ArceKV: Towards Workload-driven LSM-compactions for Key-Value Store Under Dynamic Workloads

Junfeng Liu
junfeng001@e.ntu.edu.sg
Nanyang Technological University
Singapore

Haoxuan Xie
haoxuan001@e.ntu.edu.sg
Nanyang Technological University
Singapore

Siqiang Luo
siqiang.luo@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Key-value stores underpin a wide range of applications due to their simplicity and efficiency. Log-Structured Merge Trees (LSM-trees) dominate as their underlying structure, excelling at handling rapidly growing data. Recent research has focused on optimizing LSM-tree performance under static workloads with fixed read-write ratios. However, real-world workloads are highly dynamic, and existing workload-aware approaches often struggle to sustain optimal performance or incur substantial transition overhead when workload patterns shift. To address this, we propose ElasticLSM, which removes traditional LSM-tree structural constraints to allow more flexible management actions (i.e., compactions and write stalls) creating greater opportunities for continuous performance optimization. We further design Arce, a lightweight compaction decision engine that guides ElasticLSM in selecting the optimal action from its expanded action space. Building on these components, we implement ArceKV, a full-fledged key-value store atop RocksDB. Extensive evaluations demonstrate that ArceKV outperforms state-of-the-art compaction strategies across diverse workloads, delivering around 3× faster performance in dynamic scenarios.

PVLDB Reference Format:
Junfeng Liu, Haoxuan Xie, and Siqiang Luo. ArceKV: Towards Workload-driven LSM-compactions for Key-Value Store Under Dynamic Workloads. PVLDB, 19(5): 958 - 972, 2026.
doi:10.14778/3796195.3796208

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/NTU-Siqiang-Group/ArceKV.

1 INTRODUCTION

Key-value (KV) stores map unique keys to values for fast data access and are widely used in distributed caching, large-scale databases, and cloud services [13, 18, 20, 38, 41, 52, 76, 79]. Log-Structured Merge Trees (LSM-trees) are fundamental data structures underpinning KV stores, widely supporting modern databases and applications [20, 30, 41, 52, 53, 95]. For example, Netflix deploys and optimizes Apache Cassandra [53], which is supported by LSM-trees, to effectively handle write-intensive workloads [68]. The LSM-tree improves write performance by organizing data as KV entries and deferring expensive in-place updates. It organizes data into multiple hierarchical levels, each with exponentially increasing capacities, structured as sorted runs. New KV entries are first appended to a main-memory buffer (or MemTable); when this buffer fills up, the entries are sorted, compacted, and merged as a larger sorted run into the next level. This background compaction process cascades downwards whenever a level reaches its capacity threshold.

Practical Challenge: Self-adaptation for dynamic workloads. In LSM-tree-based key-value stores, a major challenge lies in online handling dynamically changing workloads. Prior studies [12, 22, 35] have shown that real-world applications often exhibit significant workload variability, driven by daily usage patterns and operational shifts. For example, Meta analyzed access patterns from five distinct applications and found that each exhibits highly diverse workload behaviors, with substantial variation occurring even within a single day [6]. This underscores the need to efficiently manage fluctuating ratios of key lookups and entry updates. While many workload-aware methods have been proposed to optimize LSM-tree systems for a given workload, a key challenge remains unresolved for evolving workloads.

Existing workload-aware methods compute a structural configuration, including level capacities, the number of sorted runs, and their sizes to guide compactions and manage write stalls for a given workload. However, when the workload changes, the optimal configuration often changes as well, requiring the system to adapt accordingly. While methods like Moose [58] and Wacky [27] deliver excellent performance under static workloads, they do not provide mechanisms for transitioning between configurations, making them unsuitable for dynamic workloads. Naively or greedily resizing runs and merging data during such transitions may introduce latency spikes, as more aggressive write stalls [26, 64] are often required to reach the desired structure. Dostoevsky [26] not only computes a desirable configuration but also introduces a *lazy* adaptation strategy, adjusting the size and number of runs in a level only when it is fully compacted into the next. While this approach avoids costly data reorganization, it responds slowly to workload changes and depends on a sufficient number of updates to complete the transition. In contrast, Ruskey [64] proposes a *middle-ground* strategy called FLSM, which balances between greedy and lazy adaptation. It recalculates the structural configuration when performance degradation is observed and adjusts the active sorted runs during compactions at this level. Although this design accelerates responsiveness, it still relies on sufficient updates to trigger compactions, limiting its ability to adapt promptly under read-intensive workloads. In summary, the existing *recomputing and transitioning structure* approaches fail to achieve an excellent tradeoff between responsiveness to the changes and the transitioning overhead.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796208

Table 1: Comparison between ElasticLSM and existing workload-aware LSM-tree structures. The example assumes a three-level LSM-tree and a MemTable size of F.

Methods	Structural Configuration				LSM Management Actions			
	LSM structure	Size Ratios	Level Capacities	#Sorted Run	Trigger Compaction	Picked Runs	Write stall	Dynamic Workloads
Dostoevsky	Fluid Tree	$\{T, T, T\}$	$\{TF, T^2F, T^3F\}$	$\{K, K, Z\}$	Fullness of a level	Adjacent or same level(s)	#files in L0 > K	Lazy / Greedy
Ruskey	FLSM	$\{T, T, T\}$	$\{TF, T^2F, T^3F\}$	$\{K_1, K_2, K_3\}$	Fullness of a level	Adjacent or same level(s)	#files in L0 > K_1	Moderate
Moose	Generalized LSM	$\{r_1, r_2, r_3\}$	$\{r_1F, r_1r_2F, r_1r_2, r_3F\}$	$\{\sqrt{r_1}, \sqrt{r_2}, \sqrt{r_3}\}$	Fullness of a level	Adjacent levels	#files in L0 > $\sqrt{r_1}$	Not applicable
ArceKV	ElasticLSM	Removed	Removed	Removed	Workload Dependent	Runs from multiple levels	Workload Dependent	Consistently Optimizing

Our Vision: Focus on the transition procedure, not on the final structure. Existing approaches are limited by their **rigid transition actions**, often aiming to directly reach a target LSM-tree structure without considering performance during the transition. We argue that under dynamic workloads, the focus should shift from morphing into a pre-defined structure to **consistently optimizing performance** throughout the transition. While it is possible to compute the optimal LSM-tree for a given workload, blindly transitioning toward it may overlook more effective actions that yield better overall system performance.

Building on this insight, we propose two novel designs tailored to dynamic workloads:

ElasticLSM: Expanding the Transition Action Space. Existing LSM-trees rely on predefined structural configurations that fix the capacity and number of sorted runs per level, triggering compactions only when level capacity thresholds are exceeded. While this yields predictable costs, it limits flexibility under dynamic workloads. For example, proactively compacting runs across multiple levels— even when they are not full—during a read-intensive phase can further reduce runs and improve read performance. To enable such flexibility, we introduce ElasticLSM, which removes rigid limits on level capacities, run counts, and run sizes (Table 1). ElasticLSM follows a more flexible and workload-dependent policy, treating the LSM-tree as a flexible collection of sorted runs, each tagged with a timestamp, size, and key range. Compactions and write stalls can be triggered or deferred according to the current workload, and may involve runs from one or multiple levels, subject only to preserving the LSM-tree's intrinsic timestamp ordering. This expanded design allows ArceKV to explore a broader set of valid actions, opening more opportunities to optimize performance.

Arce: Lightweight Compaction Evaluation. While expanding the action space increases flexibility, it also complicates decision-making. Unlike structurally fixed LSM-trees, where compactions and stalls follow fixed rules with predictable amortized costs, the system must make online decisions in which each action impacts future ElasticLSM states and costs. This turns the search for a globally optimal action sequence into an intractable, NP-hard problem (see Section §3.3). To address this, we introduce the **A**daptive **R**untime **C**ompaction **E**ngine (Arce), a score-based evaluation framework that balances both short-term penalties and long-term benefits of compaction actions. With properly tuned parameters, this method restricts the attention to a small set of compactions that must be at least partially involved in the optimal sequence.

Based on Arce, we implement ArceKV on top of RocksDB, a widely used industrial LSM-tree storage engine, and evaluate its performance against state-of-the-art compaction policies, including Leveling [37], Tiering [53], LazyLeveling [26], Ruskey [64], and Moose [58]. Results show that ArceKV achieves high update performance comparable to update-optimized designs while also maintains top-tier read performance compared to read-optimized designs under static workloads. It also adapts rapidly to workload shifts, within 20 million operations and without exhibiting significant latency spikes. Overall, ArceKV outperforms RocksDB, the most adaptive among the baselines. Across the two evaluated workloads, ArceKV delivers an average performance improvement of 2.17× to 2.92× compared to Tiering and LazyLeveling, and 2.00× and 1.41× relative to 1-Leveling, which serves as the strongest baseline in our experiments. We further compare ArceKV with several industrial-grade databases, including Pebble [52], RocksDB [33], Cassandra [53], and WiredTiger [19]. ArceKV delivers over 10× speedup compared to Cassandra and WiredTiger, and performs 3× better than Pebble.

Contributions. In summary, we make the following contributions:

- We identify the limitations of existing compaction policies under dynamic workloads and propose a new compaction engine Arce that dynamically selects the most effective compaction and write stall threshold to adaptively balance read and write performance.
- We design a score-based model that efficiently estimates the benefit of each compaction and stall threshold pair, providing a near-optimal solution to the underlying NP-hard decision problem.
- We implement ArceKV on top of RocksDB and demonstrate its effectiveness through extensive evaluations against several state-of-the-art compaction strategies and industrial databases.

2 BACKGROUND

2.1 LSM-tree

LSM-tree is a persistent, multi-level indexing structure for key-value stores, which aims to obtain efficient write performance by transforming expensive in-place update into sequential update. All updates, insertions, and deletions are initially turned into a key-value entry and then sorted in a main memory buffer (or MemTable). It will be flushed into the disk as a new sorted run (or SSTable

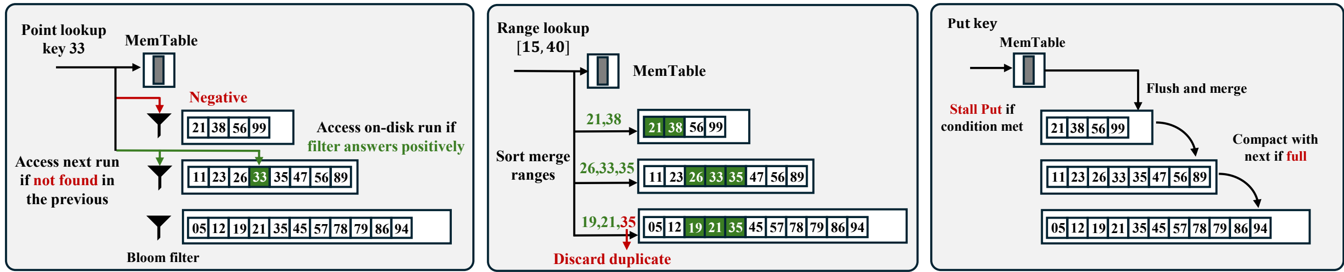


Figure 1: How the three basic operations, point lookup, range lookup, and update, are performed in an LSM-tree system.

in RocksDB, SST for short) when it is full. These SSTs are organized into several levels, with each level having a capacity T times larger than the previous one. In LSM-trees using a typical Leveling compaction policy, such as Pebble [52], SSTs at the same level are non-overlapping and collectively form a single sorted run. In contrast, Tiering-based systems like ScyllaDB [80] allow each level to maintain up to T key-overlapping sorted runs, reducing compaction size and improving write performance. LSM-based systems usually support three basic operations, shown in Figure 1:

Point Lookup. Given a key, the key-value store returns its associated value if it exists. The search proceeds by scanning each sorted run sequentially, stopping once the value is found. This process relies on the LSM-tree's timestamp ordering across levels: the smallest timestamp in the i-th level must be no smaller than the largest timestamp in the $(i-1)$-th level. Within the same level, runs may have overlapping timestamps. If multiple versions of a key exist at a given level, the system returns the most recent one based on timestamp comparison. Without this cross-level timestamp order, point lookups would require searching all levels for every query, severely degrading performance. Each sorted run is equipped with a *Bloom filter*, an in-memory structure that quickly determines whether a key may exist in the run. Its accuracy is controlled by the bits-per-key (BPK) parameter, representing the ratio of filter memory to the number of keys. The false positive rate (FPR) follows $FPR = O\left(e^{-BPK\cdot(\ln 2)^2}\right)$. Let s be the total number of sorted runs; the I/O cost of a point lookup is then $O(s \cdot FPR + 1)$.

Range Lookup. Different from point lookup, the LSM-tree retrieves all the entries within a specified key range from all the sorted runs. And then it sort merges the results from each sorted runs and produces a final result. Specifically, as most LSM-tree systems leverage *iterator* to iteratively produce the final result, which reads the first data block (usually sized one I/O block) from each sorted run and then fetches the entries one by one from each sorted runs. Suppose the search range contains l entries, each of size E bytes, and the I/O block size is B bytes, the I/O cost is $O(s + \frac{lE}{B})$.

Update. In an LSM-tree, new key-value pairs are first inserted into an in-memory buffer called the MemTable. Once the MemTable reaches its threshold size, it is flushed to disk as a new sorted run. Updates to existing keys are handled using the same out-of-place insertion mechanism, appending the new version without modifying prior entries. When the size of a level exceeds its predefined capacity, a *compaction* is triggered to merge its sorted runs with those in the next level.

Modern LSM-tree key-value systems execute queries and updates on foreground threads, while use background threads to asynchronously handle the flush and compaction when the MemTable or levels become full.

2.2 Write Stall Controller

The write stall controller is a critical component in most LSM-tree-based storage systems, including RocksDB [33], Pebble [52], Cassandra [53], and InfluxDB [47]. It controls the number of sorted runs at the first level (L0) by deliberately stalling incoming writes when they exceed a configurable threshold to maintain a designated number of sorted runs in the system. When a stall is triggered, the new incoming update will be forced to wait for several microseconds. Existing workload-aware methods [26, 27, 58, 64] stall writes when the number of sorted runs in the first level (L0) exceeds the predefined maximum in the structural configuration.

2.3 Open Challenges

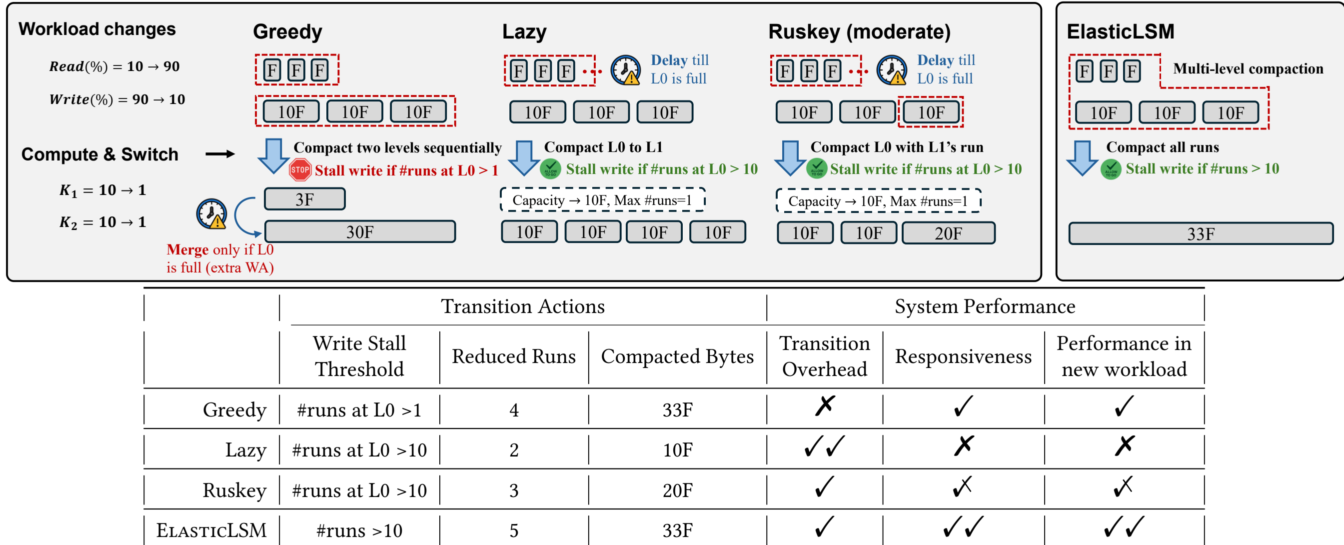
Existing approaches such as Wacky, Moose, Dostoevsky, and Ruskey can derive effective LSM configurations for static workloads, but struggle to transition between configurations with both low cost and high responsiveness. As illustrated in Figure 2, a read-intensive workload (90% reads) favors reducing the number of sorted runs from 10 to 1. A *greedy transition* adapts quickly but incurs high overhead by stalling writes when $K_1 = 1$. In contrast, the *lazy strategy* and Ruskey defer L0 adjustments, reducing transition cost but causing prolonged performance degradation before convergence.

This trade-off stems from transition mechanisms that focus only on reshaping the structure, rather than maintaining performance throughout the transition. Although a configuration may be optimal for a given workload, existing methods do not optimize system behavior during the transition itself. We argue that **sustaining optimal performance under dynamic workloads requires continuously adapting actions to the current workload and system state.**

3 ARCE: ADAPTIVE COMPACTION DECISION

To achieve this, we first decouple compaction behavior from rigid structural configuration parameters, such as fixed level capacities and prescribed run counts, which traditionally enforce a static and inflexible compaction schedule. We then introduce ElasticLSM, an enhanced LSM-tree design that enables the system to **compact runs selected across levels** and **initiate compactions at workload-dependent timing**, yielding two principal benefits:

- **Flexible Run Selection:** Merging sorted runs across multiple levels into one run in a single compaction improves responsiveness to read-intensive workloads and helps reduce write amplification. Also, selectively merging runs within a single level during write-intensive workloads reduces compaction overhead while slightly improving read performance.
- **Workload Dependent Timing:** By permitting compaction to be scheduled dynamically in response to workload conditions, the system can delay or advance compactions and write stalls as needed. This flexibility improves adaptability to workload



	Transition Actions			System Performance		
	Write Stall Threshold	Reduced Runs	Compacted Bytes	Transition Overhead	Responsiveness	Performance in new workload
Greedy	#runs at L0 >1	4	33F	✗	✓	✓
Lazy	#runs at L0 >10	2	10F	✓✓	✗	✗
Ruskey	#runs at L0 >10	3	20F	✓	✓✗	✓✗
ElasticLSM	#runs >10	5	33F	✓	✓✓	✓✓

Figure 2: The example illustrates how existing structural transition policies: Greedy, Lazy, and Moderate (Ruskey), perform during and after the transition, compared with ElasticLSM's continuous optimization approach. "Responsiveness" denotes the speed at which each method completes the transition. Performance ratings are denoted as follows: ✗ = worst, ✓✗ = mediocre, ✓ = good, and ✓✓ = best.

shifts and mitigates the risk of performance bottlenecks under write-heavy scenarios.

For example, as shown in Figure 2, by removing structural configuration parameters, ElasticLSM can compact all runs across levels in a single operation while simultaneously raising the write stall threshold. This combination avoids transition costs and delivers even better responsiveness than the Greedy approach.

In the following, we first describe how to identify action candidates after removing parameters (Section §3.1), then present a theoretical model of system cost under this setting (Section §3.2) to guide Arce in selecting the most suitable actions over time (Section §3.3).

3.1 ElasticLSM: Expanded Action Space

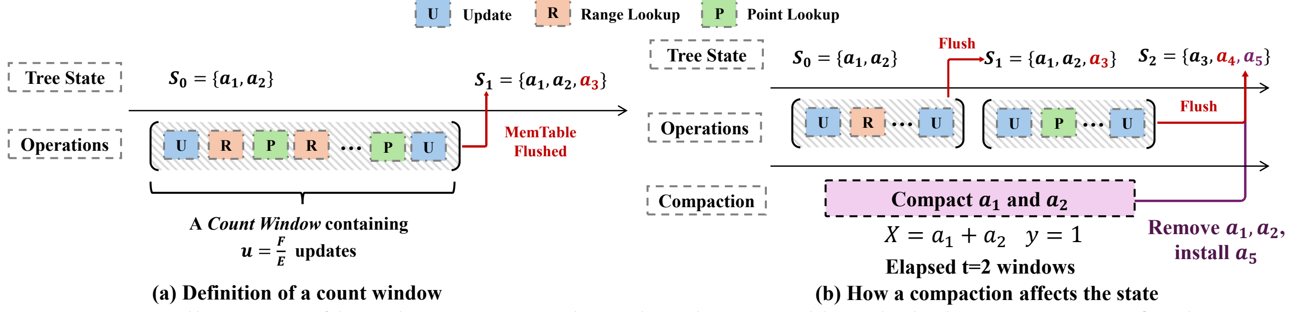
ElasticLSM maintains a collection of sorted runs across levels, each potentially varying in size and count. Without fixed structural parameters on level capacities or maximum run counts, the system must explicitly decide when and how to perform its two core management actions: compaction and write stall. Write stall in ElasticLSM is straightforward: updates are throttled only when the total number of sorted runs exceeds a tunable threshold c, with a stalling rate k. This flexibility allows the system to better balance read and write throughput. Both parameters can be tuned independently, as detailed in Section §3.4. In the following, we elaborate the more complex action – compaction.

Extensive Compaction Options. Any level can contain an arbitrary number of sorted runs of varying sizes after removing structural constraints like level capacities, sorted runs number, and run sizes. However, this flexibility does not imply that we can freely merge any subset of runs. The core requirement of an LSM-tree is to maintain timestamp ordering across levels: timestamp at shallower level must be larger than that in deeper levels, while within the same level, sorted runs can have overlapping timestamps (see Section §2). As shown in Figure 4, within the same level, the timestamp ranges of runs can be overlapped while the bottom level must have disjoint and smaller timestamps compared to the top level. Additionally, we restrict compactions to proceed downward, following the LSM-tree tradition, to avoid complicating the timestamp order of runs within a level.

Based on these rules, we identify three compaction patterns that produce valid compaction candidate set:

- **Pattern 1 (Intra-level)**: Compact any more than one sorted runs at i-th level, and place the result to the i-th level.
- **Pattern 2 (Adjacent-level)**: Compact all the sorted runs from the i-th level with zero or more sorted runs at the $(i+1)$-th level, and place the result to the $(i+1)$-th level.
- **Pattern 3 (Multi-level)**: Compact all sorted runs from the i-th to the j-th level ($j > i+1$) with zero or more sorted runs at the $(j+1)$-th level and place the result at the $(j+1)$-th level.

In general, Pattern 1 enables intra-level compaction, Pattern 2 performs traditional adjacent-level compaction, and Pattern 3 supports multi-level compaction. While these patterns enable a wide range of compaction candidates, the resulting candidate set can be extremely large and computationally expensive to process exhaustively. To address this, we apply heuristic pruning. Our observation is that, for similar sizes of compacted data, reducing a greater number of sorted runs generally yields better lookup performance. Therefore, for Pattern 1, instead of enumerating all possible combinations of runs within a level, we first sort the runs by sizes in ascending order. We then iteratively build compaction candidates by starting with the smallest run and incrementally adding one more run at a time, continuing until all runs are included. Each intermediate compaction is added to the candidate set. A similar strategy is applied for Pattern 2, where the runs in $(i+1)$-th level are also sorted and incrementally included. For Pattern 3, although a similar incremental approach can be applied, the resulting compaction candidate set can still grow to an enormous size when the total number of levels is large. Therefore, in practice, we typically



(a) Definition of a count window (b) How a compaction affects the state

Figure 3: Illustration of how the tree state evolves when the MemTable is flushed or a compaction finishes.

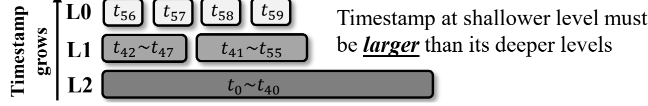


Figure 4: Example of timestamp ranges of different levels.

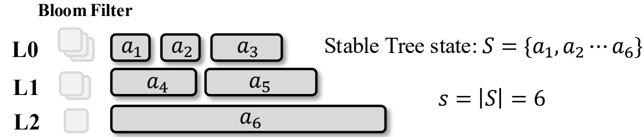


Figure 5: An example of an ElasticLSM within a count window.

limit the number of levels to fewer than 8[1]. By doing this pruning, Arce is able to rapidly find the valid compaction set in 30us in our experiment.

3.2 System Cost Modeling

Since ElasticLSM greatly expands the action space, it is crucial to understand how different compaction strategies and write stall parameters influence overall performance before making decisions. In traditional LSM-trees, operational costs are straightforward to predict because compactions and stalls follow fixed patterns. In contrast, our flexible design makes cost estimation more challenging, as the tree state (i.e., the sorted runs in the tree) can evolve by more flexible and unpredictable actions. To address this, we introduce a *Windowed-State Cost Modeling* method, which partitions the long running operation sequence into multiple state-stable windows, where tree state is generally unchanged. We then estimate the three operational costs within each window and define the rules for state transitions between consecutive windows.

Count Window: Maintaining a Stable Tree State. As discussed in Section §2, both range and point lookup costs depend on the number of sorted runs. In ElasticLSM, removing structural constraints enables more flexible compactions, but also makes the number of runs highly unpredictable. We observe that the most frequent change in run count occurs when a full MemTable is flushed to the first level (L0), whereas compactions, although they also modify the run count, generally take longer to complete. Based on this observation, we partition foreground operations into consecutive windows, each containing $u = F/E$ updates, where F is the MemTable size and E is the entry size. We term these *count windows* (or simply, windows), inspired by stream processing techniques [34]. The number of range lookup and point lookup within a window are denoted as r and p respectively. And naturally, we can describe the workload pattern by (r, u, p) tuple. Within a window, we maintain a relatively stable *tree state* S, defined as the set of sorted runs and their sizes. As illustrated in Figure 5, the example shows a stable tree state within a window containing six sorted runs of sizes a_1 to a_6 across three levels. This stable state allows us to estimate the cost of the three primary operations within the window as follows.

Operational Cost in a Window. For a point lookup, the LSM-tree may scan up to all $s = |S|$ runs to locate the target key. Each run is equipped with a Bloom filter with false positive rate α, so I/O to access a data block is required only when the filter returns a positive result. In the worst case, exactly one run yields a true positive, while the others incur I/O only on false positives with probability α. The resulting cost is given in Equation 1, where I_r denotes the I/O time to access a data block.

For a range lookup, the system first locates the start position and retrieves the corresponding block from each run, incurring a cost of $s \cdot I_r$. It then sequentially scans l entries (range length) from each run, with I/O cost $lE/B \cdot I_r$, where E is the entry size and B the data block size. Since this scanning phase depends only on l and not on the LSM-tree state, we omit it from subsequent optimization (see Equation 2).

For updates, prior methods tie write stalls to compaction, with stall time proportional to compacted bytes as dictated by structural constraints. In contrast, ElasticLSM decouples compaction from stalling: updates are slowed by a tunable rate k only when s exceeds an independent threshold c. The update cost is thus the flush I/O cost plus the stall penalty $k \cdot \mathbb{I}(s > c)$, where $\mathbb{I}$ returns 1 if $s > c$ and 0 otherwise, and I_w is the I/O time to write a block (Equation 3).

$$\textbf{Point Lookup Cost} \quad P(s) = (\alpha \cdot s + 1) \cdot I_r \tag{1}$$

$$\textbf{Range Lookup Cost} \quad R(s) = s \cdot I_r \tag{2}$$

$$\textbf{Update Cost} \quad U(s) = (F/B) \cdot I_w + k \cdot \mathbb{I}(s > c) \tag{3}$$

Evolving Tree State Between Windows. Once the cost within a single window is known, estimating the cost of the i-th window requires understanding how the run count in the tree state changes from s_{i-1} to s_i. Such changes occur through two background actions: MemTable flushes and compactions. As shown in Figure 3(a), flushing a MemTable simply adds a new sorted run of size a_3 to the state, yielding $s_i = s_{i-1} + 1$. In contrast, compaction alters the tree state more intricately, since its completion time is uncertain and typically not aligned with window boundaries. To address this, we note that a compaction in the background thread completes when the total I/O time of foreground operations equals (or exceeds) the compaction's I/O time when having sufficient I/O bandwidth.

[1] The default number of levels in RocksDB is 7.

Specifically, if a compaction of size X bytes starts in the i-th window, it will finish in the $(i+t)$-th window, where the cumulative I/O cost of foreground operations over t windows matches the compaction's I/O time. The foreground I/O time in t windows without other concurrent compactions is given in Equation 4. To preserve a stable tree state within each window, we consider the compaction to take effect in the next window after completion. The value of t is computed using Equation 5. Experimental results (Figures 11(c) and (d)) show that this rounding has minimal impact on the accuracy of theoretical cost model.

$$f(s,t) = \sum_{i=0}^{t-1} r \cdot R(s+i) + u \cdot U(s+i) + p \cdot P(s+i) \tag{4}$$

$$t = \min\left\{ t \in \mathbb{Z}^+ \,\middle|\, f(s,t) \geq \frac{X}{B}(I_r + I_w) \right\} \tag{5}$$

Example 3.1. As shown in Figure 3(b), for a compaction of size X that removes y sorted runs, if the estimated completion time is after 2 windows, its effect will be applied in the third window by removing the compacted runs (e.g., a_1 and a_2) and installing the result (e.g., a_5).

The number of sorted runs of windows evolves by:

$$s_{i+1} = \begin{cases} s_i + 1, & \text{No compaction completes at i+1 window} \\ s_i + 1 - y, & \text{Compaction reducing y runs completes} \end{cases} \tag{6}$$

3.3 Arce: Decide the Intermediate Compaction

Objective Function. Based on the cost model and state-evolution rules defined above, we can express the average cost for a given workload (r, u, p) with stall parameters c and k, after performing m compactions, as:

$$C = \frac{\sum_{i=1}^{m} f(s_i, t_i)}{\sum_{i=1}^{m} t_i (r + u + p)} \tag{7}$$

subject to the update rule:

$$s_{i+1} = s_i + t_i - y_i \tag{8}$$

Here, each compaction has X_i bytes, reduces y_i sorted runs, spans over t_i count windows, and starts with s_i sorted runs. The function $f(s_i, t_i)$ represents the cumulative cost over window t_i, as defined earlier. This problem is fundamentally a search problem to find out m compaction to minimize the average cost. Evaluating only short-term compaction candidates (i.e., small m) is computationally efficient, but tends to favor smaller, quickly completed compactions and larger stall thresholds c, which yield short-term benefits by reducing run count more rapidly. However, such strategies may overlook larger compactions that, although expensive upfront, offer substantial long-term benefits. For example, merging two 40GiB runs may yield sustained lookup improvements for the next 40GiB of inserted data. Exploring deeper compaction sequences to capture these long-term gains introduces significant computational overhead and can be proven to be NP-hard. Formal proof is provided in our technical report [1].

Lemma 3.2. *Deciding m compactions to minimize Equation 7 is NP-hard.*

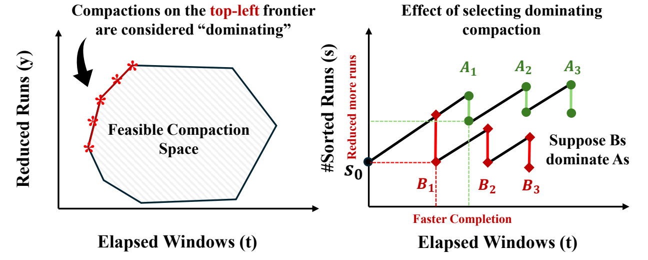


Figure 6: The left panel defines dominating compactions, while the right panel illustrates their benefits.

Fortunately, it is unnecessary to determine the full sequence of m compactions in advance. Instead, we only need to identify the first compaction to execute at each decision point. This raises a key question: Can we design a principled method to quantify both the short-term penalty and long-term benefit of an intermediate compaction candidate?

Short-Term Effect. The immediate drawback—or penalty—of executing a compaction is that it occupies a background compaction worker, potentially causing SSTs to accumulate at L0. This accumulation can degrade read performance and even trigger a write stall. We model the short-term cost as:

$$E_s(s,t) = \overbrace{I_r \cdot t \cdot (r + \alpha \cdot p)}^{\text{Read slowdown}} + \underbrace{uk \cdot \max\left(0, s + t - c\right)}_{\text{Write stall penalty}} \tag{9}$$

Here, t denotes the estimated duration of compaction with sorted runs s in the system.

Long-Term Effect. Compaction reduces the number of sorted runs, which benefits all future reads within the current decision window. We define the long-term benefit of a compaction that reduces y sorted runs as:

$$E_l(y) = (r + \alpha \cdot p) \cdot I_r \cdot y \tag{10}$$

Effectiveness Score. By integrating both effects, we define the overall effectiveness of a compaction spanning t windows and reducing y runs as:

$$E(s,t,y) = M \cdot E_l(y) - E_s(s,t) \tag{11}$$

The parameter M scales the long-term benefit and is determined by the current tree state, workload characteristics, and write stall threshold. Section 3.4 provides guidance on selecting the appropriate (M, c, k) under different scenarios. Given a fixed (M, c, k), Arce can select the compaction with the highest effective score among many compaction candidates.

Optimality Analysis. The effectiveness score not only significantly improves the efficiency of compaction selection but also reveals an important structural property among compaction candidates—domination. Formally, we say that compaction A dominates compaction B (denoted as $B \prec A$) if and only if A reduces more sorted runs as B while requiring less compaction time. Using the score-based evaluation defined in Equation 11, Arce ensures that only non-dominated candidates are selected under any given parameter configuration (M, c, k). We refer to these as *dominating compactions*, which collectively form the left frontier in a two-dimensional space, where the x-axis represents elapsed time t and the y-axis represents the number of reduced sorted runs y, as illustrated in Figure 6.

Algorithm 1: FindBestParams(M, c, k)

Input: Current tree state S and workload (r, u, p)
Output: Best parameters (M, c, k)
1 bestCost $\leftarrow \infty$;
2 bestM, bestc, bestk $\leftarrow$ null ;
3 **foreach** *valid* (M, c, k) **do**
4 totalCost $\leftarrow 0$;
5 $S' \leftarrow S$;
6 **for** $i \leftarrow 0$ **to** *MaxIterTime* **do**
7 Select compaction reducing y runs and spanning t windows based on (M, c, k) ;
8 totalCost $\leftarrow$ totalCost + f(|S'|,t);
9 totalOps $\leftarrow$ totalOps + $t \cdot (r + u + p)$;
10 $S' \leftarrow S'$ removes compacted runs and installs result ;
11 avgCost $\leftarrow$ totalCost / totalOps
12 **if** *avgCost* < *bestCost* **then**
13 bestM $\leftarrow M$;
14 bestc $\leftarrow c$;
15 bestk $\leftarrow k$;
16 bestCost $\leftarrow$ avgCost ;
17 **return** (*bestM*, *bestc*, *bestk*)

Lemma 3.3. *If $A \prec B$, the effectiveness score of A is less than B.*

Proof. The long-term effect of B will increase by $(r + \alpha \cdot p) \cdot I_r \cdot (y_2 - y_1)$, while the short-term effect of B will decrease by at least $I_r \cdot (t_1 - t_2) \cdot (r + \alpha \cdot p) + uk \cdot \max(0, t_1 - t_2 + s - c)$. Therefore, the effectiveness score of B is larger than A. □

In our extended version [1], we show that every dominating compaction can be selected by some (M, c, k), and at least part of the dominating compaction must occur in the optimal sequence.

3.4 Parameter Selection

Achieving approximately optimal compaction selection requires properly setting the parameters. However, determining the optimal values of the three parameters (M, c, k) over time is itself an NP-hard problem. Fortunately, it is not necessary to determine all parameters simultaneously. Instead, we only need to ensure that the parameter values chosen at each decision point are suitable, and we can update them periodically as the system evolves. To this end, we adopt a simple yet effective simulation-based approach. We iteratively explore a wide range of candidate (M, c, k) combinations and evaluate their effectiveness by simulating continuous compaction decisions. For each configuration, we estimate the average system cost over a sufficiently long period. The parameter set yielding the lowest cost is then selected. This process is detailed in Algorithm 1.

The underlying intuition is that when the tree state (e.g., total data volume and number of sorted runs) and the workload remain relatively stable, there exists a tuple of parameters (M, c, k) that can continuously guide the selection of the most suitable compactions to minimize system cost. A new parameter tuple is required only when any of them varies beyond a predefined recomputing threshold d ($d \in (0, 1)$). In our implementation, we use $d = 0.1$, which strikes a balance between simulation overhead and responsiveness, ensuring

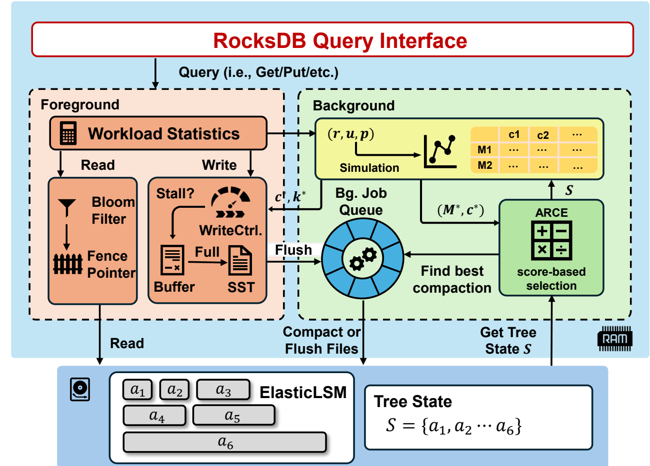


Figure 7: Overview of the architecture of ArceKV

satisfactory performance without frequent re-selection. A detailed evaluation of this threshold is provided in Section §5. To further reduce simulation time, we employ several optimization techniques, including candidate pruning and multi-threaded computation, as described in Section §4.

4 ARCEKV: WORKLOAD-DRIVEN KV STORE

As shown in Figure 7, ArceKV, built upon Arce introduced earlier, incorporates four key components: Workload Statistics, ElasticLSM, Background Simulation, and Arce's score-based compaction selection. The Workload Statistics module monitors operations and reports windowed counts (r, u, p) every one million operations. The Background Simulation module implements Algorithm 1, outputting c and k to RocksDB's write controller to manage write stalls, and forwarding M, c, and k to Arce's score-based selection module to determine optimal compactions according to Equation 11. ElasticLSM manages SSTables organized into sorted runs and levels, which Arce dynamically compacts based on these decisions. All components are implemented within RocksDB. The following section details the implementation.

Parallel Simulation. Unlike tree traversal, the score-based simulation is inherently parallelizable, as each (M, c, k) tuple can be evaluated independently. Distributing the computation across multiple threads can significantly accelerate the algorithm. By default, ArceKV uses 16 threads to run these simulations, which complete well within a single window. Additionally, the effectiveness scores are computed through linear transformations of compaction size X, reduced runs y, and elapsed windows t. This structure allows us to vectorize the score computation across all compaction candidates using the `Eigen` library [39]. `Eigen` applies SIMD (Single Instruction, Multiple Data) optimizations under the hood, further improving simulation efficiency.

Parameter Pruning. To reduce computational overhead, we adopt coarse-grained parameter tuning. For M, we use a step size of 5, as nearby values yield similar compaction choices; for c, we use a step size of 2, since closely spaced (M, c) pairs produce comparable results. We also set upper bounds: M is capped at the smallest value selecting the compaction with the largest reduction in sorted runs (upper-right candidate in Figure 6(a)), and c is limited to less than $4\times$ the current run count, as exceeding this should already

Table 2: Operation Ratios Composition

	A	B	C	D	E	F	G	H	I	J
range(%)	98	1	1	49	2	49	40	40	20	33
update(%)	1	98	1	2	49	49	40	20	40	33
point(%)	1	1	98	49	49	2	20	40	40	33

trigger re-selection based on the change threshold in the previous section. For the write stall penalty k, performance changes significantly only when it is doubled or halved. Thus, we initialize $k = 6$ (RocksDB's default stall rate) and test two additional values by successive doubling, as finer granularity provides diminishing returns. Finally, we cap the simulation iterations for each parameter tuple at `MaxIterTime` (typically 400) to ensure completion before the tree state drifts, while keeping the duration long enough to capture long-term benefits. Under a balanced workload ($r = u = p$), a window lasts over 200 ms and simulation completes within 150ms, keeping the system responsive during parameter selection.

Multi-threading Extension. The cost model in Section §3.2 assumes a single foreground thread and one background compaction worker, whereas real systems typically execute multiple foreground threads concurrently. Although the per-operation I/O cost remains unchanged, parallel execution reduces the number of elapsed compaction windows. Following Cosine [14], we model this effect using Amdahl's Law [5]. With η cores and parallelizable fraction ϕ, the speedup is $g = \frac{1}{1-\phi(1-1/\eta)}$, yielding an adjusted compaction duration $t' = t/g$. Based on profiling, we set $\phi = 0.5$.

To support multiple background workers, ArceKV tracks available compaction threads. When multiple workers are idle, newly flushed SSTables are assumed to compact immediately and no penalty is applied; otherwise, a delay penalty is introduced. The system also monitors thread, memory, and I/O utilization, applying a large penalty when resources become saturated.

5 EVALUATION

This section presents the experimental evaluation of ArceKV. All experiments are conducted on a machine equipped with an Intel Core i9-13900K CPU (5.40GHz), 128GB of RAM, and a 1TB NVMe SSD, running 64-bit Ubuntu 22.04 with an ext4 file system. To simulate realistic deployment scenarios, where not all system memory is allocated to RocksDB (e.g., TiKV recommends allocating 70% [69]), We follow Disco [106] in using the cgroup command to cap total memory usage at 75GiB. In terms of baselines, we include both widely adopted compaction strategies, such as Leveling [37], 1-Leveling [33], and Tiering [53], and recently published academic state-of-the-art LSM-tree systems with available artifacts, including LazyLeveling [26], Moose [58], Ruskey [64], and CAMAL [98]. These systems are all capable of handling point lookups, range queries, and updates issued in random order and speed using a unified handler thread. To ensure fair comparison and practical generalizability, we adopt RocksDB's default configuration, using one background compaction thread and one flush thread. Additionally, to broaden the evaluation of effectiveness, we include industrial-grade systems such as Pebble [52], WiredTiger [19], and Cassandra [53] as baseline

Baselines. The following systems and compaction strategies are used as baselines:

- **Leveling (abbr. Lvl)**: Maintains at most one sorted run per level and increases level capacity using a fixed size ratio T. This policy is optimal for read-intensive workloads.
- **Tiering (abbr. Tier)**: Allows up to T sorted runs per level, also growing capacity by size ratio T. It is designed to favor write-intensive workloads by minimizing compaction overhead.
- **1-Leveling (abbr. 1-L)**: The default compaction style in RocksDB. Unlike traditional Leveling, it allows up to 20 sorted runs at the first level, making it particularly effective for read-heavy workloads and also adaptive to workloads with a portion of writes.
- **LazyLeveling (abbr. LL)**: Structurally similar to Tiering but maintains only one sorted run at the largest level. This hybrid design improves performance for mixed read-write workloads.
- **Moose (abbr. MSE)**: Leverages a dynamic programming algorithm to configure an LSM-tree structure that achieves an optimal balance among point lookups, range queries, and updates based on the given workload.
- **Ruskey (abbr. RKY)**: Uses a reinforcement learning (RL) model to guide structural transitions, reducing the overhead of adapting to new workloads. Ruskey fixes the size ratio at $T = 10$, while the number of sorted runs at each level is determined by the RL policy.
- **CAMAL (abbr. CAM)**: Employs active learning to optimize structural parameters, such as size ratio and the number of sorted runs per level as well as the allocation of memory to Bloom filter, cache, and Memtable.

Implementation of Baselines. To ensure a fair comparison, all compaction policies except Ruskey[2] and CAMAL[3] are implemented on top of the Moose codebase[4], a framework based on RocksDB that supports configurable numbers of sorted runs and level capacities. Leveling, Tiering, and LazyLeveling are implemented using this framework, each configured with a size ratio of $T = 10$. For Moose, the size ratios and the number of sorted runs per level are determined by its dynamic programming algorithm. In our evaluation, we set these values to 40, 40, 41 for size ratios and 6, 6, 6 for sorted runs, ensuring they accommodate the total number of entries to be ingested. **ArceKV (abbr. ARC)** implements on top of RocksDB and sets the total level to 4, the I/O cost $I_w = 15$us and $I_r = 12$us based on the system profiling result. We use a Bloom filter with 10 bits-per-key for all baselines, following RocksDB's default implementation. Keys are fixed at 24 bytes, and values at 1,000 bytes and the write buffer is 2MB which is consistent with configurations commonly used in prior studies [26, 58, 64].

Dynamic Workload Generation. To construct realistic dynamic workloads, we primarily follow the approach used in Ruskey, which involves combining multiple sub-workloads with varying read-write ratios. To further enhance flexibility and realism, we distinguish between point lookups and range lookups when composing these workloads. We adopt several typical workload configurations evaluated in Endure [43] and Moose [58], with their compositions summarized in Table 2. We define two primary compound workloads:

[2]The implementation of Ruskey was obtained from the original authors. It is built on RocksDB.

[3]https://github.com/NTU-Siqiang-Group/CAMAL

[4]https://github.com/NTU-Siqiang-Group/MooseLSM

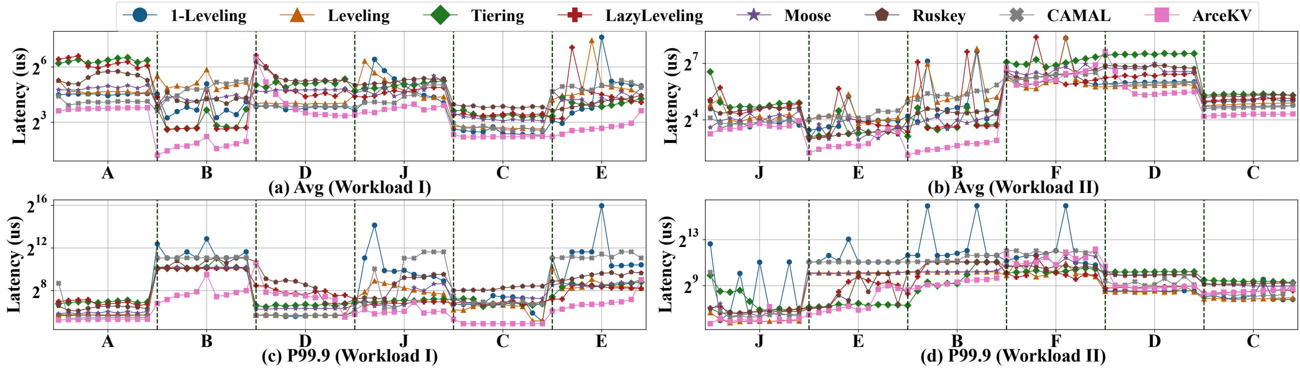


Figure 8: Average latencies and P99.9 latencies for all the methods under Workload I and Workload II.

	Workload I						
	A	B	D	J	C	E	AvgTput
1-L	3.52x	2.29x	**3.20x**	1.14x	2.60x	1.04x	1.45x
Lvl	3.07x	1.00x	2.78x	1.11x	2.21x	1.00x	1.19x
Tier	1.00x	4.21x	1.31x	1.16x	1.35x	2.50x	1.03x
LL	1.08x	3.65x	1.20x	1.47x	1.35x	1.33x	1.00x
MSE	2.76x	1.85x	1.68x	1.26x	1.57x	1.91x	1.40x
RKY	1.82x	1.92x	1.00x	1.00x	1.00x	2.39x	1.07x
CAM	4.65x	1.25x	3.07x	1.68x	2.23x	1.29x	1.57x
OURS	**6.04x**	**10.60x**	1.96x	**2.81x**	**3.08x**	**5.89x**	**2.92x**
	Workload II						
	J	E	B	F	J	C	AvgTput
1-L	2.18x	1.18x	1.26x	1.45x	2.84x	1.24x	1.53x
Lvl	1.74x	1.00x	1.00x	1.53x	**3.19x**	1.42x	1.51x
Tier	1.00x	1.72x	4.34x	1.00x	1.00x	1.00x	1.00x
LL	1.23x	1.18x	1.48x	1.37x	2.17x	1.20x	1.37x
MSE	1.99x	1.76x	4.10x	1.39x	1.73x	1.54x	1.53x
RKY	1.45x	2.00x	3.00x	1.59x	1.58x	1.05x	1.42x
CAM	2.30x	1.01x	1.54x	1.67x	2.99x	1.61x	1.72x
OURS	**2.74x**	**2.60x**	**10.62x**	**1.64x**	2.79x	**2.10x**	**2.17x**

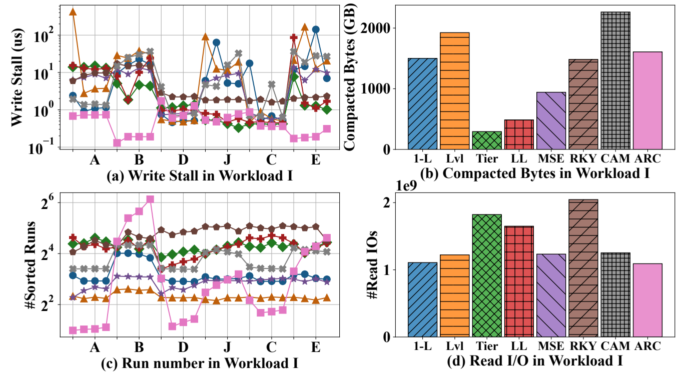


Figure 9: The table (left) shows the normalized throughput of each sub-workload in Workload I and Workload II; The figures (right) presents the change of write stall, total compaction bytes, the change of sorted runs, and the total read I/O when executing Workload I.

- **Workload I**: This compound workload consists of A, B, D, J, C, and E, with each sub-workload containing 40,960,000 operations, totaling 245,760,000 operations. It inserts approximately 74GiB of new data.
- **Workload II**: This workload comprises J, E, B, F, D, and C, with 40,960,000 operations per sub-workload, totaling 245,760,000 operations and inserting 94GiB new data.

The workload ratios in Workload I shift more abruptly—from a high write ratio to a low one—while Workload II exhibits a more gradual transition, with the write ratio slowly decreasing over time. For multi-threaded evaluation and comparison with industrial KV stores, we use Workload III, composed of G, H, and I workloads, each with 20,480,000 operations. This setup avoids extreme read- or write-heavy cases that are less reflective of production scenarios. Before running any test workload, we preload the system by sequentially writing 40GiB of data (about 40 million entries) to ensure a stable compaction state, enabling fair evaluation of write-friendly strategies like Tiering and LazyLeveling.

5.1 System Performance

Overall, ArceKV demonstrates consistently strong performance under both Workload I and Workload II. Figure 8 compares seven methods across two long-running workloads, Workload I and Workload II, each composed of multiple subworkloads with diverse read-write ratios, totaling up to 140GiB of data. Across both workloads, ArceKV consistently maintains top-tier performance, ranking first in nearly all subworkloads, except during the transition from workload B to workload D. This specific transition involves a sharp shift from an extremely write-intensive workload (B) to a read-intensive one (D), posing a significant challenge for systems to adapt promptly. Despite this abrupt change, ArceKV still performs competitively, only slightly trailing behind the two read-optimized baselines, 1-Leveling and Leveling. In Workload II, the transition from F to D is less drastic. Here, ArceKV performs similarly with 1-Leveling and Leveling. While these two baselines are optimized for read-heavy workloads, they struggle in write-intensive scenarios due to frequent and aggressive compactions. Conversely, Tiering and LazyLeveling allow more sorted runs per level and adopt lazier compaction strategies, reducing write stall and performing well under write-heavy workloads, but at the cost of degraded read performance due to excessive run accumulation.

Moose does not offer a robust solution for adapting structural configurations across varying workloads. The configurations it generates differ significantly (e.g., size ratios changing from 7,7,7,6,5 to 24,23,24), making direct transitions between them impractical without incurring severe performance degradation. To mitigate this, we compute a unified configuration for Moose based on the total number of inserted entries and the average workload composition. Similarly, since CAMAL is also designed for static workloads, we provide its trained model with the average workload composition to derive the structural parameters. While both CAMAL and Moose perform well overall under this setting, their performance deteriorates significantly in extreme workloads such as B and E, falling

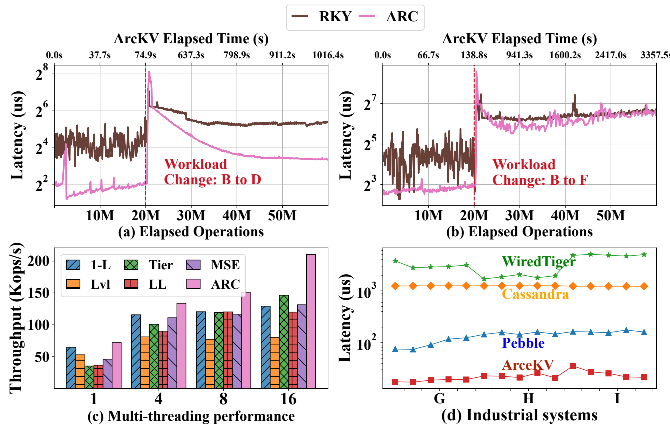


Table 3: Performance comparison under diverse query distribution workloads in the YCSB benchmark.

	Normalized Throughput					
	YCSB-A	YCSB-B	YCSB-C	YCSB-D	YCSB-E	YCSB-F
1-L	1.44×	1.00×	1.94×	1.00×	3.89×	1.42×
Lvl	1.00×	1.25×	1.53×	1.13×	2.79×	1.00×
Tier	2.77×	1.25×	1.04×	1.14×	1.00×	2.77×
LL	2.11×	1.29×	1.00×	1.24×	1.07×	2.17×
MSE	1.87×	1.84×	1.62×	1.57×	2.89×	1.89×
RKY	2.42×	2.05×	1.55×	1.25×	1.45×	2.72×
CAM	1.37×	2.03×	1.49×	1.60×	2.45×	1.37×
OURS	**4.18×**	**2.26×**	**2.34×**	**1.89×**	**5.47×**	**4.21×**

Figure 10: (a) and (b) illustrate the stabilization speed of ArceKV; (c) reports its performance under multi-threaded workloads; and (d) compares ArceKV with other industrial key–value stores. The table on the right summarizes ArceKV's performance under skewed workloads (e.g., YCSB). Throughputs (operations per second) of all systems are normalized relative to the minimum value observed in each workload.

well behind the top-performing baselines. In contrast, Ruskey dynamically computes the most suitable configuration and employs an efficient adaptation strategy. While Ruskey performs satisfactorily in scenarios with gradual workload changes or sufficient updates, its adaptation can lag under abrupt shifts or under read-intensive workload. As shown in Figure 9(a) and (b), Ruskey achieves a convergence rate comparable to ArceKV under workload F with sufficient updates, but adapts more slowly during transitions to highly read-intensive workloads such as D.

Furthermore, as shown in Figure 8(c) and (d), ArceKV maintains consistently low P99.9 latency over time, without incurring significant write stall or read overhead compared to other methods. This demonstrates the robustness and stability of ArceKV under varying workload conditions.

ArceKV orchestrates compactions and stalls more intelligently through the flexible LSM structure ElasticLSM. Figure 9 presents additional metrics from Workload I to highlight the benefits of ElasticLSM's structural flexibility. Under read-intensive workloads such as A and D, ElasticLSM is able to reduce the number of sorted runs more quickly than even Leveling by performing multi-level compactions. In contrast, under write-intensive workloads like B, ElasticLSM defers compactions more aggressively than Tiering and LazyLeveling by allowing more sorted runs to accumulate in the system. This enables ArceKV to maintain relatively low write stall time while still performing compactions as eagerly as Leveling when necessary. As a result, ArceKV achieves nearly the same read I/O efficiency as 1-Leveling over time, demonstrating its ability to strike a dynamic balance between compaction aggressiveness and write stall control.

ArceKV exhibits superior scalability under concurrent workloads than other baselines. As shown in Figure 10(c), we evaluate six methods (excluding Ruskey and CAMAL) using Workload III under 1, 4, 8, and 16 foreground threads. Ruskey and CAMAL are omitted because they rely on collecting performance metrics from the system and lacks synchronization mechanisms for multi-threaded scenarios. Additionally, the Moose framework does not support multiple background workers, so we fix the number of background threads to one and vary only the number of foreground query threads. Overall, as the number of threads increases, all methods show improved throughput, but ArceKV achieves the greatest gains. This is because higher thread counts intensify updates, accelerating sorted run accumulation. Methods like 1-Leveling and Leveling struggle with frequent write stalls due to low stall thresholds, while Tiering and LazyLeveling perform better thanks to higher thresholds. In contrast, ArceKV dynamically adjusts its stall threshold and compaction scheduling (Section §4) to account for multi-threading, effectively balancing read and write performance and delivering the highest scalability.

ArceKV outperforms other industrial key-value stores. As shown in Figure 10(d), ArceKV achieves the highest throughput when evaluated against Pebble, WiredTiger, and Cassandra on Workload III. WiredTiger offers strong write performance but suffers on range lookups due to its size-tiered compaction, which merges up to 15 chunks [67], leading to more sorted runs and higher lookup overhead. Cassandra incurs substantial local overhead from its distributed consistency mechanisms, resulting in lower performance. Pebble performs slightly worse than RocksDB, likely due to Go's garbage collection and system call costs. In contrast, ArceKV efficiently manages on-disk data and dynamically adjusts write stalls, consistently outperforming these baselines.

ArceKV delivers strong performance under query-skewed workloads. To evaluate ArceKV's ability to handle skewed query distributions, we benchmark it against six alternative methods using the YCSB suite. The workloads include: YCSB-A (Read-Write balanced), YCSB-B (Read-heavy), YCSB-C (Read-only), YCSB-D (Read-heavy with latest keys), YCSB-E (Range-heavy with latest keys), and YCSB-F (Read-Update balanced). Among them, workloads YCSB-C and YCSB-B follow a Zipfian distribution, YCSB-D focuses on the most recent insertions, while YCSB-A, YCSB-E, and YCSB-F use a uniform distribution. Across all workloads, ArceKV consistently achieves top-tier performance.

ArceKV performs exceptionally well in write-read mixed workloads because it avoids re-simulating parameter configurations when the workload shifts, resulting in consistent and robust performance. Moreover, by broadening the compaction space through ElasticLSM, ArceKV is able to identify more effective actions to optimize system behavior.

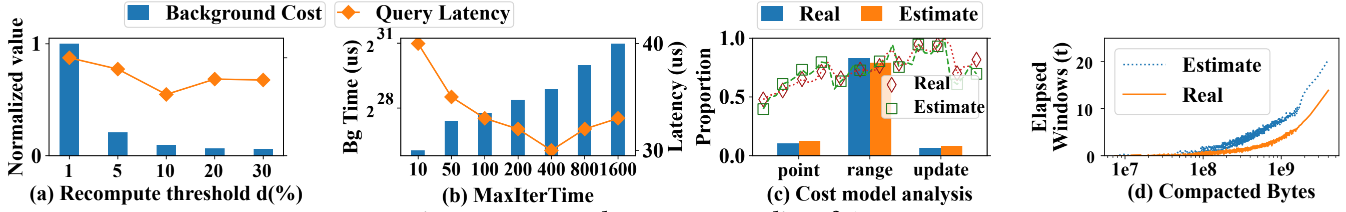


Figure 11: Internal parameter studies of ArceKV

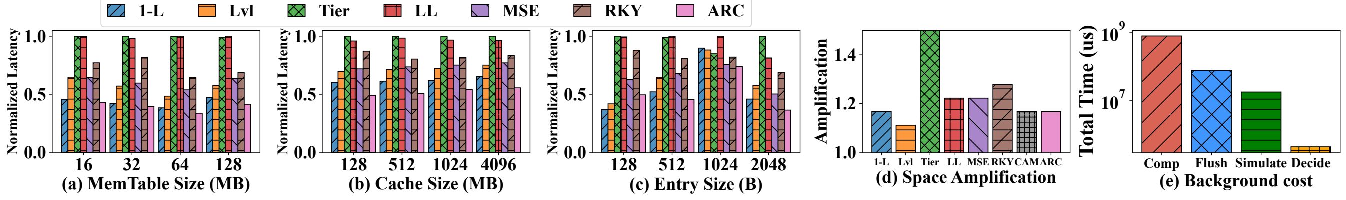


Figure 12: Common LSM parameter studies

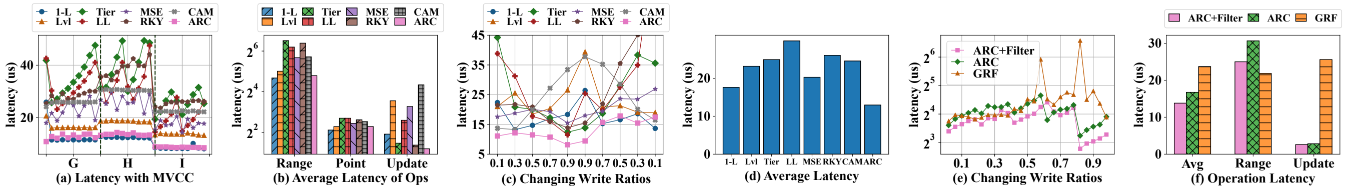


Figure 13: Case studies of various cases: (a) and (b) are the latencies of long multi-versioned chain scenario; (c) and (d) are latencies of continuous workloads; (e) and (f) are latency of ArceKV integrated with range filter.

5.2 Parameter Studies

Recomputing Threshold d Figure 11(a) shows the background cost and foreground query latency under different values of the recomputing threshold d, which determines how much the tree state or workload must change before recomputing the parameter tuple (M, c, k). When d is too small, parameters are recomputed too frequently, resulting in substantial background overhead that may slightly impact foreground query latency. As d increases, the background cost drops sharply and stabilizes beyond $d = 0.1$, where query latency also reaches its lowest point. This suggests that $d = 0.1$ offers a good balance between minimizing recomputation cost and maintaining responsiveness to dynamic workload changes.

Simulation `MaxIterTime`. This parameter controls the number of iterations (i.e., compaction decisions) used to evaluate the effectiveness of a given parameter tuple (M, c, k) under the current workload and tree state in the simulation. As shown in Figure 11(b), setting `MaxIterTime` too high can prolong the simulation and delay the timely adaptation of parameters, potentially degrading performance. Conversely, setting it too low may fail to capture the long-term effects of compactions, leading to suboptimal parameter choices. Through empirical evaluation, we find that 400 iterations offer a good balance between responsiveness and evaluation accuracy.

Justification of ArceKV's Cost Model. To evaluate the accuracy of the cost model proposed in Section §3, we compare the model's predictions against actual system latency during the execution of sub-workload J in Workload II. As shown in Figure 11(c), the predicted cost closely follows the trend of the observed latency over time. For each operation, we decompose both the theoretical and real costs into three categories: point lookups, range lookups, and updates. The results indicate that the estimated cost for each operation type aligns well with the actual measured latency, validating the effectiveness of our model.

Accuracy of Estimating Compaction Windows. In Section §3.2, we estimate the number of count windows spanned by a compaction using Equation 5. Figure 11(d) presents over 500 estimated compaction durations, ranging from 4MB to nearly 20GB, compared against their actual elapsed windows when running sub-workload J in Workload II. The results indicate that while estimation accuracy decreases slightly for larger compactions with longer execution times, the absolute error remains bounded within 3 windows.

Common Parameter Studies of LSM-Trees. To assess the robustness of our method, we evaluate ArceKV using Workload III under a variety of commonly used LSM-tree configurations. As these parameters are tuned by CAMAL's model, we exclude it from this evaluation. These include buffer sizes ranging from 16MB to 128MB, cache sizes from 128MB to 4096MB, and entry sizes from 128B to 2048B, presented in Figure 12(a) to (c). Across all 12 tested configurations, ArceKV consistently delivers the strongest or near-best performance, demonstrating its adaptability and resilience to diverse system settings.

Space Amplification. Although ArceKV focuses on read and write performance rather than space efficiency, its space amplification remains well controlled, as shown in Figure 12(d). Under workload J, which includes 36GiB of new updates and 18GiB of duplicates, Leveling yields the lowest amplification, but ArceKV follows closely with only a 0.05 increase. This is because ArceKV's lookup-driven compactions still merge overlapping runs, effectively removing duplicates and limiting space growth.

Background Cost in ArceKV. Figure 12(e) shows the CPU time spent on background tasks in ArceKV, including compaction, flushing, simulation, and decision-making. With suitable values for d and `MaxIterTime`, simulation adds only 2% overhead, and decision-making contributes less than 1%, due to the efficiency of score-based selection and SIMD acceleration.

5.3 Case Studies

Read Amplification under MVCC. When updates concentrate on a small key set, write amplification stays low but long version chains can hurt read performance. To evaluate ArceKV under this condition, we simulate a multi-version workload with 1 million hot keys over 100 million operations, following Workload III (updates, point, and range lookups). We report average latency and read amplification, as requested by the reviewer, in Figure 13(a) and (b). In theory, Leveling and 1-Leveling are well suited to this setting due to the small size increase (1 GiB) and aggressive pruning. Similarly, ElasticLSM detects small incoming runs, assigns low compaction penalties (Equation 11), and converges to Leveling-like behavior, achieving comparable read latency and amplification.

Continuously Changing Workload. We gradually vary the write ratio from 10% to 90% and back to 10%, creating a smooth workload gradient rather than abrupt changes. At each ratio, we issue about 500,000 mixed read and write operations to reflect realistic fluctuations. As shown in Figure 13(c) and (d), ArceKV achieves the best or near-best performance throughout execution and the lowest average latency. These results demonstrate that ArceKV maintains stable, adaptive performance under smoothly evolving workloads, enabled by its workload-aware compaction and write-stall mechanisms.

ArceKV with Range Filters. As shown in Figure 13 (e) and (f), to assess how range filters [16, 21, 83] enhance ArceKV, we integrate Grafite [21], a lightweight and easily pluggable range filter, and compare it with GRF [83], a state-of-the-art LSM-tree system featuring range filtering. Grafite in ArceKV is serialized into SSTables during compaction and deserialized during queries. Additionally, we record the effective ratio of the filters (defined as the reduction in run accesses divided by the total number of run accesses) and multiply this ratio by the cost of range lookups. We evaluate mixed workloads with update ratios from 10% to 90%. Overall, ArceKV+Grafite improves throughput by about 20% over the original version and achieves roughly half the latency of GRF. The gap arises because GRF's LazyLeveling policy improves point lookups and updates but adapts poorly to shifting update ratios, whereas its advanced position encoding gives it an edge in pure range lookups. Incorporating such encoding into ArceKV remains a direction for future work.

6 RELATED WORK

Key-value stores. Over the past decade, extensive research has advanced key-value stores. Hardware-focused studies [9, 32, 54, 82, 85, 92, 93, 102] optimize for modern storage technologies, including advanced SSDs [17, 32, 92, 93, 99], RDMA [86, 88], non-volatile memory [48, 54, 103, 107], and disaggregated memory architectures [40, 77, 91]. These systems improve parallelism [85] and write throughput [9, 82, 102] by aligning architectures with hardware capabilities. In cloud environments, several works [45, 49, 79] integrate cloud-specific overheads into system design, while others rethink key-value store architectures entirely [3, 55, 63, 88, 101, 110], enabling new system-level innovations. Unlike our work, these efforts target diverse environments and architectures, rather than focusing specifically on LSM-tree optimization.

Optimization of LSM-based key-value stores. There has been rapid progress in optimizing LSM-trees, driven by rethinking their core components. Innovations include advanced compaction policies [14, 26, 27, 43, 44, 46, 58, 66, 74, 84, 105] and update-friendly compaction schemes [29, 71, 73, 75, 90, 96, 97], both aiming to balance write amplification and responsiveness. Other work improves filtering structures like Bloom filters [25, 28, 56, 70, 104, 108, 109], range filters [16, 51, 62, 100], and cache policies [87, 89] to cut unnecessary I/O. Additional directions leverage emerging hardware [4, 31, 57, 81, 82, 85, 102], bridge LSM-trees with update-in-place architectures [99], and optimize via key-value separation [23, 24, 59], disaggregated storage [10], selective flushing [7], and improved memory/concurrency management [11, 36, 50, 61, 78]. Other studies target tail-latency reduction [8, 60, 75], exploit data characteristics [2, 65, 72, 94], or adapt LSM-trees to cloud environments [14, 15, 42]. Most, however, optimize LSM-based key-value stores without explicitly incorporating workload characteristics into their design.

Cost-Model-Based Compaction Optimization. Unlike traditional Leveling and Tiering policies, recent LSM-tree optimization techniques employ cost models that quantify the cost of different operations to identify the most efficient compaction strategy [26, 27, 58, 64, 66, 84, 98]. These approaches can be broadly classified into two categories: workload-unaware and workload-aware methods. The former seek generally robust performance across diverse workloads, whereas the latter aim to achieve optimal efficiency for specific workload patterns. Workload-unaware methods such as LazyLeveling [26], EcoTune [84], Moose [58], Spooky [29], and QLSM-Bush [27] model the costs of various operations and theoretically determine structural parameters or compaction behaviors that provide stable performance across diverse workloads. In contrast, workload-aware approaches such as Vertiorizon [66], CAMAL [98], Endure [43], and SMoose [58], explicitly incorporate the operation probabilities of a given workload into their cost models, enabling more accurate parameter tuning for targeted scenarios. However, while workload awareness improves specialization, adapting between parameter configurations as workloads shift can incur substantial transition costs. Ruskey [64] addresses this issue using reinforcement learning to mitigate transition inefficiencies. By contrast, ArceKV not only considers workload composition but also explicitly models and optimizes the *transition process* itself, enabling continuous performance optimization rather than discrete parameter transitioning.

7 CONCLUSION

Existing LSM-based key-value stores perform poorly under dynamic workloads due to static compaction and write-stall policies. We propose ArceKV, a workload-driven system that adaptively schedules compactions and adjusts write-stall thresholds based on the current workload and LSM-tree state. Experiments show that ArceKV consistently outperforms state-of-the-art approaches under evolving workloads.

8 ACKNOWLEDGEMENT

This research is supported by the Ministry of Education, Singapore, under its AcRF Tier 2 programme (MOE-T2EP20224-0005), and NTU SUG-NAP (022029-00001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] 2025. Technical Report of ArceKV. https://www.arxiv.org/pdf/2508.03565.

[2] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. 2018. Lightweight cardinality estimation in LSM-based systems. In *Proceedings of the 2018 International Conference on Management of Data*. 841–855.

[3] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. 2019. Fast key-value stores: An idea whose time has come and gone (HotOS'19 talk slides).

[4] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.

[5] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities *(AFIPS '67 (Spring))*. Association for Computing Machinery, New York, NY, USA, 483–485. doi:10.1145/1465482.1465560

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. doi:10.1145/2254756.2254766

[7] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 363–375.

[8] 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)*. 753–766.

[9] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1544–1556.

[10] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–316.

[11] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1863–1875.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[14] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.

[15] 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. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–28.

[16] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An Optimal Disjoint Segmented Learned Range Filter. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1911–1924.

[17] Yen-Ting Chen, Ming-Chang Yang, Yuan-Hao Chang, Tseng-Yi Chen, Hsin-Wen Wei, and Wei-Kuan Shih. 2018. Co-optimizing storage space utilization and performance for key-value solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 1 (2018), 29–42.

[18] Cloudflare, Inc. 2024. Cloudflare Workers KV. https://developers.cloudflare.com/workers/runtime-apis/kv/.

[19] Source Code. 2024. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[20] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[21] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming Adversarial Queries with Optimal Range Filters. *Proc. ACM Manag. Data* 2, 1, Article 3 (March 2024), 23 pages. doi:10.1145/3639258

[22] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 313–324.

[23] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[24] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2020. Learning How To Learn Within An LSM-based Key-Value Store. *CoRR* (2020).

[25] 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*. 79–94.

[26] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. doi:10.1145/3183713.3196927

[27] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[28] 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*. 365–378.

[29] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[30] DGraph. 2024. DGraph. https://dgraph.io/.

[31] Chen Ding, Kai Lu, Quanyi Zhang, Zekun Ye, Ting Yao, Daohui Wang, Huatao Wu, and Jiguang Wan. 2025. DFlush: DPU-Offloaded Flush for Disaggregated LSM-based Key-Value Stores. *Proc. ACM Manag. Data* 3, 3, Article 147 (June 2025), 28 pages. doi:10.1145/3725284

[32] Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. 2023. Dotori: A Key-Value SSD Based KV Store. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1560–1572.

[33] Facebook. 2024. RocksDB. https://github.com/facebook/rocksdb.

[34] Apache Flink. 2025. Apache Flink Documentation: Windows. https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/operators/windows/.

[35] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. Workload analysis and demand prediction of enterprise data center applications. In *2007 IEEE 10th International Symposium on Workload Characterization*. IEEE, 171–180.

[36] 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*. 1–14.

[37] Google. 2024. LevelDB. https://github.com/google/leveldb/.

[38] Google Cloud. 2024. Google Cloud Memorystore. https://cloud.google.com/memorystore.

[39] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. https://eigen.tuxfamily.org.

[40] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. 2024. Aceso: Achieving Efficient Fault Tolerance in Memory-Disaggregated Key-Value Stores. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 127–143.

[41] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[42] 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*. 651–665.

[43] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2022. Endure: a robust tuning paradigm for LSM trees under workload uncertainty. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1605–1618.

[44] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24.

[45] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn.. In *CIDR*.

[46] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn.. In *CIDR*.

[47] Influxdata. 2024. InfluxDB. https://www.influxdata.com/.

[48] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB:Single-LevelKey-Value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 191–205.

[49] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. 2016. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 57–70.

[50] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, et al.

2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151.

[51] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proceedings of the 2022 International Conference on Management of Data*. 1670–1684.

[52] Cockroach Labs. 2024. CockroachDB. https://github.com/cockroachdb/cockroach.

[53] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[54] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4023–4037.

[55] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.

[56] Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering. In *Proceedings of the ACM Web Conference 2022*. 2759–2767.

[57] Yuhong Liang, Tsun-Yu Yang, and Ming-Chang Yang. 2021. KVIMR:Key-Value Store Aware Data Management Middleware for Interlaced Magnetic Recording Based Hard Disk Drive. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 657–671.

[58] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[59] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[60] Chen Luo and Michael J Carey. 2019. On performance stability in LSM-based storage systems (extended version). *arXiv preprint arXiv:1906.09667* (2019).

[61] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.

[62] 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*. 2071–2086.

[63] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. 2009. Modular data storage with Anvil. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 147–160.

[64] 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

[65] Dingheng Mo, Junfeng Liu, Fan Wang, and Siqiang Luo. 2025. Aster: Enhancing LSM-structures for Scalable Graph Database. *Proc. ACM Manag. Data* 3, 1, Article 12 (Feb. 2025), 26 pages. doi:10.1145/3709662

[66] Dingheng Mo, Siqiang Luo, and Stratos Idreos. 2025. How to Grow an LSM-tree? Towards Bridging the Gap Between Theory and Practice. *Proc. ACM Manag. Data* 3, 3, Article 173 (June 2025), 25 pages. doi:10.1145/3725310

[67] MongoDB. 2025. WiredTiger API: WT_SESSION Struct Reference. http://source.wiredtiger.com/mongodb-5.0/struct_w_t___s_e_s_s_i_o_n.html.

[68] Netflix. 2024. How Netflix optimizes use of Apache Cassandra® for massive scale. https://www.youtube.com/watch?v=n_SXhW-x0WA.

[69] PingCAP. 2025. TiKV Tuning Guide. https://docs.pingcap.com/tidb/stable/tikv-configuration-file/.

[70] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In *Proceedings of the 2018 International Conference on Management of Data*. 551–566.

[71] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[72] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[73] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.

[74] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2022. Constructing and analyzing the LSM compaction design space. *arXiv preprint arXiv:2202.04522* (2022).

[75] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.

[76] Amazon Web Services. 2024. Amazon ElastiCache. https://aws.amazon.com/elasticache/.

[77] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. 2023. FUSEE: A fully Memory-DisaggregatedKey-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 81–98.

[78] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building Workload-Independent Storage with VT-Trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 17–30.

[79] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 729–730.

[80] Cloudius Systems. 2024. ScyllaDB. https://www.scylladb.com/.

[81] Risi Thonangi and Jun Yang. 2017. On log-structured merge for solid-state drives. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 683–694.

[82] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. Noftl-kv: Tackling write-amplification on kv-stores with native storage management. In *Advances in database technology-EDBT 2018: 21st International Conference on Extending Database Technology, Vienna, Austria, March 26-29, 2018. proceedings*. University of Konstanz, University Library, 457–460.

[83] 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

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

[85] 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 *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[86] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. 2023. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 441–459.

[87] Xiaoliang Wang, Peiquan Jin, Yongping Luo, and Zhaole Chu. 2024. Range Cache: An Efficient Cache Component for Accelerating Range Queries on LSM-Based Key-Value Stores. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 488–500.

[88] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Xstore: Fast rdma-based ordered key-value store using remote learned cache. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–32.

[89] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. 2020. AC-Key: Adaptive caching for LSM-basedKey-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 603–615.

[90] 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 *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[91] Ziwei Xiong, Dejun Jiang, and Jin Xiong. 2024. DiStore: A Fully Memory Disaggregation Friendly Key-Value Store with Improved Tail Latency and Space Efficiency. In *Proceedings of the 53rd International Conference on Parallel Processing*. 607–617.

[92] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. 2024. IONIA:High-Performance Replication for Modern Disk-based KV Stores. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 225–241.

[93] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the design of LSM-tree Based OLTP storage engine with persistent memory. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1872–1885.

[94] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1976–1989.

[95] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[96] Ting Yao, Jiguang 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. In *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*. 1–13.

[97] Ting Yao, Jiguang 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, 4 (2017), 1–28.

[98] Weiping Yu, Siqiang Luo, Zihao Yu, and Gao Cong. 2024. CAMAL: Optimizing LSM-trees via Active Learning. 2, 4 (2024).

[99] Yu, Geoffrey X and Markakis, Markos and Kipf, Andreas and Larson, Per-Åke and Minhas, Umar Farooq and Kraska, Tim. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.

[100] 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*. 323–336.

[101] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, and Si Wu. 2022. DEPART: Replica Decoupling for Distributed Key-Value Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 397–412.

[102] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 225–237.

[103] Yinan Zhang, Huiqi Hu, Xuan Zhou, Enlong Xie, Hongdi Ren, and Le Jin. 2023. PM-Blade: A Persistent Memory Augmented LSM-tree Storage for Database. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3363–3375.

[104] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. 2018. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[105] Fuheng Zhao, Zach Miller, Leron Reznikov, Divyakant Agrawal, and Amr El Abbadi. 2025. Autumn: A Scalable Read Optimized LSM-Tree Based Key-Value Stores with Fast Point and Range Reads. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2824–2837.

[106] Wenshao Zhong, Chen Chen, Xingbo Wu, and Jakob Eriksson. 2025. Disco: A Compact Index for LSM-trees. *Proc. ACM Manag. Data* 3, 1, Article 33 (Feb. 2025), 27 pages. doi:10.1145/3709683

[107] Yijie Zhong, Zhirong Shen, Zixiang Yu, and Jiwu Shu. 2023. Redesigning High-Performance LSM-based Key-Value Stores with Persistent CPU Caches. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1098–1111.

[108] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing bloom filter cpu overhead in lsm-trees on modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. 1–10.

[109] Zichen Zhu, Yanpeng Wei, Ju Hyoung Mun, and Manos Athanassoulis. 2025. Mnemosyne: Dynamic Workload-Aware BF Tuning via Accurate Statistics in LSM trees. *Proc. ACM Manag. Data* 3, 3, Article 190 (June 2025), 28 pages. doi:10.1145/3725327

[110] Zeying Zhu, Yibo Zhao, and Zaoxing Liu. 2024. In-MemoryKey-Value Store Live Migration with NetMigrate. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 209–224.