Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space

JUNFENG LIU* and FAN WANG*, Nanyang Technological University, Singapore
DINGHENG MO, Nanyang Technological University, Singapore
SIQIANG LUO†, Nanyang Technological University, Singapore

Mainstream LSM-tree-based key-value stores face challenges in optimizing performance for point lookup, range lookup, and update operations concurrently due to their constrained configurations. They typically follow fixed patterns to specify the level capacity and the number of sorted runs per-level. This confines their designs to a restricted space, limiting opportunities for broader optimizations.

To address this challenge, we consider a more flexible configuration that enables independent adjustments of the number of runs per-level, size ratio, and Bloom filter settings at each LSM-tree level. By carefully analyzing the cost of each operation based on the new design space, we unveil two critical insights for optimizing the tradeoff among the three operations. Firstly, achieving efficient point lookup requires a large last level. Secondly, there is a specific correlation between the number of runs per level and size ratio that is advantageous for overall update and range lookup performance.

Based on these insights, we introduce Moose, a structure delivering an impressive overall performance for point lookup, range lookup, and update concurrently. Furthermore, we also introduce a new framework, Smoose, to navigate the design space for adapting specific workloads. We implemented Moose and Smoose on top of RocksDB and experimental results demonstrate that our proposed approach outperforms state-of-the-art LSM-tree structures across diverse workloads.

CCS Concepts: • **Information systems** → **Record and block layout**; **Data structures**; **Data layout**.

Additional Key Words and Phrases: LSM-tree, data structure, optimization

ACM Reference Format:
Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 175 (June 2024), 26 pages. https://doi.org/10.1145/3654978

1 INTRODUCTION

Log-Structured Merge Trees, abbreviated as LSM-trees, play a pivotal role as the foundational data structures underpinning widely adopted industrial key-value stores like Google LevelDB [23], Meta RocksDB [22], Apache Cassandra [32], LinkedIn Voldemort [35], and MongoDB WiredTiger [10]. These LSM-trees drive the advancement of mainstream NoSQL database technology. LSM-trees support three fundamental operations: *point lookup*, *range lookup*, and *update (or write)*. These operations empower key-value stores to construct a wide array of applications, ranging from

*Both authors contributed equally to this research.
†Corresponding Author

Authors' addresses: Junfeng Liu, JUNFENG001@e.ntu.edu.sg; Fan Wang, FAN008@e.ntu.edu.sg, Nanyang Technological University, Singapore; Dingheng Mo, Nanyang Technological University, Singapore, DINGHENG001@e.ntu.edu.sg; Siqiang Luo, Nanyang Technological University, Singapore, siqiang.luo@ntu.edu.sg.

ACM 2836-6573/2024/6-ART175
https://doi.org/10.1145/3654978

Table 1. Our proposed Moose expands the configuration space of LSM-tree, offering a desired optimal three-way tradeoff. The structural flexibility level is marked by symbols: ✗ for fixed, ✓̸ for conditionally tunable, and ✓ for entirely flexible. In conventional LSM-trees, a fixed size ratio and run number are applied across all levels. Dostoevsky [15] extends this design space by employing two adjustable run numbers for the last level and non-last levels, respectively. LSM-Bush [16], to some extent, relaxes constraints on both size ratio and run number. It adopts a descending size ratio and maintains either a single run or a number equal to size ratio of runs per level. LSM-tree generalization offers entirely independent configuration options for the size ratio and number of runs per level. The structures within each configuration space are listed to assess their optimality for different workloads. Note that "optimal" indicates the optimality within its own designated configuration space.

Configuration Space				**Optimality within configuration sapce**			
Domains	Structural Flexibility			Structures	Point lookup	Update	Range query
	Size ratio	#Runs per-level	Filter BPK per-level				
Conventional LSM-tree	✗	✗	✗	Leveling [23]	**Optimal**	Low	**Optimal**
				Tiering [32]	Low	**Optimal**	Low
Dostoevsky [15]	✗	✓	✓	Lazy Leveling [15]	Moderate	High	Low
LSM-Bush [16]	✓̸	✓̸	✓	QLSM-Bush [16]	Moderate	**Optimal**	Low
LSM-tree Generalization*	✓	✓	✓	**Moose***	**Conditional Optimal**	**Optimal Tradeoff**	

social graph processing [4] and time-series data systems [31, 46] to cryptocurrencies [45], online services [19] and spatial databases [39]. A point lookup outputs a value corresponding to the queried key; a range lookup scans a key range and outputs the values mapped to the keys located in the range; an update in an LSM-tree admits a new key-value entry into the data structure, with a special bit marking whether the write represents an insert or a delete.

Recent LSM-tree development centers on optimizing the costs of its three core operations. Initially designed, the LSM-tree organizes data as key-value pairs across multiple exponentially increasing levels, each level representing a consolidated *sorted run* with keys sorted from small to large. A level is sort-merged to the subsequent level when reaching its capacity. At a high level, the LSM-tree employs an out-of-place update mechanism for efficient batched updates, while can impact read performance. Therefore, recent key-value stores like RocksDB and LevelDB use Bloom filters in each sorted run to significantly improve read performance by reducing disk I/Os for point lookups. Moreover, the point lookup performance has been further optimized by an influential work Monkey [14], which strategically utilizes the filter memory budgets across levels and effectively improves point lookup performance. Doestoevsky [15], built on Monkey, explores a tradeoff between tiering compaction (write-optimized) and leveling compaction (read-optimized), suggesting performance tuning based on workload. Later, LSM-Bush [16] introduces a more flexible structure by partly relaxing capacity ratios between adjacent levels which obviously enhances write performance.

While these works offer fundamental improvements and inspirations, we witness two open problems.

Problem 1: To what extent can we optimize point lookups, range lookups and updates simultaneously? Recent studies have pointed out that the intrinsic tradeoff exists among different LSM-tree operations, such as point lookups, range lookups, and updates, suggesting that one cannot expect the co-existence of optimum costs for all these operations. For example, Dostoevsky shows that tuning compaction policies effectively achieves different balance between read costs and write costs. However, there lacks theoretical understanding in balancing among different operations especially when considering the three fundamental operations (point lookups, range lookups and

updates) together. In particular, given a colossal configuration space for LSM-trees, can we quickly determine which configuration yields a data structure that makes an optimal tradeoff between two operations, while the third operation is conditionally optimal given the cost of the previous two? Answering this question is challenging because when optimizing towards one operation, it can negatively impact the balance between the other two, resulting in intricate cost analysis.

Problem 2: Can we explore a more flexible configuration space in LSM-tree designs? Previous studies model the I/O complexity based on a configuration space that consists of size ratios between two adjacent levels, numbers of runs per-level, as well as bits-per-key settings in Bloom filters. However, these studies have typically worked with constrained configuration spaces. Examples of these constraints include maintaining a fixed size ratio (e.g., Dostoevsky requires a fixed size ratio between adjacent levels), enforcing a specific number of sorted runs (e.g., Leveling compaction policy suggests only one sorted run in each level), or adhering to a predefined pattern of size ratios (e.g., LSM-Bush considers doubling exponential size ratios as levels grow larger). This gives rise to a natural question – can we derive a better data structure when we remove these constraints and open up more configuration flexibility? To seek out superior configurations within an expanded space, one naive approach is to exhaustively explore all possibilities within this space, but apparently this leads to a prohibitive search complexity due to the sheer number of possible settings. To effectively explore all possible LSM-tree designs, therefore, a new analytical framework is required to intricately formulate the I/O cost of each operation in the new configuration space, so as to discover a small subset of the promising settings for further verification.

Our insight: There is a sweet spot amongst the three operational costs with an expanded configuration space. Unlike previous studies, we consider a configuration space that consists of all possible level-wise settings including size ratio, number of runs, and Bloom filter bits (subject to a total memory budget). Importantly, we allow for distinct configurations to be assigned to individual levels within this framework. To effectively explore the vast configuration space, we model the I/O costs of point lookups, range lookups, and updates in the expanded space, from which we discover a set of promising designs. Our crucial observation is that the optimal point lookup performance hinges primarily on the largest level, whereas the optimal tradeoff between range lookups and updates depends largely on size ratios across all levels. Given this observation, we find that a sweet spot to balance the three operational costs may happen when the size ratio and the number of runs per level vary across the levels, which justifies the necessities of examining the LSM-tree design over an expanded space.

Moose and Smoose. Based on these insights, we present a new LSM-structure called Moose based on the theoretical analysis mentioned earlier. Moose achieves an asymptotically optimal point lookup cost when the largest level capacity is given and the cost tradeoff between range lookups and updates follows an optimal tradeoff curve. This is a non-trivial instance-optimality result that bridges the three fundamental LSM-tree operational costs, which has rarely been systematically studied before. Table 1 compares our proposal against existing studies. The left side of the table shows that our LSM-tree generalization entails a configuration space that gives the largest flexibility while the right side of the table illustrates the cost optimality of each structure *within their own configuration space*. Clearly, Moose considers the highest structural flexibility, and is the only method that offers optimality interpretations covering three operations. To further pinpoint the best setting of the largest level, we design Smoose, a workload-aware version of Moose, to autonomously determine the capacity of the largest level with respect to a given workload consisting of point lookups, range lookups, and updates.

Contributions. We summarize our contribution as follows:

- **All-flexible configurations.** We are the first to consider an LSM-tree design space that simultaneously allows varying Bloom filter bits-per-keys, the numbers of runs, and size ratios across

each level of the LSM-tree. We establish a new set of cost analysis based on the largely expanded configuration space.

- **New LSM-structures with optimality guarantees.** We present Moose, a new LSM-tree design that is selected by optimizing the costs over the vast configuration space. Given the capacity of the largest level, it secures an asymptotic optimal point lookup cost, conditioned on an optimal tradeoff between range lookup cost and update cost along a specific Pareto curve. Moreover, we further present Smoose, built upon on Moose, to pinpoint the hyperparameters in response to diverse workloads.
- **Extensive experiments over RocksDB.** We embed our design into RocksDB, a widely adopted industrial KV-store, and evaluate its performance against state-of-the-art designs. Experimental results show that Moose and Smoose are often the best-rank methods among the seven methods we study, over a wide spectrum of workloads.

2 BACKGROUND

LSM-tree. The LSM-tree comprises a main memory buffer, typically denoted as level-0, and subsequent levels that are located on disks. The capacity of these levels is controlled by size ratio, often set to a fixed integer T, which represents the capacity ratio between two consecutive levels. Data insertions are initially buffered in Level-0 until the level capacity is reached thus prompting a flush which sort-merges the data to the next level. During the merge, only the most recent version is retained for entries with the same keys and the obsolete versions are discarded. This cyclical process, occurring between any two levels, expands the number of levels as more data is inserted. With a memory buffer (i.e. Level-0) size of F, the capacity of Level-i is $F \cdot T^i$, and the last level can hold approximately $N \cdot \frac{T-1}{T}$ entries, where N represents the total number of entries. Therefore, the number of levels can be derived with a given entry number N and size ratio T as $L = \lceil \log_T(\frac{N}{F} \cdot \frac{T-1}{T}) \rceil$. Please note that the size ratio between consecutive levels should be in the range $[2, N/F]$ to ensure the multi-level structure of the LSM-tree and maintain increasing capacity.

Merge Policies and Sorted Runs. Merge policy refers to the operations executed when a certain Level-i is full and is required to be merged to the next level. Typical merge policies include *leveling* and *tiering*. In leveling policy, each LSM-tree level only contains one sorted run, and filling-up a level triggers a sort-merge of the level to the run at the next level. In tiering policy, in contrast, each LSM-tree level may have at most T sorted runs, and the full Level-i triggers a merge amongst the runs at this level and a new sorted run will be placed into the next level.

Point Lookups. Point lookup involves finding the value of a given key in an LSM-tree. If the key is in the memory buffer, the associated value is returned from the buffer; otherwise, the procedure searches sorted runs on disk from small to large LSM-tree levels and returns the first encountered value for the queried key. If multiple runs exist within one level, all the runs should be searched and only the latest entry is returned.

To accelerate the point lookups, *Bloom filters* are commonly employed for efficient enhancement. These probabilistic structures determine the key existence in a set and are typically cached in memory during system startup. The accuracy of Bloom filter is governed by a tunable parameter, bits-per-key (BPK), representing the ratio of the filter memory budget to the number of keys. BPK can impact the False Positive Rate (FPR) of a Bloom filter, given by $FPR = e^{-BPK \cdot ln(2)^2}$.

When the LSM-tree conduct point lookup on a sorted run, it will first access the Bloom filter to predict the existence of the query key. Actual I/O operations are performed only if the Bloom filter indicates a positive result. The cost of point lookup is directly related to the FPR of each sorted run when the key is absent. Typically, the memory allocated to each entry is fixed for every level (i.e., BPK is a constant for all the levels) in the systems.

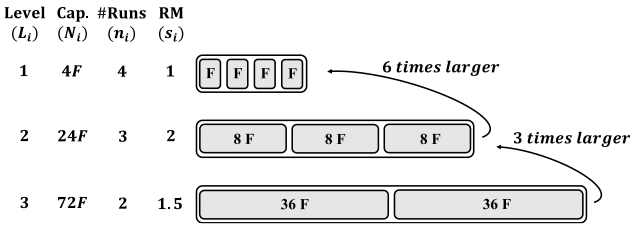

Fig. 1. An instance of LSM-tree generalization. The LSM-tree generalization enables independent configuration of the size ratio and the number of runs per level, thus providing a significantly higher degree of flexibility in structure design. In this figure, *RM* stands for run magnification, and Cap. is the abbreviation for capacity.

Range Lookups. During a range lookup, the LSM-tree retrieves the most recent versions of all entries within a specified key range. It starts from the target key at each level/sorted run and then sort-merges the retrieved entries from different level/sorted runs. If there are the same keys at different levels/sorted runs, the LSM-tree will discard the older versions.

Updates. The LSM-tree inserts new key-value pairs to its memory buffer and flushes it to the disk as a sorted run when the buffer threshold is reached. Moreover, LSM-tree adopts an out-of-place strategy to update the existing keys, which follow the same process of insertion.

3 MOOSE: THREE-WAY BALANCED LSM-TREE STRUCTURE

This section presents the detailed design of Moose (Sections 3.1 ∼ 3.5), and its workload-aware counterpart Smoose (Section 3.6).

3.1 LSM-tree Generalization and Analysis

In order to develop a structure that effectively balances all three operations of point lookup, range lookup, and update, we introduce a more flexible LSM structure model, *LSM-tree generalization*, to significantly expand the configuration space of LSM-tree. Based on the proposed structure model, we carefully break down the cost associated with each operation and evaluate the corresponding complexity with the disk access model that formulates operational costs with the associated I/O times. Our analysis varies from existing works for the employment of an extended configuration space which ultimately drives a new data structure design, named Moose.

General Configuration Space Drives New Designs. Most existing works employ a global size ratio or a restricted compaction policy to determine the capacity of each level and its associated sorted runs. Apparently, this strategy poses latent constraints on the profile of the LSM-tree thus resulting in a limited design space.

To tackle this problem, we propose LSM-tree generalization that relaxes the restriction on size ratio and compaction policy thus significantly enhancing the structural flexibility of LSM-tree. Specifically, LSM-tree generalization introduces two level-independent parameters, run number $\{n_i\}$ and run magnification $\{s_i\}$ to facilitate the refined structural adjustment, where n_i is the maximum number of sorted runs at Level-i, and s_i is the ratio between the run size of Level-i and the capacity of Level-$(i-1)$. Let N_i be the capacity of level i, then:

$$N_i = s_i \cdot N_{i-1} \cdot n_i \tag{1}$$

It immediately implies that the size ratio $r_i = \frac{N_i}{N_{i-1}} = s_i \cdot n_i$, suggesting that the setting of r_i for different level-i can be diverse, which leads to various size ratios across different levels. The run magnification s_i allows for level-wise tuning between *leveling* and *tiering* merge policy, hence controlling the merge policy in finer granularity. We present an instance of the LSM-tree

generalization structure in Figure 1 for which $\{n_i\}$ and $\{s_i\}$ are specified as $\{4, 3, 2\}$, and $\{1, 2, 1.5\}$, respectively, to illustrate the generality of our configuration space.

Our structure design principle is mostly inspired by a new set of theoretical analysis we establish based on the extended configuration space. Importantly, the new analysis gives us a better opportunity to co-optimize all three operations (range lookup, update, and point lookup). For example, our analysis shows that the number of runs (n_i) at each level should be proportional to the square root of the size ratio of that level. Under such conditions, point lookup cost becomes asymptotically optimal as long as the Bloom filter memory budget is wisely allocated across levels.

3.2 Analyzing Each Operation

Based on the extended design space of LSM-tree generalization, we deliver a careful analysis on the cost of update, range lookup, and point lookup operation.

Analyzing update cost. The I/O cost of update operation is introduced by the subsequent compaction process in which the updated entry participates. According to the design principle of LSM-tree, a compaction is triggered when any level reaches its capacity. If level-i is full, all the N_i entries will be read to memory and merged into a larger sorted run to be placed at the next level. Hence, during the compaction process, every entry in the level would be rewritten. Additionally, entries are stored in data blocks, which means one I/O operation will read/write an entire data block. Assume each data block accommodates B entries, where B is the block size, rewriting N_i entries consumes $\frac{N_i}{B}$ I/Os in total. This implies that each compaction introduces $\frac{1}{B}$ I/Os for every participant entry.

In the worst case that an entry is ultimately stored at the largest level, it must traverse the entire tree after insertion. For tiering, each entry participates in only one compaction at each level thus taking L compactions to reach the last level. This results in $O(\frac{L}{B})$ I/Os for an update operation. When it comes to leveling, for which each level contains a single sorted run, all existing data at the target run would be read and rewritten for reorganization once a new run comes. When the level achieves its maximum capacity, each entry takes part in an average of $\frac{T}{2}$ compactions resulting in the the update cost of $O(\frac{T \cdot L}{B})$.

As shown by Figure 1, the run magnification s_i is not necessarily an integer for our proposed LSM-tree generalization. To facilitate this new design, we adopt a different compaction policy beyond these two methods. When a compaction targets an empty level, a new run is created at the target level and identified as active. Entries coming from the previous level will be merged into this active run until it reaches its capacity. If the active run is full, another active run will be established and the previous one will be marked as static. Consequently, given N_i is r_i times greater than the capacity of level-$(i-1)$, it takes $s_i = \frac{N_i/n_i}{N_{i-1}}$ compactions, on average, to fill up a run. As a result, the I/O cost of each update operation U is determined by the number of compactions enabling each entry to traverse L levels, which is represented by the Equation 2.

$$U = O\left(\sum_{i=1}^{L}\left(\frac{1}{B} \cdot \frac{N_i}{N_{i-1}} \cdot \frac{1}{n_i}\right)\right) \tag{2}$$

Analyzing range lookup cost. The range lookup retrieves all entries whose key falls within the target range from the database. In the worst case, entries matching the key range specifications are scattered across all runs in the entire LSM-tree.

To retrieve them all, the range lookup can be divided into two phases: scanning phase and retrieving phase. The scanning phase begins with locating the starting point and retrieving the associated data block at each sorted run. Therefore, the I/O cost in this phase equals the total number of runs. For tiering and leveling, the scanning phase requires $O(T \cdot L)$ and $O(L)$ I/Os, respectively. For LSM-tree generalization, the I/O cost in this phase is, accordingly, $O(\sum_{i=1}^{L} n_i)$.

Table 2. List of terms used in the paper.

Term	Definition
N	total data size quantified by the number of entries
L	total level number
M	Bloom filter memory (in bits)
F	buffer size quantified by the number of entries
B	block size quantified by the number of entries
N_i	capacity of level-i quantified by the number of entries
r_i	size ratio between level-i and level-(i-1)
n_i	number of runs at level-i
s_i	run magnification of level-i
M_i	Bloom filter memory assigned to level-i
p_i	false positive rate of level-i
U, R, Z	I/O cost of update, range lookup, point lookup

Subsequently, in the retrieving phase, the LSM-tree retrieves and sort-merges the entries from each runs iterating from the starting point to the end of the specified key range. Assuming that each run holds several data blocks to read, then the total number of entries to fetch equals $t = O(\sum_{i=1}^{L} n_i \cdot B)$. (We will discuss in Section 3.5 why our approach still works even when this assumption is removed). Since the entries are fetched sequentially in the retrieving phase, $\frac{t}{B}$ additional I/Os are introduced in this phase. As a result, the I/O cost of range query operation is the combination of scanning cost and retrieving cost, as Equation 3 shows.

$$R = O\left(\sum_{i=1}^{L} n_i + \frac{\sum_{i=1}^{L} n_i \cdot B}{B}\right) = O\left(\sum_{i=1}^{L} n_i\right) \tag{3}$$

Analyzing point lookup cost. Point lookup operation retrieves the up-to-date entry possessing the target key from the LSM-tree. For instance, if an entry is flushed to Level-1 while another entry with the same key has already been stored at Level-2, two qualified entries exist simultaneously. Nevertheless, the one located at the larger level carries outdated information thus is obsolete. In order to retrieve the most recent record, the Level-0 memory buffer is initially examined to identify if the desired entry exists. If the target entry is found, retrieve it and end the operation, which requires one I/O operation. Otherwise, scan the LSM-tree level by level from the small to large level until the target is identified or all levels are traversed. If multiple runs exist within one level, all the runs should be searched and only the latest entry is returned if the entry exists.

Additionally, Bloom filters are designed for each run to boost the point lookup process. If the qualified entry does not exist in a run, its Bloom filter provides an accurate negative result. Then, we can safely skip searching in that run, thus saving one I/O operation. Whereas, Bloom filter may produce false positives, which means that we still need to access the run even if the target entry is not present. Hence, the probability of falsely retrieving a run at Level-i equals the Bloom filter false positive rate p_i. In the worst case, as indicated in the recent works [14, 15], the desired key is absent throughout the entire tree and all levels should be traversed. Therefore, the I/O cost of a point lookup operation is the sum of false positive rates at all runs as Equation 4 presents.

$$Z = O\left(\sum_{i=1}^{L} n_i \cdot p_i\right) \tag{4}$$

We denote the total Bloom filter memory budget as M, with M_i budget assigned to the i-th level. Given the Bloom filter bits per key for that level is $\frac{M_i}{N_i}$, we can derive the corresponding false positive rate with Equation 5 based on the Bloom filter's property.

$$p_i = e^{-\frac{M_i}{N_i} \cdot \ln(2)^2} \tag{5}$$

3.3 New Inspirations on the Three-way Tradeoff

Our proposed LSM-tree generalization model presents higher structural flexibility thus enabling us to develop a generic structure striking superior three-way tradeoff among the cost of range lookup, update, and point lookup operations. To this end, we investigate the properties of the three costs quantitatively and propose Moose, a structure that realizes an optimal tradeoff between range lookup and update, while achieving a conditioned asymptotic optimal point lookup cost.

The Pareto curve of range lookup and update. It is observed that there exists an intrinsic connection between the range lookup cost and update cost. Therefore, we start by examining a Pareto curve[1] of the two costs (denoted as RU-curve in the following) indicating the condition that range lookup/update cost cannot be further improved without harming another as is described by Equation 6.

$$H = R \cdot U = O\left(\sum_{i=1}^{L}\left(\frac{1}{B} \cdot \frac{N_i}{N_{i-1}} \cdot \frac{1}{n_i}\right) \cdot \sum_{i=1}^{L} n_i\right) \tag{6}$$

Lemma 3.1 (Cauchy-Schwarz Inequality). *For any two sets of positive numbers, $\{x_i | 1 \le i \le t\}$ and $\{y_i | 1 \le i \le t\}$, we have*

$$\sum_{1 \le i \le t} x_i \sum_{1 \le i \le t} y_i \ge \left(\sum_{1 \le i \le t} \sqrt{x_i y_i}\right)^2,$$

where the equality holds when $\frac{x_i}{y_i} = \frac{x_j}{y_j}$ for any $1 \le i, j \le t$.

By applying the Cauchy-Schwarz inequality, the impact of n_i can be removed from the expression of H. Then we achieve a more explicit expression of the RU-curve as specified by the Equation 7. This equation indicates that the optimal RU-curve is obtained when $n_i = k\sqrt{\frac{N_i}{N_{i-1}}}$ (derived by using the condition that Cauchy-Schwarz equality holds) if the capacity of each level N_i is determined. The parameter k is a knob allowing us to tune along the RU-curve.

$$H \ge O\left(\frac{1}{B}\left(\sum_{i=1}^{L}\sqrt{\frac{N_i \cdot n_i}{N_{i-1} \cdot n_i}}\right)^2\right) = O\left(\frac{1}{B}\left(\sum_{i=1}^{L}\sqrt{\frac{N_i}{N_{i-1}}}\right)^2\right) \tag{7}$$

Currently we are capable of finding an RU-curve presenting the optimal tradeoff between range lookup and update cost for a set of $\{N_i\}$. Meanwhile, there are multiple valid level capacity allocation strategies for a given data size N, each of which would introduce a specific RU-curve. Therefore, selecting the optimal RU-curve and the corresponding level capacity distribution $\{N_i\}$ is another vital problem.

Acute readers may observe that the RU curve obtained through Equation 7 could be further optimized with the Arithmetic Mean-Geometric Mean (AM-GM) Inequality:

[1]Beyond which lookup cost cannot be improved without harming update cost, and vise versa.

$$H \geq O\left(\frac{1}{B} \cdot \left(L \cdot \left(\prod_{i=1}^{L} \sqrt{\frac{N_i}{N_{i-1}}}\right)^{\frac{1}{L}}\right)^2\right) = O\left(\frac{1}{B} \cdot L^2 \cdot \left(\frac{N_L}{N_0}\right)^{\frac{1}{L}}\right) \tag{8}$$

The equality is held when $\sqrt{N_i/N_{i-1}} = \sqrt{N_j/N_{j-1}}$ for any i and j. It implies that the size ratio for each level should be identical throughout the tree for balancing the cost of range lookup and update. However, strictly adhering to this configuration of globally fixed capacity ratio may lead to a sub-optimal point lookup performance, which we will discuss at Section 3.4.

Co-optimizing with point lookups. Based on our previous analysis, the I/O cost of a point lookup operation is calculated by summing the product of the run number n_i and the respective Bloom filter false positive rate p_i across all levels. Therefore, there are two factors affecting the point lookup efficiency. Similar to range lookup operation, the point lookup cost is positively related to the number of runs. Consequently, the point lookup performance can be co-tuned with that of range lookup by adjusting run number regulator k along the RU-curve. Whereas, the impact of false positive rate is more complex due to its dependence on the Bloom filter memory assignment strategy. Existing studies [14, 16] have demonstrated that assigning identical bits per key for each entry is suboptimal for point lookup improvement. Therefore, we aim to propose an optimal Bloom filter memory assignment strategy for our proposed LSM-tree generalization to expedite point lookup operation. Finally, our analysis indicates that, considering both the run number n_i and false positive rate p_i, expanding the capacity of the last level N_L is advantageous for co-optimizing point lookups conditioned on optimized update and range lookup performance.

To achieve the most effective Bloom filter memory assignment, we reevaluate the point lookup cost. Assuming the ith level takes up M_i of Bloom filter memory, the false positive rate can be expressed as $exp(-\frac{M_i}{N_i} \cdot \ln(2)^2)$, where $\frac{M_i}{N_i}$ represents the Bloom filter bits per key. Subsequently, we decompose the point lookup cost at a granular level, with each entry contributing $\frac{n_i \cdot p_i}{N_i}$ to the entire cost. Consequently, we can optimize the I/O cost of the point lookup operation according to Equation 9 without introducing any additional assumption on the capacity ratio.

$$\begin{aligned}
Z &= O\left(\sum_{i=1}^{L} N_i \cdot \frac{n_i \cdot p_i}{N_i}\right) \\
\textbf{(By AM-GM inequality)} \quad &\geq O\left(N \cdot \left(\prod_{i=1}^{L} \left(\frac{n_i}{N_i} \cdot p_i\right)^{N_i}\right)^{\frac{1}{N}}\right) \\
\textbf{(By Eq. 5)} \quad &= O\left(N \cdot \left(\prod_{i=1}^{L} \left(\frac{n_i}{N_i}\right)^{N_i} \cdot e^{-\sum_{i=1}^{L} M_i \cdot \ln(2)^2}\right)^{\frac{1}{N}}\right) \\
&= O\left(N \cdot \left(\prod_{i=1}^{L} \left(\frac{n_i}{N_i}\right)^{N_i}\right)^{\frac{1}{N}} \cdot e^{-\frac{M}{N} \cdot \ln(2)^2}\right)
\end{aligned} \tag{9}$$

The inequality is due to AM-GM inequality, and the equality holds when the Equation 10 is satisfied for arbitrary two levels. This implies that to achieve the optimal Bloom filter memory allocation strategy, the false positive rate p_i for a specific level should be proportional to its run size $\frac{N_i}{n_i}$. In other words, it is advisable to allocate more bits per key to the smaller levels. This strategy is reasonable because improving the point lookup performance of the smaller levels consumes lower memory budget compared to the larger ones. For example, assuming level-$(i+1)$ is five times larger

than level-i. Then, it takes the same amount of memory to increase the bits per key from 0 to 5 for level-i while only to increase it from 0 to 1 for level-$(i+1)$. Consequently, the false positive rate of level-$(i+1)$ is roughly 6.8 times higher than that of level-i. Additionally, larger levels are more likely to be accessed by point lookup operation since they store larger portion of data. Therefore, they are inherently less sensitive to the false positive rate increment due to the higher true positive rate. As a result, allocating more memory to the smaller levels helps achieve a more favorable tradeoff that ultimately enhances overall point lookup performance.

$$\frac{n_i \cdot p_i}{N_i} = \frac{n_j \cdot p_j}{N_j} \tag{10}$$

This insight is consistent with the existing study Monkey [14]. Specifically, Monkey is a specialized case of our method with a fixed size ratio and number of runs across all levels. In comparison to this approach, our Bloom filter memory allocation strategy is more inclusive and thus applicable to more flexible LSM-tree structures. This, in turn, empowers us to achieve a superior three-way tradeoff when combined with the design space provided by the LSM-tree generalization.

To achieve this, we incorporate the insights gained through RU-curve optimization into the point lookup cost analysis for further enhancement. As Equation 11 depicts, the I/O cost of point lookup operation Z is currently formulated solely in terms of the level capacity $\{N_i\}$ after introducing the condition $n_i = k\sqrt{\frac{N_i}{N_{i-1}}}$. Accordingly, we can adjust the profile of the LSM-tree by adjusting $\{N_i\}$ to enhance point lookup performance while retaining its superiority on the range lookup and updates.

$$\begin{aligned} Z &= \left(N \cdot \left(\prod_{i=1}^{L} \left(\frac{n_i}{N_i} \right)^{N_i} \right)^{\frac{1}{N}} \cdot e^{-\frac{M}{N} \cdot \ln(2)^2} \right) \\ &= O\left(N \cdot k \cdot \left(\prod_{i=1}^{L} \left(\frac{1}{\sqrt{N_i \cdot N_{i-1}}} \right)^{N_i} \right)^{\frac{1}{N}} \cdot e^{-\frac{M}{N} \cdot \ln(2)^2} \right) \end{aligned} \tag{11}$$

Let $\mathcal{H}(N_1 \cdots N_L) = k \cdot N \cdot (\prod_{i=1}^{L} (\frac{1}{\sqrt{N_i \cdot N_{i-1}}})^{N_i})^{\frac{1}{N}} \cdot e^{-\frac{M}{N} \cdot \ln(2)^2}$ and $\mathcal{U}(N_1 \cdots N_L) = \sum_{i=1}^{L} N_i \log(N_i N_{i-1})$. It is obvious that $\mathcal{H}(N_1 \cdots N_L)$ is positively correlated to $2^{-\mathcal{U}(N_1 \cdots N_L)}$, and hence inversely correlated to $\mathcal{U}(N_1 \cdots N_L)$. Consequently, we should find the maximum value of $\mathcal{U}(N_1 \cdots N_L)$ to minimize the point lookup cost. Moreover,

$$\sum_{i=1}^{L} N_i \log N_i \leq u(N_1 \cdots N_L) \leq \sum_{i=1}^{L} N_i \log(N_i N_i) = 2 \sum_{i=1}^{L} N_i \log N_i$$

This equation demonstrates that maximizing $\mathcal{U}(N_1 \cdots N_L)$ is closely related to maximizing $\sum_{i=1}^{L} N_i \log N_i$. Considering the convex property of function $f(x) = x \log x$ and the capacity constraint $\sum_{i=1}^{L} N_i = N$, an increased N_L yields advantages for optimizing the point lookup performance.

3.4 DP Aided Three-way Tradeoff

For obtaining an ideal LSM-tree structure that delivers optimal overall performance on range lookup, update, and point lookup simultaneously, we revisit the insights derived from RU-curve and point lookup optimization. To minimize the point lookup cost, a large capacity should be assigned to the last level. Meanwhile, a fixed size ratio is advantageous for reaching the optimal RU-curve that is negatively related to the sum of the size ratio for each level. In this scenario, an enlarged last level capacity would inevitably increase the size ratio between the last two levels, thus affecting the RU-curve optimality. Therefore, we have to make a compromise when applying these insights on LSM-tree structure design for achieving the optimal overall performance.

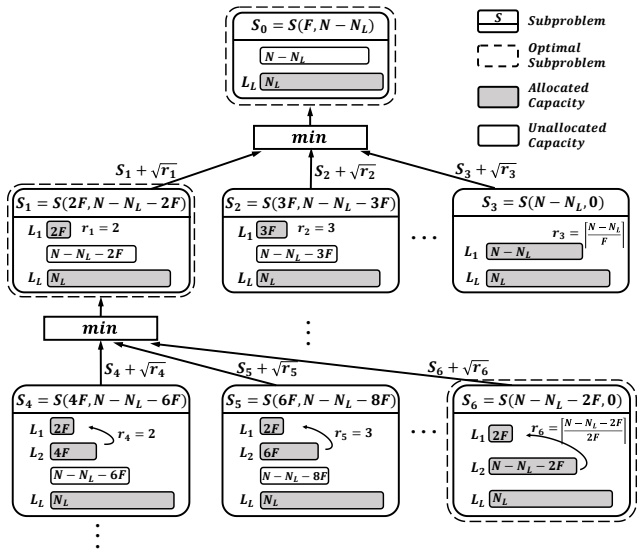

Fig. 2. Dynamic programming is able to produce RU-curve optimal structure for a given entry number N, buffer size F, and last level capacity N_L by decomposing the original problem into various sub-problems recursively.

To this end, we employ a specialized last level capacity N_L for improving the point lookup efficiency. Based on this determined N_L, we subsequently identify an optimal capacity configuration for the non-last levels to enhance the overall performance of update and range lookup via approaching the conditioned optimal RU-curve. According to Equation 7, the optimality of RU-curve hinges directly on the sum of the square roots of each level size ratio, denoted to A that equals $\sum_{i=1}^{L} \sqrt{\frac{N_i}{N_{i-1}}}$.Therefore, we can examine the level capacity configuration $\{N_1, N_2, \cdots, N_{L-1}\}$ for given N and N_L by minimizing the cost A with dynamic programming. Interestingly, we will show that during this optimization, the impact on the point lookup is well bounded, as we will analyze in Section 3.5.

Dynamic Programming (DP). To derive optimal configuration $\{N_1, N_2, \cdots, N_{L-1}\}$, we solve this problem recursively. To start with, we define the state of this problem as $S(N_d, N_r)$, where N_d is the initial capacity and N_r is the remained capacity. Accordingly, the original problem is denoted as $S(F, N - N_L)$. A sub-problem can be produced by allocating certain amount of remained capacity to a level and noted as $S(N_d \cdot r, N_r - N_d \cdot r)$, where r is the size ratio that incurs $\sqrt{r}$ cost to reach this sub-problem. Eventually, the cost of the original problem is the minimum value among the costs of all possible sub-problems plus the cost for reaching that particular state.

As depicted in Figure 2, we can enumerate all potential capacities N_d for level-1 and determine the optimal capacity configurations among them. The state $S(F, N - N_L)$ is essentially determined by one of these configurations that yields the minimum cost. To further illustrate this process, let us consider the scenario that $r \cdot F$ capacity is allocated to level-1. The state of this sub-problem $S(r \cdot F, N - N_L - r \cdot F)$ is associated with an optimal configuration $\{r \cdot F, N_2', \cdots, N_{L-1}'\}$ that achieves the minimum cost. According to the cost formulation of A, this particular sub-problem results in a cost of $\sqrt{r} + S(r \cdot F, N - N_L - r \cdot F)$, as Equation 12 presents.

$$S(N_d, N_r) = \min_{2 \le r \le \lceil \frac{N_r}{N_d} \rceil} \{S(rN_d, N_r - rN_d) + \sqrt{r}\} \tag{12}$$

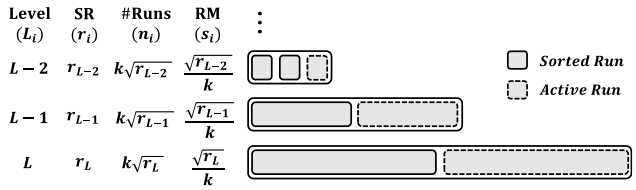

Fig. 3. An illustration for the largest three levels of the default Moose, in which, *SR* and *RM* stands for size ratio and run magnification, respectively. The size ratio is configured as $\{9, 6, 5\}$ corresponding to a run number of $\{3, 2, 2\}$.

Moreover, we can exhaustively list all sub-problems related to level-1 by iterating through r from 2 to $\lceil N - N_L/F \rceil$. As previously mentioned, $S(F, N - N_L)$ is the minimum cost among all sub-problems, as demonstrated in Figure 2. Similarly, we can further decompose each sub-problem to derive its state by enumerating all potential capacities of level-2. During this process, if N_r is less than twice N_d, it indicates that the remaining capability is insufficient to establish a new level. In such cases, we practically allocate the remaining N_r capacity to the previous level and update its state to $\sqrt{N_L/(N_d + N_r)}$. Following this approach, we can recursively break down the sub-problems and assemble their results to construct the solution for the original problem.

To improve computational efficiency, we implement memoization for each state to quickly access them in the subsequent iterations without recalculation. Given that both initial capacity N_d and remaining capacity N_r are integer multiples of the buffer size, there are at most $\lceil N/F \rceil^2$ unique states in the entire process. This significantly reduces the time and computational complexity of the algorithm. In practice, our dynamic programming algorithm can reach the RU-curve optimal structure in just a few seconds.

Moose. Building upon the preceding discussion, we propose Moose, a versatile structure striking superior three-way tradeoff among point lookup, range lookup and update. Moose incorporates an independent last level capacity N_L to adjust point lookup performance. Subsequently, based on the pre-determined N_L, it identifies a capacity configuration $\{N_1, N_2, \cdots, N_{L-1}\}$ approaching conditioned optimal RU-curve via dynamic programming algorithm, which boosts the overall update and range lookup efficiency. Following the insights gained from RU-curve optimization, Moose maintains $k \cdot \sqrt{r_i}$ sorted runs at each level that is controlled by the run number regulator k to adjust the range lookup and update performance.

Apparently, the Moose structure can be adjusted by tuning two parameters N_L and k. This allows it to achieve various tradeoffs among the three operations thus adapting to different workloads. For instance, when point lookup dominates the workload, it is advisable to use a larger N_L. If the range lookup overweights the updates, a smaller k is more suitable. Therefore, to determine these two parameters for specific workloads, a workload aware counterpart Smoose is proposed in Section 3.6. Given the optimal N_L is determined by workload-specific information which is inaccessible for Moose, we select a sound value, $N_L = 0.8N$, as a default setting whose efficacy and robustness is further validated in Section 4. Moreover, as the Moose instance presented in Figure 3 illustrates, we set k to 1 for the default configuration. Considering the per-level run number may not equal 1 nor the size ratio, we maintain an active run at each level as depicted in Figure 3. During the compaction process, files coming from the previous level are merged with the active run until it reaches its maximum capacity. Should this occur, a new active run is created, and the previous one is designated as static.

With increasing number of unique keys inserted into Moose, a preemptive compaction is triggered on the last level when its maximum capacity is reached, which expands the size N_L. Then,

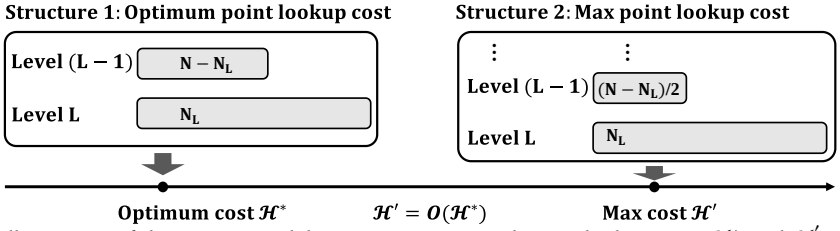

Fig. 4. An illustration of the structures delivering optimum and worst lookup cost, $\mathcal{H}^*$ and $\mathcal{H}^{'}$, respectively. The point lookup performance of Moose is asymptotically optimal since $\mathcal{H}^{'} = O(\mathcal{H}^*)$.

we can deduce a new N based on their relationship (e.g., $N_L = 0.8N$). After that, the introduced algorithm can be repeated to obtain a new structure for the updated N.

3.5 Theoretical Insights

In this section, we show several conclusive optimality results based on the design in Section 3.4. We give the following lemmas.

Lemma 3.2. *Given the last level capacity N_L, the expected point lookup cost in Moose is asymptotically optimal conditioned on optimized range lookup and update costs.*

Proof. Recall the discussion at the end of Section 3.3. The point lookup cost $\mathcal{H}$ deteriorates with the increment of $\mathcal{U}$ based on a given RU-curve. Hence we can evaluate the point lookup cost by examining the maximum and minimum values of $\mathcal{U}(N_1, \ldots, N_L)$ among various legit $N_1, \ldots, N_{L-1}$ as indicated in Figure 4.

$$\begin{aligned}\mathcal{U}(N_1, \ldots, N_L) &= \sum_{i \le L} N_i \log(N_i N_{i-1}) \\ &= N_L \log N_L + N_L \log N_{L-1} + \sum_{i \le L-1} N_i \log(N_i N_{i-1}) \\ &\le N_L \log N_L + N_L \log(N - N_L) \end{aligned} \tag{13}$$

The last inequality holds because $\mathcal{U}(\cdot)$ is maximized when $N_{L-1} = N - N_L$ and all other $N_i = 0$. Inspired by the AM-GM inequality, since $\sum_{i \le L} N_i$ remains constant and $N_i \log(N_i N_{i-1})$ escalates with increased $N_i \cdot N_{i-1}$, the maximum should be reached when the $N_i \cdot N_{i-1}$ values presents substantial deviation. To prove this, let us consider all N_i ($i \ge 3$) are fixed and optimize $\mathcal{U}(\cdot)$ by co-tuning N_1 and N_2, where $\mathcal{U}(N_1, \ldots, N_L) = \Psi + N_3 \log N_2 + N_2 \log N_2 + N_2 \log N_1$ and $\Psi = \sum_{i=4}^{L} N_i \log(N_i N_{i-1}) + N_3 \log N_3$ is a value independent of N_1 and N_2. Since $N_1 + N_2$ is a fixed value now (because all N_i ($i \ge 3$) are fixed and $N_1 + N_2 = N - \sum_{i \ge 3} N_i$), it is easy to verify that $\mathcal{U}(\cdot)$ is maximized when $N_1 = 0$ and $N_2 = N - \sum_{i=3}^{L} N_i$, namely the last two levels are merged. Repetitively applying the procedures on the last two levels, giving us that $N_{L-1} = N - N_L$ is the setting that maximizes $\mathcal{U}(\cdot)$. On the other hand,

$$\begin{aligned}\mathcal{U}(N_1, \ldots, N_L) &= \sum_{i \le L} N_i \log(N_i N_{i-1}) \ge N_L \log N_L + N_L \log N_{L-1} \\ &\ge N_L \log N_L + N_L \log \frac{N - N_L}{2} \end{aligned} \tag{14}$$

The last inequality holds as each level multiplies capacity of the preceding one, thus $N_{L-1} \ge \sum_{i=1}^{L-2} N_i$ and $N_{L-1} + \sum_{i=1}^{L-2} N_i = N - N_L$. Assume the point lookup cost ($\mathcal{H}^{'}(\cdot)$) in our structure is due to setting $N_1^{'}, N_2^{'}, \ldots, N_{L-1}^{'}, N_L$, and the optimum cost ($\mathcal{H}^*(\cdot)$) is by setting $N_1^*, N_2^*, \ldots, N_{L-1}^*, N_L$. Note

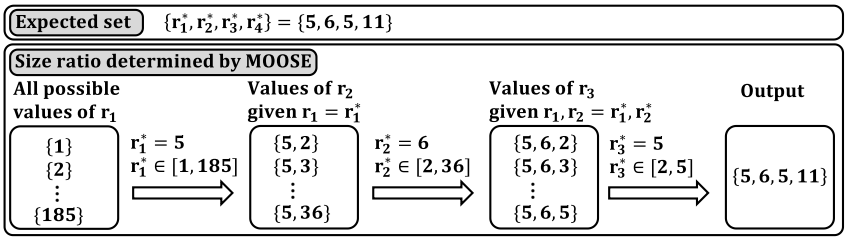

Fig. 5. The expected set falls within Moose's searching space since it explores all possible values of each size ratio.

that $\mathcal{H}(N_1, \ldots, N_L) = \Upsilon \cdot (2^{-0.5\mathcal{U}(N_1,\ldots,N_L)})^{\frac{1}{N}}$, where $\Upsilon = N \cdot k \cdot e^{-\frac{M}{N}(\ln 2)^2}$. Our performance is asymptotically optimal as shown

$$\begin{aligned}
\mathcal{H}'(N_1', \ldots, N_{L-1}', N_L) &= \Upsilon \cdot (2^{-0.5\mathcal{U}(N_1,\ldots,N_L)})^{\frac{1}{N}} \\
\textbf{(By Eq. 14)} &\leq \Upsilon \cdot (N_L^{N_L}(\frac{N - N_L}{2})^{N_L})^{\frac{-0.5}{N}} \\
&= \Upsilon \cdot (N_L^{N_L}(N - N_L)^{N_L})^{\frac{-0.5}{N}} \cdot 2^{\frac{0.5N_L}{N}} \\
\textbf{(By Eq. 13)} &\leq \mathcal{H}^*(N_1^*, \ldots, N_{L-1}^*, N_L) \cdot 2^{\frac{0.5N_L}{N}} \\
&= O(\mathcal{H}^*(N_1^*, \ldots, N_{L-1}^*, N_L))
\end{aligned}$$

□

Lemma 3.3. *Given the last level capacity N_L, the expected product of range lookup cost and update cost in Moose is asymptotically optimal.*

Proof Sketch. Let $RU(r_1, r_2 \ldots, r_L)$ denote the expected product of range lookup and update cost of a given size ratio set $r_1, r_2 \ldots, r_L$. Let $\{r_1^*, r_2^* \ldots, r_{L*}^*\}$ denote the size ratios determined by Moose.

Given $RU(r_1, r_2 \ldots, r_L)$ is the optimal value, the associated value of Moose satisfies $RU(r_1, r_2 \ldots, r_L) \leq RU(r_1^*, r_2^* \ldots, r_{L*}^*)$. We would like to prove the equality holds because the expected set lies within Moose's searching space. To start, there must be $r_1 = r_1^*$ as Moose scrutinizes every feasible value of r_1. Otherwise, the stipulated condition in Equation 12 would be contravened presenting

$$S(r_1^* F, N - r_1^* F) + \sqrt{r_1^*} > S(r_1 F, N - r_1 F) + \sqrt{r_1}$$

Subsequently, we demonstrate that $r_i = r_i^*$ for any $i < L^*$. To this end, let us assume that $\forall i \leq k$, $r_i = r_i^*$ holds. Let $N_d = F \cdot \prod_{i=1}^{k} r_i$ and $N_r = N - F \cdot \sum_{i=1}^{k} \prod_{j=1}^{i} r_j$. Similarly, $r_{k+1} = r_{k+1}^*$ must be valid, otherwise we would achieve

$$S(r_{k+1}^* N_d, N_r - r_{k+1}^* N_d) + \sqrt{r_{k+1}^*} > S(r_{k+1} N_d, N_r - r_{k+1} N_d) + \sqrt{r_{k+1}}$$

which violates the logic of DP algorithm presented in Equation 12. Subsequently, we can establish $r_1 = r_1^*$ to $r_i = r_i^*$ for any $i < L^*$. To assist understanding, we provide a specific example in Figure 5.

All things considered, $RU(r_1, r_2 \ldots, r_L) \leq RU(r_1^*, r_2^* \ldots, r_{L*}^*)$ holds only when size ratios $\{r_1, \ldots, r_L\}$ is identical to Moose's setting $\{r_1^*, r_2^* \ldots, r_{L*}^*\}$. Therefore, the expected product of range lookup cost and update cost in Moose is asymptotically optimal. This completes the proof.

□

Assumptions and Relaxations. Common in most theoretical studies, the cost analysis entails assumptions that the analysis is based on the worst case scenarios, and hence it necessitates the evaluation in real KV-stores of our designs (as will be shown in Experiment Section).

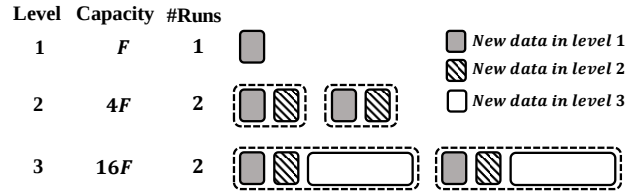

Fig. 6. An example of space amplification analysis. In the worst case, the new data at level 1, level 2, and level 3 have 5, 4, and 2 copies in the LSM-tree respectively. Thus the space amplification is 2.625.

Here we also offer more analysis to explain why our approach still works when removing our previous assumption on the total size of data to be retrieved for a range query $t = O(\sum_{i=1}^{L} n_i B)$. The assumption is applied for bounding the cost of additional $\frac{t}{B}$ I/Os for range lookups in retrieving phase. Considering a more general scenario, let $\bar{t}$ be the average retrieved data size of range lookups, then their average additional I/Os is $\frac{\bar{t}}{B}$. By applying the previous analytical approach for these range lookups and point lookups, there is still a Pareto Curve for them with respect to the expected cost of $U \cdot (\bar{R} - \frac{\bar{t}}{B})$, where $\bar{R}$ refers to the average cost for these range lookups.

Space Amplification. Space amplification is the ratio of the database size to the actual dataset size. For instance, suppose an LSM-tree has 10 runs and each run stores precisely the same entries. The corresponding space amplification is 10 since each unique entry takes up 10 times larger space than its own data size. In other words, the space amplification of LSM-tree is represented by the average number of copies for each unique entry. As Figure 6 presents, in the worst case, each run at a certain level contains identical entries, and these entries are duplicated at each larger level. Therefore, level i contains $(\frac{N_i}{n_i} - \frac{N_{i-1}}{n_{i-1}})$ new entries to the smaller levels (here we let $\frac{N_0}{n_0} = 0$). Each new entry appears at all runs at level i to level L counting to $\sum_{i=1}^{L} n_i$ copies. Additionally, there are $\frac{N_L}{n_L}$ unique entries in total given all unique entries are included by any run at the largest level. Hence we calculate the average number of replicas to derive the space amplification SA using Equation 15.

$$\begin{aligned} SA &= \left(\sum_{i=1}^{L} (\sum_{j=i}^{L} n_j \cdot (\frac{N_i}{n_i} - \frac{N_{i-1}}{n_{i-1}})) \right) \Big/ \frac{N_L}{n_L} \\ &= \frac{n_L}{N_L} \cdot \sum_{j=1}^{L} n_j (\sum_{i=1}^{j} (\frac{N_i}{n_i} - \frac{N_{i-1}}{n_{i-1}})) = \frac{N}{N_L} \cdot n_L \end{aligned} \tag{15}$$

In our configuration, the run number of level L is determined by the ratio between its capacity and that of level $L-1$, which gives $n_L = k \cdot \sqrt{N_L / N_{L-1}}$. Then, we apply the capacity constraint $N_{L-1} < N - N_L$ to identify the upper bound of the space amplification SA. Hence we can formulate SA with N_L and analyze the space efficiency of Moose, as illustrated in Equation 16. As the formula indicates, the space amplification is well bounded when the last level capacity N_L is determined. Therefore, our DP aided three-way tradeoff method presents a satisfactory space efficiency.

$$SA = k \cdot \sqrt{\frac{N_L}{N_{L-1}}} \cdot \frac{N}{N_L} \leq k \cdot N \cdot \sqrt{\frac{1}{N_L (N - N_L)}} \tag{16}$$

As indicated by Equation 16, the space amplification for Moose in default setting is less than 2.5. We further evaluate the real space amplification in Section 4 for more comprehensive analysis.

3.6 Smoose: Workload Aware Moose

Our proposed Moose encompasses three distinct features: the specialized last level capacity N_L, adaptive size ratio $\{r_i\}$, and adjustable run number $n_i = k \cdot \sqrt{r_i}$. Theoretical proof has confirmed

that Moose is able to achieve a commendable three-way tradeoff due to its unique design and flexibility. However, it is important to note that there does not exist a universally optimal structure that consistently delivers peak performance across all workloads. Fortunately, by tuning the three parameters, Moose empowers various tradeoffs among point lookup, range lookup, and update performance to facilitate diverse work conditions. Therefore, we introduce workload aware Moose, dubbed as Smoose, to select the optimal Moose configuration for a certain workload. To achieve this, we propose a tractable structure adaptation method for navigating the design space of Moose, complemented by a comprehensive searchable cost model established to evaluate the tradeoff performance of each potential Moose configuration.

Comprehensive searchable cost model. Our comprehensive searchable cost model is a workload-aware function that quantitatively assesses three-way tradeoff result and overall performance of a Moose structure. This function formulates the average I/O cost of an operation through weighted averaging. To this end, we employ the proportion of range lookup, update, and point lookup operations, denoted as s, u, and z, respectively, to weight their associated I/O costs S, U, and Z as presented in Equation 17.

$$\mathscr{L} = s \cdot S + u \cdot U + z \cdot Z \tag{17}$$

Previous section has discussed the I/O costs associated with different operation types. As the size ratios r_i are derived using the DP algorithm, Smoose only requires adjusting N_L and k to instantiate a particular structure. Consequently, we can reformulate the costs as follows: $S = k\sum_{i=1}^{L}\sqrt{r_i}$, $U = \frac{1}{kB}\sum_{i=1}^{L}\sqrt{r_i}$, and $Z = k\sum_{i=1}^{L} p_i\sqrt{r_i}$. Please note that the cost of non-zero result point lookup is not specialized in the model since its probabilistic is uncertain for Smoose. Moreover, this cost model can be conveniently expanded if the relevant information is furnished [14]. This cost model assists the subsequent tractable structure adaptation process to evaluate the performance of each specific structure within the design space, enabling the identification of the most favorable configuration.

Tractable structure adaptation. A given workload characterized by operation proportions $\{s, u, z\}$ tends to favor a particular tradeoff strategy among point lookups, range lookups, and updates that is associated with a particular Moose configuration. At the same time, this configuration provides the lowest cost in our proposed comprehensive searchable cost model among all potential candidates for the given workload. Hence, we can locate this configuration by exploring within the design space of Moose to minimize the cost model. For this purpose, we introduce a tractable structural adaptation approach to investigate the design space and identify the appropriate configuration based on a specific workload.

Moose has two adjustable parameters: the capacity of last level N_L, and the run number regulator k. To achieve the optimal three-way tradeoff, we design a three-step approach to determine the two parameters. (1) To start with, we enumerate the last level capacity N_L to primarily optimize the tradeoff between point lookup performance and the RU-curve. Given the last level accommodates a dominant portion of the entire data, we iterate through N_L value from $0.5N$ to N, using the buffer size F as step size empirical. (2) For each iteration of N_L, we determine the size ratio $\{r_i\}$ with dynamic programming. This step mitigates the level capacity explosion problem thus resulting in a fine-tuned RU-curve. (3) Subsequently, we enumerate k to adjust the run numbers for attaining optimal overall performance concerning given N_L and $\{r_i\}$, by which the significance of range cost is explored via tuning the tradeoff between range lookup and update. As a result, the desired structure is the most outstanding one among all searched instances in the process which minimizes the comprehensive searchable cost model.

Running example: Assuming N is 100M, we initialize the last level capacity N_L as 50M to obtain corresponding size ratio set, $\{4, 4, 4, 2\}$ with DP algorithm. Then, we fine-tune the number of runs $\{n_i\}$ by varying k from 0.5 to 2 (note $n_i = k\sqrt{r_i}$), among which $k = 1$ yields the minimum cost

Table 3. The output of Smoose in 10 different workloads. "ris" and "nis" are the sequence of size ratios and number of runs of each level, respectively.

	A	B	C	D	E
ris	20,26,19	10,11,11,8	27,27,13	26,20,19	7,7,7,8,3
nis	1,1,1	10,11,11,7	2,2,1	1,1,1	7,7,7,8,3
	F	G	H	I	J
ris	11,11,12,6	11,11,12,6	11,12,11,6	7,7,7,7,3	11,12,11,6
nis	2,2,2,1	2,2,2,1	1,1,1,1	2,2,2,2,1	2,2,2,1

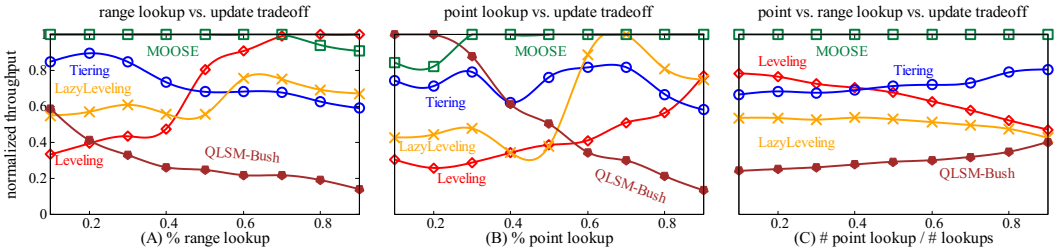

Fig. 7. Evaluating the trade-off between different operations. Moose outperforms all the other baselines in most workloads.

according to Equation 17. Hence, we derive the optimal structure for $N_L = 50$M characterized by the size ratio $\{4, 4, 4, 2\}$ and $k = 1$. Next, we increase N_L from 50M to 99M by 1M and repeat the above procedure to search the desirable structure reaching minimum cost.

Moreover, the tractable structure adaptation presents satisfactory computation efficiency. It takes an average of 1.52s, with a maximum of 4.43s, to determine a desirable structure as N varies from 1GB to 16GB. Therefore, while initially designed for static workload tuning, the impressive computational efficiency of Smoose suggests its potential for supporting future dynamic applications. To be specific, minor workload variations could be accommodated without adaptation and maintain sound performance. While substantial changes in workload require structural transformation to modify capacities and run numbers at each level. Hence, a lazy strategy [41] and preemptive compaction [16] could be utilized to reduce transformation costs while maintaining competitive overall performance.

4 EVALUATION

This section presents the experimental evaluation of Moose and Smoose under diverse workloads. The outcomes indicate that both Moose and Smoose demonstrate remarkable competitiveness compared to various baselines. The experiments are conducted on a server equipped with an Intel(R) Xeon(R) W-2235 CPU at 3.80GHz, 32GB of DDR4 main memory, and a 512GB SATA SSD, running a 64-bit Ubuntu 20.04 on an ext4 partition.

Implementation. Moose and Smoose are implemented on top of RocksDB [22], a well-known LSM-tree storage engine. We extend the basic Bloom filter policy in RocksDB to Monkey Filter policy for all the baselines, which allows us to set a different bits-per-key at different levels. As for Moose (abbr. MSE), we employ the default setting, $N_L = 0.8N$ and $n_i = \sqrt{r_i}$ for run numbers, while calculating the size ratios r_i using dynamic programming.

Baselines. We conduct a comparative analysis of our methodology against several conventional baseline approaches, which encompass Leveling [23] (abbr. Lv), Tiering [32] (abbr. Tr), Lazy-Leveling [15] (abbr. LL), and QLSM-Bush [16] (abbr. Bsh). For Leveling, Tiering, and Lazy-Leveling, we set the size ratio T to 10, in accordance with the default size ratio of RocksDB [22]. In the case of

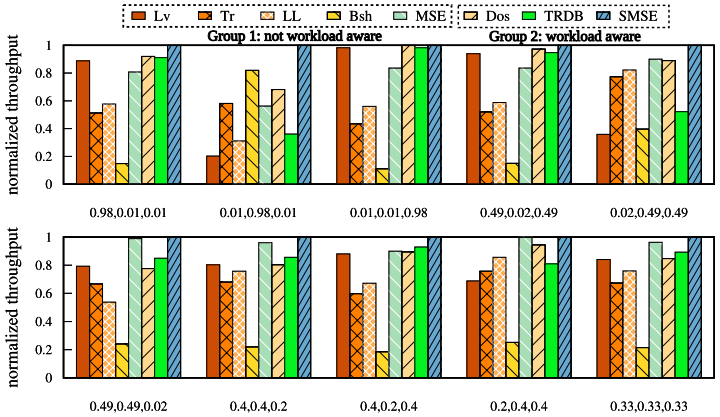

Table 4. Workload composition and performance ranking

	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

Work-	Not workload aware					Workload aware		
Load	Lv	Tr	LL	Bsh	**MSE***	Dos	TRDB	**SMSE***
A	4	7	6	8	5	2	3	1
B	8	4	7	2	5	3	6	1
C	3	7	6	8	5	1	3	1
D	4	7	6	8	5	2	3	1
E	8	5	4	7	2	3	6	1
F	4	6	7	8	2	5	3	1
G	4	7	6	8	2	4	3	1
H	5	7	6	8	3	4	2	1
I	7	6	4	8	1	3	5	1
J	5	7	6	8	2	4	3	1
Avg.	5.2	6.3	5.8	7.3	<u>3.2</u>	3.1	3.7	<u>1</u>

Fig. 8. The composition of the workload is graphed on the x-axis as (range lookup, update, point lookup). In a general sense, Moose surpasses the other baselines in group 1, with the exception of certain highly read/write intensive workloads.Smoose outperforms all other baseline systems across all types of workloads in group 1 and group 2. The table on the right denotes the ranking of each method under each workload. The number underlined is the top-ranked in each group.

QLSM-Bush, we employ $T = 2$ and $X = 2$, following the recommendations outlined in the original paper [16]. The buffer size for all the baseline systems is set at 2MB. Furthermore, we allocate a Bloom filter budget of 5 bits per key, in accordance with the default configuration in RocksDB.

We assess the performance of Smoose (abbr. SMSE) with three-way-mixed workloads and make a comparative evaluation against another workload-aware designs, Dostoevsky [15] (abbr. Dos). To ensure equitable comparisons, we configure both methods using identical workload parameters (i.e., the proportion of each operations). In addition, to illustrate the practical effectiveness, we also compare the result of the tuned RocksDB (abbr. TRDB) [15].

Experiment design. In data preparation phase, we bulk load approximately 11GB data into the database, featuring random key-value pairs. Each pair is comprised of a 24B key and a 1000B value. Subsequently, each baseline will be subjected to testing across a range of workloads, each consisting of 2,000,000 operations.

Overall, Moose exhibits desirable performance and robustness across a spectrum of different workloads. In Figure 7 (A), we evaluate mixed workloads comprising range lookups and updates, with varying proportions of range lookups from 10% to 90%. As the proportion of range lookups increases, Leveling is preferred for its fewer sorted runs. For workloads with substantial updates, Tiering is favored as it is optimized for writes. Lazy-Leveling balances update throughput and range lookup performance, outperforming Tiering in the latter. However, QLSM-Bush, prioritizing updates, compromises range lookup performance, leading to inadequate overall performance. Moose outperforms other baselines in this experiment, except for two range-lookup-intensive workloads, where it is slightly less efficient than Leveling. This validates Moose's ability to achieve a superior tradeoff between range lookups and updates.

In Figure 7 (B), the tested workload mixes point lookups and updates, varying the proportion of point lookups from 10% to 90% to examine Moose's ability on point lookup-update tradeoff. QLSM-Bush excels in substantial update scenarios but degrades with heavy reads. Tiering is preferable for update-dominant workloads, while Leveling is better for read-dominant scenarios. Lazy-Leveling achieves a balanced performance by combining Tiering and Leveling policies. Moose showcases

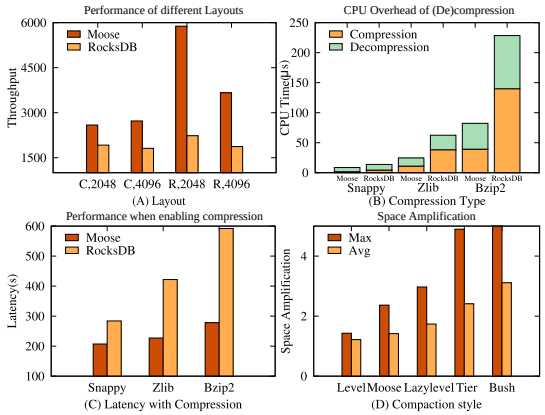

Fig. 9. (A) presents the capability of Moose under different storage layouts (noted as "Layout,Key-Value-Size", "C" means "Column Store" while "R" means "Row Store") compared to RocksDB; (B) and (C) shows the performance after integrating different compression policies; (D) presents the actual space amplification of various methods

Table 5. The number of IO per query, compacted bytes (GB), and the number of compaction in format "#IO/compaction byte/#compaction".

	Not workload aware					Workload aware		
	Leveling	Tiering	LazyLevel	QLSM-Bush	**MOOSE***	Dostoevsky	TRDB	**SMOOSE***
A	4.23/1/6	17.9/10/1	14.0/2/1	139/0/0	5.98/2/6	3.45/1/5	3.45/1/206	2.29/1/6
B	4.25/31/561	0.96/6/103	2.20/16/103	1.63/2/4	1.68/13/656	1.13/8/102	2.56/19/1084	0.68/4/100
C	0.21/31/6	0.97/6/1	0.40/15/1	2.34/2/0	0.28/13/6	0.19/8/6	0.28/19/247	0.14/4/6
D	2.24/1/12	10.1/1/2	7.91/1/2	72.9/0/0	3.22/1/12	1.85/1/11	1.98/2/254	1.46/1/13
E	2.32/17/283	1.10/3/52	0.80/3/52	3.13/1/2	1.02/6/328	0.89/5/189	1.63/10/447	0.80/3/73
F	4.56/33/302	11.5/6/52	10.9/16/52	67.9/2/2	4.48/12/329	4.59/21/302	4.10/22/511	3.61/12/329
G	3.67/29/247	9.18/6/42	7.48/15/42	59.8/2/2	3.66/11/268	3.69/29/246	3.37/24/451	2.89/17/268
H	2.69/19/123	8.68/3/21	6.89/3/21	57.6/2/1	3.24/7/134	2.70/19/123	2.66/15/332	2.10/11/133
I	2.73/19/246	4.98/3/42	3.99/3/42	30.1/2/2	2.24/7/268	2.22/13/295	2.59/13/320	1.90/9/268
J	2.94/22/203	7.57/3/34	6.08/3/34	49.5/2/1	3.06/9/221	2.95/17/203	2.79/17/406	2.35/12/222

nearly ideal throughput in write-intensive workloads and remarkably outperforms all the baselines under most workloads, demonstrating the capability of Moose in maintaining a desired point lookup-update tradeoff.

In Figure 7 (C), to further explore the tradeoffs between mixed read operations, including point lookups and range lookups, we vary the ratio of point lookup over the total number of reads. The total number of reads is fixed to 50% of the entire operations while the remaining are updates. For workloads with a lower proportion of range lookups, Tiering and QLSM-Bush perform better due to their retention of significant sorted runs. Conversely, in range-lookup-intensive scenarios with a low proportion of point lookups, Leveling excels. The cost of Lazy-Leveling is relatively higher than Tiering due to the presence of 50% updates. Despite maintaining numerous sorted runs to reduce update costs, Lazy-Leveling lags behind Leveling in this experiment when there is a high proportion of range lookups. Moose outperforms all other approaches in all workloads, indicating its ability to strike the best tradeoff among the three types of operations.

In general, our proposed Moose delivers more impressive and robust performance for different operation portions for range lookups and updates in comparison with all baselines, indicating Moose's positioning along a superior RU-curve. Besides, it does not affect the point lookup performance thus striking outstanding performance for all workloads. Despite the remarkably changing workloads, Moose consistently proves highly competitive, showcasing its robustness and effective optimization compared to existing designs.

Exploring a wider space enables Smoose to outperform all baselines in three-way mixed workloads. The baselines in this experiment can be divided into two groups, the non-workload-aware group and the workload-aware group. Following the setting of existing works (e.g., [8, 26]), we conduct the three-way-mixed workloads for Leveling, Tiering, QLSM-Bush, Moose, Dostoevsky, and Smoose. The composition of operations in each workload and the result are depicted in Figure 8. Generally, Smoose outperforms all baselines or demonstrates comparable performance across all workloads. Specifically for the non-workload-aware group, no one baseline can be the best setting under all the workloads, but Moose performs the best or near the best in most cases, indicating its strong robustness. This can be attributed to Moose's adeptness in achieving a well-balanced tradeoff

among the three operations. Though Moose cannot maintain such advantage as the workloads become not even, its performance is still around the best settings.

To bridge this gap, Smoose leverages the proportions of each operation type, enabling the dynamic programming algorithm to adjust the contribution of each operation. Therefore, Smoose consistently outperforms or achieves comparable results to other methods, including Moose. In comparison to other well-established workload-aware baselines, Dostoevsky and tuned RocksDB, Smoose demonstrates superiority in most cases though Dostoevsky presents comparative performance with Smoose under workload C. This outcome is attributed to the broader design space explored by Smoose. As previously mentioned, Smoose can vary the size ratios across different levels and obtain different run numbers at varying levels with fewer constraints by adjusting the parameter k. In contrast, Dostoevsky maintains a fixed size ratio across different levels and sets only two distinct run numbers for the last level and all the non-last levels. Consequently, Dostoevsky may overlook the better design space explored by Smoose. Similarly, though we can tune RocksDB with various size ratios for different workloads, the narrow design space restrains its capability to achieve competitive performance except for some read-intensive scenarios.

Let us consider workload G, where the composition of range lookups, updates, and point lookups is 40%, 40%, and 20%. Dostoevsky tunes within its configuration space and derives a Leveling-like structure with T=10 for this workload. This outcome aligns with expectations, as read operations (range lookups and point lookups) constitute 60% of the overall workload, favoring structures with a lower number of runs. The experimental results further confirm that this configuration surpasses all competitors except for Moose and Smoose. However, Moose's size ratios and run numbers fall outside the search space of Dostoevsky. This configuration achieves better performance because it takes into account the optimization tradeoff between range lookups and updates simultaneously. Moreover, though Moose already demonstrates exceptional performance, Smoose manages to discover a more desirable structure by fine-tuning parameters such as k and N_L. Through adjustments to the dynamic programming cost function that considers the workload composition, Smoose conducts a more precise search. Furthermore, Smoose can adapt N_L according to the proportion of point lookups, resulting in a more balanced performance across all three operations. Smoose selects size ratios of 11 or 12 for all non-last levels with $N_L = 0.85N$ and run numbers set to 2 for all non-last levels, which also lie outside the search space of Dostoevsky. The output of Smoose for different workloads is shown in Table 3.

To further illustrate the capability of Smoose and Moose, we present the raw metrics, such as the number of IO and compaction, and the compacted bytes in Table 5. It is obvious that in all cases, Smoose demonstrate an outstanding performance while Moose can also perform robustly within the non-workload-aware group.

Overall, Moose consistently exhibits outstanding performance, albeit with slight variations when one type of operation dominates. Therefore, we introduce Smoose to utilize workload information. Compared to Dostoevsky, Smoose's structural flexibility enables it to potentially identify more desirable choices for specific workloads.

Discussion 1: Impact of various k. Figure 10 (A) shows how the choices of k might affect the performance on reads and updates. In this experiment, we range k from $1/\sqrt{r_{max}}$ to $\sqrt{r_{max}}$ and gather the cumulative latency for each specific operation type, where r_{max} represents the maximum size ratio. Since the run number for each level must be a non-zero integer and not exceed the size ratio, thus rounding is required when this rule is violated. Evidently, as k increases, the performance of lookups tends to decrease because higher k values may increase the number of sorted runs at each level, potentially deteriorating read performance. Conversely, reduced WA per level may enhance update performance. Notably, the cost of point lookups is significantly lower than other

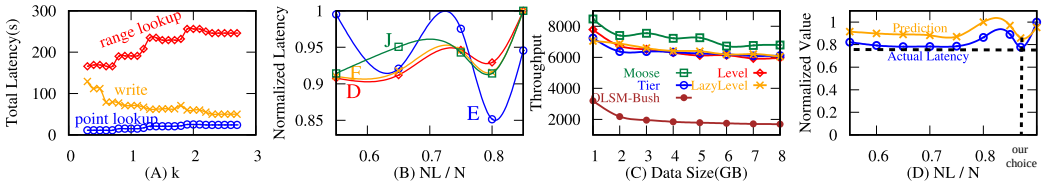

Fig. 10. (A) presents total latency of each operation indicating the positive/negative correlation between k and update/lookups, respectively. (B) shows the reason why we select $N_L = 0.8N$ as Moose's default setting; (C) presents Moose's consistent superiority on growing N; (D) shows the accurate prediction of Smoose

operations, prompting our search algorithm to prioritize optimizing range lookups and updates, even if it means sacrificing some point lookup performance.

Discussion 2: Impact of different choices of N_L. In Moose we employ the hyper-parameter N_L to tune the tradeoff between point lookups and overall performance of range lookups and updates. To substantiate our default choice of $N_L = 0.8N$ in Moose, we maintain a constant value $k = 1$ and employ workload D, E, F, and J, the two-way/three-way uniformed workloads, to examine its efficacy, as is shown in Figure 10 (B). Evidently, a local minimum point exists near $N_L = 0.8N$ across all workloads. Specifically, for point lookup intensive workloads like D and E, a larger N_L is preferable. However, this may compromise the tradeoff between range lookups and updates. For achieving satisfactory overall performance, $N_L = 0.8N$ presents optimal point lookup performance without significantly compromising range lookup and update performance across all workloads. Though $N_L = 0.65N$ seems competitive in the figure, it cannot win the comprehensive gains compared to our choice. For instance, the size ratios and run numbers for them are $\{12, 17, 18, 2\}/\{3, 4, 4, 1\}$ and $\{7, 7, 7, 6, 5\}/\{3, 3, 3, 2, 2\}$, respectively. The latter suffers higher cumulative overall cost due to its smaller N_L though presenting marginally better update cost.

Discussion 3: Performance with growing N. In Section 3.4, we address the scenario of growing N. In Figure 10 (C), we examine Moose with 8GB data inserted under a balanced workload. Generally, Moose showcases superiority throughout the entire process. LazyLeveling slightly outperforms Leveling and Tiering, while QLSM-Bush exhibits poor performance due to a high number of range lookup operations.

Discussion 4: Prediction accuracy of cost model. We vary the last level capacity and compare predicted costs with actual latencies. For clarity, we normalize them respectively in Figure 10 (D) since predicted costs pertain to I/O times. The alignment between predictions and actual values underscores the efficacy of our cost model in guiding structural tuning. Moreover, the structure identified by Smoose consistently corresponds to the actual optimal latency.

Discussion 5: Effectiveness in practical environments. We expand the scope of Moose to accommodate varying key-value sizes and storage layouts (column/row store) to assess its effectiveness in practical systems. Moreover, we incorporate column accessing into the workload to realize the performance enhancement presented in most associated database systems [5, 24, 51]. In Figure 9 (A), Moose outperforms RocksDB significantly, affirming its effectiveness across diverse access patterns and storage layouts. Specifically, adopting column layout does not compromise the overall superiority of Moose across varying key-value sizes. This is attributed to the fact Moose directly optimizes the comprehensive I/O cost. Since column accessing, though retrieves only portions of the value, it presents similar I/O costs to regular access operations and can still benefit from Moose. Regarding another crucial industry feature, SSTable compression, we evaluate Moose with Snappy, Zlib, and BZip2 compression algorithms and compare its performance with RocksDB in Figure 9 Parts (B) and (C). Obviously, Moose exhibits significantly lower latency. This is because an SSTable

is compressed or decompressed to/from disks only for write or read operations. Consequently, the compression time is generally proportional to I/O times. Hence the additional overhead introduced by compression/decompression could be mitigated by Moose concurrently.

Discussion 6: Space amplification evaluation. Figure 9 (D) evaluates the actual space amplification (SA) with N growing from 1GB to 12GB. Notably, Leveling exhibits the smallest SA, while Moose ranks the second smallest. Note that, Leveling represents the optimal compaction style in terms of SA due to its having only one sorted run at each level. This experiment validates the satisfactory performance of Moose concerning SA.

Summary. Overall, while each design has adept workloads, Moose achieves desirable performance when the workload changes. We further introduce Smoose, a versatile and searchable design with only a few knobs, to adapt to specific workloads and outperform the existing continuum design as presented in the experiments.

5 RELATED WORK

Key-value stores. Over the past decade, numerous research has emerged concerning key-value stores (KVS). Some hardware oriented studies [6, 20, 33, 52, 55, 57, 60, 64] design specialized KVS to deploy on modern storage devices, including advanced SSDs [20, 52, 60] and persistent memory [33, 66, 68], to enhance the parallel processing capabilities [57] and write performance [6, 55, 64]. For facilitating cloud servers, Idreos *et al.* [28] present a KVS design space consisting of various hardware and data layout strategies [3, 49]. Based on this, Cosine [8] is proposed to quantify the overhead of KVS on the cloud and tune for specific workload distribution. Conway *et al.* [11] propose Mapped SplinterDB to improve lookup performance of SplinterDB. Wang *et al.* [58] reduce the software overhead for the KVS designed in memory disaggregated architecture. Compared with our work, these studies are oriented towards enhancing diverse key-value stores or full DBMS like Spanner [5], rather than being specifically tailored to the LSM-trees.

LSM-tree performance optimization. To enhance point lookup performance, Monkey [14] assigns more Bloom filter BPK for smaller levels. Li *et al.* [67] and Zhu *et al.* [69] reach the same target by applying Bloom filter modification. Moreover, Chucky [17] and SSCF [34] replace the Bloom filter with alternative components, which allows them to reach superior point lookup performance with reduced storage overhead. Besides, SA-LSM [63] and Bi-directional LSM-tree [65] change data distribution to speed up lookup operations of frequently accessed data for handling the skewed workload. Vu [56] *et al.*study the incremental spatial partitioning problem to decrease the lookup latency for big data [21].

To boost range lookup efficiency, various range filters have been proposed. Zhang *et al.* [62] record the inserted keys with fast succinct tries to utilize the sequential information. Luo *et al.* [38] proposed a Bloom filter based method to encode the existence of keys with hierarchical structure, which is subsequently enhanced by bloomRF [42]. Based on these works, Proteus [30] and REncoder [59] are proposed to employ hybrid structures. Moreover, Vaidya *et al.* [54] introduced learned model which is followed by OASIS [9] and presented satisfactory performance. To facilitate adversarial queries, Grafite [12] is designed with impressive space complexity.

For pursuing higher update performance, many works utilize modified compaction schema including delayed compaction [44], light-weight compaction [53, 61], and compaction warm-up [1]. To start with, Patrick [43] proves a fixed size ratio is update-friendly for Level-like structures. Wisckey [36] and HashKV [7] demonstrate that key-value separation is another approach to improve update performance. Additionally, Dai *et al.* [13] propose Bourbon, a learned index, to accelerate the point lookup operations for these methods. Meanwhile, Nova-LSM [25] parallelizes compaction by storage and process separation. Luo *et al.* [37] and Kim *et al.* [29] identify memory

contention between buffer and Bloom filter, allowing them to adjust point lookup and update trade-off by tuning memory allocation strategy. For mitigating the issue of space amplification, Sarkar *et al.* [47], Dayan *et al.* [18], and Alkowaileet *et al.* [2] modify the compaction granularity [40] to expedite garbage collection. These approaches often focus on single or dual optimization objectives, rather than simultaneously optimizing point lookup, range lookup, and update. Besides, they are typically orthogonal to our work, as they rarely involve tuning the LSM-tree structure.

LSM-tree structure tuning. The works optimizing the LSM-tree performance via tuning the LSM-tree structure are more related to our work. Endure [26] leverages a globally consistent size ratio and selects compaction policies between leveling and tiering. Sarkar *et al.* [48] present the design space of LSM-tree compaction and evaluate various compaction strategies. Dostoevsky [15] enables more flexible compaction by adjusting the number of runs per level, ranging from 1 to the size ratio, for the last and non-last levels. Ruskey[41] proposes a further extended design space by allowing flexible compaction for all levels and utilizes a reinforcement learning algorithm [50] to explore it. K-LSM [27] also considers flexible compactions across levels, while integrating a novel tuning approach to determine the structure for facilitating workload uncertainty. It proposed a general solution that could also empower Smoose to handle workload uncertainty with modified cost model. For example, uncertain workload distribution can be described with KL-Divergence and formulated as $\mathcal{W} = \{\hat{w} \in \mathbb{R}^3 | \hat{w} \geq 0, \hat{w}^T e = 1, I_{KL}(\hat{w}, w) \leq \rho\}$, where ρ describes the divergent range, w is the target workload indicating the proportion of range read(s), update(u), point lookup(z) at each dimension, and I_{KL} is the KL-Divergence between the target workload and the uncertain workload $\hat{w}$. Accordingly, the cost function of Smoose turns out to be $\max_{\hat{w} \in \mathcal{W}}(\hat{s}S + \hat{u}U + \hat{z}Z)$, which could be minimized using Lagrange Multiplier for deriving the suitable structure. Moreover, in our paper, we further broadened and explored the LSM-tree design space by considering the level-wise adjustment of size ratios and run numbers. This introduces notable challenges, necessitating the non-trivial optimization techniques proposed in our work. Dayan *et al.* [16] explore this constraint by introducing an exponentially decreasing size ratio between adjacent levels and propose QLSM-Bush, which achieves satisfactory update performance. Comparatively, we explore a more extensive design space thus empowering the LSM-tree to achieve a superior three-way trade-off.

6 CONCLUSION

Current LSM-tree key-value stores struggle to optimize all the three operations concurrently due to rigid configurations. To address this, we remove constraints on run number, size ratio, and Bloom filter, facilitating a comprehensive theoretical analysis of the cost for each operation. We present Moose, with a specialized last level optimized for point lookup and a flexible size ratio benefiting range lookup and update. In response to diverse workloads, we introduce Smoose, a workload-aware structure, to achieve an optimal tradeoff. Both Moose and Smoose demonstrate a desirable three-way tradeoff and outstanding performance, as validated theoretically and experimentally. In the future, it would be interesting to integrate our work into actual DBMS [5, 24, 51] or LSM-Tree Index to examine the efficacy and attain further gains.

ACKNOWLEDGMENTS

This research is supported by NTU-NAP startup grant (022029-00001). We thank the anonymous reviews for their valuable suggestions.

REFERENCES

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

[2] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. 2019. An LSM-based Tuple Compaction Framework for Apache AsterixDB (Extended Version). *arXiv preprint arXiv:1910.08185* (2019).

[3] Wail Y Alkowaileet and Michael J Carey. 2021. Columnar formats for schemaless LSM-based document stores. *arXiv preprint arXiv:2111.11517* (2021).

[4] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *SIGMOD*. 1185–1196.

[5] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. 2017. Spanner: Becoming a SQL system. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 331–343.

[6] 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.

[7] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.

[8] 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.

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

[10] Source Code. 2022. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[11] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[12] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming Adversarial Queries with Optimal Range Filters. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–23.

[13] 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.

[14] 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.

[15] 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. https://doi.org/10.1145/3183713.3196927

[16] 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.

[17] 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.

[18] 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.

[19] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[20] 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.

[21] Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh, Majid Saeedan, Akil Sevim, AB Siddique, Samriddhi Singla, Ganesh Sivaram, Tin Vu, and Yaming Zhang. 2021. Beast: Scalable exploratory analytics on spatio-temporal data. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3796–3807.

[22] Facebook. 2022. RocksDB. https://github.com/facebook/rocksdb.

[23] Google. 2022. LevelDB. https://github.com/google/leveldb/.

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

[25] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: a distributed, component-based LSM-tree key-value store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.

[26] 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.

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

[28] 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*.

[29] 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.

[30] 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.

[31] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. PVLDB 11, 6 (2018), 677–690.

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

[33] 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.

[34] 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.

[35] LinkedIn. 2022. Voldemort. http://www.project-voldemort.com.

[36] 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.

[37] 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.

[38] 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.

[39] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. 2018. Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *Proceedings of the VLDB Endowment* 11, 5 (2018), 594–606.

[40] Qizhong Mao, Mohiuddin Abdul Qader, and Vagelis Hristidis. 2020. Comprehensive comparison of LSM architectures for spatial data. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 455–460.

[41] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *arXiv preprint arXiv:2308.07013* (2023).

[42] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2023. bloomRF: On performing range-queries in Bloom-Filters with piecewise-monotone hash functions and prefix hashing. In *Advances in database technology: Proceedings of the 26th International Conference on Extending database Technology (EDBT), 28th March-31st March 2023, Ioannina, Greece*, Vol. 26. Open Proceedings. org, Univ. of Konstanz, 131–143.

[43] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[44] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.

[45] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. {mLSM}: Making Authenticated Storage Faster in Ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[46] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. Littletable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 125–138.

[47] 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.

[48] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2022. Constructing and Analyzing the LSM Compaction Design Space (Updated Version). *arXiv preprint arXiv:2202.04522* (2022).

[49] 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.

[50] AB Siddique, Samet Oymak, and Vagelis Hristidis. 2020. Unsupervised paraphrasing via deep reinforcement learning. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 1800–1809.

[51] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020*

ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[52] 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.

[53] 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.

[54] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644.

[55] 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.

[56] Tin Vu, Ahmed Eldawy, Vagelis Hristidis, and Vassilis Tsotras. 2021. Incremental partitioning for efficient spatial data analytics. *Proceedings of the VLDB Endowment* 15, 3 (2021), 713–726.

[57] 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.

[58] Ruihong Wang, Jianguo Wang, Prishita Kadam, M Tamer Özsu, and Walid G Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2835–2849.

[59] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2036–2049.

[60] 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.

[61] 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.

[62] 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.

[63] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM: optimize data layout for LSM-tree based storage using survival analysis. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2161–2174.

[64] 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.

[65] Xin Zhang, Qizhong Mao, Ahmed Eldawy, Vagelis Hristidis, and Yihan Sun. 2022. Bi-directional Log-Structured Merge Tree. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management*. 1–4.

[66] 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.

[67] 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)*.

[68] 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.

[69] 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.

Received October 2023; revised January 2024; accepted February 2024