latency due to queuing caused by compaction even under a low query arrival rate. In Figure 8, all algorithms achieve a query throughput of more than 10 KOPS. However, in Figure 9a, all algorithms still exhibit a high tail latency of several milliseconds. The reason for this is that when queries arrive as a stream, there are moments when neither queries nor compaction tasks are present, leaving the CPU cores free. At other times, compaction tasks may cause query requests to queue, resulting in higher latency. In other words, the CPU cores for all algorithms are under-utilized. A potential solution to this problem is to adjust the compaction policy based on the query arrival rate to ensure that the CPU cores remain fully utilized. For example, if there are no incoming queries, all available threads can be used to greedily compact sorted runs into the last level. In this scenario, there is no need to reserve CPU or I/O resources for query processing. We propose this query-stream-dependent compaction strategy as an interesting direction for future work.

5.4 Performance under Skewed YCSB workload

Our previous experiments were conducted with a uniform workload, where nearly all queries required SSD I/Os. In this experiment, we use the YCSB benchmark to generate a skewed workload

[3]In fact, the queuing latency will finally become infinite if the query arrival speed is higher than the average query throughput. We only measure the tail latency within a compaction round.

	Optane (KOPS)	NVMe (KOPS)
EcoTune	174.80	122.86
Lazy Leveling	157.85	109.90
Leveling	29.52	23.09
Moose	142.11	104.07

Table 9. Average query throughput on different SSDs with YCSB workload.

consisting of 30% long-range scans and 70% point reads. The write speed is set to 100 MB/s. The results, shown in Table 9, demonstrate that EcoTune outperforms lazy leveling by approximately 15%, though this advantage is less pronounced compared to the uniform workload. Conversely, EcoTune's advantage over the leveling policy is significantly greater, achieving 6× average throughput. The reason is that aggressive compaction frequently invalidates cached data blocks, as reported by [83]. EcoTune uses an aggressive compaction policy at the beginning of a compaction round and the leveling policy is aggressive all the time, which lower the cache hit rate for each algorithm.

5.5 Performance under High Parallelism and Different Write Speed

When sufficient CPU resources are available, the average query throughput is bottlenecked by SSD bandwidth. In fact, our model does not differentiate between CPU or I/O bottlenecks. We also conduct experiments at lower write speeds to demonstrate that our EcoTune performs well under various settings.

High Parallelism Performance. In Figure 10a and Figure 10b, we use 64 threads to evaluate the performance of different policies under high parallelism. The experiment targets the I/O bottleneck cases, so we assign each thread its own CPU core to avoid any contention on CPU cores. Our EcoTune outperforms all baselines.

The Impact of Write Speed. In previous experiments, we fixed the write speed at 100 MB/s, which is significantly higher than most real-world workloads. We are also interested in examining each compaction policy's query performance when the write speed is lower. We vary the write speed from 20 MB/s to 100 MB/s, with the long-range scan ratio set to 30%. The experimental results are shown in Figure 10c anc Figure 10d. When the write speed is low, the leveling policy achieves higher query throughput than the lazy leveling policy and Moose. A lower write speed means a longer compaction round. For each compaction policy, the compaction time occupies a small portion of the overall compaction round, and the leveling policy has fewer sorted runs and faster query execution. Our EcoTune still identifies the optimal policy and achieves higher query throughput than the leveling policy.

6 Conclusion

We have introduced a novel perspective on the compaction process in LSM-trees. Given a specific write speed, our compaction policy aims to maximize average query throughput. We have observed the competitive relationship between compaction and query operations. To address this, we propose a DP algorithm that optimally balances the competition between compaction and queries.

Acknowledgement

This work was supported (in part) by the Shanghai Qi Zhi Institute Innovation Program SQZ202406.

References

[1] [n. d.]. Apache. Cassandra. http://cassandra.apache.org.
[2] [n. d.]. Apache. HBase. http://hbase.apache.org/.
[3] [n. d.]. CockroachDB. https://github.com/cockroachdb/cockroach.

[4] [n. d.]. Facebook. RocksDB. https://github.com/facebook/rocksdb.
[5] [n. d.]. Intel Optane SSD DC P5800X Series 800GB. https://www.intel.com/content/www/us/en/products/sku/201860/intel-optane-ssd-dc-p5800x-series-800gb-2-5in-pcie-x4-3d-xpoint/specifications.html.
[6] [n. d.]. LinkedIn. Voldemort. http://www.project-voldemort.com.
[7] [n. d.]. Rocksdb Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.
[8] [n. d.]. RocksDB. Universal Compaction. https://github.com/facebook/rocksdb/wiki/universal-compaction.
[9] [n. d.]. Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb.
[10] [n. d.]. Solidigm D7-P5620 SSD. https://www.solidigm.com/products/data-center/d7/p5620.html.
[11] [n. d.]. WiredTiger. https://github.com/wiredtiger/wiredtiger.
[12] 2015. M. Annamalai. Zippydb: a modern, distributed keyvalue data store. https://www.youtube.com/watch?v=DfiN7pG0D0k.
[13] Ildar Absalyamov, Michael J. Carey, and Vassilis J. Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. *Proceedings of the 2018 International Conference on Management of Data* (2018). https://api.semanticscholar.org/CorpusID:4851108
[14] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction Management in Distributed Key-Value Datastores. *Proc. VLDB Endow.* 8 (2015), 850–861. https://api.semanticscholar.org/CorpusID:14622760
[15] Wail Y. Alkowaileet, Sattam Alsubaiee, and Michael J. Carey. 2019. An LSM-based tuple compaction framework for Apache AsterixDB. *Proceedings of the VLDB Endowment* 13 (2019), 1388 – 1400. https://api.semanticscholar.org/CorpusID:204788800
[16] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark D. Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *ACM SIGMOD Conference*. https://api.semanticscholar.org/CorpusID:11759711
[17] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:22824631
[18] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:196810469
[19] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau
[20] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13 (1970), 422–426.
[21] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proc. VLDB Endow.* 11 (2018), 1863–1875. https://api.semanticscholar.org/CorpusID:49572006
[22] Zhichao Cao, Siying Dong, Sagar Vemuri, and David Hung-Chang Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies*. https://api.semanticscholar.org/CorpusID:211137004
[23] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:260548458
[24] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *Proc. VLDB Endow.* 15 (2021), 112–126. https://api.semanticscholar.org/CorpusID:245811523
[25] Subarna Chatterjee, Mark F. Pekala, Lev Kruglyak, and Stratos Idreos. 2024. Limousine: Blending Learned and Classical Indexes to Self-Design Larger-than-Memory Cloud Storage Engines. *Proc. ACM Manag. Data* 2, 1, Article 47 (mar 2024), 28 pages. doi:10.1145/3639302
[26] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (2011), 266–277. https://api.semanticscholar.org/CorpusID:41957
[27] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An Optimal Disjoint Segmented Learned Range Filter. *Proc. VLDB Endow.* 17, 8 (may 2024), 1911–1924. doi:10.14778/3659437.3659447
[28] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1, Article 46 (may 2023), 27 pages. doi:10.1145/3588726
[29] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming Adversarial Queries with Optimal Range Filters. *Proc. ACM Manag. Data* 2, 1, Article 3 (mar 2024), 23 pages. doi:10.1145/3639258

[30] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. doi:10.1145/3035918.3064054

[31] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *Proceedings of the 2016 International Conference on Management of Data* (2016).

[32] 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. *Proceedings of the 2018 International Conference on Management of Data* (2018).

[33] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[34] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. doi:10.1145/3448016.3457273

[35] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proc. VLDB Endow.* 15 (2022), 3071–3084.

[36] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *ArXiv* abs/2103.02515 (2021).

[37] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017).

[38] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[39] Tomer Even, Guy Even, and Adam Morrison. 2022. Prefix Filter: Practically and Theoretically Better Than Bloom. *ArXiv* abs/2203.17139 (2022).

[40] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014).

[41] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 32, 14 pages. doi:10.1145/2741948.2741973

[42] Thomas Mueller Graf and Daniel Lemire. 2019. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *arXiv: Data Structures and Algorithms* (2019).

[43] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 651–665. doi:10.1145/3299869.3314041

[44] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[45] Andrew Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *Proc. VLDB Endow.* 15 (2021), 1605–1618.

[46] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Slavin 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 *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:58013807

[47] IM Junsu, Jinwook Bae, Daegu Gyeongbuk, Chanwoo Chung, Sungjin Lee, Junsu Im, Jinwook Bae, and Chanwoo Chung. 2021. Design of LSM-tree-based Key-value SSDs with Bounded Tails. *ACM Transactions on Storage (TOS)* 17 (2021), 1 – 27. https://api.semanticscholar.org/CorpusID:235441154

[48] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. 2022. Separation or Not: On Handing Out-of-Order Time-Series Data in Leveled LSM-Tree. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 3340–3352. doi:10.1109/ICDE53745.2022.00315

[49] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 821–837. https://www.usenix.org/conference/atc21/presentation/kassa

[50] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Murtadha Ai Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, Chen Luo, Ian Maxon, and Pouria Pirzadeh. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50 (2020), 1114 – 1151. https://api.semanticscholar.org/CorpusID:214469621

[51] Eric Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. *Proceedings of the 2022 International Conference on Management of Data* (2022).

[52] Haridimos Kondylakis, Niv Dayan, Konstantinos Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *Proc. VLDB Endow.* 11 (2018), 677–690. https://api.semanticscholar.org/CorpusID:3569962

[53] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proc. VLDB Endow.* 12 (2019), 502–515. https://api.semanticscholar.org/CorpusID:85529414

[54] Haoyu Li, Liuhui Wang, Qizhi Chen, Jianan Ji, Yuhan Wu, Yikai Zhao, Tong Yang, and Aditya Akella. 2023. ChainedFilter: Combining Membership Filters by Chain Rule. *Proceedings of the ACM on Management of Data* 1 (2023), 1 – 27. https://api.semanticscholar.org/CorpusID:261242942

[55] Junkai Liang and Yunpeng Chai. 2021. CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1032–1043. doi:10.1109/ICDE51399.2021.00094

[56] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in a Colossal Configuration Space. *Proc. ACM Manag. Data* 2, 3, Article 175 (may 2024), 26 pages. doi:10.1145/3654978

[57] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies*. https://api.semanticscholar.org/CorpusID:11367463

[58] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies*.

[59] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-based Storage Systems. *ArXiv* abs/1906.09667 (2019). https://api.semanticscholar.org/CorpusID:195345039

[60] Chen Luo and Michael J. Carey. 2020. Breaking Down Memory Walls: Adaptive Memory Management in LSM-based Storage Systems. *Proc. VLDB Endow.* 14 (2020), 241–254. https://api.semanticscholar.org/CorpusID:236120784

[61] 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 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. doi:10.1145/3318464.3389731

[62] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. 2018. TOAIN: A Throughput Optimizing Adaptive Index for Answering Dynamic kNN Queries on Road Networks. *Proc. VLDB Endow.* 11 (2018), 594–606. https://api.semanticscholar.org/CorpusID:5595327

[63] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14 (2020), 1 – 13.

[64] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks. *Proceedings of the VLDB Endowment* 13 (2020), 3217 – 3230. https://api.semanticscholar.org/CorpusID:221539348

[65] 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. *Proc. ACM Manag. Data* 1, 3, Article 213 (nov 2023), 25 pages. doi:10.1145/3617333

[66] 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. *ArXiv* abs/2308.07013 (2023). https://api.semanticscholar.org/CorpusID:260886977

[67] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martín Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. *Proceedings of the 2021 International Conference on Management of Data* (2021). https://api.semanticscholar.org/CorpusID:233238457

[68] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: Making Authenticated Storage Faster in Ethereum. In *USENIX Workshop on Hot Topics in Storage and File Systems*. https://api.semanticscholar.org/CorpusID:46996220

[69] Kai Ren, Qing Zheng, Joy Arulraj, and Garth A. Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proc. VLDB Endow.* 10 (2017), 2037–2048. https://api.semanticscholar.org/CorpusID:3625489

[70] Sean C. Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. LittleTable: A Time-Series Database and Its Uses. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017). https://api.semanticscholar.org/CorpusID:28501514

[71] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020). https://api.semanticscholar.org/CorpusID:216034805

[72] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proc. VLDB Endow.* 14 (2021), 2216–2229.

[73] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012). https://api.semanticscholar.org/CorpusID:207194816

[74] Hui Sun, Guanzhong Chen, Yinliang Yue, and Xiao Qin. 2023. Improving LSM-Tree Based Key-Value Stores With Fine-Grained Compaction Mechanism. *IEEE Transactions on Cloud Computing* 11 (2023), 3778–3796. https://api.semanticscholar.org/CorpusID:264981626

[75] Risi Thonangi and Jun Yang. 2017. On Log-Structured Merge for Solid-State Drives. *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (2017), 683–694. https://api.semanticscholar.org/CorpusID:852089

[76] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: A Learning-Enhanced Range Filter. *Proc. VLDB Endow.* 15, 8 (jun 2022), 1632–1644. doi:10.14778/3529337.3529347

[77] Tobias Vinçon, Sergey Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage Management. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:266407

[78] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proc. ACM Manag. Data* 2, 3, Article 141 (may 2024), 27 pages. doi:10.1145/3654944

[79] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *European Conference on Computer Systems*. https://api.semanticscholar.org/CorpusID:3339913

[80] Zepeng Wang and Shu Yin. 2023. RBC: A bandwidth controller to reduce write-stalls and tail latency. 213–222. doi:10.1145/3605573.3605601

[81] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. REncoder: A Space-Time Efficient Range Filter with Local Encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2036–2049. doi:10.1109/ICDE55515.2023.00158

[82] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:13430182

[83] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rong yao Chen, Jianying Wang, and Gui Huang. 2020. Leaper. *Proceedings of the VLDB Endowment* 13 (2020), 1976 – 1989. https://api.semanticscholar.org/CorpusID:221082134

[84] Ting Yao, Ji guang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. 2017. A Light-weight Compaction Tree to Reduce I / O Amplification toward Efficient Key-Value Stores. https://api.semanticscholar.org/CorpusID:13577976

[85] Ting Yao, Ji guang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Transactions on Storage (TOS)* 13 (2017), 1 – 28. https://api.semanticscholar.org/CorpusID:25350030

[86] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16 (2022), 99–112. https://api.semanticscholar.org/CorpusID:252924530

[87] 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 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. doi:10.1145/3183713.3196931

[88] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *USENIX Conference on File and Storage Technologies*. https://api.semanticscholar.org/CorpusID:211567164

[89] Zichen Zhu. 2023. SHaMBa: Reducing Bloom Filter Overhead in LSM Trees. In *PhD@VLDB*. https://api.semanticscholar.org/CorpusID:259848936

Received October 2024; revised January 2025; accepted February 2025