Low Unleashes More: Separating low-Access Data for Optimal Read/Write in LSM-based Key-Value Stores

Zizhao Wang
SIAT, UCAS
zz.wang@siat.ac.cn

Yunjue Gu
SIAT, CAS
yj.gu1@siat.ac.cn

Wenhan Feng
ISCAS, University of CAS
fengwenhan21@otcaix.iscas.ac.cn

Jintao Meng
SIAT, CAS
jt.meng@siat.ac.cn

Yang Wang
SIAT, CAS
yang.wang1@siat.ac.cn

André Brinkmann
Johannes Gutenberg University Mainz
brinkman@uni-mainz.de

ABSTRACT

LSM-based key-value stores are widely used in modern applications, but still suffer from significant write amplification due to frequent compactions. Existing work has employed hot-cold separation strategies to mitigate write amplification. However, these approaches are often based on inappropriate assumptions, making their strategies effective only in limited scenarios.

LuMDB, which is presented in this paper, addresses this problem based on four insights and optimizations. ❶ A compaction model to analyze **key factors** that impact the compaction. ❷ A **theoretical model** is proposed to analyze the optimal hot-cold separation strategy. ❸ A **workload skewness detection** automatically determines whether to use hot-cold separation and how to set the threshold for hot data identification. ❹ A ***two-phase partitioning leveling*** (**2PPL**) methodology is developed to store cold data using the separation, which can automatically determine the partition range configuration based on memtable size and workload skewness. A Tiering strategy is used to process and store the remaining hot data in LuMDB when separation is applied. For cases not suitable for hot-cold separation, LuMDB stores all data with 2PPL. Extensive experiments show that LuMDB can reduce write amplification by up to 8.1×, approaching its minimum theoretical limit, and increase read throughput by up to 3.7× in the YCSB benchmark.

PVLDB Reference Format:
Zizhao Wang, Yunjue Gu, Wenhan Feng, Jintao Meng, Yang Wang, and André Brinkmann. Low Unleashes More: Separating low-Access Data for Optimal Read/Write in LSM-based Key-Value Stores. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Zizhao-Wang/LuMDB.

1 INTRODUCTION

Log-Structured Merge (LSM) tree-based key-value (KV) stores are used to handle write-intensive workloads with high write throughput [36]. They are widely embedded in modern applications, such as BigTable [10], LevelDB [18], RocksDB [32], MongoDB [6], and Cassandra [9]. These stores serve as the storage backend for numerous scenarios, including LLM system development [13], blockchain systems [27, 40], web applications [29, 35], and IoT applications [11, 46].

LSM-based KV stores optimize write performance by buffering writes first in a memtable and then organizing data across multiple levels. However, the multi-level structure can increase read latency because reads must check the memtable and each level sequentially. To mitigate this, LSM trees perform compaction [45] to reorganize data across levels, which improves read performance and reduces space amplification [44] by merging different versions of data into one. However, this process results in write amplification (WA), where the amount of data written to disk exceeds the user writes [26]. High write amplification consumes more disk bandwidth, increases write stalls [55] and accelerates disk wear [55].

An effective way to reduce write amplification is to exploit workload characteristics [1], such as data access patterns. Many studies [2, 7] assume that frequent compaction is mainly caused by some frequently accessed data (typically the top 1~10%) under skewed workloads [1]. As a result, these studies focus on separating this hot data from cold data. These approaches can be categorized into two main directions: 1) Some studies try to identify hot data at runtime and periodically move these data to faster storage devices [2, 12]. 2) Other studies [7, 38, 48] assume that under skewed workloads, hot data is primarily stored in the top levels of the LSM tree and thus, these top levels are responsible for most of the write I/Os. Therefore, they suggest separating the top levels and storing them on faster storage devices.

Our experiments show that these approaches are only effective for high-skew workloads or for special configurations where their assumptions are met. ❶ In the first case, less hot data in the LSM trees does not guarantee more write amplification reduction. For example, the effectiveness of hot-cold data separation decreases under low-skew workloads. ❷ In the second case, the assumption that top levels induce more write I/Os is only valid for special configurations, e.g., for a small memtable.

These restricted assumptions prevent current hot-cold data separation schemes for LSM trees from effectively reducing write amplification. Therefore, we try to propose a hot-cold separation method that can work under all kinds of workloads. To do that, we first propose a model to analyze the compaction process and then summarize the key factors affecting read and write performance. We then perform a theoretical analysis to obtain a more precise relationship between these factors and write amplification under workloads of different skewness. Finally, we find that 1) the more skewed the workload, the greater the benefit of separation, and

2) when hot-cold separation is required, the more thorough the hot-cold separation, the better the effect.

Based on these findings, we propose LuMDB (Low unleashes More), the first LSM-based KV store that uses the variance of data access frequencies to represent and exploit data skewness. LuMDB includes a novel *two-phase partitioning leveling*, which automatically determines the partition range in Level 0 (L_0) based on the memtable size. During compaction in L_0, LuMDB computes data skewness and identifies hot data at runtime. If separation is applied, LuMDB repartitions L_1, storing cold data in 2PPL and hot data in Tiering. If separation is not required, all data is stored in *two-phase partitioning leveling*.

In summary, the contributions of this paper include:

- For the first time, we find that almost all hot-cold separation approaches in existing LSM-based KVs rely on assumptions that are only valid under high-skew workloads or in special configurations.
- We present a model of the compaction process and summarize some key factors that affect write amplification and write/read performance. Based on the key factors, we analyze the trade-offs of some widely used LSM tree-based KV stores.
- For the first time, we provide a theoretical analysis to determine the optimal hot-cold separation design for an LSM tree-based KV store. Specifically, we figure out when separation should be integrated and what the optimal separation bound is for all skewnesses of workloads.
- Based on the theoretical analysis result, we propose LuMDB. In LuMDB, we integrate a novel hot data definition, a novel hot data identification mechanism, and a novel data structure, i.e., *two-phase partitioning leveling*, to store data.
- We evaluate LuMDB under YCSB workloads of different skewness. We find that LuMDB can reduce write amplification to approximately 1, approaching the theoretical minimum of write amplification and achieve a reduction of up to 8.1 times compared to other state-of-the-art KV stores. Additionally, LuMDB can improve read performance by up to 3.7 times and 2.5 times for point and range queries.

The rest of this paper is organized as follows. Section 2 reviews the background of current LSM-based KV stores. Section 3 outlines the motivation for our work. Section 4 discusses our model for analyzing current compaction policies. Section 5 introduces our theoretical analysis. Section 6 describes our proposed LuMDB. Finally, Section 7 presents the evaluation results of LuMDB. Section 8 discusses related work. Section 9 concludes the paper.

2 BACKGROUND

2.1 LSM tree-based KV stores

LSM trees are designed for write-intensive workloads [1] and trade off their high write performance by partially sacrificing read performance [42]. LSM trees typically store data in the form of KV pairs in an underlying file system (e.g., EXT4 [28]) using a multi-level structure denoted from L_0 to L_{max}. The size of each level increases by a factor f, which is usually set to 10. In LSM tree-based KV stores, data is first written to an in-memory memtable, which is often organized as a skiplist [37]. When the memtable reaches its capacity, it is converted to an *immutable* [18] and can no longer be modified. The immutable is then sequentially written to disk at L_0 as a sorted strings table (SSTable), which is also known as **minor compaction**.

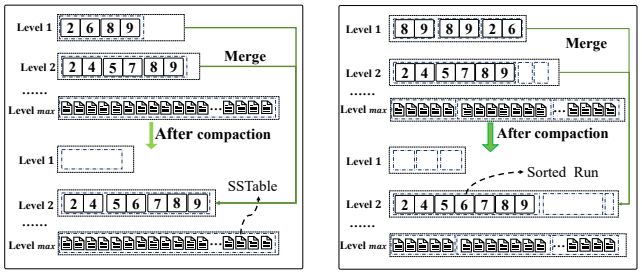
The read process in LSM-based KV stores involves first checking the memtable, then the immutable, and finally traversing the disk levels from L_0 down L_{max} until the desired value is found [25]. There are two types of read operations: point queries, which search for a value based on a user-provided `key`, and range queries, which start from a user-provided `key` and return the next `iter_len` valid KV pairs. Both query types may need to traverse several levels, which increases read latency [58]. To reduce unnecessary checks, Bloom filters [5] are often used to quickly determine if a key can be found in a particular SSTable.

2.2 Compaction in LSM-based KV stores

To improve read performance and reduce space amplification [17], LSM-based KV stores employ a process called **compaction**. Compaction can reorganize data across levels, consolidating scattered or redundant entries to ensure faster reads and better space utilization. When an in-disk level (L_i) reaches its capacity, known as level saturation [42], it triggers a compaction process with the next level (L_{i+1}). Compaction involves reading SSTables from both levels, merging duplicate entries, removing outdated versions of KV pairs, and writing the consolidated data to L_{i+1}. After compaction, each level contains a single sorted sequence of SSTables, also known as a run [26].

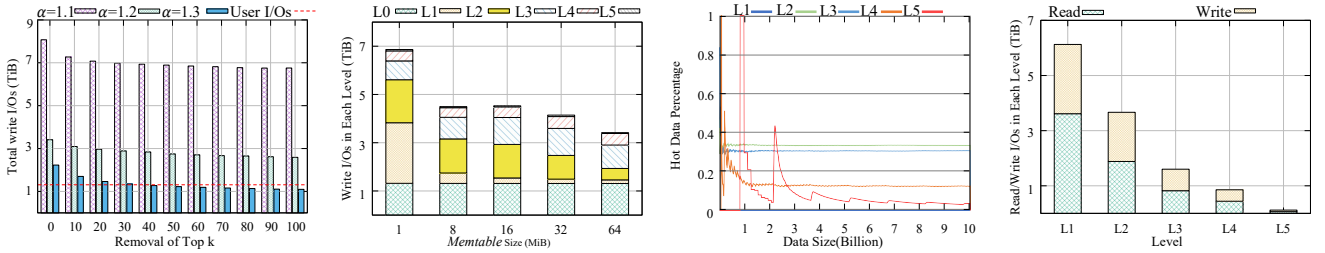
Under read-intensive workloads, more frequent compaction can reduce the average read latency, providing a trade-off between write amplification and read latency. In contrast, under write-intensive workloads [45], frequent compaction significantly increases write amplification [43].

Figure 1a shows that this approach, known as Leveling, allows only one version of each key to exist in each level (except for L_0) [45] after compaction. An alternative data organization method



(a) Leveling **(b) Tiering**

Figure 1: In the above leveling, when L_1 is full, some SSTables in L_1 and all overlapping SSTables in L_2 are selected. These SSTables are then merged in memory and re-written to L_2. In the above tiering, each level can accommodate several runs of overlapping key ranges. When these runs reach their capacity limit in L_1, all active runs of L_2 are selected and these runs are merged and written to L_2.



(a) Total I/Os of remaining data **(b) I/Os while varying memtable** **(c) Hot data cause compactions** **(d) Read and Write I/Os by level**

Figure 2: LevelDB performance diagnosis. We write 10 billion 144-byte KV pairs (1.3 TiB) for each experiment. (a) When the memtable size is set to 64 MiB, the write I/O reduction of the remaining data after removing the top k KV pairs does not change much for bigger k and $\alpha = 1.1$. (b) When $\alpha = 1.2$, the write I/Os of each level decreases while increasing the memtable size. (c) Asymmetry in read/write I/Os by level when $\alpha = 1.1$. (d) The percentage of compactions triggered by hot data at different levels.

is called Tiering, where each level can contain multiple Runs (typically three) [30]. Compared to Leveling, Tiering offers advantages in reducing write amplification by delaying cross-level merges (see Figure 1b).

2.3 Hot-Cold Separation in Leveling

Many studies propose hot-cold data separation to reduce write amplification in LSM-based KVs for skewed workloads [1, 14, 19]. These approaches can be broadly categorized into: 1) **data handling approaches**, where KV pairs are classified into *hot* (frequently accessed) and *cold* (less frequently accessed) based on their key access frequencies, with hot data stored in faster storage and cold data in slower storage. And: 2) **data structure handling approaches**, where top levels of the LSM tree are treated as hot levels and lower levels as cold levels. Data structure handling approaches assume that hot data is primarily stored in the hot levels and that these levels contribute to most data rewrites during compaction. Consequently, they suggest placing hot levels on faster storage, such as persistent memory (PM) [53].

Both approaches have typically been implemented and evaluated based on modifications of LevelDB and RocksDB. Our experiments in Appendix A show that Tiering-based KV stores (e.g. PebblesDB) typically have a larger capacity of L_0 and a lower f compared to Leveling-based approaches. This configuration results in more data at the top levels being merged in batches with the next level; data writes are distributed across multiple levels. We found that for this configuration, using hot-cold data separation provides little improvement in write amplification for both high-skew and low-skew workloads and also has a negative impact on read performance.

3 MOTIVATION

Although many studies have optimized LSM trees through hot-cold data separation, we find that most of them rely on partially inappropriate assumptions about real-world workloads. In particular, we conducted a series of experiments and found that: 1) hot data separation works well only for high-skew workloads; 2) top levels of LSM do not have more data rewrites; and 3) cold data are the real villains of compaction.

3.1 How effective is separating hot data?

Data handling approaches assume that **hot data causes frequent compactions and therefore leads to a higher write amplification**. Therefore, they move hot data from slower storage (e.g., HDDs) to faster storage (e.g., SSDs or DRAM). However, this is not always effective. Figure 2a shows the total write I/Os (TWI) computed by measuring the data rewritten in each compaction after removing the top k entries from each memtable (the memtable size is 64 MiB and the KV size is 144 bytes) for workloads with varying skewness (i.e., α[1]) of the underlying Zipfian distribution [33].

Regardless of the workload, removing more high-frequency entries consistently results in a decrease in TWI. However, the magnitude of the decrease varies depending on the skewness of the workloads. For example, under low-skew workloads (e.g., α=1.1), removing the top 100 entries brings only a 16.30% reduction in TWI, while reducing the number of KV writes by 40.56%. For high-skew workloads (e.g., α=1.3), the reduction is much more significant, reaching 50.8%, with 79.75% of the KV writes removed.

Furthermore, as shown in Figure 2b, we observe that by simply increasing the memtable size, we can achieve a significant reduction in TWI. For example, increasing the memtable size from 1 MiB to 64 MiB with $\alpha = 1.2$ reduces the TWI from 6.85 TiB to 3.41 TiB. Larger memtables reduce the number of compactions and reduce the TWI more effectively. This approach also avoids the trade-off of sacrificing additional read performance. These results highlight two conclusions for data handling approaches:

- **Limited benefit from removing top frequency data under low-skew workloads**. For example, under a low-skew workload with α=1.1, removing 40.56% (i.e., top 100 entries) of the data only results in a 16.30% reduction in TWI.
- **Greater benefit from larger memtable under high-skew workload**. As shown in Figure 2b, increasing the size of the memtable provides significant benefits for high-skew workloads (e.g., α=1.3). This benefit is similar to removing the top 100 most frequent entries from each batch, which also reduces the total write I/Os by 50%. This indicates that while removing the top k entries is effective under high-skew workloads, increasing the memtable size yields even more significant benefits.

[1]In this paper, α refers to the parameter in the Zipfian formulation

3.2 Do top levels cause more data rewrites?

Data structure handling approaches assume that hot data is primarily stored in the top levels and this hot data is responsible for most write I/Os, suggesting that top levels be separated into faster storage devices. However, this assumption only works under certain conditions. Figure 2b shows the write I/Os (WI) at each level of LevelDB for different memtable sizes. As the memtable size increases from 1 MiB to 64 MiB, the WI decreases significantly, i.e., from 6.86 TiB to 3.41 TiB. However, the WI at each level shows different trends. For example, when the memtable is 1 MiB, the WI at L_1 and L_2 is significantly higher than that at other levels, accounting for 77.41% of the TWI. These results align with trends observed in the evaluation of splitDB [7].

However, the value drops to 29.64% when the memtable is 64 MiB. We observe that as the size of the memtable increases, compaction in the upper levels of the LSM tree is triggered more quickly since larger SSTables reach the level capacity limit more easily. The experiments in Appendix B show that LevelDB shows the same trend under different skewed workloads. In short, the WI in the top levels decreases as the size of the memtable increases. These observations lead to the following conclusions:

- **Top levels of LSM trees dominate WI only for small memtables.** As the memtable size increases, each level fills faster, causing data to be merged into lower levels more quickly. The assumption that hot data exist primarily at the top levels is only reasonable when the memtable in LSM-based KVs is very small (e.g. for the 1 MiB memtable in SplitDB [7]). In addition, a larger memtable will cause data to sink to lower levels while providing greater benefits in terms of WA reduction.

3.3 Who is the real villain of compaction?

Existing approaches primarily aim to separate hot data from cold data, driven by the observation that a small fraction of data is accessed frequently, especially at the top levels of the LSM tree, and the belief that **hot data is the primary cause of numerous compactions**. Therefore, these approaches focus on **separating hot data to mitigate write amplification.**

But is this true? Let us consider a two-level compaction scenario. Before compaction, F_1 in L_i contains {1, 3, 5, 7} and F_2 in L_{i+1} contains {1, 3, 5, 7, 18, 20}. A compaction requires two conditions. Firstly, L_i reaches its compaction. Secondly, there are two files that contain overlapping data (range), i.e., {1, 3, 5, 7} in both F_1 and F_2. If no such file exists, the LSM will trivially move SSTables instead of conducting a compaction. Since the first condition only matters with the amount of data, we use the hot data percentage in the overlapping data set to determine whether hot data triggers compaction. For example, if {3, 5, 7} are the top 1% most frequent keys ("hot"), we think hot data trigger 75% of compaction. If only {3} is the "hot" key, we think 25% of compaction is triggered by hot data. For multiple compactions, we use the average number of their percentages to represent the overall percentage of compaction triggered by hot data.

Figure 2c shows the compactions percentage triggered by hot data at different levels when hot data is defined as the top 1% frequently accessed data. As the amount of data written increases, the percentages of compaction triggered by hot data in L_1, L_2 and L_3 remains stable (around 0%, 30%, and 30%.) And the percentages in L_4 and L_5 decrease significantly (remains 12% and 3%).

Another concern is whether these compactions triggered by hot data significantly increase write amplification, even if their percentage is small. When each compaction happens, some data in L_i and L_{i+1} is read into memory to sort and merge and rewritten to disk (L_{i+1}) finally. This data can be divided into two types. The first type of data has multiple versions in L_i and L_{i+1}, and only the latest version will be written to L_{i+1}. we call this data **version-updating data** (VU-data). The second type of data has only one version in L_i and L_{i+1}, and this version will be written to L_{i+1}, we call this data **no-version-updating data** (NVU-data). The WI of each compaction is the WI sum for **VU-data** and **NVU-data**.

Usually, hot data acts as **VU-data** during compaction because its occurrence frequency is higher and, thus, easier to leave multiple versions in two adjacent LSM levels. Therefore, if hot data plays a critical role in write amplification, the **VU-data** will take up much of the total WA during compaction. Considering the definition of **VU-data**, read I/Os will be significantly bigger than write I/Os in the compaction since each version of the data requires a read, but only one write is conducted.

To ensure whether this is possible, we conducted an experiment to record the write I/Os and read I/Os during compaction at each level. Figure 2d shows that when α=1.1, after writing 1.3 TB of data, the amount of data reads and writes caused by compaction at each level, except for L_1, is nearly equal. This result shows that **VU-data** rarely occur in the compaction process. Moreover, hot data, which is mainly considered by existing works, is not the critical part of compaction or write amplification. **In contrast, cold data, which is usually NVU data, is the real villain of compaction.**

4 CURRENT MERGE POLICIES ANALYSIS

In this section, we present a model to analyze the impact of compaction on the write/read process in some widely used LSM-based KV stores.

4.1 Compaction Model

As described in Section 2, users append data to the memtable. When the memtable reaches its capacity limit, it flushes the data to L_0. The LSM tree triggers compaction at a level when data accumulates beyond the capacity of the level and some data overlap. In particular, the compaction process can be divided into three phases: 1) **compaction-read:** reading data from adjacent levels, 2) **sort-merge:** sorting them and merging different versions of the same key in memory, and 3) **compaction-write:** rewriting the data into new SSTables. Thus, we analyze the factors that influence each phase and how they affect write performance.

The **compaction-read** phase is unrelated to TWI, and we therefore do not consider it in our analysis. For the **sort-merge** phase, LSM trees require memory to store KV pairs and CPU cycles to determine their order after merging. More frequent compactions increase resource consumption and lengthen this phase, resulting in higher average write latency. To quantify the time, CPU, and memory costs, we can use the **frequency of compactions (CF)** as a measure.

Table 1: Technical Trade-offs in Current Compaction Policies

Merge Policy	Write Performance			Read Performance	
	Compaction Frequency		Extra Data Writes	Depth/Files	Dispersion
	f	c_0	NVU/VU Ratio		
Leveling [18, 21, 32]	High	Low	High	Low	Low
Tiering [30, 39, 41]	Low	High	Low	High	High
Partitioning Leveling [52]	High	Low	Very Low	Very Low	Very Low
L_1-Leveling [15]	Moderate	High	Moderate	Moderate	Moderate
L_l-Leveling [16]	Moderate	Low	Moderate	Moderate	Moderate
LuMDB	**High**	**High**	**Extremely Low**	**Extremely Low**	**Extremely Low**

CF depends on two main factors: the size ratio (f) between adjacent levels and c_0, the capacity of L_0. f determines how quickly each level reaches its capacity. A larger f implies a greater increase in size between two adjacent levels, meaning that data is less likely to sink. Thus, write amplification will be severe because the LSM tree will trigger more compactions between the two levels. Conversely, a smaller f allows data to sink faster. Thus, there is a clear positive correlation between the size of f and write amplification. However, a smaller f is not always preferable because it increases the depth of the LSM tree, which increases the average number of levels to check during lookups.

A larger size c_0 of L_0 induces the capacity of every level to be larger, which reduces the number of levels but increases the number of SSTables in each level. Therefore, the amount of data that needs to be sorted and merged in each compaction increases, which decreases write performance. However, a point query can be faster since a point query performs a binary search in each level, and the number of levels is reduced.

For the **compaction-write** phase, we define **extra data writes (EDWs)** as the additional amount of data written to disk during compaction. EDWs characterize the write I/Os overhead of compaction, and higher EDWs consume more I/O bandwidth and reduce the lifetime of SSDs [50]. We argue that EDWs are directly determined by the VU-data / NVU-data ratio, which reflects workload skewness. For low skewness, the NVU-data ratio increases, meaning less data can be merged, and more data must be rewritten to disk, resulting in higher EDWs.

Based on the above analysis, we can use the two key metrics, CF and EDWs, to measure the cost of compaction on write performance. It is important to note that CF and EDWs are not independent. The amount of EDWs depends primarily on the ratio of VU-data to NVU-data. Usually, a higher CF is acceptable when EDWs are small because CPU/memory operations are much faster than disk I/Os [4]. Therefore, write performance should be measured by **both CF and EDWs.**

Besides write performance, developers are also concerned about read performance [8]. Read performance depends on the tree's depth, each level's capacity(determines the number of SSTables that can be stored in one level), and the data's dispersion within the LSM tree. For a read operation, the LSM tree must perform a binary search of metadata of SSTable and actual data in each level until the key is found. Increasing either the tree depth or the capacity of each level increases the number of binary searches or the cost of each binary search separately, thus increasing read latency. In addition, data dispersion also affects range query efficiency. For example, if data in the same range (e.g., keys 80-90) is spread across different levels, the cost of range scans will be higher than if all this data is stored in one level. Partitioning the LSM to group data of the same range into a single file can also accelerate the read process [54].

4.2 Analysis of the write/read performance of existing merge policies

Table 1 summarizes the design choices of five widely used LSM tree-based KV stores and highlights how their choices affect their read and write performance. Leveling typically sets f to 10 and c_0 to 10 MiB. In contrast, Tiering sets f from 2 to 4 and c_0 to 64 MiB. In Tiering, when L_i is full, all runs from L_i are merged with all runs from L_{i+1}. This approach reduces the capacity of each level by lowering f and increasing c_0 while keeping the total amount of data constant. As a result, Tiering can make data sink to lower levels more quickly, thus avoiding triggering more compactions at each level, reducing the CF, and lowering EDWs, which ultimately improves write performance. However, when considering reading performance, Tiering usually sets lower f and causes the number of layers to increase, causing the data to be more spread out, which negatively affects read performance. In contrast, Leveling has fewer levels and better read performance.

The L_1-Leveling and L_l-Leveling approaches use Tiering at the first layer and last layer, respectively. As a whole, their f values lie between those of Leveling and Tiering. The L_1-Leveling approach has a larger c_0. For L_1-Leveling, there are fewer CF and smaller EDWs at the top levels, making it suitable for reducing write stalls. Conversely, L_l-Leveling has fewer CF at the last levels, and due to the larger capacity of the last level, the reduction in EDWs is more significant. However, this comes at the cost of potentially worse read performance because more data is stored in the last level.

Partitioning Leveling adds multiple partitions at each level, where the f value is typically the same as Leveling, but c_0 increases exponentially. The c_0 of each partition matches the c_0 of Leveling. With the same total amount of data, the amount of data written to each partition decreases, finally leading to a reduction in CF and total EDWs.

5 HOW CAN LSM TREES ACHIEVE LOWER WA?

In Section 3, we found that the current hot-cold data separation approaches used in LSM tree-based KV stores may not always work under skewed workloads. In this section, we use a theoretical analysis to find out whether a better LSM tree-based KV store design can work for all types of workloads.

An LSM tree-based KV design must answer two questions. First, what data structure should it use for data management? As described in Section 4, an LSM tree-based KV store can use leveling, tiering, or other less common data structures, or a hybrid approach to manage data and balance read/write performance based on user requirements. Second, what strategies do we use to utilize data structure to manage data. For example, TRIAD [2] uses hot-cold separation to classify data and store hot data in memory while storing cold data in leveling. They can also use other strategies that put all data into leveling or tiering. In this section, we focus on the effectiveness of hot-cold separation, whether hot-cold separation is useful, and what is the optimal threshold of hot data for various workloads.

We assume that all workloads fit the Zipfian distribution and that their skewness is controlled by a parameter α. According to existing studies [1, 20, 34], real-world workloads often exhibit skewed access patterns, which are commonly modeled using Zipfian distributions [3, 24]. These Zipfian distributions approximate the Zipf distribution as the amount of data D becomes large and thus $D \to \infty$ [49]. The probability mass function of the Zipf distribution is defined as $P(x) = \frac{1/x^{\alpha}}{\zeta(\alpha)}$, where $\zeta(\alpha) = \sum_{i=1}^{D} 1/i^{\alpha}$ is the truncated Riemann zeta function [47] and $P(x)$ is the probability of the x^{th} element appearing.

From the analysis in Section 4, we understand that analyzing write amplification essentially means analyzing EDWs. Therefore, in this section, we use EDWs as our objective function. Table 2 shows all terms used in our analysis.

Table 2: Terms used in formal analysis

Term	Definition
D	total number of key-value pairs
$D_{\text{hot}}, D_{\text{cold}}$	amount of KV pairs for hot-cold data
α	parameter of Zipf distribution
T	threshold of hot data
k_{hot}	number of distinct keys in D_{hot}
c_0	unit of capacity in L_0
f	size ratio between adjacent levels
r_c, r_h	size ratio of capacity of L_0 for different compaction policies
p	number of partitions
$EDWs$	extra data writes
ACM	average count of merge for each key-value pair

5.1 EDWs with hot-cold separation

In this section, we analyze EDWs for hot and cold data under a skewed workload after separation, respectively.

EDWs for hot data: We define T as the threshold of how often a key must occur in a data set of size D in the expected case to be a hot key. The number of distinct hot keys k_{hot} therefore satisfies $\frac{1/k_{\text{hot}}^{\alpha}}{\zeta(\alpha)} \times D \geq T$. We can deduce that

$$k_{\text{hot}} = \left\lfloor \left(\frac{D}{T \cdot \zeta(\alpha)} \right)^{1/\alpha} \right\rfloor,$$

and the amount of total hot data $D_{hot} = D \times \sum_{i=1}^{k_{\text{hot}}} \frac{1/i^{\alpha}}{\zeta(\alpha)}$. Note that in LumDB, hot-cold separation is performed during compaction of L_0, so that D in the resulting equations must be set to the maximum number of keys stored in L_0.

Without sorting and merging, hot keys could appear multiple times in an SSTable. However, only the latest version of each key's KV pair is rewritten during the rewrite phase. For each hot key, the number of times it will be rewritten is determined by whether or not it appears every time that L_0 is filled. Every L_0 will be filled with $k_h c_0$ data, so it triggers maximal $\frac{D_{\text{hot}}}{k_h c_0}$ compactions for hot data. And the i^{th} key appears $\left\lceil \frac{Di^{-\alpha}}{\zeta(\alpha)} \right\rceil$ times under the entire workload. If the i^{th} key appears when every compaction is triggered (but this is limited by the frequency of this key), then this is the corresponding upper bound. In contrast, the i^{th} key appears in at least $\left\lceil \frac{Di^{-\alpha}}{k_h c_0 \zeta(\alpha)} \right\rceil$ compactions (since it has to fill at least that many L_0), which will produce the fewest EDWs, i.e., lower bound. This result can apply to all LSM-based data structures. So, the $EDWs_{\text{hot}}$ is

$$EDWs_{\text{hot}} = \sum_{i=1}^{k_{\text{hot}}} r_i, \; r_i \in \left[\left\lceil \frac{Di^{-\alpha}}{k_h c_0 \zeta(\alpha)} \right\rceil, \min\left(\left\lceil \frac{Di^{-\alpha}}{\zeta(\alpha)} \right\rceil, \frac{D_{\text{hot}}}{k_h c_0} \right) \right],$$

Thus, the $EDWs_{\text{hot}}$ is determined by k_{hot} as

$$EDWs_{\text{hot}} \approx m \times k_{\text{hot}}, \quad m \in \left[\frac{1}{k_{\text{hot}}} \sum_{i=1}^{k_{\text{hot}}} \left\lceil \frac{Di^{-\alpha}}{k_h c_0 \zeta(\alpha)} \right\rceil, \frac{D_{\text{hot}}}{k_h c_0} \right). \quad (1)$$

where m is a parameter determined by the arrival order.

EDWs for cold data: According ot [16], every key will be rewritten $\frac{f}{2} \log_f \left(\frac{D_{\text{cold},i}}{c_0} \right)$ times with a uniform distribution in Leveling. Although the definition of a hot key may vary, we generally consider it to be hot if it occurs at least twice. Therefore, a thorough hot-cold separation involves separating keys that occur **twice or more** in the entire workload. If $T = 2$, we can easily extend this result with Leveling to Tiering and PL. The average count of merge (ACM) for each KV pair in i^{th} partition is approximately $\frac{f}{2} \log_f \left(\frac{D_{\text{cold},i}}{r_c c_0} \right)$ ($i = 1, 2, \cdots, p$), and $D_{\text{cold}} = D - D \times \sum_{i=1}^{k_{\text{hot}}} \frac{1/i^{\alpha}}{\zeta(\alpha)} = \sum_{i=1}^{p} D_{\text{cold},i}$. So, the estimation of EDWs for cold data is

$$\begin{aligned} EDWs_{\text{cold}} &= \sum_{i=1}^{p} \overline{ACM}_i \times D_{\text{cold},i} \approx \sum_{i=1}^{p} \frac{f D_{\text{cold},i}}{2} \log_f \left(\frac{D_{\text{cold},i}}{r_c c_0} \right) \\ &= \frac{f D_{\text{cold}}}{2} \log_f \left(\frac{D_{\text{cold}}}{p r_c c_0} \right). \end{aligned} \quad (2)$$

Total EDWs for skewed workloads. The total EDWs for a skewed workload with hot-cold separation is the sum of (1) and (2) as

$$EDWs_{\text{skew}} \approx \frac{f D_{\text{cold}}}{2} \log_f \left(\frac{D_{\text{cold}}}{p r_c c_0} \right) + m \times k_{\text{hot}}, \quad (3)$$

where $m \in \left[\frac{1}{k_{\text{hot}}}\sum_{i=1}^{k_{\text{hot}}}\left\lceil\frac{Di^{-\alpha}}{r_h c_0 \zeta(\alpha)}\right\rceil, \frac{D_{\text{hot}}}{r_h c_0}\right)$.

The expression (3) provides an overall estimate of EDWs under skewed workloads. Here, D_{cold} and k_{hot} are mainly controlled by α and T, while c_0 and f are fundamental parameters in LSM trees, representing the capacity of L_0 and the size ratio between adjacent levels. LSM tree-based KV stores make trade-offs between read, write, and space by adjusting p, r_c, and r_h. For instance, for leveling and tiering, they do not perform partitioning, so $p = 1$ and $r_{c/h}$ is set to 1 and 3 respectively as tiering keeps three runs in L_0. But in PL, it maintains p independent LSM trees with leveling, so $p \geq 1$ and $r_c = 1$. Clearly, when the same hot-cold separation is applied for a given workload (i.e., with a and T held constant), the larger p, r_c, and c_0 are, the smaller the EDWs for skewed workloads become. Meanwhile, as the growth factor f increases, the term $\log_f(\frac{D}{pr_c c_0})$ decreases logarithmically, while the multiplicative factor $\frac{f}{2}$ increases linearly. So $EDWs_{\text{skew}}$ increases with f.

5.2 EDWs without hot-cold data separation

Without separation, the main difference of the merge process is at $L_0 \rightarrow L_1$. Because the capacity of L_1 is f times L_0, a uniform workload with on expectation nearly no duplicate keys in L_0 and L_1 needs to trigger f compactions to fill up L_1 eventually. If the workload is skewed, after each compaction, the amount of data to be rewritten decreases due to duplicate keys (i.e., hot data), and we need to trigger more than f compactions to fill L_1 eventually.

Consequently, the estimation of EDWs from L_i to L_{i+1} for subsequent levels ($i \geq 1$) can be approximated as similar to that in cold data with hot-cold separation. We compare the EDWs in two cases: $T = 1$ and $T = 2$. When $T = 1$, we write KVs with the same configuration for cold data with hot-cold separation. We use $(\cdot)^n$ to represent $T = n$. k^1_{hot} is the number of distinct keys in the whole data set D while k^2_{hot} is that in D^2_{hot}. If we do hot-cold separation, all keys that exist in D^2_{cold} are the keys that occur once (i.e., $k^2_{\text{cold}} = D^2_{\text{cold}}$), so we have $k^1_{\text{hot}} = D^2_{\text{cold}} + k^2_{\text{hot}}$.

We can deduce that $\overline{ACM}$ at partition i for each level is

$$\overline{ACM}_{L_0 \rightarrow L_1, i} \approx \frac{f}{2},$$

$$\overline{ACM}_{L_1 \rightarrow L_{i+1}, i} \approx \frac{f}{2}\left[\log_f\left(\frac{D^2_{\text{cold},i}}{r_c c_0}\right) - 1\right] \approx \frac{f}{2}\left[\log_f\left(\frac{D^2_{\text{cold}}}{pr_c c_0}\right) - 1\right].$$

If we do not perform hot-cold data separation, duplicate keys will only be written once after the compaction rather than rewriting all KV pairs. The average count of merges is then reduced by about $\frac{a_i f}{2}\log_f(\frac{D^2_{\text{cold},i}}{r_c c_0})$ for each key in partition i, where $a_i \in [0, 1]$ represents the probability of having the same key in the two sets of data triggered the compaction. So EDWs specifically reduce the rewrites by $\sum_{i=1}^{p}\frac{a_i f}{2}k^2_{\text{hot},i}\log_f(\frac{D^2_{\text{cold},i}}{r_c c_0})$ approximately in total. So, the EDWs under a skewed workload without separation $EDWs^{T=1}_{\text{skew}}$ are approximately equal to

$$\frac{fD}{2} - \sum_{i=1}^{p}\frac{a_i f k^2_{\text{hot},i}\log_f\left(\frac{D^2_{\text{cold}}}{pr_c c_0}\right)}{2} + \frac{fk^1_{\text{hot}}}{2}\left(\log_f\left(\frac{D^2_{\text{cold}}}{pr_c c_0}\right) - 1\right) \quad (4)$$

5.3 An optimal LSM-based KV store

Do we need hot-cold separation for low-skew workloads?

From (3) and (4), we have

$$EDWs^{T=1}_{\text{skew}} = EDWs^{T=2}_{\text{skew}} + \frac{fD^2_{\text{hot}}}{2} - \left(m + \frac{\bar{a}f}{2}\log_f\left(\frac{D^2_{\text{cold}}}{pr_c c_0}\right)\right)k^2_{\text{hot}},$$

where exists $\bar{a}$ satisfying $\sum_{i=1}^{p}\frac{a_i f}{2}k^2_{\text{hot},i} = \frac{\bar{a}f}{2}k^2_{\text{hot}}$. In terms of minimizing EDWs, whether or not to apply hot-cold separation is determined by $\frac{fD_{\text{hot}}}{2} - \left(m + \frac{\bar{a}f}{2}\log_f\left(\frac{D^2_{\text{cold}}}{pr_c c_0}\right)\right)k^2_{\text{hot}}$, where $D^2_{\text{hot}}, D^2_{\text{hot}}$ and k^2_{hot} are all related to α. And in Appendix C, we calculated the first-order partial derivatives of D_{cold} and k_{hot} with respect to α, obtaining the results (C1) and (C4):

$$C1: \ \frac{\partial k^2_{\text{hot}}}{\partial \alpha} \leq 0 \quad C4: \ \frac{\partial D^2_{\text{cold}}}{\partial \alpha} \leq 0,$$

which means that both k^2_{hot} and D^2_{cold} increase whereas D^2_{hot} decreases as the α decreases. So it is possible that $\frac{fD^2_{\text{hot}}}{2} - (m + \frac{\bar{a}f}{2}\log_f\left(\frac{D^2_{\text{cold}}}{pr_c c_0}\right))k^2_{\text{hot}} \leq 0$ under a low-skew workload which a and m depend on the arrival order of hot and cold data. **In summary, ❶ the lower the skewness, the higher the probability that non-separation will generate lower WA; ❷ for the more skewed data, we should do hot-cold separation.**

Whether an optimal T exists in separation? We substitute $m = \frac{1}{k_{\text{hot}}}\sum_{i=1}^{k_{\text{hot}}}\lceil\frac{Di^{-\alpha}}{r_h c_0 \zeta(\alpha)}\rceil$ into expression (3) to get the lower bound of EDWs(i.e., $EDWs^{lb}_{\text{skew}}$). In Appendix D, we calculate the first-order partial derivative of $EDWs^{lb}_{\text{skew}}$ with respect to T and get the result (D2) as

$$\frac{\partial EDWs^{lb}_{\text{skew}}}{\partial T} = \frac{1}{\alpha}\left(\frac{D}{T\zeta(\alpha)}\right)^{1/\alpha} \times \left(\frac{f}{2}\log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right) + \frac{fD_{\text{cold}}}{2pr_c c_0 \ln f \log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right)} - \frac{1}{r_h c_0}\right)$$

As c_0 is a large value as well, so the derivative of $EDWs^{lb}_{\text{skew}}$ is positive. Under the same workload with separation (i.e., $T \geq 2$), the larger T means larger $EDWs_{\text{skew}}$. **In conclusion, the most thorough hot-cold separation (i.e., $T = 2$) is the optimal choice.**

6 LUMDB

Building on the analysis in Section 5, we introduce LuMDB, a novel KV store that computes the workload skewness and identifies hot data during compaction. For high-skew workloads, LuMDB employs *two-phase Partitioning Leveling* for managing cold data and uses Tiering for hot data management. For low-skew workloads, LuMDB stores all data with *two-phase Partitioning Leveling*.

6.1 An Overview of LuMDB

The basic principle of LuMDB is to separate hot data from cold data under high-skew workloads while avoiding separation under low-skew workloads. In particular, such a separation raises the following questions: ❶ How does LuMDB compute skewness? ❷

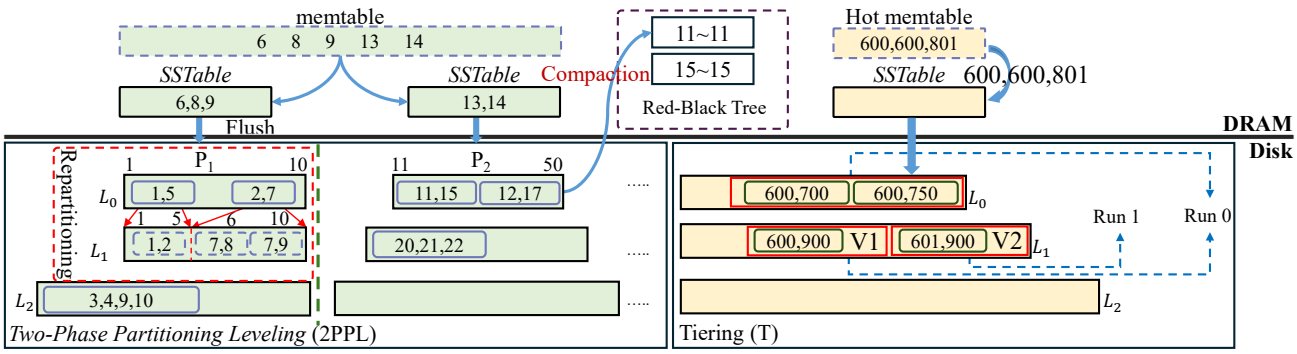


Figure 3: Overall architecture of LuMDB.

How does LuMDB identify hot data? ❸ How does LuMDB manage and store cold and hot data?

For ❶ and ❷, we design a novel compaction-combined mechanism to compute skewness and identify hot data during each compaction with L_0. During compaction, LuMDB counts the frequency of each key and computes the **variance** of the frequency as the measure of skewness. The skewness then determines whether separation should be applied in LuMDB. We identify hot data by checking whether the occurrence count of a key exceeds T. Compared to identifying hot data before writing to disk or memtables [2], this mechanism offers two advantages: 1) it does not introduce any additional I/Os in compaction-read and compaction-write, and the computation in CPU and memory costs little time and 2) identification accuracy improves significantly because the amount data becomes larger. We discuss the details of the mechanism in Section 6.2.

The question ❸ is to determine the optimal approach for managing hot and cold data after separation. In general, LuMDB manages cold data in a novel structure, *two-phase partitioning leveling (2PPL)*, and manages hot data in Tiering. After separation, the NVU-data/VU-data ratio in cold data is very high, meaning that the amount of data read and rewritten during compaction is almost equal. To address this, 2PPL uses partitioning to reduce CF by increasing the number of partitions. Partitioning can also allow developers to set appropriate partition capacity of L_0 and L_1 in 2PPL, which makes it able to deal with different memtable sizes.

Typically, Tiering can reduce the CF by storing the same data with different versions in different Runs [26], thus suitable for managing hot data. However, a concern of Tiering is whether it can bring negative impacts on read performance. Since all the data stored in Tiering are hot data, it rarely descends to the lower levels. This is because hot data has a high percentage of VU-data during compaction, and much data is discarded after each compaction, so very little data is written to the next level, making it difficult to fill a level. For example, when $\alpha = 1.3$, data in Tiering only sink to the first run of L_1 after writing 1.3 TiB data. Therefore, we argue that using Tiering to store hot data will not introduce significant additional overhead on read performance.

6.2 Hot Data Identification

We now introduce our hot data identification process. First, LuMDB assigns $T = 2$ as discussed in Section 5, which means keys with a frequency exceeding the number of T inside L_0 are classified as hot keys.

As shown in Figure 3, LuMDB maintains two separate memtables: `hot_mem_` for hot data and `mem_` for cold data. Initially, the `hot_mem_` is not used, and all incoming KV pair writes are directed to the `mem_`. When the `mem_` becomes full, data in `mem_` are flushed to the appropriate partition of L_0 in 2PPL. We discuss the detailed design in Section 6.3.

During the compaction in L_0, LuMDB counts the frequency of each key. Then calculates the **variance** of the key frequency (`var_`) as a measure of skewness. If `var_` is less than or equal to a predefined `sep_skewness`, LuMDB does not conduct the hot-cold separation, treats all data as cold, and writes them to 2PPL.

If `var_` $\geq$ `sep_skewness`, LuMDB records all identified hot keys in memory. In particular, it use hot range, i.e., a range like [Key_min, Key_max] to represents all the keys which $\geq$ Key_min and $\leq$ Key_max are hot keys. And LuMDB uses a red-black tree [23] to organize all the hot ranges identified during runtime. In some scenarios, almost every hot range will have only one hot key (e.g., in Figure 3, LuMDB identifies two hot ranges: [11-11] and [15-15]), and take more time to determine whether a key is hot. However, in general, our method of using hot range can be efficient in hot key management. Subsequently, all KV writes to LuMDB are checked against these ranges to determine whether data are written to `hot_mem_` or `mem_`.

For read process, LuMDB first checks whether the incoming key is hot key, i.e., hitting any hot range recorded. If it is a hot key, LuMDB reads its value according to its Tiering structure. If it is not a hot key. LuMDB calculate which partition in 2PPL storing this key and checks each level of this partition to get its newest value.

6.3 Hybrid Partitioning Leveling and Tiering

Next, we present the detailed design of LuMDB's *two-phase partitioning leveling* and tiering.

6.3.1 Two-Phase Partitioning Leveling. After separation, each key in the cold data section occurs infrequently. This leads to high EDWs per compaction regardless of the structure used because the

amount of data read and rewritten during compaction is nearly equal. Therefore, minimizing CF becomes a primary goal for cold data management. In addition, since much of the queried data may be cold in read-intensive scenarios, improving read performance for cold data is equally important.

Pure leveling or tiering is not suitable for cold data. The EDWs will be very high for leveling because it can trigger too many compactions. In addition, tiering will significantly reduce read performance because it involves many runs. Therefore, we propose *two-phase partitioning leveling* (2PPL), which distributes data across multiple partitions in LuMDB. Compared to pure leveling with a single partition, the design can distribute data to more partitions, thus reducing the CF in each partition and ultimately reducing the EDWs. Compared to tiering, partitioning leveling can provide better read performance because the data in each partition becomes much smaller.

The next question is how to partition the distribution. As shown in Figure 4, a simple approach is to use a fixed partition, like proposed in WaLSM [12]. However, this approach has two issues: 1) Predefined partitions do not guarantee identical amounts of data per partition. For example, as shown in Figure 4, P_2 contains only one KV pair after compaction, causing small I/Os which increase the write latency and write stalls. This contradicts the original goal of LSM trees, i.e., avoiding the small I/Os. 2) Fixed partitions are tailored to a specific distribution and cannot work when data distribution changes over time.

To address the above two problems, we propose a dynamic partitioning strategy used in 2PPL. When the first minor compaction is triggered, LuMDB initiates the partition creation process. Specifically, data in this minor compaction are not fully written to a single SSTable. LuMDB guarantees that the number of keys (or the amount of data) in one partition does not exceed a pre-defined limit (`min_file_size`), e.g., 1 MiB. As illustrated in Figure 4, with `min_file_size` set to four, the first four KV pairs are written to the first partition in L_0, creating partition P_1 (1-10). The remaining KV pairs are written to the second partition, creating P_2 (600-1002). Finally, we store the data in a partitioning leveling (PL) through our proposed dynamic partitioning strategy. We can note that the final partitioning created by PL is also related to the order of data arrivals.

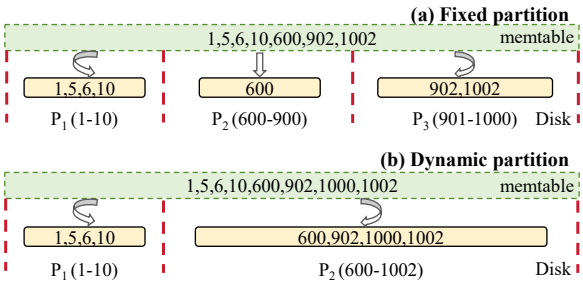


Figure 4: (a) Fixed partitioning pre-defines partitions. It may create small SSTables for some partitions, leading to small I/Os. (b) For dynamic partitioning, data are sequentially written to SSTables from the memtable. A new partition is created when the SSTable size reaches the `min_file_size`.

However, the effectiveness of PL is affected by two factors: skewness and `mem_` size. The `mem_` size is limited by server resources, and if the `mem_` is relatively small, PL may create very few partitions. In addition, high workload skewness results in a smaller amount of cold data, and in such a situation, the feature of PL that multiple versions of data can appear simultaneously will cause more compaction and lower read performance.

With small `mem_` sizes, LuMDB will not create more partitions. For example, if the `mem_` size of LuMDB is 2 MiB, the `min_file_size` is 1 MiB, LuMDB will only create two partitions in L_0. Thus, after the partition initialization process, 2PPL introduces **repartitioning** process during the first compaction from L_0 to L_1. In particular, the process involve **repartitioning** each partition exists in L_0 during the first compaction from L_0 to L_1 and allows overlapping ranges of different SSTables at L_1. Due to allowing overlapping SSTables in both L_0 and L_1, 2PPL can create more partitions compared to PL, which increases the number of files in L_1 that must be queried, trading off read performance for reducing EDWs. For example, in Figure 3, LuMDB repartitions the P_1 (1-10) into (1-5) and (6-10). Additionally, under low-skew workloads, LuMDB's 2PPL does not apply hot-cold separation. Similarly, it does not introduce repartitioning in L_1. This is because if repartitioning were applied, hot data would be confined to a small partition, leading to a significant increase in the NVU-data ratio within the partitions containing cold data, which would cause a substantial rise in LuMDB's EDWs.

This trade-off can be parameterized by changing the capacity of L_0 and L_1. Under higher skewness, where the amount of cold data is less, 2PPL adaptively sets a small capacity of L_0 (even zero), facilitating faster data sinking into the lower levels. Additionally, for a larger `mem_size`, many partitions can already be created at L_0. To adapt to varying conditions, LuMDB dynamically changes the capacity of L_0 and L_1 based on `mem_` size and skewness, progressively reducing these thresholds as `mem_size` and skewness increase. In LuMDB, the capacity of L_0 and L_1 are pre-set for different sizes of `mem_`. For example, when $\alpha = 1.2$ and 1 MiB `mem_`, the capacity of L_0 and L_1 is 6 and 3.

6.3.2 Tiering. After separation, each key in the hot data is stored in LuMDB's Tiering. Compared to traditional Tiering, Tiering in LuMDB records the age of data. When a compaction finishes, LuMDB creates the runs with older data firstly thus these data can be quicker found when a read operation asks them. The idea behind it is that older data is less frequently updated compared to newer data. For example, as shown in the Tiering part of Figure 3, when a compaction from L_0 to L_1 happens, LuMDB will select the older run (i.e., Run 0) of L_1 as a victim to merge with Run 0.

7 EVALUATION

7.1 Experimental Setup

Benchmark. We used the YCSB benchmark [14] to generate all read and write workloads for the evaluation, as well as for the experiments in Section 3. The workload skewness was set by adjusting the Zipfian parameter α. For all evaluations, we set the key size to 16 bytes and the value size to 128 bytes.

Comparison Baselines. We designed two sets of KV store baselines. First, to show the effectiveness of each technique in LuMDB, we designed a set of KV stores that implement a subset of LuMDB's techniques. We designed two baselines (L, PL) without hot-cold data separation and three baselines (LaT, PLaT, 2PPLaT) with hot-cold data separation. In particular, L manages all data in Leveling, while PL manages data in Partitioning Leveling. LaT, PLat and 2PPLaT store hot data in Tiering. However, LaT stores cold data in Leveling, PLat in Partitioning Leveling, and 2PPLaT in *Two-Phase Partitioning Leveling*. For LuMDB, when separation is applied, it is 2PPLaT; when separation is not applied, it is PL. Second, to evaluate whether LuMDB can outperform SOTA KV stores, we compared LuMDB with LevelDB 1.23 [18], RocksDB 9.4.0 [32], and PebblesDB [39]. For all these KVs, we modified their `write_buffer_size` to control the memtable size and left all other configurations at their default values. We measured **total write I/O (TWI)** as a metric to quantify WA.

Settings. All experiments were conducted on an Intel server with two 28-core Intel(R) Xeon(R) Platinum 8180 CPUs @ 2.50GHz and 512 GB of RAM. The server, running Ubuntu 22.04 LTS with Linux kernel GNU/Linux 6.5.0-27-generic x86_64, uses a CORSAIR MP400 SSD (7 TB) as the experimental storage platform. The SSD achieves maximum sequential read and write speeds of 3400 MiB/s and 3000 MiB/s, respectively.

7.2 Write Performance

In this section, we evaluate the effectiveness of LuMDB in reducing write amplification and providing better write performance. We first evaluate the effectiveness of WA reduction by comparing **TWI** between LuMDB and all baselines for workloads with different skewness, memtable size, and hot data threshold. We then compare LuMDB's **tail latency** and **average latency** to SOTA KV stores.

Figure 5 shows the TWI after writing 100M KV pairs for different baselines under low-skew (α = 1.1) and high-skew (α = 1.3) workloads. When α = 1.1, LuMDB does not apply hot-cold separation. When α = 1.1 and the memtable size is 1 MiB, LuMDB can reduce the TWI by 5.8× compared to the worst case L. When α = 1.3, it keeps basically the same trend as when α = 1.1. The difference is that the TWI for 2PPLaT, PLaT, and PL is only slightly higher than that of LuMDB because there is very little cold data in the entire workload at this point, and these baselines use the same methods to manage hot data as LuMDB.

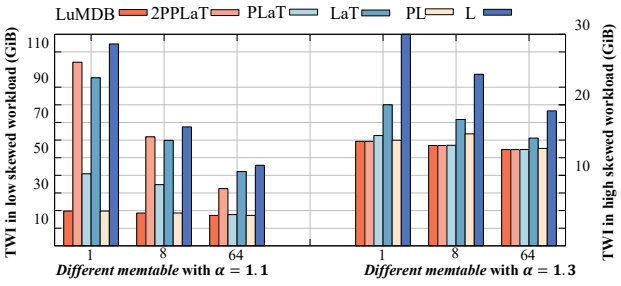


Figure 5: TWI for different memtable sizes when α = 1.1 and α = 1.3

Besides the effectiveness of LuMDB, we also observe some important findings about the performance of these baselines. When α = 1.1 and the memtable size is 1 MiB, the TWI of 2PPLaT is only lower than the TWI of L. This is because a low-skew workload contains a large amount of cold data. In such a situation, the partitions created by 2PPLaT in L_0 are very few, and they cannot significantly reduce the number of compactions. As the memtable size increases, the number of partitions that 2PPLaT can create gradually increases, and the number of compactions is reduced, so that the TWI reduction becomes noticeable. This is consistent with our conclusion in Section 5 that it is better not to perform **hot-cold separation** for a low-skew workload. Furthermore, when α = 1.3, we find that the TWI of almost all partitioning methods is smaller than that of LaT. This proves that partitioning based on hot-cold separation can effectively reduce the TWI.

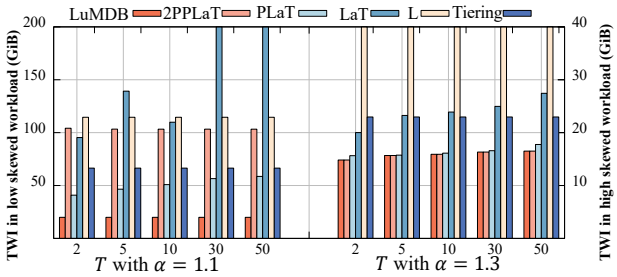
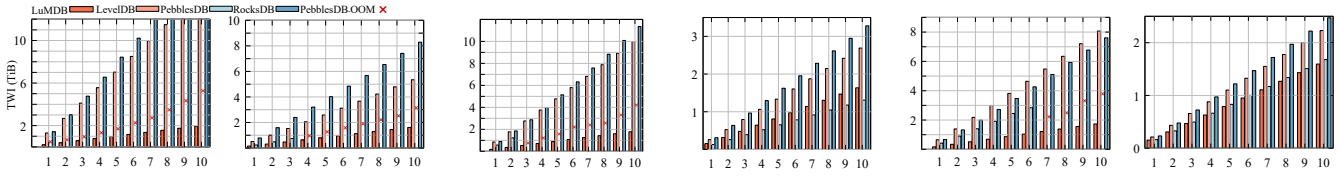


Figure 6: TWI depending on T when α=1.1 and α=1.3.

Figure 6 presents the TWI when increasing T (i.e., hot data threshold) for low and high skew workloads. Regardless of whether α = 1.1 or α = 1.3, LuMDB can always provide the lowest TWI compared to all other baselines, which is 23.1× (α = 1.1) and 3.5× (α = 1.3) better than the worst TWI of LaT and L. Besides, we observe that regardless of α, as T increases, the TWI of Tiering decreases, while the TWI of cold data in all approaches increases. Therefore, the overall TWI of hot-cold separation methods depends on whether the increase in the TWI of cold data is greater than or less than the decrease in Tiering's TWI, where they store their hot data. For example, the TWI for both PLaT and LaT increases as T grows from 2 to 50 (e.g., for LaT, TWI increases by a factor of 3.8 when α = 1.1). Overall, this is consistent with the second conclusion we reached in Section 5, i.e., T = 2 is the optimal separation threshold.

For 2PPLaT, the TWI increases by 11% when α = 1.3, whereas at α = 1.1, it decreases by 0.8%. This is because, at α = 1.1, the increase in TWI for 2PPLaT is less than the decrease in TWI of Tiering. This effect arises from the more fine-grained partitions that exist in 2PPLaT, which enables more compactions driven by version updates in some partitions. This results in fewer EDWs for 2PPLaT compared to LaT. However, we can see that the TWI of 2PPLaT is still higher than that of LuMDB and PLaT when α = 1.1. This is because, when α = 1.1, there is a large amount of cold data, and the introduction of more partitions prevents the hot data from effectively increasing the VU-data ratio. As a result, 2PPLaT exhibits naturally larger EDWs compared to LuMDB. This finding aligns with our analysis in Section 6.3.1.



(a) 1 MiB memtable, α=1.1(L) and α=1.3(R) **(b) 8 MiB memtable, α=1.1(L) and α=1.3(R)** **(c) 64 MiB memtable, α=1.1(L) and α=1.3(R)**

Figure 7: Total write I/Os while writing 1.3 TB data with different memtable size

We then evaluate the write performance of LuMDB against several widely used LSM tree-based KV stores. Figure 7 shows the TWI as the skewness and memtable size increase. We see LuMDB always maintain the lowest TWI. In particular, when the memtable size is 1 MiB and α = 1.1, LuMDB's TWI remains around 1.9 TiB, while LevelDB and RocksDB have TWI values that are 7.4 and 8.1 × higher than LuMDB's, respectively. Because PebblesDB has to maintain a large number of Bloom filters in memory, it exhausted memory resources after writing 750 million KV pairs. As α increases to 1.3 with 1MiB memtable, LuMDB's TWI continues to decrease to 1.6 TiB, while LevelDB and RocksDB have TWI values that are 3.33 and 4.63 × that of LuMDB. In this setting, PebblesDB completes 3.96 billion writes with a TWI of 0.9 TiB.

When the memtable size increases to 64 MiB (α = 1.1), the TWI of all KVs decreases significantly whatever skewness. For example, LuMDB reaches a TWI of 1.7 TiB, while LevelDB and RocksDB decrease to 8.0 and 7.6, respectively. PebblesDB only finishes 4.75 billion KV writes. Additionally, we find that increasing the memtable size from 8 MiB to 64 MiB only reduces the TWI of RocksDB and LevelDB by 18.9% and 33.3%, respectively, indicating that simply increasing the memtable size does not fundamentally reduce WA. For α = 1.3, the trend is similar to that of 1.1; there is the smaller drop in the TWI of these KVs since their TWI at 1MiB is not high.

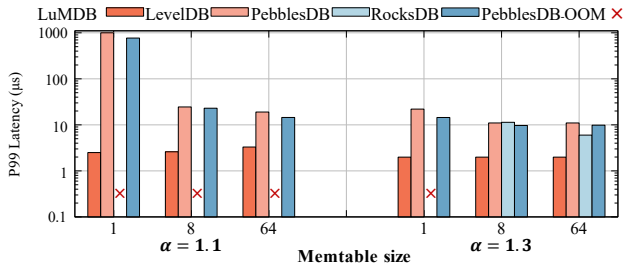


Figure 8: Tail latency

Figure 8 and Figure 9 show the 99th percentile (P99) write latency and average latency of these KVs during the above writing process. We can see that whatever memtable size changes, LuMDB maintains the lowest and most stable tail latency. This is because LuMDB will create multiple partitions, so the data written to each partition is much smaller, reducing the CF of each partition and thus keeping latency low for most write operations. For example, when α = 1.1 and memtable is 64 MiB, LuMDB is 5.7 times lower than LevelDB for P99 latency.

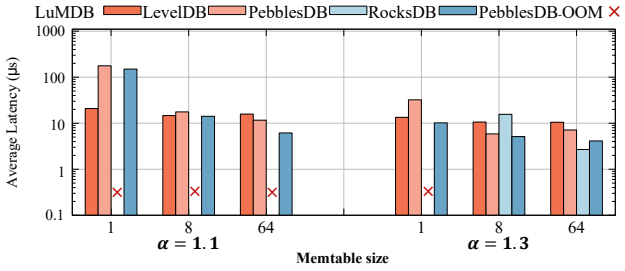


Figure 9: Average latency

However, for average latency, LuMDB's is higher than the other KVs as memtable increases. This is because LuMDB's effective hot-cold separation results in storing a large amount of cold data by 2PPL. While increasing the size of the memtable creates more partitions, this only decreases the CF of each partition, but not the EDWs, since all these cold data are almost NVU-data. Thus, the average write latency of LuMDB decreases very little as memtable increases. In contrast, other KV stores, which do not perform hot-cold separation, store more hot data in the larger memtable, resulting in more VU-data and less data being rewritten after compaction. This reduces their average write latency more significantly.

Note that for all kv stores, compaction is so expensive, especially in single threaded implementations like LuMDB, that the slowest 1% of requests not included in the P99 percentile can make the average latency slower than that P99 percentile. We will show that the small overhead in some settings of LuMDB in terms of average write latency pays off in read performance and in the YCSB benchmarks.

7.3 Read Performance

In this section, to evaluate read performance, we executed 0.5M range and point queries after prewriting 1 billion KV pairs under three different workloads (α ranging from 1.2 to 1.4). The distribution of all queried keys is uniform.

Figure 10 shows that LuMDB consistently outperforms other databases for both point and range queries. For point queries, at α = 1.2, LuMDB achieves a throughput of 7.1k, which is 1.68 times higher than LevelDB, the worst performer. At α = 1.3, LuMDB's throughput (46k) is particularly outstanding, reaching 4.12 times that of RocksDB. At α = 1.4, LuMDB's throughput (75k) is slightly lower than LevelDB's, but the difference is minimal (0.02 ×). This shows the advantage of partitioning in LuMDB. In the range query scenario, as skewness increases, the throughput of all databases

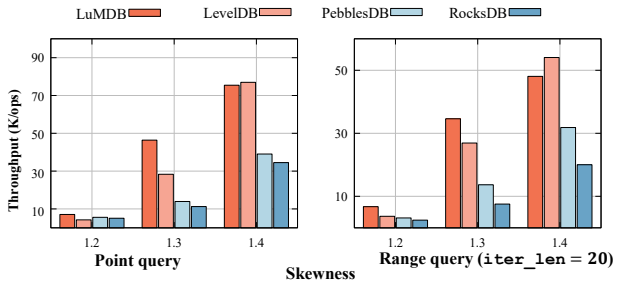


Figure 10: Performance of point queries and range queries

behaves similarly to point queries, with LuMDB maintaining its overall performance advantage. Specifically, at $\alpha = 1.3$, LuMDB's throughput is 4.59 × higher than the throughput of RocksDB. During our monitoring of the disk activity, we observed that RocksDB consistently performs small-sized read I/Os during the range query and point query workloads, which we suspect is the reason for its lower throughput.

7.4 Performance under YCSB Benchmark

Finally, we run the Yahoo! Cloud Serving Benchmark (YCSB) [14] to evaluate the overall performance of LuMDB. All KV stores were preloaded with 1 billion (134.3 GB) KV pairs, following a Zipfian distribution with $\alpha = 1.2$. The results of the YCSB benchmarks are shown in Figure 11.

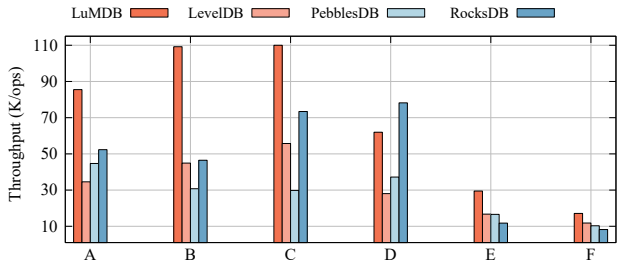


Figure 11: Throughput of YCSB benchmarks

In the YCSB benchmark, all lookups and updates follow a Zipfian distribution, while the inserts in YCSB D and YCSB E are single inserts. LuMDB provides the best performance for all workloads except YCSB D. In YCSB A, LuMDB is 2.47× faster than LevelDB. As the update-to-lookup ratio increases (from YCSB C to YCSB A), LuMDB's throughput gradually increases from 1.5× to 1.6× that of RocksDB, proving that LuMDB effectively avoids write amplification and provides better performance for write-intensive workloads. In read-intensive workloads such as YCSB C, LuMDB still provides the best throughput compared to SOTA LSM tree-based KV stores (e.g., 3.7 faster than PebblesDB). In YCSB D, LuMDB lags behind RocksDB by 20.8%. In this workload, the insertion of new data causes LuMDB to trigger the merging of multiple partitions, causing the throughput to drop.

The evaluation results show that LuMDB can also perform well on workloads that consist mainly of complicated operations. For example, in the scan-intensive YCSB E and read-modify-write YCSB F workloads, LuMDB still provides the highest throughput (2.9k in YCSB E and 1.7k in YCSB F) among all baselines, which is 2.5× that of RocksDB in YCSB E and 1.5× that of LevelDB in YCSB F.

8 RELATED WORK

Data structure handling approaches LogStore [31] divides the LSM tree into three levels. L_0 and L_1 are stored on faster SSDs, while L_2 is stored on slower HDDs. It maintains histograms per level in DRAM to evaluate the hotness of each SSTable, merging the coldest SSTable in DRAM with the L_2 SSTable during each compaction. MirrorKV [48] categorizes LSM trees into key LSM trees and value LSM trees, placing the top two levels in a block store and the remaining levels in an object store. In addition, MirrorKV retains the hottest blocks (e.g. 10%) when compacting from L_1 to L_2. SplitDB [7] stores the top five levels (L_0-L_4) in persistent memory, which occupies 1/10 of the space of the entire LSM tree, to reduce the number of compactions that happen on SSDs. NoveLSM [22] adds an NVM memtable to serve writes to reduce write stalls. It also uses I/O parallelism to search multiple levels at once to reduce query latency. Similarly, MatrixKV [55] creates a larger PM component to reduce write amplification and write stalls.

Data handling approaches. TRIAD [2] separates the top 10 most frequently updated KV pairs in each batch and stores these hot KV pairs in memory while flushing cold KV pairs to disk. It writes WALs for crash recovery, which impacts range query performance due to unsorted data in the WAL. Similarly, WaLSM [12] identifies hot data and stores it in PM. It also introduces partitioning and virtual partitioning techniques to manage tiering policies in SSDs and uses Q-learning to dynamically adjust the index to workload shifts. Mutant [56] tracks the frequency of SSTables in a cloud environment and periodically adjusts the location of SSTables so that the most frequently accessed SSTables are stored in faster storage. PrismDB [38] uses Optane SSDs [51] as the faster tier and stores records in unordered slabs. It uses a B+ tree in DRAM to index these records. SA-LSM [57] uses statistical learning (also known as survival analysis) to predict cold data and demote cold records from faster to slower storage. However, it does not support promoting hot records back to faster storage.

9 CONCLUSION

This paper presents LuMDB, a novel LSM-based KV store that can achieve ultra-low write amplification and better performance for both read/write operations through meaningful hot-cold data separation. Compared to existing works that are limited to high-skew workloads and special configurations, LuMDB leverages the analysis results based on a general compaction model and formal derivation, which allows it to adopt an optimized hot-cold separation strategy for workloads with different skew. To better manage data without hot-cold separation and cold data with hot-cold separation, LuMDB also introduces the novel data structure 2PPL. Our evaluation shows that LuMDB can reduce write amplification by up to 8.1 ×, which is close to the theoretical limit.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. {TRIAD}: Creating synergies between memory, disk and log in log structured {Key-Value} stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 363–375.

[3] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768.

[4] William Lloyd Bircher and Lizy K John. 2011. Complete system power estimation using processor performance events. *IEEE Trans. Comput.* 61, 4 (2011), 563–577.

[5] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[6] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. 2019. *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media.

[7] Miao Cai, Xuzhen Jiang, Junru Shen, and Baoliu Ye. 2024. SplitDB: Closing the Performance Gap for LSM-Tree-Based Key-Value Stores. *IEEE Trans. Comput.* 73, 1 (2024), 206–220.

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[9] Apache Cassandra. 2014. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra* 13 (2014).

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

[11] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. 2015. A big data modeling methodology for Apache Cassandra. In *2015 IEEE International Congress on Big Data*. IEEE, 238–245.

[12] Lixiang Chen, Ruihao Chen, Chengcheng Yang, Yuxing Han, Rong Zhang, Xuan Zhou, Peiquan Jin, and Weining Qian. 2023. Workload-Aware Log-Structured Merge Key-Value Store for NVM-SSD Hybrid Storage. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2207–2219.

[13] Danny Chiao and David Simmons. 2023. https://cloud.google.com/blog/products/databases/how-feast-feature-store-streamlines-ml-development

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[15] 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.

[16] 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*. 505–520.

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

[18] Google. 2021. https://www.google.com/chrome/index.html

[19] Kecheng Huang, Zhiping Jia, Zhaoyan Shen, Zili Shao, and Feng Chen. 2021. Less is More: De-amplifying I/Os for Key-value Stores with a Log-assisted LSM-tree. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 612–623.

[20] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert Van Renesse. 2014. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. 1–7.

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

[22] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning {LSMs} for Nonvolatile Memory with {NoveLSM}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 993–1005.

[23] Charles Eric Leiserson, Ronald L Rivest, Thomas H Cormen, and Clifford Stein. 1994. *Introduction to algorithms*. Vol. 3. MIT press Cambridge, MA, USA.

[24] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with {In-Network} coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406.

[25] Chen Luo and Michael J Carey. 2019. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proceedings of the VLDB Endowment* 12, 5 (2019).

[26] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[27] Nishchit Malasana. 2023. Why do cryptocurrencies use LevelDB? https://0xnishchit.hashnode.dev/why-do-cryptocurrencies-use-leveldb

[28] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. Citeseer, 21–33.

[29] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[30] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*. 477–489.

[31] Prashanth Menon, Thamir M Qadah, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2020. Logstore: A workload-aware, adaptable key-value store on hybrid storage systems. *IEEE Transactions on Knowledge and Data Engineering* 34, 8 (2020), 3867–3882.

[32] Meta. 2024. https://rocksdb.org/

[33] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[34] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.

[35] Gregorius Ongo and Gede Putra Kusuma. 2018. Hybrid database system of MySQL and MongoDB in web application development. In *2018 international conference on information management and technology (ICIMTech)*. IEEE, 256–260.

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

[37] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[38] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J Freedman. 2023. Efficient Compactions between Storage Tiers with PrismDB. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 179–193.

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

[40] 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)*.

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

[42] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, designing, and optimizing LSM-based data stores. In *Proceedings of the 2022 International Conference on Management of Data*. 2489–2497.

[43] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A dictionary for LSM compactions. In *Proceedings of the 2022 International Conference on Management of Data*. 2429–2432.

[44] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM design space and its read optimizations. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3578–3584.

[45] 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).

[46] Zhaoyan Shen, Yuanjing Shi, Zili Shao, and Yong Guan. 2018. An efficient LSM-tree-based SQLite-like database engine for mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 9 (2018), 1635–1647.

[47] EC Titchmarsh. 1986. *The theory of the Riemann zeta-function*. The Clarendon Press Oxford University Press.

[48] Zhiqi Wang and Zili Shao. 2023. MirrorKV: An Efficient Key-Value Store on Hybrid Cloud Storage with Balanced Performance of Compaction and Querying. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[49] Wikipedia. 2024. Zipf's law. https://en.wikipedia.org/wiki/Zipf%27s_law

[50] Guanying Wu and Xubin He. 2012. Delta-FTL: Improving SSD lifetime via exploiting content locality. In *Proceedings of the 7th ACM european conference on Computer Systems*. 253–266.

[51] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an unwritten contract of intel optane {SSD}. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*.

[52] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. {LSM-trie}: An {LSM-tree-based}{Ultra-Large}{Key-Value} Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[53] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182.

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

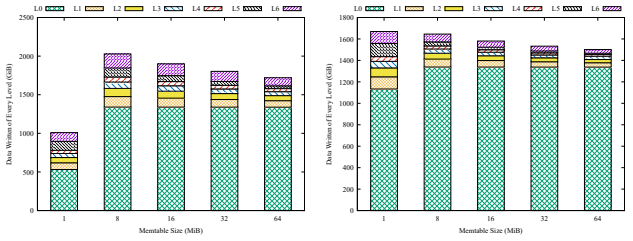
[55] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. {MatrixKV}: Reducing Write Stalls and Write Amplification in {LSM-tree} Based {KV} Stores with Matrix Container in {NVM}. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 17–31.

[56] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 162–173.

[57] 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.

[58] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. {REMIX}: Efficient Range Query for {LSM-trees}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.

A WRITE-I/OS OF EACH LEVEL IN TIERING



(a) $\alpha = 1.3$ **(b)** $\alpha = 1.4$

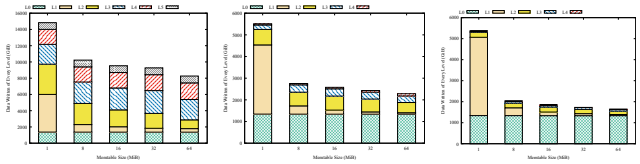
Figure 12: WI in each level with various memtable varying α

Figure 12 describes the WI of each level of PebblesDB under different memtable sizes (1 MiB to 64 MiB) and different skewnesses α = 1.3 and 1.4). Regardless of whether $\alpha = 1.3$ or $\alpha = 1.4$, PebblesDB maintains 7 levels due to its larger L_0 and smaller f compared to LevelDB. PebblesDB does not exhibit significantly higher WI at the top levels compared to other levels, regardless of the memtable size. As shown in Figure 12a, except L_0, at $\alpha = 1.3$ with a 1 MiB memtable, the WI of the L_5 and L_6 is notably higher than that of the other levels, comprising 22.7% of the total WI, while the highest WI among the remaining levels only accounts for 8% of the total TWI. As the memtable size increases to 64 MiB, the WI at L_6 decreases to 6.1% of the total WI, and the WI progressively increases from L_6 to L_1, with L_1 contributing 4.9% of the total WI.

While the separation of top levels may be justified in high skewness, the TWI of PebblesDB is already relatively low. For instance, as shown in Figure 12b, when the memtable size is 1 MiB, PebblesDB's WAF reaches its peak but remains at only 1.47. Furthermore, when the memtable size increases to 64 MiB, the WAF is further reduced to 1.11. Thus, making more efforts to separate the top levels is not warranted.

B WRITE-I/OS OF EACH LEVEL IN LEVELING

Figure 13 illustrates the WI at each level of the LevelDB under different memtable sizes (1 MiB to 64 MiB) and varying skewness



(a) $\alpha = 1.1$ **(b)** $\alpha = 1.3$ **(c)** $\alpha = 1.4$

Figure 13: WI in each level with various memtable varying α

(α = 1.1, 1.3, and 1.4). We observe a trend of decreasing WI at the upper levels as the memtable size increases. For instance, when $\alpha = 1.1$, as the memtable grows from 1 MiB to 64 MiB, the WI of the top two levels decreases from 40.5% to 21.7% of the overall WI. With higher skewness, a smaller portion of the data is accessed more frequently, reducing the probability of generating more levels since most data updates occur in the upper layers. For example, at $\alpha = 1.3$ and $\alpha = 1.4$, as the memtable increases from 1 MiB to 64 MiB, the WI of the top two levels in LevelDB decreases slightly from 82.3% and 94.4% to 61.6% and 84.5%, respectively. When skewness exceeds a certain threshold (e.g., $\alpha \geq 1.3$), it may be beneficial to separate the top levels to reduce overall WA. However, this separation becomes less impactful than the overall WA. For example, at $\alpha = 1.3$ with a 64 MiB memtable, LevelDB's WAF is only 1.7056. When α increases to 1.4, the WA further reduces to 1.2, which is almost negligible. When high skewness, separation may offer limited benefits in overall performance and may not be cost-effective because faster storage is more expensive.

C RELATION BETWEEN $EDWs_{\text{skew}}$ AND α

When other settings remain unchanged, different α in skewed data distribution will result in different $EDWs_{\text{skew}}$. Since the sum and product of two monotonic functions remain monotonic even when the function value is greater than zero, we only need to prove the monotonicity of k_{hot} and D_{cold} separately.

First, we calculate the first derivative of k_{hot} with respect to α.

$$\frac{\partial k_{\text{hot}}}{\partial \alpha} = -k_{\text{hot}} \left[\frac{1}{\alpha^2} \ln\left(\frac{D}{T\zeta(\alpha)}\right) + \frac{1}{\alpha\zeta(\alpha)} \sum_{i=1}^{D} (\frac{\alpha}{i}) i^{-\alpha} \right].$$

Generally, $\frac{D}{T\zeta(\alpha)}$ is greater than 1, making $\ln \frac{D}{T\zeta(\alpha)}$ positive. Additionally, α is a real number greater than 1. Therefore, we deduce that,

$$\frac{1}{\alpha^2} \ln\left(\frac{D}{T\zeta(\alpha)}\right) + \frac{1}{\alpha\zeta(\alpha)} \sum_{i=1}^{D} (\frac{\alpha}{i}) i^{-\alpha} \geq 0.$$

While k_{hot} is a positive integer, we have

$$\frac{\partial k_{\text{hot}}}{\partial \alpha} \leq 0. \tag{C1}$$

Then, we calculate the first derivative of D_{cold} concerning α.

$$\frac{\partial D_{\text{cold}}}{d\alpha} = -D \frac{\partial \sum_{i=1}^{k_{\text{hot}}} \frac{1/i^{\alpha}}{\zeta(\alpha)}}{\partial \alpha}$$

Since we have demonstrated that k_{hot} is a decreasing function from (C1), we can simplify the calculation with a fixed upper limit for $\sum_{i=1}^{k_{\text{hot}}} \frac{1/i^{\alpha}}{\zeta(\alpha)}$. In other words, we only need to prove the

monotonicity of $\sum_{i=1}^{k} \frac{1/i^{\alpha}}{\zeta(\alpha)}$ with respect to α. The first deviation of $\sum_{i=1}^{k} \frac{1/i^{\alpha}}{\zeta(\alpha)}$ is

$$\frac{\partial \sum_{i=1}^{k} \frac{1/i^{\alpha}}{\zeta(\alpha)}}{\partial \alpha} = \frac{\sum_{i=1}^{k}(-\alpha)i^{-\alpha-1}\zeta(\alpha) - \sum_{i=1}^{k} i^{-\alpha} \sum_{i=1}^{D}(-\alpha)i^{-\alpha-1}}{\zeta(\alpha)^2}$$
$$= \frac{-1}{\zeta(\alpha)^2}\left[\sum_{i=1}^{k}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=1}^{D} i^{-\alpha} - \sum_{i=1}^{k} i^{-\alpha}\sum_{i=1}^{D}(\frac{\alpha}{i})i^{-\alpha}\right]. \tag{C2}$$

In the above formulation, $\frac{1}{\zeta(\alpha)^2}$ is larger than zero, so we can only analyze whether $\sum_{i=1}^{k}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=1}^{D} i^{-\alpha} - \sum_{i=1}^{k} i^{-\alpha}\sum_{i=1}^{D}(\frac{\alpha}{i})i^{-\alpha}$ is less than zero.

$$\begin{aligned}
&\sum_{i=1}^{k}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=1}^{D} i^{-\alpha} - \sum_{i=1}^{D}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=1}^{k} i^{-\alpha} \\
&= [\sum_{i=1}^{k}(\frac{\alpha}{i})i^{-\alpha}(\sum_{i=1}^{k_{\text{hot}}} i^{-\alpha} + \sum_{i=k+1}^{D} i^{-\alpha}) \\
&- (\sum_{i=1}^{k}(\frac{\alpha}{i})i^{-\alpha} + \sum_{i=k+1}^{D}(\frac{\alpha}{i})i^{-\alpha})\sum_{i=1}^{k} i^{-\alpha} \\
&= \sum_{i=1}^{k}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=k+1}^{D} i^{-\alpha} - \sum_{i=k+1}^{D}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=1}^{k_{\text{hot}}} i^{-\alpha} \\
&\le \sum_{i=b+1}^{k} i^{-\alpha}\sum_{i=k+1}^{D} i^{-\alpha} - \sum_{i=b+1}^{k} i^{-\alpha}\sum_{i=k+1}^{D}(\frac{\alpha}{i})i^{-\alpha} \\
&+ \sum_{i=1}^{b}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=k+1}^{D} i^{-\alpha} \\
&= \sum_{i=b+1}^{k}(1-\frac{\alpha}{i})i^{-\alpha}\sum_{i=k+1}^{D} i^{-\alpha} + \sum_{i=1}^{b}(\frac{\alpha}{i})i^{-\alpha}\sum_{i=k+1}^{D} i^{-\alpha} \\
&= (\sum_{i=b+1}^{k}(1-\frac{\alpha}{i})i^{-\alpha} + \sum_{i=1}^{b}(\frac{\alpha}{i})i^{-\alpha})\sum_{i=k+1}^{D} i^{-\alpha},
\end{aligned}$$

where $b = \lfloor \alpha \rfloor$. As $b < \lfloor k/2 \rfloor$, we have

$$(\sum_{i=b+1}^{k_{\text{hot}}}(1-\frac{\alpha}{i})i^{-\alpha} + \sum_{i=1}^{b}(\frac{\alpha}{i})i^{-\alpha})\sum_{i=k_{\text{hot}}+1}^{D} i^{-\alpha} \le 0. \tag{C3}$$

From (C2) and (C3), we can deduce that

$$\frac{\partial \sum_{i=1}^{k} \frac{1/i^{\alpha}}{\zeta(\alpha)}}{\partial \alpha} \ge 0,$$

then

$$\frac{\partial D_{\text{cold}}}{\partial \alpha} \le 0. \tag{C4}$$

In summary, we have demonstrated here that D_{cold} and k_{hot} are decreasing functions of α. That is, with an identical amount of data D as α increases (indicating a tendency toward high skewness in workload), the amount of cold data and the count of distinct keys in hot data decrease accordingly.

D RELATION BETWEEN $EDWs_{\text{skew}}$ AND T

We calculate the first-order partial derivative of $EDWs^{lb}_{\text{skew}}$ with respect to T.

$$\frac{\partial EDWs^{lb}_{\text{skew}}}{\partial T} = \frac{\partial \frac{fD_{\text{cold}}}{2}\log_f\left(\frac{D_{\text{cold}}}{pk_c c_0}\right)}{\partial T} + \frac{\partial \sum_{i=1}^{k_{\text{hot}}} \frac{Di^{-\alpha}}{r_h c_0 \zeta(\alpha)}}{\partial T}.$$

About D_{cold}, only the upper limit of the finite series k_{hot} is related to T. First, we calculate $\frac{\partial D_{\text{cold}}}{\partial T}$.

We have the following conclusion for differentiation for a finite series. We define a finite series $S(x)$:

$$S(x) = \sum_{i=1}^{f(x)} g(i).$$

We have

$$\frac{d}{dx}S(x) = g(f(x)) \cdot f'(x).$$

So,

$$\frac{\partial D_{\text{cold}}}{\partial T} = -D\frac{\partial \sum_{i=1}^{k_{\text{hot}}} \frac{1/i^{\alpha}}{\zeta(\alpha)}}{\partial T} = -D\frac{(k_{\text{hot}})^{-\alpha}}{\zeta(\alpha)}\frac{\partial k_{\text{hot}}}{\partial T}$$
$$= -T\frac{\partial k_{\text{hot}}}{\partial T}. \tag{D1}$$

where

$$\frac{\partial k_{\text{hot}}}{\partial T} = -\frac{1}{\alpha T}\left(\frac{D}{T\zeta(\alpha)}\right)^{1/\alpha}.$$

Thus,

$$\begin{aligned}
\frac{\partial EDWs^{lb}_{\text{skew}}}{\partial T} &= \frac{f\log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right)}{2}\frac{\partial D_{\text{cold}}}{\partial T} \\
&+ \frac{fD_{\text{cold}}}{2}\frac{1}{\ln f \log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right) pr_c c_0}\frac{\partial D_{\text{cold}}}{\partial T} \\
&+ \frac{1}{r_h c_0}\frac{\partial D_{\text{cold}}}{\partial T} \\
&= \frac{\partial D_{\text{cold}}}{\partial T}\left(\frac{f}{2}\log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right) + \frac{fD_{\text{cold}}}{2pr_c c_0 \ln f \log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right)} - \frac{1}{r_h c_0}\right) \\
&= \frac{1}{\alpha}\left(\frac{D}{T\zeta(\alpha)}\right)^{1/\alpha} \times \\
&\left(\frac{f}{2}\log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right) + \frac{fD_{\text{cold}}}{2pr_c c_0 \ln f \log_f\left(\frac{D_{\text{cold}}}{pr_c c_0}\right)} - \frac{1}{r_h c_0}\right).
\end{aligned} \tag{D2}$$