How to Grow an LSM-tree?

Towards Bridging the Gap Between Theory and Practice

DINGHENG MO, Nanyang Technological University, Singapore
SIQIANG LUO, Nanyang Technological University, Singapore
STRATOS IDREOS, Harvard University, USA

LSM-tree based key-value stores are widely adopted as the data storage backend in modern big data applications. The LSM-tree *grows* with data ingestion, by either adding levels with fixed level capacities (dubbed as vertical scheme) or increasing level capacities with fixed number of levels (dubbed as horizontal scheme). The vertical scheme leads the trend in recent system designs in RocksDB, LevelDB, and WiredTiger, whereas the horizontal scheme shows a decline in being adopted in the industry. The growth scheme profoundly impacts the LSM system performance in various aspects such as read, write and space costs. This paper attempts to give a new insight into a fundamental design question – *how to grow an LSM-tree* to attain more desirable performance?

Our analysis highlights the limitations of the vertical scheme in achieving an optimal read-write trade-off and the horizontal scheme in managing space cost effectively. Building on the analysis, we present a novel approach, Vertiorizon, which combines the strengths of both the vertical and horizontal schemes to achieve a superior balance between lookup, update, and space costs. Its adaptive design makes it highly compatible with a wide spectrum of workloads. Compared to the vertical scheme, Vertiorizon significantly improves the read-write performance trade-off. In contrast to the horizontal scheme, Vertiorizon greatly extends the trade-off range by a non-trivial generalization of Bentley and Saxe's theory [7], while substantially reducing space costs. When integrated with RocksDB, Vertiorizon demonstrates better write performance than the vertical scheme, while incurring about six times less additional space cost compared to the horizontal scheme.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; • **Information systems → Data structures**.

Additional Key Words and Phrases: LSM-tree, key-value stores, index structure.

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

1 Introduction

LSM-tree is a key-value data structure that organizes data into multiple increasing levels. It appends incoming data entries into the buffer, whose filled-up triggers a merge-sort of the buffer into the next level. The process can cascade down whenever the affected level reaches its capacity to ensure desired read performance and minimize the space amplification. Due to its desired performance, the LSM-tree is extensively employed as the backbone of various data storage systems. This includes its utilization in key-value stores (KV-stores) like RocksDB [20], Cassandra [34], and ScyllaDB [53], NewSQL stores such as CockroachDB [33], as well as time-series databases like InfluxDB [30].

Authors' Contact Information: Dingheng Mo, Nanyang Technological University, Singapore, dingheng001@ntu.edu.sg; Siqiang Luo, Nanyang Technological University, Singapore, siqiang.luo@ntu.edu.sg; Stratos Idreos, Harvard University, USA, stratos@seas.harvard.edu.

ACM 2836-6573/2025/6-ART173
https://doi.org/10.1145/3725310

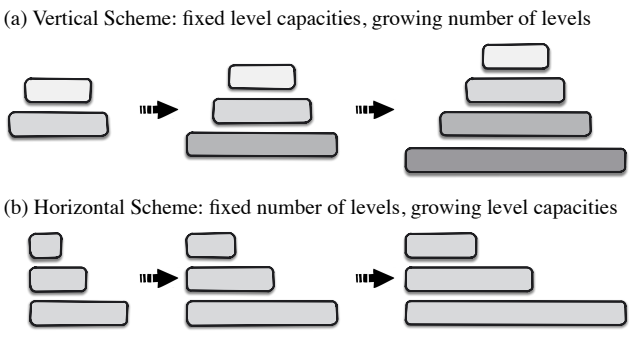


Fig. 1. Different growth schemes of LSM-trees.

Vertical and Horizontal Growth Schemes. The growth scheme governs how LSM-trees expand and maintain their hierarchical structure amidst the continuous influx of new data entries. Specifically, the LSM-tree relies on compactions to transfer data from one level to the next, ensuring that each level's size is significantly larger than the preceding one. The growth scheme fundamentally controls the timing of these compactions at each level. Currently, the growth schemes adopted in mainstream LSM-based key-value stores fall into two major categories, the vertical scheme and the horizontal scheme. With the vertical scheme, data ingestion into the LSM-tree would expand it vertically by creating additional levels. Specifically, each LSM-tree level has a fixed capacity that increases exponentially with the number of levels, as Figure 1 (a) depicts. Compactions are initiated from one level to the subsequent level as soon as the level is full. Under the horizontal scheme, the total number of levels on the disk is confined to a fixed value. As illustrated in Figure 1 (b), each level grows horizontally with a heuristic (see Section 3) that generally maintains the relative capacity between levels, implying that their capacities would gradually expand as data flows in. Beyond the last level, each level initiates a compaction into the subsequent level after a certain number of compactions from its preceding level. The required number of compactions gradually increases as the levels grow.

Growth Scheme	Vertical	Horizontal
Leveling Merge	LevelDB, RocksDB, WiredTiger, BadgerDB ...	BigTable, HBase, AsterixDB
Tiering Merge	Cassandra, ScyllaDB	Not exist

Table 1. The landscape of the two growth schemes.

The Decline of Horizontal Scheme V.S. Its Desired Theoretical Property. The horizontal scheme had been adopted by many pioneering key-value storage systems, including Bigtable [10], HBase [23], and AsterixDB [4]. However, a noteworthy trend in recent ten years is that most of the new generation key-value storage systems employ the vertical scheme, such as LevelDB [25], RocksDB [20], WiredTiger [13], Cassandra [34], ScyllaDB [53], and BadgerDB [12], as Table 1 summarizes. Furthermore, a majority of the prominent NoSQL databases in recent times, such as CockroachDB [33], TiDB [47], Dgraph [19], and PolarDB [26], have opted for the vertical scheme in their storage backend. This is probably attributed to that the vertical scheme offers the advantage of being easier to manage, and it allows for a finer compaction granularity to mitigate space amplification and write stalls. Contrary to this trend, interestingly, an early theory back in 1980's [7] revealed that the horizontal-growth-style data structure could offer an optimal trade-off between read and write costs even before the formal invention of the LSM-tree, and this theory is

Designs	Vertical-Leveling	Vertical-Tiering	Horizontal-Leveling	Horizontal-Tiering **(ours)**	VERTIORIZON **(ours)**
Update-Heavy	★	★★★★	★★★	★★★★★	★★★★★
Lookup-heavy	★★★★★	★★	★★★★★	★★★	★★★★★
Space	★★★★	★★★★	★★★	★★★	★★★★

Table 2. Comparing existing growth schemes and ours.

further explicitly discussed in the context of LSM-trees by Mathieu et al. [45]. This controversy strongly indicates that the horizontal growth scheme might be undervalued in practical system design choices.

The Problem: The Imperfect LSM-tree Growth Scheme Hurts System Performance. How to grow (a.k.a., expand) the LSM-trees with fast data writes is among the most critical design choices, profoundly impacting the performance of LSM-tree based systems. For instance, in a cloud environment using LSM-based storage, the chosen growth scheme impacts the system latency, write amplification, and space cost during data ingestion. A suboptimal growth scheme leads to higher write amplification, requiring more disk scans, which shortens hardware lifespan and raises cloud costs. Additionally, ineffective growth schemes result in increased latency and space usage, driving up costs for users storing and querying data. This is because the growth scheme dictates the timing of compactions, and inappropriate compaction timing can lead to higher write and read amplification (as we detail later in Sections 3 and 4), thereby increasing latency. Furthermore, different schemes support varying compaction granularities, while full compaction in particular may cause high space amplification, resulting in increased storage cost. Existing growth schemes thus face limitations that hinder system scalability in various ways. The vertical scheme, used by RocksDB and LevelDB, is not optimal in balancing read-write trade-offs because it performs compaction at each level with a fixed frequency, leading to reduced system throughput as data scales. The horizontal scheme, employed by BigTable, has only been implemented with the leveling merge policy, making it difficult to handle update-intensive workloads—a common workload type in modern systems. It also requires full compaction during merges, resulting in high space overhead. This motivates us to revisit the strengths and weaknesses of the two growth schemes, and ask for the possibility of better designs.

Key Intuition 1. There is an increasing need for a systematic discussion on the foundational topic of LSM-tree growth schemes, due to the growing scale and complexity of LSM-based systems. One important recent design is to adopt the tiering merge policy (i.e., each level is compacted as a new sorted run in the subsequent level), rather than the default leveling merge policy (i.e., each level is compacted to the existing sorted run in the subsequent level), to improve write performance at the expense of read performance. As demonstrated by Dayan and Idreos [15], the vertical growth scheme can be easily combined with the tiering policy, increasing its flexibility in balancing read-write trade-offs. However, the integration of the horizontal scheme with the tiering merge policy remains unexplored, and no such design has been adopted previously (Table 1). As a result, the horizontal scheme offers limited flexibility in balancing read and write performance, which is particularly undesirable for workloads with frequent updates. Our first intuition is, *what if we can design a horizontal-tiering scheme that couples horizontal scheme with tiering merge policy?* Such a design, if feasible, could significantly elevate the status of the horizontal scheme and revive interest in this largely overlooked approach. However, creating an effective horizontal-tiering scheme is notoriously difficult and has not been accomplished since the invention of LSM-trees.

Key Intuition 2. Since the strengths and weaknesses of the two major growth schemes complement each other, our second intuition is *whether it is possible to combine the two to achieve the best of both worlds*. There are countless potential ways to hybridize these schemes, but ensuring the combination fully leverages the strengths of both is challenging. Moreover, the optimal growth scheme often depends on the specific workload. For example, a read-intensive workload might favor a leveling-based scheme over a tiering-based one. This implies the necessity of a growth scheme that can be adaptive to different workloads, adding to the sophistication in hybridizing the schemes.

Our Proposal. We present two novel techniques, namely Horizontal-Tiering and Vertiorizon , to address the challenges. Horizontal-tiering enhances the horizontal scheme with the integration of the tiering policy (response to Intuition 1). This technique will revoke the recognition of the largely overlooked horizontal scheme, and thus highly impact the landscape of LSM-tree growth schemes. On top of this enhanced scheme, we further propose a new design Vertiorizon, which is a novel self-designing LSM-tree growth scheme that harnesses the strengths of both vertical and horizontal schemes, obtaining both theoretical merits and desired practical performance (response to Intuition 2). As shown in Figure 10 (a), Horizontal-Tiering effectively expands the optimal read-write trade-off to a much wider spectrum. As shown in Table 2, compared to existing schemes like Vertical-Leveling, Vertical-Tiering, and Horizontal-Leveling, Vertiorizon offers very robust performance across all dimensions. We summarize our contributions as follows.

- We carefully review the pros and cons of the two major growth schemes for LSM-trees, and motivates the fundamental problem in designing better growth schemes.
- We present the horizontal-tiering scheme, which is the first horizontal scheme that can be effectively applied under the tiering merge policy since the invention of LSM-trees. We also theoretically show that our designed horizontal-tiering scheme achieves the minimum accumulative cost during compaction. This is a non-trivial generalization of Bentley and Saxe's theory [7].
- We present Vertiorizon, an innovative growth scheme that embodies the strengths of both the vertical and horizontal schemes, which also offers a nice self-designing feature to work in an adaptive manner with different workloads.
- We design a novel algorithm to adapt our design to work under more skewed workloads. This is accompanied with a deep analysis to justify the rationale behind such designs.
- We present typically how Vertiorizon can be embedded in a mature LSM-system by introducing a novel dynamic Bloom filter layout tailored to our design.
- We carry out extensive experimental assessments of Vertiorizon in real-world system environments. We demonstrate that Vertiorizon achieves up to 3.2 times throughput than the vertical scheme (employed by RocksDB), and incurs about 6 times less additional space amplification compared to the horizontal scheme.

2 Background

LSM-tree. The LSM-tree [46] is a persistent multi-level index structure designed to store key-value entries. An LSM-tree achieves efficient write performance by transforming random writes to the disk into sequential writes. Specifically, when a key-value entry is inserted, the LSM-tree first temporarily stores it in a main memory buffer. Once the buffer reaches its capacity, all entries within it are merged and sorted by keys into the first level on the disk. To gain better write performance, LSM-tree organizes data across various sorted levels on the disk, arranged in ascending order of size. When certain conditions are met, the LSM-tree would merge one level into the subsequent level through compaction, continuing in this manner up to the largest level. In this way, the write amplification associated with merging data from the buffer or a specific level to the subsequent one is duly mitigated. Write amplification quantifies the average count of physical rewrites for each

entry, due to the recurrent data rewrites to the storage driven by the compaction process inherent to the LSM-tree. Read amplification measures the average number of disk pages read for a single lookup, as a search might scan through several sorted runs within an LSM-tree to find a specific key. Space amplification evaluates the additional space cost needed on average for each unique entry.

Full and Partial Compactions. Usually, a compaction merges and sorts the full data from one level with the next, known as full compaction. However, in LSM-trees, a deeper level could hold vast amounts of data, and full compaction in such cases can cause major performance issues. This not only slows down the system but also leads to substantial space amplification, as original data cannot be disposed of until compaction is fully done. The partial compaction mitigates these problems, allowing merging only a small fraction of the data with the subsequent level in each compaction.

Tiering and Leveling Merge Policy. Under the leveling merge policy, a compaction from Level i to Level $i+1$ merge-sorts all data from Level i with all the existing data in Level $i+1$. Additionally, each buffer flush merge-sorts its data with the existing data in the first level. In contrast, the tiering merge policy handles compactions or buffer flushes to a level by adding incoming data as a new separate sorted run in that level, without merging it with the existing data there. In this way, each entry is involved in only one compaction at each level. Generally, the leveling merge policy is more read-friendly, while tiering performs better in write-intensive scenarios.

Point Lookup and Range Lookup. Point-lookup locates the entry for a given key. During the lookup, the sorted run is probed from the newest to the oldest, until the entry containing the queried key is found. Modern LSM-based systems typically store the starting key of each disk block as an index in memory. Since each LSM-tree run is sorted, this setup allows for a direct pinpoint of the sole block likely to contain a particular key, ensuring a maximum of one I/O for each lookup at every sorted run. This process is further accelerated by applying Bloom filter at each sorted run. The filters will be loaded into the main memory when the system starts. To check whether a key exists in a run, the Bloom filter associated with the run is first probed. An I/O is incurred only when the filter returns true, otherwise the access to the sorted run can be exempted. In contrast, range-lookup aims to extract the entries for a given key range. The process is similar to point-lookup, except that (1) Bloom filter is not applied; and (2) every sorted run is accessed.

Read-write Trade-off. The "read-write trade-off" in an LSM-tree is the balance between lookup and update costs. By changing its structure, the LSM-tree can optimize for either lookups or updates, but improving one comes at the expense of the other. Different growth schemes produce different trade-off curves, showing how much one cost must be sacrificed to improve the other. A design with a better read-write trade-off requires less compromise in one cost to achieve a certain performance level in the other.

3 Analysis of Growth Schemes

Vertical Scheme. In the vertical scheme, the capacity of each level is fixed, with a size ratio T between two adjacent levels. A compaction happens when the data in the level reaches the predefined capacity, subsequently triggering a merge of the data in that level to the next level. A new level will be created to accommodate the compaction of the current largest level. As a result, the LSM-tree grows vertically (i.e., additional levels are gradually created) as the overall data volume expands. Given the buffer size B, the capacity of Level i is set as $B \cdot T^i$, a compaction from Level i to Level $i+1$ is triggered when the data volume at Level i surpasses its capacity.

Horizontal Scheme. In the horizontal scheme, the number of levels is strictly fixed. Consequently, with the influx of more data, the LSM-tree broadens horizontally, meaning that each level's capacity would increase over time. As shown in Algorithm 1, let ℓ be the maximum number of levels, and C_i

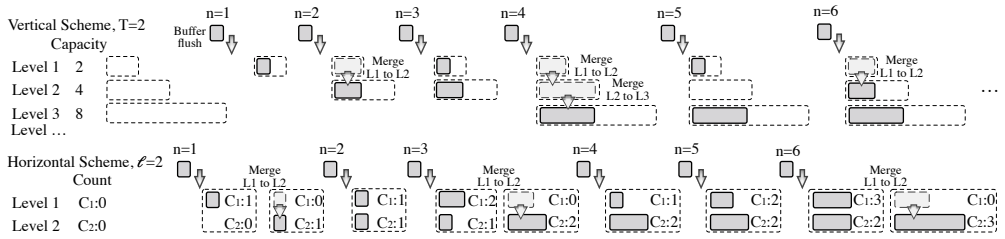


Fig. 2. Running examples of the vertical and horizontal growth schemes.

be the times of compactions conducted from Level $i-1$ to Level i since the previous compaction from Level i to Level $i+1$. Especially, for the first level, C_1 is the number of flushes from memory buffer to Level 1 since the previous compaction from Level 1 to Level 2. For the largest level, C_ℓ is the number of flushes from Level $\ell-1$ to Level ℓ. A full compaction from Level i to Level $i+1$ is triggered whenever $C_i > C_{i+1}$. Intuitively, the horizontal scheme dynamically controls the data volume ratio between two adjacent levels by their relative numbers of compactions.

Running Example (Figure 2). Let n be the number of times a full memory buffer is flushed to disk. We set size ratio $T = 2$ for the vertical scheme, and let the maximum number of levels ℓ be 2 in the horizontal scheme. Initially, each level of the vertical scheme has a predetermined capacity, with the first level having a capacity that can hold 2 buffers, and the second level can hold 4 buffers. In contrast, the horizontal scheme's levels do not have a predetermined capacity. Under the vertical scheme, when the number of buffer flushes $n = 1$, the LSM-tree consists solely of Level 1, accommodating data equivalent to the buffer size. When $n = 2$, the data volume of Level 1 reaches its capacity after two buffer flushes, and a compaction from Level 1 is triggered, with Level 2 created as the target level. The subsequent compaction from Level 1 to Level 2 occurs at $n = 4$, at which point Level 2 also reaches its capacity after two compactions from Level 1. Consequently, a new Level 3 is then established, enforcing a compaction from Level 2 to Level 3. In this way, LSM-tree under the vertical scheme continuously expands vertically. On the other hand, the horizontal scheme assigns each level a compaction counter, starting at 0. Each buffer flush increases the counter C_1 at Level 1 by one. After the first buffer flush ($n = 1$), we have $C_1 = 1$ and $C_2 = 0$. Given these values, the compaction condition for Level 1 under the horizontal scheme, $C_1 > C_2$, is met, thereby triggering a compaction from Level 1 to Level 2. After this compaction, C_1 is reset to 0, while C_2 increases by one. When $n = 3$, C_1 equals 2, which surpasses $C_2 = 1$, thus

Algorithm 1: Horizontal Scheme

```
   Input: Maximum number of levels ℓ
1  for i = 1 to ℓ do
2      C_i ← 0
3  for each flush from buffer do
4      C_1 ← C_1 + 1;
5      for i = 1 to ℓ − 1 do
6          if C_i > C_{i+1} then
7              Trigger a full compaction from Level i to Level i + 1;
8              C_{i+1} ← C_{i+1} + 1, C_i ← 0;
```

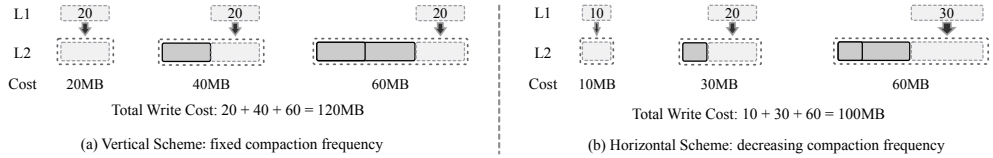


Fig. 3. Illustrating the cost of different schemes.

triggering another compaction from Level 1 to Level 2. Subsequently, C_1 is reset to 0 again, while C_2 rises to 2. A similar process recurs at $n = 6$.

Horizontal Scheme Achieves a Better Read-Write trade-off (a.k.a., Bentley and Saxe's theory [7]). The compaction from Level i to Level $i + 1$ involves merge-sorting all data from Level i with existing data in Level $i + 1$, keeping each level as a single sorted run. This process becomes increasingly costly as the size of Level $i + 1$ grows, especially if equal amounts of data are merged each time, as is typical in the vertical scheme. For example, as shown in Figure 3 (a), the vertical scheme moves a total of 60MB data from Level 1 to Level 2 in three equal compactions of 20MB, which results in compaction costs of 20MB, 40MB, and 60MB, leading to a total of 120MB of writes. Intuitively, a more efficient approach is to perform compaction more frequently when Level 2 is smaller and less frequently as it grows. By merging at 10MB, 20MB, and then 30MB, the compaction costs become 10MB, 30MB, and 60MB, reducing the total write cost to 100MB, as illustrated in Figure 3 (b). This is precisely the strategy employed by the horizontal scheme.

In fact, Bentley and Saxe [7] (later refined by Mathieu et al. [45] for LSM-tree context) have demonstrated the optimality of the horizontal scheme, that the horizontal scheme incurs the minimal write cost for a given number of levels. Since the read cost of an LSM-tree under the leveling merge policy is proportional to the number of levels, the horizontal scheme consistently achieves the optimal trade-off between the read and write costs.

Limitations of the Horizontal Scheme. The previous analyses conclude that the traditional vertical scheme is suboptimal, while the horizontal scheme can provide better overall read-write trade-off. Nevertheless, as we mentioned, most recent key-value stores [12, 13, 20, 25, 33, 34, 53] tend to adopt the vertical scheme. This somewhat contradicts the theoretical results presented above. We conjecture that this discrepancy is probably due to two practical limitations of the horizontal scheme.

Limited capability in handling update-heavy workloads. Update he-avy workloads are common in practice. The vertical scheme effectively accommodates such workloads by switching to tiering merge policy. In contrast, the horizontal scheme has only been designed for the leveling merge policy. Moreover, as we will show shortly, directly applying the tiering merge policy may easily lead to sub-optimal performance. To address this limitation, we propose the *horizontal-tiering* scheme, which is the first horizontal scheme algorithm that can be effectively applied under the tiering merge policy. This approach will be explained in detail in Section 4.

Gap between theory and practice. Horizontal scheme is based on *full compaction* granularity, whereas file-level partial compaction granularity is the standard in most industrial systems, and has demonstrated practical advantages in certain aspects of empirical system performance, including significantly lower space amplification and improved worst-case throughput. This discrepancy arises because a single full compaction merges the entire contents of one level into the next level all at once. When the sizes of the participating levels are large, this process incurs substantial space amplification (since, for consistency, the data involved in compaction must be backed up until the compaction completes) and causes write stalls (as data in preceding levels cannot be transferred while compaction is underway). In contrast, each partial compaction merges only a single file from one level with a small set of overlapping files in the next level, thereby affecting only a fraction of

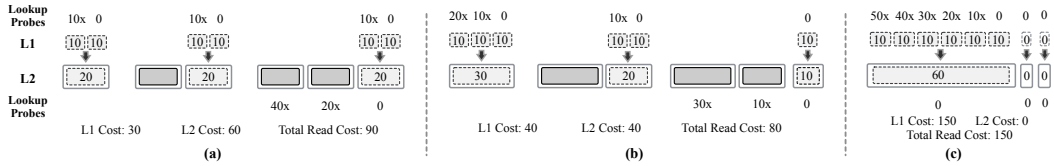


Fig. 4. Different compactions lead to varying read costs.

the data and considerably mitigating these issues. This gap between theory and practice motivates us to propose a novel design that integrates both the horizontal and vertical scheme, as described in Section 5.

4 The Horizontal-Tiering Scheme

In this section, we introduce the horizontal-tiering scheme, which serves as the backbone of our proposed method, Vertiorizon. The horizontal-tiering scheme redefines the horizontal algorithm by integrating the tiering merge policy, which governs when the data in a level is compacted into the subsequent level as a new isolated sorted run. While there are numerous compaction-timing choices in an LSM-tree, our proposed horizontal-tiering scheme employs a compaction sequence that achieves the optimal read-write performance trade-off, or precisely, achieves the minimum write cost when fixing the number of levels. This is the first significant extension of the horizontal growth scheme to the tiering merge policy, resulting in a much broader range of optimal read-write performance trade-offs for horizontal schemes. Although the proof of these optimality properties is quite intricate, the algorithm of the final scheme is easy to implement, making it appealing from an engineering standpoint as well.

4.1 High-Level Idea

Under the tiering merge policy, we first claim that the write amplification, in the long run, is the same *regardless of when* we compact data from a level to its subsequent level. This is because, with the tiering policy, each entry incurs only a single write amplification (i.e., rewritten onto the disk once) when passing through each level, exactly at the time when it is moved onto the level. As a result, the key to optimizing overall performance lies in selecting the optimal timing for compacting data from Level i into a new run at Level $i + 1$, in order to minimize read amplification.

A simple approach is to follow the vertical scheme by using the original Algorithm 1, but instead of merge-sorting data with the existing content in Level $i + 1$, the compaction would simply move data from Level i into a new run in Level $i + 1$. However, this naive strategy would result in significantly higher read amplification, because it would quickly create many runs in a level, even when the total data size in that level is still relatively small. Since the read cost is roughly proportional to the number of runs, this approach would lead to suboptimal read performance. This underscores the critical role of compaction timing in optimizing read efficiency.

To further explore the complexity of this challenge, we consider three examples in Figure 4, where a total of 60MB of data is moved from Level 1 to Level 2 across 3 compactions as data is written into the workload. We further assume there are x lookups when 1MB of data is ingested, and each lookup incurs an I/O for every run present at that time. [1] Figure 4 (a) describes an equal-frequency

[1]For simplicity, this assumption excludes common memory optimizations typically employed in LSM-trees, such as Bloom filters and block caches. However, our conclusions remain valid even when considering these optimizations. For example, in an LSM-tree utilizing Bloom filters and block caches, if the workload is uniform, there is still a consistent probability (approximately equal to the Bloom filter's false positive rate) that a data block retrieval will occur on each run during every lookup. In this context, the insight presented in this example continues to hold.

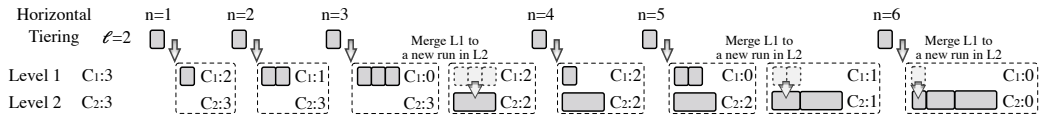


Fig. 5. Running example of the horizontal-tiering scheme.

compaction, where a compaction from Level 1 to Level 2 happens every insertion of 20MB data. Figure 4 (b) illustrates an increasing-frequency compaction strategy, where compactions occur after ingesting 30MB, 20MB, and 10MB of data, respectively. In contrast, Figure 4 (c) presents an extreme case where three compactions happen consecutively after inserting 60MB of data. Now, consider the total read cost during the insertion of 60MB. We will analyze the costs at Level 1 and Level 2 separately.

At Level 1, each run is 10MB in size, and if compactions are less frequent, earlier runs remain in the level for longer, increasing the read cost. Thus, Case (c) results in significantly more I/Os at Level 1 (150 times of I/O operations), compared to Case (a) and Case (b), which incur 30 and 40 times of I/O operations, respectively. However, compacting more frequently increases the cost at Level 2, as runs are generated earlier and serve more lookups. For example, Case (a) has a higher cost at Level 2 because runs stay longer in this level compared to the other two cases. Among the three, Case (b) achieves the lowest overall I/O cost. Its first compaction delayed until 30MB of data has been written, and subsequent compactions occur more frequently, triggered after 20MB and 10MB of written data, respectively. In this manner, by the third compaction, the two earlier sorted runs in Level 2 have existed for shorter duration compared to the case in Figure 4 (a). The first run has only undergone 30 times of lookups, while the second run has experienced just 10 times of lookups, as compared to 40 and 20 times of lookups in Case (a). This suggests that a decreasing-frequency compaction strategy may help reduce read amplification.

Building on this insight, an intuitive approach is to perform compactions to Level i less frequently when it is relatively empty, because the sorted runs written at this stage will exist for a longer period and thus contribute more to read amplification. In contrast, when Level i is close to full, we can add new sorted runs at shorter intervals, since they will not persist long and will therefore have a smaller impact on read amplification. We formalize the insight into developing the horizontal-tiering scheme below.

4.2 Algorithms

Building upon our previous analysis, we introduce a new horizontal-tiering scheme that extends the original horizontal scheme to better accommodate update-heavy workloads, providing a superior read-write trade-off compared to vertical-tiering. As shown in Algorithm 3, the overall algorithm is similar to the default horizontal scheme under leveling, with two key differences. First, the compaction counters of all levels are initialized to an integer k instead of zero (Lines 4–5). Second, when a buffer flush or compaction occurs, the compaction counter for the target level decrements instead of incrementing. Furthermore, a level's compaction is triggered when its counter reaches zero, and after compaction, its counter is reset to the current counter value of its subsequent level (Lines 6–12).

These two differences cause compactions to become increasingly frequent as the counters decrease. To illustrate this point, we present a simple example in Figure 5, where $\ell = 2$ and $k = 3$, meaning that all levels' compaction counters are initialized to 3. With each buffer flush, the compaction counter C_1 at Level 1 decreases by 1. When $n = 3$ (after three buffer flushes), C_1 is reduced to zero and meets the compaction condition $C_i = 0$, which triggers a compaction from Level 1 to Level 2. Following this compaction, C_2 decreases by 1, and C_1 is reset to the current value of C_2, now 2. After two additional buffer flushes, C_1 decreases to 0 once again, triggering another

Algorithm 2: Horizontal-tiering Scheme

```
   Input: Maximum number of levels ℓ, data size N
1  k ← 0;
2  while C(k+ℓ−1, ℓ) < N/B do
3      k ← k + 1
4  for i = 1 to ℓ do
5      C_i ← k
6  for each flush from buffer do
7      C_1 ← C_1 − 1;
8      for i = 1 to ℓ − 1 do
9          if C_i = 0 then
10             Trigger a full compaction from Level i to Level i + 1;
11             C_{i+1} ← C_{i+1} − 1;
12             for j = 1 to i do
13                 C_j ← C_{i+1};
```

compaction. Under the tiering merge policy, the data from Level 1 is merged into a new sorted run in Level 2. Subsequently, C_2 decreases from 2 to 1, and C_1 is reset to 1. The next compaction then occurs after only one buffer flush. As C_2 decreases due to compactions, the compactions from Level 1 to Level 2 become increasingly frequent. This behavior aligns with the best-performing increasing-frequency compaction strategy depicted in Figure 4(b).

The algorithm initializes the compaction counters for all levels to k (Lines 1–5), where k is the smallest integer satisfying the inequality $\frac{N}{B} \leq \binom{k+\ell-1}{\ell}$. The reason for such setting is as follows. In running the algorithm, the compaction counters for each level will gradually decrease, until the compaction counters of all levels reach zero. Therefore, the initial value of the compaction counter must be set based on the total data size N[2], where a larger data size corresponds to a larger initial compaction counter value k. We present a novel lemma to formalize the relationship between k and the data size, which explains the initial setting of k.

Lemma 4.1. *Under the horizontal-tiering scheme, if we initially set the compaction counters of all levels to k, then after* $\binom{k+\ell-1}{\ell}$ *buffer flushes, the compaction counters of all levels will decrease to zero.*

4.3 Optimality Analysis

The traditional horizontal scheme is known to achieve the optimal read-write trade-off under the leveling merge policy. Our proposed horizontal-tiering algorithm generalizes this optimality to the tiering merge policy. As we mentioned, the optimality analysis can be reduced to analyzing the total read cost, when the number of levels is fixed. Next, we show that the compaction sequence described in Algorithm 2 achieves the optimal read cost.

We first state the problem setting as follows. Between consecutive buffer flushes, compactions can be performed between adjacent levels to control the number of runs of Level 1 and other levels, reducing future write costs. A compaction that starts from Level l_1 and ends at Level l_2 would merge

[2]Specifically, N represents the data size that the LSM-tree will have written by the end of Algorithm 2. In practical applications, N may be set according to the capacity of the storage device or a more precise value based on user estimates. In general, as long as N falls within a reasonable range, deviations in its value do not significantly affect the practical performance of Algorithm 2.

all runs from Levels l_1, $l_1 + 1$, ..., $l_2 - 1$ into Level l_2. Different growth schemes result in varying compaction timings, which ultimately lead to different read overheads. For example, if compactions are too infrequent or not performed at all, sorted runs accumulate in Level 1, making lookups increasingly costly. The timing of each compaction can be represented by a triple $p = (I, l_1, l_2)$, where I denotes that the compaction occurs after the I-th buffer flush, and l_1 and l_2 indicate the starting and ending levels, respectively. This scenario can then be formulated into the following problem.

Problem 1. *Consider a workload that would incur a total of n sequential buffer flushes in the LSM-tree, which has a maximum of ℓ levels. Our objective is to find the optimal compaction sequence $S^* = \{p_1^*, p_2^*, ...\}$ to minimize the total read cost of processing this workload. We define this problem as $\psi(n, \ell)$.*

In a tiered LSM-tree, different growth schemes correspond to specific compaction sequences in Problem 1, resulting in varying costs. We give the following theorem stating that the compaction sequence employed by our horizontal-tiering algorithm incurs the minimal cost. Here we provide only a proof sketch, and the detailed proof is available in our technical report [1].

Theorem 4.2. *When there are total data of size $N = \binom{k+\ell-1}{\ell} \cdot B$ written into the LSM-tree, the compaction operations in the horizontal-tiering scheme described in Algorithm 2 is optimal for Problem 1.*

Proof Sketch. We can adopt a dynamic programming approach to solve Problem 1 by decomposing it into smaller subproblems. Consider the problem $\psi(n, \ell)$, where n is the number of buffer flushes and ℓ is the number of levels. Suppose that in the optimal compaction sequence $S^* = \{p_1^*, p_2^*, \dots\}$, the first compaction to the largest level p_f^* occurs after the i-th buffer flush. We can then partition the original problem into two subproblems – Subproblem $\psi(i, \ell - 1)$ that addresses the first i buffer flushes with the first $\ell - 1$ levels, and subproblem $\psi(n - i, \ell)$ that involves the remaining $n - i$ buffer flushes and all ℓ levels. If the optimal compaction sequences for these subproblems are S_1 and S_2 respectively, then the optimal solution to the original problem satisfies $S^* = \{S_1, p_f^*, S_2\}$.

We first explain why the portion of S^* preceding p_l^* must be S_1. Before the first compaction to the largest level in S^*, the data entering the LSM-tree with the first i buffer flushes remains within the first $\ell - 1$ levels since no compaction to Level ℓ occurs before p_f^*. This scenario exactly matches problem $\psi(n-i, \ell)$ because S^* is the optimal compaction sequence that minimizes the total read cost, and the portion of S^* preceding p_f^* must correspond to the optimal solution of $\psi(i, \ell-1)$. Otherwise, we could replace that portion with the optimal sequence S_1, resulting in a lower total read cost, contradiction ensues. Similarly, the portion of S^* following p_f^* must correspond to the optimal solution of $\psi(n - i, \ell)$, since the portion of data already moved into the largest level would not be merged with and affect any further data entering the LSM-tree. If it were not optimal, substituting it with S_2 would yield a compaction sequence with a lower read cost. By recursively applying this decomposition, we can solve Problem 1 by breaking it down into subproblems until they become trivial (e.g., when $n = 1$ or $\ell = 1$). This dynamic programming approach constructs the optimal compaction sequence by combining the solutions of all these subproblems.

Furthermore, we can prove the following claim by mathematical induction. For any n and ℓ, if in the optimal compaction sequence corresponding to $\psi(n, \ell)$, the last compaction to the largest level occurs after the i-th buffer flush, then i must satisfy

$$\binom{m-1}{\ell} \le n - i \le \binom{m}{\ell}, \text{ and } \binom{m-1}{\ell-1} \le i \le \binom{m}{\ell-1} \tag{1}$$

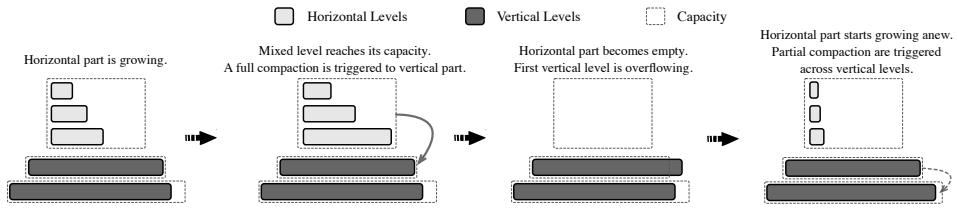


Fig. 6. General process in Vertiorizon.

where m is the integer that satisfies $\binom{m}{\ell} \leq n \leq \binom{m+1}{\ell}$. Having identified the corresponding timing of compaction p_f^*, we can divide any problem into two subproblems. For each of these subproblems, we apply this equation to determine their first compaction to the largest level p'_f. This recursive process continues until all resulting subproblems can no longer be subdivided, that is, when either $n = 1$ or $\ell = 1$. The collection of all compactions determined through this process constitutes an optimal compaction sequence, which matches the compaction operations in Algorithm 2. □

5 The Vertiorizon Growth Scheme

In the previous section, we introduced the enhanced horizontal growing scheme, which maintains high efficiency under update-heavy workloads. However, it still suffers from high space amplification and write stalls due to its reliance on full compaction. This limitation prompts us to propose Vertiorizon, a novel LSM growing scheme that combines both horizontal and vertical approaches to provide robust and comprehensive performance.

5.1 Overall Layout

Vertiorizon divides the LSM-tree into two sections from top to bottom – horizontal part and vertical part, such that the upper horizontal part utilizes the horizontal scheme for improving the overall performance, while the lower vertical part manages the largest two levels, aiming to mitigate space amplification and alleviate severe write stalls through partial compaction.

Horizontal Part. This part comprises the upper levels of the LSM-tree, where the vast majority of compactions occur. These levels follow the horizontal growth scheme to achieve better efficiency. The horizontal part has a predetermined capacity, which is n times the buffer size (we will discuss how to set n shortly). When the data size reaches this capacity, a full compaction is performed to merge all data into the vertical part. The structure of the horizontal part is flexible such that it can be adjusted according to the characteristics of the workload. Specifically, it has two configurable knobs. (1) Merge Policy: this allows switching between a leveling merge policy and a tiering merge policy. (2) Number of Levels ℓ: under the leveling policy, a smaller number of levels leads to better read performance; under the tiering policy, fewer levels result in better write performance. Section 5.2 describes how Vertiorizon self-tunes these parameters.

Vertical Part. The vertical part comprises the two largest levels of the LSM-tree, each possessing fixed capacities similar to those in the vertical scheme. When the horizontal part reaches its total capacity, all data is merged into the first level of the vertical part through full compaction. Once the first level becomes full, data is moved to the second level—the largest level—via partial compaction. Like the horizontal part, the vertical part can switch between leveling and tiering merge policies and includes an adjustable parameter, i.e., the size ratio T between adjacent levels in the vertical scheme. Specifically, by default, the capacity of the first level in the vertical part is T times the total size of the horizontal part, which is $n \cdot T \cdot B$. The second level's capacity is T times that of the first level, resulting in $n \cdot T^2 \cdot B$. Because partial compaction between the two levels in the vertical

part causes only minimal space amplification and write stalls, Vertiorizon can roughly reduce space amplification and write stalls by a factor of T compared to an LSM-tree that employs full compaction throughout.

One may wonder why we empirically fix two levels in the vertical part. The rationale behind this is that the primary purpose of the vertical part is to avoid excessive space amplification and write stalls by enabling partial compaction between the largest levels By configuring the largest two levels to use the vertical scheme and applying partial compaction between them, Vertiorizon can significantly reduce space amplification and write stalls by roughly a factor of T compared to an LSM-tree that employs full compaction throughout. We found that practically such setting is quite sufficient and effective. Further increasing the number of levels in the vertical part would require proportionally reducing the scale of the horizontal part, thereby diminishing its benefits. Note that, even though the number of levels in the vertical part is fixed, we can still trade between write amplification and space amplification by tuning the parameter T. Increasing T can further decrease overall space amplification and write stalls but at the cost of increased write amplification in the vertical part.

Running Example. Figure 6 displays a typical process of Vertiorizon when ingesting new data. Initially, the horizontal part is ready to grow, while the first level of the vertical part holds data volume near its full capacity. Once the horizontal part is full, it starts a compaction to the first vertical level. This process empties all the horizontal levels. Subsequently, as the data volume in the first vertical level exceeds its capacity, data begins to gradually move into the largest level via partial compactions, after which the horizontal part begins a new phase of growth.

Dynamic Resizing of Horizontal Part. To maintain the structure of the LSM-tree as the data volume scales up, Vertiorizon dynamically increases the parameter n, which controls the capacity of the horizontal part. Specifically, n is initiated with a moderate value according to the data scale. When the largest level of the vertical part reaches its capacity, we increment n by a factor of $1/T$ after the next full compaction clears the horizontal part, and adjust the capacity of the vertical part accordingly. This adjustment incurs no additional overhead because it is after a full clear of the horizontal part. In this way, the overall capacity of the LSM-tree under Vertiorizon can gradually expand.

Optimize Write Amplification by Adjusting Size Ratio. We can further optimize Vertiorizon's write amplification by adjusting the size ratio of the two largest levels in the vertical part. These two levels are special because they transition from a level employing full compaction to that with partial compaction, opening up an opportunity to optimize the overall write amplification.

Since the vertical part employs partial compaction, the data volume in its first level consistently remains close to its capacity. Consequently, the compaction from the horizontal part to the first level of the vertical part will always incur a write amplification factor of T, because the data volume in this level is maintained at T times the capacity of the horizontal part. In contrast, the average write amplification incurred when compacting from the first level to the second level of the vertical part is only $\frac{T+1}{2}$. This reduction occurs because partial compaction leads to uneven data density within the level, as established in previous studies [35]. Due to these differences, setting equal size ratios of T results in different write amplifications for compactions at the two levels of the vertical part, leading to suboptimal overall write amplification. Specifically, we set the size ratio between the horizontal part and the first level of the vertical part to T', and the size ratio between the first and second levels of the vertical part to T^2/T'. This setting ensures that the capacity of the largest level in Vertiorizon remains unchanged, thereby maintaining the overall scale of the LSM-tree. Under this setup, the write amplification at the first level of the vertical part becomes T', while the write amplification at the second level is adjusted to $\left(T^2/T' + 1\right)/2$. Note that

$$T' + (\frac{T^2}{T'} + 1)/2 \geq 2\sqrt{T' \cdot \frac{T^2}{2T'}} + \frac{1}{2} = \sqrt{2}T + \frac{1}{2} \tag{2}$$

The above inequality is due to AM-GM Inequality, and the equality holds when $T' = T/\sqrt{2}$.

5.2 Self-Tuning for Various Workloads

Cost Model for Horizontal Part. Vertiorizon self-tunes its horizontal part to ensure it reaches the optimal point on the read-write trade-off curve for any given workload. To this end, we have carefully analyzed the cost of each operator and proposed a cost model for the horizontal part.

Point and Range Lookup Cost. A point lookup probes each sorted run from top to bottom. As the total data range is at least the size of the largest level in the vertical scheme, which is significantly larger than the total data size in the horizontal part, the probability of an arbitrary key appearing in the horizontal part is very low. Therefore, we model the lookup cost of the horizontal-leveling scheme by multiplying the number of levels ℓ with the false positive rate of the Bloom filters f (i.e. $R_l = \ell \cdot f$). The analysis of the lookup cost for the horizontal-tiering scheme is quite intricate. Since each run exists for varying and irregular periods, they contribute differently to the overall read cost. Below we present a novel analysis results in the following lemma stating the lookup cost under horizontal-tiering scheme. Please refer to our technical report [1] for a complete proof.

Lemma 5.1. *Let m be the integer that satisfies $\binom{m}{\ell} \leq n \leq \binom{m+1}{\ell}$. Then the lookup cost of the horizontal-tiering scheme can be computed by the following equation.*

$$R_t = \left(\ell \cdot \binom{m}{\ell+1} + (m - \ell + 1) \cdot \left(n - \binom{m}{\ell}\right)\right) \cdot \frac{f}{n} \tag{3}$$

In terms of incurred overhead, the primary distinction between range lookups and point lookups lies in their interaction with the runs in the horizontal part of the LSM-tree. Specifically, for each existing run, point lookups typically result in a disk I/O operation only in cases of false positives of Bloom filters, whereas range lookups invariably incur a disk I/O for every run. Aside from this difference, the derivations for point lookups and range lookups are identical. Therefore, the range lookup costs under leveling and tiering satisfy $Q_l = \frac{R_l}{f}$ and $Q_t = \frac{R_t}{f}$, respectively.

Update Cost. With tiering, each entry would only be involved in one compaction at each level. Therefore, the I/O cost per update under the horizontal-tiering scheme can be modeled by dividing the number of levels ℓ by the page size in entries P, reminding that each disk I/O moves an entire page of entries in bulk. Furthermore, we employ the following lemma to calculate the update cost of the horizontal-leveling scheme.

Lemma 5.2. *Let m be the integer that satisfies $\binom{m}{\ell} \leq n \leq \binom{m+1}{\ell}$. Then the update cost of the horizontal-leveling scheme can be represented as the following equation.*

$$W_l = \left(\ell \cdot \binom{m+1}{\ell+1} + (m+1) \cdot \left(n - \binom{m}{\ell}\right) - (\ell - 1) \cdot n\right) \cdot \frac{1}{n \cdot P} \tag{4}$$

Navigate Toward the Best Design. For a given design, we can weight its I/O cost of the update, point lookup, and range lookup W, R, and Q with their respective proportion in the workload, denoted as w, r, and q, to derive the average per operation I/O cost incurred by this design.

$$\zeta = w \cdot W + r \cdot R + q \cdot Q \tag{5}$$

We explore all possible configurations for the horizontal part of Vertiorizon. Specifically, leveraging the convexity of the cost functions under both leveling and tiering merge policies with respect to the number of levels ℓ, we initiate our search from the minimum feasible value, $\ell = 2$ and

increase ℓ. The optimal setting for each merge policy is identified by locating the saddle point of the cost function ζ, where the function transitions from decreasing to increasing. Finally, we select the superior design between the leveling and tiering merge policies. To enhance the efficiency of self-tuning, we employ binary search, noting that the maximum value of ℓ cannot exceed n. Therefore, the complexity of the self-tuning process is only $O\left(\log_2 n\right)$, which is efficient practically.

5.3 Optimization for Skewed Workloads

As an optional technique, we can further optimize our technique by making use of skewness information. We adopt the definition of skewed distribution from previous works [11], which introduces a sharp distinction between hot and cold keys for analytical convenience. The skewed distribution comprises two distinct uniform distributions: U_h and U_c, where U_h represents a small set of frequently updated hot keys, and U_c denotes a much larger set of regular cold keys. We assume that the hot key set has a high probability of appearing in each buffer flush, and it is likely that each subsequent run containing all the keys from the hot set. For instance, under a uniform workload, merging two runs of size D during compaction results in a new run of size $2D$ because the keys are unlikely to overlap. In contrast, under a skewed workload, both runs contain the hot keys, and obsolete versions are discarded during compaction. This leads to a run size of only $2D - U_h$, while U_h redundant hot keys are eliminated. Therefore, under the leveling merge policy, each level grows more slowly with each compaction as the workload becomes more skewed. Whereas, under the tiering merge policy, the impact of workload skewness on performance is less significant because new data flowing in each level does not merge with existing data.

We then discuss how we adapt the compaction strategy when the impact of skewness is more pronounced, i.e., under the leveling merge policy. The write amplification of a level, incurred by the compaction of its upper level, is proportional to the current size of the level. Under skewed workloads, the growth of levels is slower than in a uniform workload because more duplicate entries appear in a compaction, leading to a slower increase in write amplification. Consequently, we see a diminished benefit of performing compaction, which can reduce the write amplification of subsequent compactions. Building on this insight, it is beneficial to align the compaction frequency with the workload skewness. Interestingly, only minor modifications to Algorithm 1 are necessary. Specifically, we introduce a coefficient $\alpha = \frac{U_h}{B}$ to quantify the degree of workload skewness, where B is the buffer size. This coefficient also reflects the slowdown in the growth of the first level's size under a skewed workload, as each hot key has a high probability of appearing in each buffer flush. As the workload becomes more skewed, or α increases, compactions should occur less frequently since the level grows slower. This adjustment is implemented by increasing the compaction trigger threshold of the first level based on skewness α. While there are multiple ways to achieve the adjustment, in this paper, we change the condition in Line 6 in Algorithm 1 to $C_i > C_{i+1} + \delta$ when $i = 1$, where δ is the largest integer satisfying

$$\frac{\alpha}{1-\alpha} \geq \frac{\delta \cdot (\delta + 1)}{2}. \tag{6}$$

In Equation 6, the term $\alpha/(1-\alpha)$ represents a skewness-related coefficient, reflecting the ratio of hot keys in each flush. Notably, it shows an approximately quadratic relationship with δ. Equation 6 is grounded in a deep intuition comparing the benefits and costs in adjusting δ. Due to space limitation, we refer interested readers to our technical report [1] for a rigorous derivation of this equation.

5.4 Vertiorizon as a Backbone

For existing LSM-based designs using the vertical scheme, our improved Vertiorizon can serve as a backbone to replace the vertical scheme in enhancing their system performance.

For example, we consider the well-known LSM-based design lazy-leveling, proposed in Dostoevsky [15]. In lazy-leveling, the largest level of the LSM-tree is configured using the leveling merge policy, while the remaining levels employ tiering. We can embed Vertiorizon into this design by replacing all the tiering levels with the horizontal part of Vertiorizon utilizing the horizontal-tiering scheme. Specifically, suppose the original lazy-leveling design has L levels. To maintain consistency during embedding, we set the number of levels ℓ in the horizontal part to $L-1$ and adjust its capacity to $B \cdot T^{L-1}$, corresponding to the size of the largest level being replaced. In this way, we integrate Vertiorizon into lazy-leveling without altering the original configuration. Since the number of levels using tiering remains unchanged after the replacement, the update cost is the same as that of the original lazy-leveling design. Meanwhile, the lookup cost is improved thanks to the nice property of horizontal scheme (Theorem 4.2).

Introducing New Bloom Filter Layout. Some LSM-based designs employ the Monkey Bloom filter layout [14] to improve their point lookup performance. The idea is to allocate higher bits-per-key for Bloom filters at smaller levels. However, the optimization strategy rests on the assumption that the data size within every level consistently remains at that level's capacity. This assumption is not accurate under the full compaction granularity employed in many state-of-the-art LSM-based designs [18, 36, 55]. Because each full compaction empties a level after merging all its data. Consequently, the level would progress from being vacant to reaching its full capacity before another compaction occurs. This problem also impedes the compatibility of the Monkey layout with Vertiorizon.

To address this problem, we propose a new dynamic Bloom filter layout, which is designed to achieve better read optimization under the horizontal scheme by more precisely modeling full compaction processes. Unlike the static allocation of the Bloom filter under the Monkey layout, the dynamic layout adaptively adjusts the Bloom filter's allocation in response to changes in data volume for each level and incorporates future changes in data volume into the model's considerations. However, directly adjusting Bloom filter's allocation requires fetching the original data indexed by the Bloom filter from disk into memory in order to reconstruct the Bloom filter based on the new bits per key, which incurs significant additional overhead. To circumvent this issue, we never directly adjust Bloom filter's allocation under the dynamic layout. Each time a full compaction occurs between two levels and data is loaded into memory, we simultaneously reallocate and reconstruct the Bloom filter associated with those two levels.

6 Additional Related Works

LSM-tree-based Designs. Recent LSM-tree-based optimizations include exploring innovative compaction policies and routines [11, 14–16, 18, 28, 29, 48, 50, 51], improving Bloom filters [14, 17, 62, 63] and range filters [32, 40, 60], as well as harnessing novel hardware for performance gains [3, 27, 54, 56, 59, 61]. Techniques have also been investigated for key-value separation [9, 37], managing in-memory hot entries [5], enhancing concurrency [24, 52], and prefix indexing [40, 57, 60]. Furthermore, some efforts have been dedicated to practical performance enhancements by reducing tail latency [6, 38, 51], exploiting workload characteristics [2, 49, 58], and refining the memory distribution [8, 31, 39]. To the best of our knowledge, most of these works are either based on the vertical growth scheme or are orthogonal to the growth scheme, as the vertical scheme is typically considered the default in LSM-trees.

Studies on Horizontal Growth Scheme. Over the past decade, several studies [41–45] have recognized the advantages of the horizontal scheme and have focused on its analysis and application. Mathieu *et al.* [44] analyzed and optimized the horizontal growth scheme in BigTable from the perspective of competitive ratios. In contrast, Mao *et al.* [41–43] conducted a series of

experimental studies comparing the vertical and horizontal schemes under regular and spatial workloads. Although these studies are insightful, their efforts primarily concentrated on theoretical and experimental investigations of the existing horizontal growth scheme using the leveling merge policy, without proposing new designs. In this paper, we further enhance the applicability of the horizontal scheme through a novel design. Specifically, we introduce the horizontal-tiering scheme, which extends the horizontal scheme to the tiering merge policy, and present Vertiorizon, an enhanced growth scheme that embodies the strengths of both the vertical and horizontal schemes.

7 Evaluation

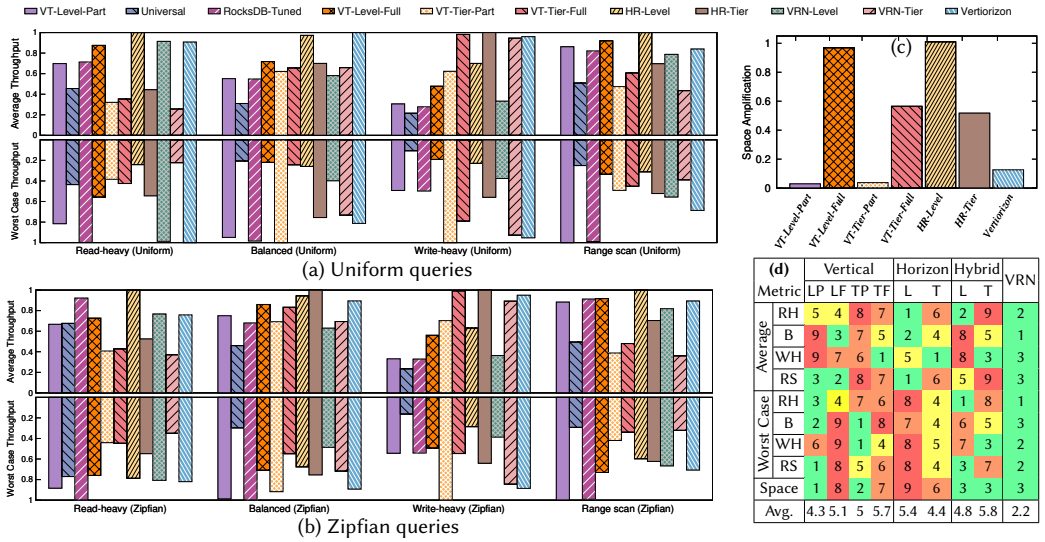
By default, our experiments are conducted on a server with Intel(R) Gold 6326 CPU @ 2.90GHz processor, 256GB DDR4 memory, and 1TB NVMe SSD, running 64-bit Ubuntu 20.04.4 LTS on an ext4 partition.

Baselines and Implementation. We assess Vertiorizon (abbr. VRN) against ten baselines. The implementation of Vertiorizon and all the baselines are built on Facebook's RocksDB [20], a popular LSM-tree-based key-value storage system and also been employed by many previous LSM-tree studies [9, 14–16, 18, 37, 39, 49, 59, 60].

The first four baselines are all possible combinations of compaction schemes and merge policies discussed in our technical section: vertical-leveling (abbr. VT-Level-Part), vertical-tiering (abbr. VT-Tier-Part), horizontal-leveling (abbr. HR-Level), and horizontal-tiering (abbr. HR-Tier). By default, the vertical schemes employ partial compaction, while the horizontal schemes use full compaction. This configuration makes vertical-leveling equivalent to the default setup of RocksDB. For the other three baselines, we extended RocksDB's code to implement them. We also included two additional baselines derived from the default RocksDB (i.e., Vertical-Level-Part): "Universal" and "RocksDB-Tuned". They share most of the same settings with Vertical-Level-Part but are extended in two key ways. Specifically, Universal enables RocksDB's universal compaction [22], an age-based compaction scheme aligned with the tiering merge policy, whereas RocksDB-Tuned applies additional tunings to certain RocksDB parameters specific to the vertical growth scheme. [3] Furthermore, although not commonly used in practical applications, designs based on the vertical scheme can adopt full compaction to sacrifice fine-grained file-level compaction for improved performance. This leads to two additional baselines: vertical-leveling with full compaction (abbr. VT-Level-Full) and vertical-tiering with full compaction (abbr. VT-Tier-Full). Additionally, we compare two designs based on Vertiorizon but without self-tuning. Instead, they use a fixed leveling design and a fixed tiering design with $\ell = 2$ in the horizontal part, referred to as VRN-Level and VRN-Tier, respectively.

Experimental Setting. For all methods, we retain most orthogonal parameters as RocksDB's default settings, which are known to be developer-optimized for general SSD-based applications [21]. Furthermore, following previous works [15, 28], we enable direct I/O by setting the parameters "use_direct_io_for_flush_and_compaction" and "use_direct_reads" to true in RocksDB. We configure the memory buffer size to 2 MB by setting "write_buffer_size" and "target_file_size_base" to 2,097,152, and set the LSM-tree size ratio $T = 6$ by assigning "max_bytes_for_level_multiplier" to 6. We evaluate the impact of different Bloom filter settings, varying the bits-per-key from 4 to 20 (Figure 8), while using a default value of 5 in other experiments, as this aligns with the latest RocksDB default. We also analyze the impact of block cache size, varying from 16MB to 64GB (Figure 8 (e)), with a default of either 32MB or 64GB in other experiments. Specifically, Figure 7, Figure 8 (a) and (d), Figure 10 (b) and (c), and Figure 9 (a) use a 32MB block cache, while

[3] Specifically, we enable "level_compaction_dynamic_level_bytes", set "max_bytes_for_level_multiplier" to 10, and configure "compaction_pri" to "kOldestSmallestSeqFirst".



(d)

Metric		Vertical				Horizon		Hybrid		VRN
		LP	LF	TP	TF	L	T	L	T	
Average	RH	5	4	8	7	1	6	2	9	2
	B	9	3	7	5	2	4	8	5	1
	WH	9	7	6	1	5	1	8	3	3
	RS	3	2	8	7	1	6	5	9	3
Worst Case	RH	3	4	7	6	8	4	1	8	1
	B	2	9	1	8	7	4	6	5	3
	WH	6	9	1	4	8	5	7	3	2
	RS	1	8	5	6	8	4	3	7	2
Space		1	8	2	7	9	6	3	3	3
Avg.		4.3	5.1	5	5.7	5.4	4.4	4.8	5.8	2.2

Fig. 7. Vertiorizon shows desirable performance across all scenarios and metrics. Table (d) presents the ranking of each method under various metrics. (For methods, "L" means leveling, "T" means tiering, "P" means partial, "F" means full, and "VRN" is Vertiorizon. For workloads, "RH" is read-heavy, "B" is balanced, "WH" is write-heavy, and "RS" is range scan.)

Figure 8 (b) and (c), Figure 10 (d) and (e), along with Figure 9 (b) use a 64GB block cache. These two configurations are chosen because a 32MB block cache aligns with the default setting of the latest RocksDB version, while a 64GB block cache represents scenarios with abundant memory resources. Evaluating both settings reveals different performance rankings among the baselines, providing a more comprehensive understanding of system performance. We evaluate each under various workloads with different lookup and update ratios; we initially bulk-load 10^7 key-value entries of 1KB into the LSM-tree. Each entry is generated through the YCSB benchmark with either uniform or Zipfian distribution, comprising a 128-byte key and an 896-byte value. For testing, we generate and process a workload with 4,000,000 operations, each can be a lookup or an update, and report the overall average throughput and the worst-case throughput of processing the workload. To measure the worst-case throughput, we maintain a time window containing the most recent 100,000 processed operations and record the lowest throughput observed within this window, which is then reported as the worst-case throughput. To measure space amplification, we record the peak disk space occupied by RocksDB's storage directory during runtime. By subtracting the data size from the total space used by the LSM-tree, and then dividing the difference by the size of unique entries, we derive the additional space amplification per entry.

Vertiorizon achieves desirable performance across all scenarios and dimensions. In Figure 7 (a) and (b), we compare Vertiorizon against all baselines in terms of average throughput and worst-case throughput across various YCSB workloads under both uniform and Zipfian distributions, with y-axis standardized to facilitate comparison. The workloads include balanced workload (50% updates and 50% point lookups), write-heavy workload (90% updates and 10% point lookups), read-heavy workload (10% updates and 90% point lookups), and range scan workload (25% range lookups and 75% updates). Overall, the results under the uniform and Zipfian distributions are similar. From Figure 7 we have the following discussions.

Results on read-heavy workload. (1) Leveling-based approaches consistently achieve better overall performance compared to tiering-based methods, because tiering-based methods tend to suffer from

high read amplification, resulting in lower throughput in read-heavy scenarios. (2) Under the vertical scheme, the full compaction method (e.g., VT-Level-Full) delivers superior average throughput compared to its corresponding partial compaction method (e.g., VT-Level-Part). This is because partial compaction introduces additional write amplification. However, the worst-case throughput of VT-Level-Full is lower than VT-Level-Part, because full compaction leads to significant write stalls when merging large levels. We also note that the worst-case throughput is also influenced by overall performance. For instance, VT-Tier-Part has the lowest average throughput among all vertical baselines, even though it incurs minimal write stalls with partial compaction. (3) Baselines under the horizontal scheme demonstrate higher average throughput than their vertical scheme counterparts with the same merge policy, owing to the optimal read-write trade-off of the horizontal scheme. For example, the average throughput of HR-Level is notably better than VT-Level-Full. Nevertheless, because these horizontal methods rely on full compaction, they suffer from reduced worst-case throughput due to write stalls. (4) Vertiorizon excels in both average and worst-case throughput by employing a hybrid layout that delivers strong read and write performance while alleviating write stalls. This design enables Vertiorizon to achieve strong average throughput and also maintain the best worst-case throughput among all methods.

Results on write-heavy workload. (1) All tiering-based methods exhibit desired overall performance, with HR-Tier achieving the highest average throughput and VT-Tier-Partical achieving the best worst-case throughput. In contrast, all leveling-based methods struggle in this read-heavy scenario. (2) Similar to our observations under the read-heavy workload, both VT-Tier-Full and HR-Tier experience lower worst-case throughput compared to partial or hybrid designs like VT-Tier-Part and Vertiorizon, due to write stalls caused by full compaction. However, the relative decline in worst-case throughput is less pronounced than that observed among leveling-based methods under the read-heavy workload. This is because, under the tiering merge policy, compaction data is not merged with existing data in the next level, resulting in smaller compacted run sizes compared to leveling and thus reduced write stalls. (3) Although VT-Tier-Part achieves the best worst-case throughput among tiering-based methods, it suffers from significantly lower average throughput—only 60% of HR-Tier's average throughput. This degradation is attributed to partial compaction under tiering, which introduces higher write amplification and increased read amplification, as runs at each level are not cleared after compaction and continue to accumulate. (4) Most leveling-based baselines perform poorly overall. Among these, HR-Level shows relatively better performance due to the superior read-write trade-off of the horizontal scheme. (5) Meanwhile, Vertiorizon achieves satisfactory performance in both average throughput and worst-case throughput on write-heavy workload, through self-tuning to a tiering-based design similar to VRN-Tier.

Results on balanced workload. Under the balanced workload, Vertiorizon achieves the highest average throughput by automatically tuning to a design that optimally suits balanced read-write demands. Simultaneously, Vertiorizon maintains satisfactory worst-case throughput. Most of the baselines exhibit markedly lower average throughput, except for HR-Full, which attains average throughput close to the best. This performance is attributed to the optimal read-write trade-off provided by the horizontal scheme and its settings being well-positioned on the trade-off curve for the balanced workload. VT-Level-Part and VT-Tier-Part achieve the best worst-case throughput due to their use of partial compaction. However, this also leads to their average throughput being relatively poor among the baselines. Meanwhile, the worst-case throughput of several baselines employing the tiering merge policy is unsatisfactory. Finally, although VRT-Level and VRT-Tier adopt the same hybrid layout as Vertiorizon, their fixed settings prioritize either read or write performance at the expense of the other, resulting in mediocre performance under the balanced workload.

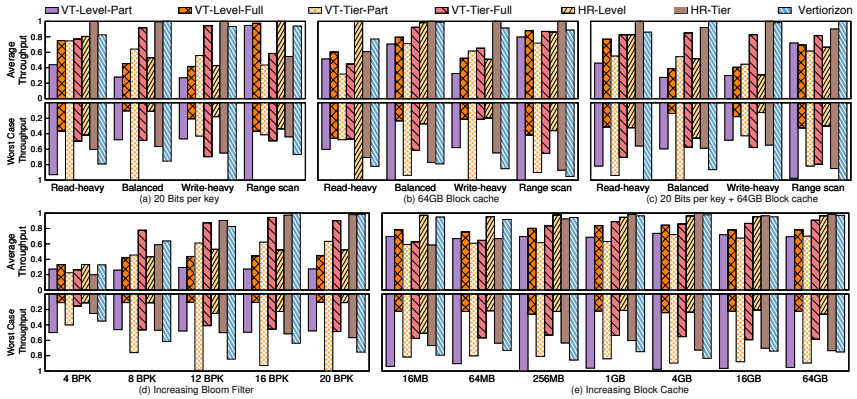


Fig. 8. The performance with varied bits per key and block cache size.

Table 3. Rankings of each method across different metrics for each case in Fig. 8 (rounded to nearest tenth).

Metrics		Vertical				Horizon		VRN
		LP	LF	TP	TF	L	T	
Average	(a)	6.0	4.5	5.5	3.5	3.3	2.5	2.0
	(b)	6.0	4.0	6.3	4.0	3.5	1.8	1.8
	(c)	6.3	5.3	5.3	3.3	5.0	1.5	1.5
	(d)	6.4	5.2	4.4	2.8	4.2	3.0	1.8
	(e)	5.7	4.4	6.9	4.4	2.1	2.7	2.1
Worst Case	(a)	2.8	6.5	3.0	3.5	6.3	4.3	1.8
	(b)	2.5	6.3	2.8	5.5	5.0	3.5	2.3
	(c)	2.3	6.5	3.0	4.0	6.5	3.5	1.5
	(d)	3.8	7.0	1.2	4.6	6.0	3.4	2.2
	(e)	1.0	6.4	2.1	5.0	6.6	4.0	2.9
Avg.		4.3	5.6	4.1	3.9	4.9	3.0	2.0

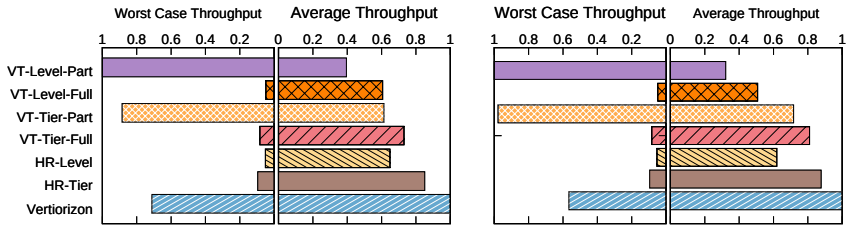
Results on range scan workload. The performance trends are similar to those observed under the read-heavy workload, because a range lookup is sustaintially more costly than a point lookup. Notably, we observe a more substantial variation in worst-case throughput across methods, as the heavier updates in this workload further exacerbates performance variability caused by write stalls.

Assessing advanced variations of RocksDB. From Figures 7(a) and (b), Universal consistently underperforms under all settings due to its simplistic trigger condition for universal compactions, which limits algorithmic efficiency. RocksDB-tuned typically surpasses Vertical-Level-Part (the default RocksDB configuration), especially under read-heavy workloads. However, its average throughput still struggles in update-intensive scenarios, as it is essentially based on the vertical scheme and the leveling merge policy.

Comparison of space amplification. Figure 7 (c) presents the space amplification under the balanced workload with uniform distribution. Among all the baselines, the two full-compaction methods, VT-Level-Full and HR-Level, entail the highest space amplification. In contrast, VT-Tier-Full and HR-Tier, which also use full compaction but with the tiering policy, have a lower space amplification. This reduction occurs because, under the tiering merge policy, compactions to a level do not merge with existing data, which would temporarily duplicate during the merge sort process. Compared methods with full compaction, the methods with partial compaction often entail much lower space amplification, as exemplified by VT-Level-Part and VT-Tier. Meanwhile, Vertiorizon has a quite desirable space amplification, as it adopts the vertical scheme with partial compaction in the largest levels.

Overall performance ranking. Figure 7 (d) presents the ranking of each method across various performance metrics, including average and worst-case throughput across all uniform workloads, as well as space amplification. Overall, the results demonstrate that Vertiorizon consistently delivers strong performance across all metrics, whereas each baseline falls short in at least one dimension of performance.

The impact of large block cache and Bloom filters. Figure 8 presents the throughput of each approach with large Bloom filters (up to 20 bits per key) and block cache (up to 64GB). The workloads for (d) and (e) follow a balanced distribution, consisting of 50% updates and 50% point lookups, uniformly distributed. Aside from these varying parameters, all other settings follow the settings in Figure 7. To examine whether Vertiorizon maintains its superiority under large Bloom filters and block cache, we present the corresponding performance rankings in Table 3, aligned with each subfigure in Figure 8.



(a) Moderate Memory Setting (b) Large Memory Setting

Fig. 9. Evaluation of Vertiorizon under very large data scale.

Several key observations emerge when comparing these rankings with Figure 7. **First**, Vertiorizon remains highly competitive across all rankings and consistently achieves the highest overall performance. While our approach may not always be the best for a specific workload, this overall ranking highlights its *robustness* across various scenarios. In contrast, every other method exhibits at least one scenario where its effectiveness diminishes. **Second**, when ample memory resources are available, all approaches are expected to achieve better performance, resulting in closer performance across them. As shown in Figure 8 (d), a higher bits-per-key setting reduces read costs, leading to increased throughput. Similarly, Figure 8 (e) shows a slight performance improvement when the block cache size is large. This improvement is primarily driven by a higher cache hit rate. However, since frequent compactions tend to invalidate the block cache, the benefit remains limited even with a larger cache size. **Third**, since large Bloom filters and block caches generally enhance point-lookup performance, methods that prioritize write optimization (e.g., tiering methods) tend to improve their rankings. Notably, HR-Tier now ranks second in average throughput, making it more comparable to Vertiorizon. However, it is important to note that HR-Tier is also a novel approach introduced in this paper.

Performance under larger datasets. In Figure 9, we compare our method with the baselines under significantly larger memory buffer size and data size, aiming to offer a comparative analysis that reflects scenarios of the big data era. As such large-scale experiments have high demand on disk space, we use an alternate server with a larger disk capacity, specifically featuring a 13th Gen Intel(R) Core(TM) i9-13900K CPU @ 4.0GHz processor, 128GB DDR4 main memory, and a 2TB NVMe SSD. In this experiment, most configuration settings align with precedent experiments in Figure 7. Except for that for all methods, the buffer size is increased to 64 MB, and the total preloaded data is increased to 500 GB, corresponding to 500,000,000 key-value entries. We then run a workload of 200,000,000 operations to evaluate each method's performance. Because loading such a large amount of data is time-consuming, we select a relatively general workload setting to conduct this experiment: the read-write balanced workload (50% updates and 50% point lookups) with a uniform key distribution. We use metrics similar in Figure 7 (a), including average throughput and worst-case throughput during workload processing. Figure 9 (a) uses a moderate memory configuration with 32MB block cache and 10 bits per key Bloom filter, while (b) employs a larger memory configuration with 64GB block cache and 20 bits per key Bloom filter.

We observe that when the data scale becomes very large, Vertiorizon excels in average throughput. In particular, its average throughput surpasses that of baselines such as HR-Level and HR-Tier, which had previously performed well under smaller-scale settings. The underlying reason is that these baselines rely exclusively on full compaction, leading to drastically severe write stalls at very large data scales and causing their worst-case throughput to deteriorate significantly. In contrast, by employing a hybrid growth scheme, Vertiorizon maintains a satisfactory worst-case throughput, trailing behind methods that fully adopt partial compaction (e.g., VT-Level-Part and VT-Level-Full).

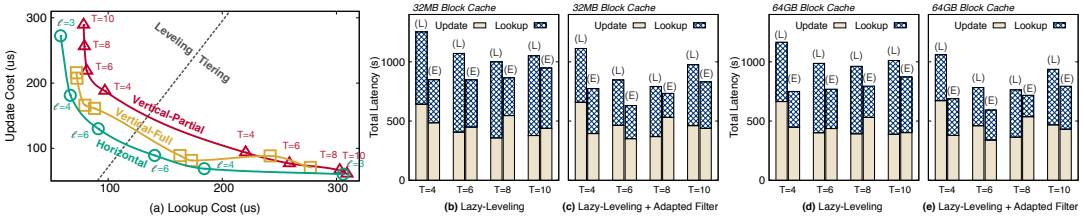


Fig. 10. (a) compares the trade-off between vertical scheme and horizontal scheme. (b) ~ (e) shows that Vertiorizon can be applied to existing designs to enhance their performance. ("L" means the lazy-leveling design and "E" means lazy-leveling embedded with Vertiorizon.)

Nevertheless, because partial compaction is overall less efficient, these methods post only mediocre results in terms of average throughput.

From horizontal-leveling to horizontal-tiering: enhanced read-write trade-off curve. In Figure 10 (a), we compare the read-write trade-off curves of the vertical and horizontal schemes. In the vertical scheme, the trade-off between read and write performance is controlled by adjusting the size ratio T and the merge policy. The horizontal scheme's trade-off, conversely, is regulated by varying the number of levels ℓ and the merge policy, and is enhanced by our horizontal-tiering algorithm under tiering merge, To evaluate these schemes comprehensively, we assess ten different designs under each compaction scheme with varying settings. Specifically, for the vertical scheme, we test both partial compaction and full compaction designs with adjacency ratios $T = 4, 6, 8, 10$ under both leveling and tiering merge policies. Similarly, for the horizontal scheme, we evaluate designs with the number of levels $\ell = 3, 4, 6$ under both merge policies. We evaluate them on a workload consisting of 20 million operations with equal amounts of updates and point lookups. For each design, we plot a point on Figure 10 (a) based on its per-lookup and per-update latency, where the horizontal axis represents the read cost and the vertical axis represents the write cost.[4] By connecting all the points corresponding to each compaction scheme, we obtain the read-write trade-off curve for that scheme in practical LSM-tree implementations.

The result shows that our proposed horizontal-tiering scheme significantly extends the trade-off curve of the original horizontal scheme, thereby completely enveloping that of both vertical-partial and vertical-full designs. The enhanced horizontal scheme's trade-off dominates the vertical scheme's, with the horizontal scheme's curve consistently closer to the origin, indicating lower overall costs. This aligns with our earlier theoretical analysis, as the compaction sequence under the horizontal scheme is always optimal. The trade-off curve presented for the horizontal scheme effectively represents the Pareto frontier for LSM-trees in terms of compaction schemes.

Vertiorizon enhances existing designs as a fundamental infrastructure. We embedded Vertiorizon into various lazy-leveling designs and compared their performance with the original configurations. Each lazy-leveling design employed different size ratios T, specifically scaling across the values 4, 6, 8, and 10. We evaluated these designs using a balanced workload comprising 4 million operations, with an equal split of 50% updates and 50% point lookups. Figure 10 (b) and (d) present the total latency for each lazy-leveling design, as well as the individual read and write latencies under 32MB and 64GB block cache configurations, respectively. In this figure, the upper portion of each bar represents the read latency, while the lower portion corresponds to the write latency. The results demonstrate that embedding Vertiorizon into any lazy-leveling design

[4]The figure shows that all methods achieve relatively low latencies in both lookups and updates, on the order of hundreds of microseconds. This is primarily attributed to the inherent efficiency of the LSM-tree data structure and the high data transfer bandwidth of the SSDs used in our experimental setup. Notably, previous studies [15, 28] have also reported that, under similar experimental conditions, RocksDB demonstrates operation latencies within the hundred-microsecond range.

consistently improves performance. This enhancement is primarily reflected in the reduction of read latency. The improvement arises because our embedding replaces the original tiering levels with the horizontal part of Vertiorizon utilizing the horizontal-tiering scheme, all while maintaining the same number of levels. This approach effectively reduces the average read amplification during runtime without increasing the write amplification. Furthermore, we also tested lazy-leveling designs with adapted filter layout [14]. As shown in Figure 10 (c) and (e), the results indicate that after incorporating Bloom filter optimizations, embedding Vertiorizon as a backbone continues to enhance the performance of lazy-leveling designs.

8 Conclusion

In this paper, we revisit two established growth schemes for LSM-trees: the horizontal scheme and the vertical scheme. Through a comprehensive analysis of their respective strengths and limitations, we revoke the recognition of the horizontal scheme and present an enhanced horizontal scheme by extending its applicability to accommodate the tiering merge policy. Building on this foundation, we propose Vertiorizon, a self-designing growth scheme that leverages the individual strengths of both horizontal and vertical schemes. We demonstrate the practical benefits of the enhanced horizontal scheme and Vertiorizon by embedding these schemes into RocksDB and conducting extensive experiments. Notably, Vertiorizon achieves robust performance across a wide range of scenarios and metrics. These results highlight the potential of Vertiorizon to serve as a foundational infrastructure for improving general LSM-tree-based systems.

Acknowledgments

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

References

[1] 2024. Technical Report. https://sites.google.com/view/lsm-scheme-technical-report.
[2] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. 2018. Lightweight cardinality estimation in LSM-based systems. In *Proceedings of the 2018 International Conference on Management of Data*. 841–855.
[3] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.
[4] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. AsterixDB: A scalable, open source BDMS. *arXiv preprint arXiv:1407.0454* (2014).
[5] 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.
[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.
[7] Jon Louis Bentley and James B Saxe. 1980. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms* 1, 4 (1980), 301–358.
[8] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1863–1875.
[9] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.
[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] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.

[12] Source Code. 2024. BadgerDB. https://github.com/dgraph-io/badger.

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

[14] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

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

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

[17] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

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

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

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

[21] Facebook. 2024. RocksDB tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[22] Facebook. 2024. Universal Compaction in RocksDB. https://github.com/facebook/rocksdb/wiki/universal-compaction.

[23] Lars George. 2011. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.".

[24] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–14.

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

[26] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[27] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[28] Andy Huynh, Harshal Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *arXiv preprint arXiv:2110.13801* (2021).

[29] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Slavin Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

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

[31] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, et al. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151.

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

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

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

[35] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of {Multi-Stage} Log-structured Designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 149–166.

[36] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *PACMMOD* 2, 3 (2024), 1–26.

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

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

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

[40] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[41] Qizhong Mao, Steven Jacobs, Waleed Amjad, Vagelis Hristidis, Vassilis J Tsotras, and Neal E Young. 2019. Experimental evaluation of bounded-depth LSM merge policies. In *2019 IEEE International Conf. on Big Data*. IEEE, 523–532.
[42] Qizhong Mao, Steven Jacobs, Waleed Amjad, Vagelis Hristidis, Vassilis J Tsotras, and Neal E Young. 2021. Comparison and evaluation of state-of-the-art LSM merge policies. *The VLDB Journal* 30 (2021), 361–378.
[43] Qizhong Mao, Mohiuddin Abdul Qader, and Vagelis Hristidis. 2023. Comparison of LSM indexing techniques for storing spatial data. *Journal of Big Data* 10, 1 (2023), 1–26.
[44] Claire Mathieu, Rajmohan Rajaraman, Neal E Young, and Arman Yousefi. 2021. Competitive data-structure dynamization. In *Proceedings of the 2021 ACM-SIAM Symp. on Discrete Algorithms (SODA)*. SIAM, 2269–2287.
[45] Claire Mathieu, Carl Staelin, Neal E Young, and Arman Yousefi. 2014. Bigtable merge compaction. *arXiv preprint arXiv:1407.3008* (2014).
[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[47] PingCAP. 2024. TiDB. https://github.com/pingcap/tidb.
[48] 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 Symp. on Operating Systems Principles*. 497–514.
[49] 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.
[50] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.
[51] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.
[52] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building Workload-Independent Storage with VT-Trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 17–30.
[53] Cloudius Systems. 2024. ScyllaDB. https://www.scylladb.com/.
[54] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. Noftl-kv: Tackling write-amplification on kv-stores with native storage management. In *Advances in database technology-EDBT 2018: 21st International Conference on Extending Database Technology, Vienna, Austria, March 26-29, 2018. proceedings*. University of Konstanz, University Library, 457–460.
[55] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *PACMMOD* 2, 3 (2024), 1–27.
[56] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.
[57] 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.
[58] 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.
[59] Yu, Geoffrey X and Markakis, Markos and Kipf, Andreas and Larson, Per-Åke and Minhas, Umar Farooq and Kraska, Tim. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.
[60] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.
[61] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 225–237.
[62] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. 2018. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.
[63] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing bloom filter cpu overhead in lsm-trees on modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. 1–10.

Received October 2024; revised January 2025; accepted February 2025