Breaking Down Memory Walls: Adaptive Memory Management in LSM-based Storage Systems

Chen Luo
University of California, Irvine
cluo8@uci.edu

Michael J. Carey
University of California, Irvine
mjcarey@ics.uci.edu

ABSTRACT

Log-Structured Merge-trees (LSM-trees) have been widely used in modern NoSQL systems. Due to their out-of-place update design, LSM-trees have introduced memory walls among the memory components of multiple LSM-trees and between the write memory and the buffer cache. Optimal memory allocation among these regions is non-trivial because it is highly workload-dependent. Existing LSM-tree implementations instead adopt static memory allocation schemes due to their simplicity and robustness, sacrificing performance. In this paper, we attempt to break down these memory walls in LSM-based storage systems. We first present a memory management architecture that enables adaptive memory management. We then present a partitioned memory component structure with new flush policies to better exploit the write memory to minimize the write cost. To break down the memory wall between the write memory and the buffer cache, we further introduce a memory tuner that tunes the memory allocation between these two regions. We have conducted extensive experiments in the context of Apache AsterixDB using the YCSB and TPC-C benchmarks and we present the results here.

PVLDB Reference Format:
Chen Luo and Michael J. Carey. Breaking Down Memory Walls: Adaptive Memory Management in LSM-based Storage Systems. PVLDB, 14(3): 241-254, 2021.
doi:10.14778/3430915.3430916

1 INTRODUCTION

Log-Structured Merge-trees (LSM-trees) [47] are widely used in modern NoSQL systems, such as LevelDB [4], RocksDB [6], Cassandra [2], HBase [3], X-Engine [30], and AsterixDB [1]. Unlike traditional in-place update structures, LSM-trees adopt an out-of-place update design by first buffering all writes in memory; they are subsequently flushed to disk to form immutable disk components. The disk components are periodically merged to improve query performance and reclaim space occupied by obsolete records.

Efficient memory management is critical for storage systems to achieve optimal performance. Compared to update-in-place systems where all pages are managed within shared buffer pools, LSM-trees have introduced additional memory walls[1]. Due to the LSM-tree's out-of-place update nature, the write memory is isolated from the buffer cache. Moreover, data management systems adopting LSM storage engines, such as MyRocks [5] on RocksDB [6], PolarDB [18] on X-Engine [30], and AsterixDB [9], must deal with multiple heterogeneous LSM-trees from multiple datasets and indexes. This requires the write memory to be efficiently shared among multiple LSM-trees. Since the optimal memory allocation heavily depends on the workload, memory management should be workload-adaptive to maximize the system performance.

Unfortunately, adaptivity is non-trivial, as it is highly workload-dependent. Existing LSM-tree implementations, such as RocksDB [6] and AsterixDB [32], have opted for simplicity and robustness over optimal performance by adopting static memory allocation schemes. For example, RocksDB sets a static size limit (default 64MB) for each memory component. AsterixDB specifies the maximum number N of writable datasets (default 8) so that each active dataset, including its primary and secondary indexes, receives 1/N of the total write memory. Both systems allocate separate static budgets for the write memory and the buffer cache. Despite their simplicity and robustness, static memory allocation schemes may negatively impact the system performance and efficiency due to sub-optimal memory allocation.

Our Contributions. In this paper, we seek to break down these memory walls in LSM-based storage systems to enable adaptive memory memory management and maximize performance and efficiency. As the first contribution, we present an adaptive memory management architecture for LSM-based storage systems. In this architecture, the overall memory budget is divided into the write memory region and the buffer cache region. Within the write memory region, the memory allocation of each memory component is purely driven by its demands, i.e., write rates, to minimize the overall write amplification. The two regions are connected via a *memory tuner* that adaptively tunes the memory allocation between the write memory and the buffer cache.

As the second contribution, we present a series of techniques for efficiently managing the write memory for LSM-trees in order to minimize their write I/O cost. We first present a new LSM memory component structure for managing the write memory for a single LSM-tree, and then propose novel flush policies for managing the write memory for multiple LSM-trees.

Our third contribution is the detailed design and implementation of a memory tuner that adaptively tunes the memory allocation between the write memory and the buffer cache to reduce the system's overall I/O cost. The memory tuner performs on-line tuning by modeling the I/O cost of LSM-trees without any a priori knowledge of the workload. This further allows the memory tuner to quickly adjust the memory allocation when the workload changes.

Proceedings of the VLDB Endowment, Vol. 14, No. 3 ISSN 2150-8097.
doi:10.14778/3430915.3430916

[1]In this paper, the term *memory wall* refers to the barriers among various memory regions that prevent efficient memory sharing.

As the last contribution, we have implemented all of the proposed techniques inside Apache AsterixDB [1]. We have carried out extensive experiments on both the YCSB benchmark [22] and the TPC-C benchmark [7] to evaluate the effectiveness of the proposed techniques. The experimental results show that the proposed techniques successfully reduce the disk I/O cost, which in turn maximizes system efficiency and overall performance.

The remainder of this paper is organized as follows. Section 2 discusses background information and related work. Section 3 presents our adaptive memory management architecture for LSM-trees. Section 4 describes the new memory component structure for managing the write memory. Section 5 presents the design and implementation of the memory tuner. Section 6 experimentally evaluates the proposed techniques. Finally, Section 7 concludes the paper.

2 BACKGROUND

2.1 Log-Structured Merge Trees

The LSM-tree [47] is a persistent index structure optimized for write-intensive workloads. LSM-trees perform out-of-place updates by always buffering writes into a memory component and appending log records to a transaction log for durability. Writes are flushed to disk when either the memory component is full, called a *memory-triggered flush*, or when the transaction log length becomes too long, called a *log-triggered flush*.

A query over an LSM-tree has to reconcile the entries with identical keys from multiple components, as entries from newer components override those from older components. A range query searches all components simultaneously using a priority queue to perform reconciliation. A point lookup query simply searches all components from newest to oldest until the first match is found. To speed up point lookups, a common optimization is to build Bloom filters [15] over the sets of keys stored in disk components.

To improve query performance and space utilization, disk components are periodically merged according a pre-defined merge policy. In practice, two types of merge policies are commonly used [42], both of which organize disk components into "levels". The leveling merge policy maintains one component per level. When a component at Level i is T times larger than that of Level $i-1$, it will be merged into Level $i+1$ to form a new component. In contrast, the tiering merge policy maintains T components per level. When a Level i becomes full with T components, they are merged together into a new component at Level $i+1$.

Partitioning. In practice, a common optimization is to range-partition a disk component into multiple (often fixed-size) SSTables[2] to bound the processing time and temporary space of each merge. This optimization is often used together with the leveling merge policy, as pioneered by LevelDB [4]. An example of a partitioned LSM-tree with the leveling merge policy is shown in Figure 1, where each SSTable is labeled with its key range. Note that L_0 is not partitioned since its SSTables are directly flushed from memory. L_0 also stores multiple SSTables with overlapping key ranges to absorb write bursts. To merge an SSTable from L_i to L_{i+1}, all of its overlapping SSTables at L_{i+1} are selected and these SSTables are merged to form new SSTables at L_{i+1}. For example in Figure 1, the

[2] An SSTable (Sorted String Table) [20] stores a set of immutable rows sorted on keys.

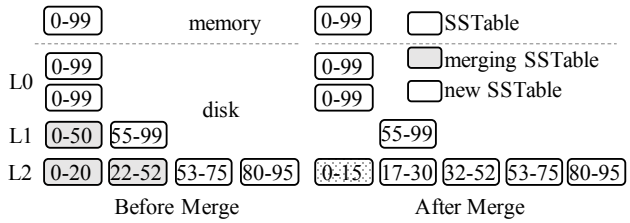

Figure 1: Example Partitioned LSM-tree

Table 1: LSM-tree Notation

Notation	Definition	Example
Global Notation		
T	size ratio of the merge policy	10
P	disk page size	4 KB/page
M_w	total write memory size	1GB
Local Notation		
e_i	entry size	100 B/entry
a_i	ratio of an LSM-tree's write memory to total write memory	20%
N_i	number of levels (excluding L_0)	3
$\|L_{l_i}\|$	size of Level L_{l_i}	10 GB
C_i	write I/O cost per entry	4 pages/entry

SSTable labeled 0-50 at L_1 will be merged with the SSTables labeled 0-20 and 22-52 at L_2, which produce new SSTables labeled 0-15, 17-30, and 32-52 at L_2. When the LSM-tree becomes too large, a new level must be added. To maximize space utilization, the new level should be added at L_1 instead of the last level [27]. The last level is always treated as full, which in turn determines the maximum sizes of other levels. When the maximum size of L_1 is larger than T times the write memory size (or the configured base level size), a new L_1 is added while all remaining levels L_i become L_{i+1}. In our work, we will focus on the partitioned leveling structure due to its wide adoption in today's LSM-based systems.

Write Memory vs. Write Cost. Here we provide a simple cost analysis to show the relationship between the write memory size and the per-entry write I/O cost. Our notation is shown in Table 1. Note that since we consider multiple LSM-trees, Table 1 contains global notation that is valid for all LSM-trees and local notation that is specific to one LSM-tree. In the remainder of this paper, for the i-th LSM-tree, we add the subscript i to denote the local notation for this LSM-tree. Note that we have further introduced the notation a to denote the write memory ratio of an LSM-tree. Thus, for the i-th LSM-tree, its write memory size is $a_i \cdot M_w$. Moreover, given a collection of K LSM-trees, we have $\sum_{i=1}^{K} a_i = 1$.

Each entry written to an LSM-tree is flushed to disk once and merged multiple times down to the last level. The per-entry flush cost is $\frac{e}{P}$ pages/entry. Merging an SSTable at Level L_i usually has T overlapping SSTables at Level L_{i+1}. Thus, to merge an entry from L_0 to the last level, the overall merge cost is $\frac{e}{P} \cdot (T+1) \cdot N$ pages/entry. Here the number of levels N can be expressed using other terms as follows. Given an LSM-tree whose write memory size is $a \cdot M_w$, the maximum size of i-th level is $a \cdot M_w \cdot T^i$. Based on the size

of the last Level $|L_N|$, we have $|L_N| \leq a \cdot M_w \cdot T^N$. Thus, N can be approximated as $\log_T \frac{|L_N|}{a \cdot M_w}$. Putting everything together, the per-entry write cost C is approximately

$$C = \frac{e}{P} + \frac{e}{P} \cdot (T+1) \cdot \log_T \frac{|L_N|}{a \cdot M_w} \quad (1)$$

As Equation 1 shows, a larger write memory reduces the write cost by reducing the number of disk levels. Thus, it is important to utilize a large write memory efficiently to reduce the write cost.

2.2 Apache AsterixDB

Apache AsterixDB [1, 9, 19] is a parallel, semi-structured Big Data Management System (BDMS) for efficiently managing large amounts of data. It supports a feed-based framework for efficient data ingestion [29, 59]. The records of a dataset in AsterixDB are hash-partitioned based on their primary keys across multiple nodes of a shared-nothing cluster [10]. Each partition of a dataset uses a primary LSM-tree to store the data records with each component begin organized as a B^+-tree. Local secondary indexes, including LSM-based B^+-trees, R-trees, and inverted indexes, can also be built to expedite query processing.

AsterixDB uses a static memory allocation scheme for simplicity and robustness [32]. It specifies static memory budgets for the buffer cache and the write memory. Moreover, AsterixDB specifies the maximum number D of writable datasets (default 8) so that each active dataset receives 1/D of the total write memory. When a dataset's write memory is full, all of its LSM-trees, including its primary index and secondary indexes, will be flushed to disk together. If the user writes to the D+1-st dataset, the least recently written active dataset will be evicted to reclaim its write memory. In this work, we use AsterixDB as a testbed to evaluate the proposed techniques and compare them to other baselines.

2.3 Related Work

LSM-trees. Recently, a large number of improvements have been proposed to optimize the original [47] LSM-tree design. These improvements include optimizing write performance [11, 14, 25, 26, 33, 37, 45, 46, 50, 62], supporting auto-tuning of LSM-trees [23, 24, 35, 53], optimizing LSM-based secondary indexes and filters [39, 44, 49], minimizing write stalls [12, 40, 54], and extending the applicability of LSM-trees [43, 51]. We refer readers to a recent survey [42] for a more detailed description of these LSM-tree improvements.

In terms of memory management, FloDB [13] presents a two-level memory component structure to mask write latencies. However, it mainly optimizes for peak in-memory throughput instead of reducing the overall write cost. Accordion [16] introduces a multi-level memory component structure with memory flushes and merges. One drawback is that Accordion does not range-partition memory components, resulting in high memory utilization during large memory merges. We will further experimentally evaluate Accordion in Section 6. Monkey [23] uses analytical models to tune the memory allocation between memory components and Bloom filters. ElasticBF [34] proposes a dynamic Bloom filter management scheme to adjust false positives rates based on the data hotness. Different from Monkey and ElasticBF, in our work Bloom filters are managed the same paged way as SSTables through the buffer cache. It should also be noted that virtually all previous research only considers the memory management of a single LSM-tree.

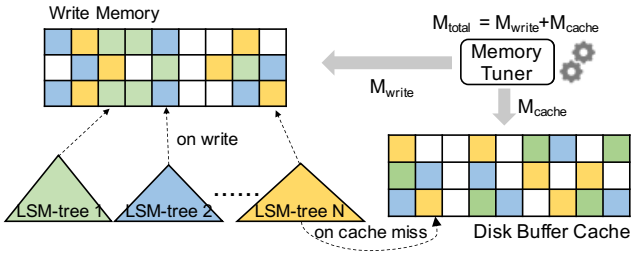

Figure 2: Memory Management Architecture

Database Memory Management. The importance of memory management, or buffer management, has long been recognized for database systems. Various buffer replacement policies, such as DBMIN [21], 2Q [31], LRU-K [48], and Hot-Set [52], have been proposed to reduce buffer cache misses. These replacement policies are orthogonal to this work because we mainly focus on the memory walls introduced by the LSM-tree's out-of-place update design.

Automatic memory tuning is also an important problem for database systems. Some commercial DBMSs have supported auto-tuning the memory allocation among different memory regions [8, 55]. Depending on the tuning goals, the memory tuning techniques can be classified as maximizing the overall throughput or meeting latency requirements. DB2's self-tuning memory manager (STMM) [55] is an example of the former, using control theory to tune the memory allocation. However, STMM targets targets a traditional in-place update system, which does not include the write memory used by LSM-trees. For the latter, the relationship between the buffer cache size and the cache miss rate must be predicted, using either analytical models [57] or machine learning approaches [56].

There has been recent interest in exploiting machine learning to tune database configurations [28, 36, 58, 61], where memory allocation is treated as one tuning knob. These approaches usually require additional training steps and user inputs. Different from these approaches, our memory tuner uses a white-box approach; it carefully models the I/O cost of LSM-based storage systems.

3 MEMORY MANAGEMENT ARCHITECTURE

In this section, we present our memory management architecture to enable adaptive memory management. In this architecture, depicted in Figure 2, the total memory budget is divided into the write memory M_{write} and the buffer cache M_{cache}. These two regions are further connected via a memory tuner, which periodically performs memory tuning to reduce the total I/O cost.

Write Memory. The write memory stores incoming writes for all LSM-trees. To maximize memory utilization, we do not set static size limits for the individual memory components. Instead, all memory components are managed through a shared memory pool. When an LSM-tree has insufficient memory to store its incoming writes, more pages will be requested from the pool. When the overall write memory usage is too high, an LSM-tree is selected to have its memory component flushed to disk.

While the basic idea of this design is straightforward, there are several technical challenges here. First, how can we best utilize

the write memory to minimize the write cost? Second, since the memory component of an LSM-tree now becomes dynamic, how can we adjust the disk levels as the write memory changes to always make optimal performance trade-offs? Finally, given a collection of heterogeneous LSM-trees of different sizes, how can we allocate the write memory to these LSM-trees to minimize the overall write cost? We will present our solutions to these challenges in Section 4.

Buffer Cache. The buffer cache stores (immutable) disk pages of the SSTables as well as their Bloom filters for all LSM-trees. Even though all LSM-trees share the same buffer cache, their merges are performed separately. As in traditional database systems, all disk pages are managed together using a predefined buffer replacement policy. For example, AsterixDB uses the clock replacement policy to manage its shared buffer cache. In this work, we mainly focus on the memory allocation given to the buffer cache instead of cache replacement within the buffer cache.

Memory Tuner. Given a memory budget, the memory tuner attempts to tune the memory allocation between the write memory and the buffer cache to reduce the total I/O cost. The key property of the memory tuner is that it takes a white-box approach by carefully modeling the I/O cost of LSM-based storage systems and thus does not require any offline training. We will describe the design and implementation of the memory tuner in Section 5.

4 MANAGING WRITE MEMORY

Now we present our solution for managing the write memory. We first describe the memory component structure of a single LSM-tree and then present techniques for managing multiple LSM-trees.

4.1 Partitioned Memory Component

4.1.1 Basic Design. With the new memory management architecture, a memory component can become very large since its size is not limited. Existing LSM-tree implementations use skiplists or B$^+$-trees to manage memory components and always flush a memory component entirely to disk. However, this reduces memory utilization for two reasons. First, an updatable B$^+$-tree has internal fragmentation, as its pages are about 2/3 full [60]. Second, after a flush a large chunk of write memory will be freed (vacated) all at once, which reduces the average memory utilization over time. [3]

To maximize the memory utilization, we propose to use a partitioned in-memory LSM-tree to manage the write memory, which is called a *partitioned memory component*. An example LSM-tree with this structure is shown in Figure 3, which has an active SSTable at M_0 for storing incoming writes and a set of memory levels for storing immutable SSTables. When a memory level M_i is full, one of its SSTables is merged into the next level M_{i+1} using a *memory merge*. We use a greedy selection policy to select SSTables to merge by minimizing the ratio between the size of the overlapping SSTables at M_{i+1} and the size of the selected SSTable at M_i, i.e., the overlapping ratio. Memory SSTables must be eventually flushed to disk. For a memory-triggered flush, SSTables at the last memory level (M_2 in Figure 3) are flushed to disk in a round-robin way. For a log-triggered flush, the SSTable with the minimum log sequence number (LSN) will be flushed (as well as all overlapping SSTables at higher levels) to facilitate log truncation.

[3] To see this, consider a single LSM-tree with a large memory component. If its memory component is flushed entirely, the average memory utilization over time will be less than 50%.

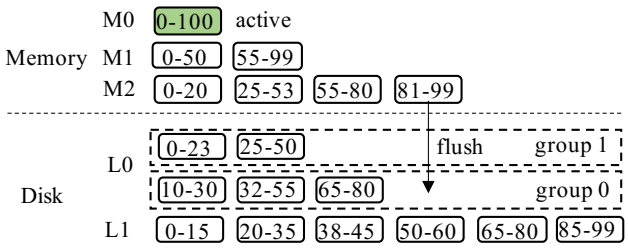

Figure 3: LSM-tree with a Partitioned Memory Component

Compared to the monolithic memory component structure used in existing systems, the proposed structure increases the memory utilization and reduces the write amplification in several ways. First, an LSM-tree achieves much higher space utilization than B$^+$-trees. For example, with a size ratio of 10, an LSM-tree achieves 90% space utilization, which is much higher than that of a B$^+$-tree. Moreover, since the proposed structure is range-partitioned, it can naturally flush one memory SSTable at a time using partial flushes so that the write memory stays full. Finally, partial flushes further reduce the write amplification by creating skews at the last level [35]. Since SSTables are flushed in a round-robin way, the flushed SSTable will have received the most updates. Thus, the key ranges of these SSTables will be denser than the average, which reduces the subsequent merge cost. In the remainder of this section, we further discuss the detailed design of the proposed structure.

4.1.2 Grouped L_0. In the original LSM-tree design (Figure 1), the disk level L_0 stores a list of (unpartitioned) SSTables ordered by their recency. When the number of SSTables at L_0 exceeds a pre-defined threshold, flushes will be paused to bound the worse-case query performance. Multiple L_0 SSTables are also merged together into L_1 to reduce the merge cost. In the partitioned memory component structure, where the flushed SSTables are range partitioned, the original L_0 structure is unsuitable since non-overlapping SSTables have no negative impact on queries. Thus, flushes should only be paused when there are too many overlapping SSTables.

To better accommodate the new memory component structure, we propose a new L_0 structure by organizing its SSTables into groups, where each group contains a set of disjoint SSTables. Groups are ordered based on their recency, where the keys in a newer group override the keys in an older group. When the total number of groups at L_0 exceeds a predefined threshold, incoming flushes will be stopped. We further use the following heuristics to reduce the number of groups at L_0 and also the write amplification. First, when an SSTable is flushed to disk, it is always inserted into the oldest possible group where all newer groups do not have any overlapping SSTables. Otherwise, if no such group can be found, a new group is created. Second, SSTables from the smallest group that contains the fewest SSTables will be merged into L_1 first. Specifically, an SSTable from this group as well as any overlapping SSTables from other L_0 groups are merged with the overlapping SSTables at L_1. To reduce write amplification, the merging SSTable is selected to minimize

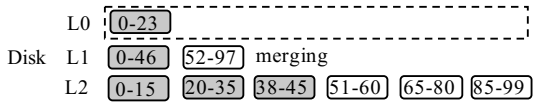

Figure 4: Example Merge for Removing L_1

the ratio between the total size of the overlapping SSTables at L_1 and the total size of the merging SSTables at L_0.

4.1.3 Adjusting Disk Levels. In the new memory component architecture, the write memory of each LSM-tree is allocated on-demand and is thus dynamic. Since the write cost of an LSM-tree depends on the number of disk levels, the number of disk levels needs to be adjusted as its write memory size changes[4].

Recall that to maximize the space utilization, levels are only added or deleted at L_1. For each disk level L_i, its maximum size is $a \cdot M_w \cdot T^i$. Here we assume that the size of each disk level $|L_i|$ is relatively stable, but the write memory allocated to an LSM-tree $a \cdot M_w$ is dynamic. When an LSM-tree's write memory size becomes too small, i.e., $a \cdot M_w \cdot T < |L_1|$, a new L_1 should be added to reduce the write cost. In this case, an empty L_1 can be added and all remaining levels L_i simply become L_{i+1}. In contrast, when the write memory size becomes too big, i.e., $a \cdot M_w \cdot T > |L_2|$, L_1 becomes redundant and can be deleted. However, implementing this strategy directly can cause oscillation when the write memory is close to this threshold. To avoid this, the deletion of L_1 can be delayed until the write memory further grows by a factor of f, i.e., $a \cdot M_w \cdot T > f \cdot |L_2|$. As we will see in Section 6, delaying the deletion of L_1 has a much smaller impact than delaying the addition of a level. In general, a larger f better avoids oscillation but may have a larger negative impact on write amplification. By default, we set f to 1.5 to balance these two factors.

To delete L_1, all existing SSTables from L_1 must be merged into L_2. Here we describe an efficient solution to delete L_1 smoothly with minimal overhead. To delete L_1, SSTables from L_0 can be directly merged into L_2 along with all overlapping SSTables at L_1. Consider the example in Figure 4. To delete L_1, the SSTable labeled 0-23 at L_0 and the SSTable labeled 0-46 at L_1 can be directly merged into L_2. This mechanism ensures that L_1 will not receive new SSTables but does not itself guarantee that L_1 will eventually become empty. To address this problem, low-priority merges are also scheduled to merge SSTables from L_1 into L_2 when there are no merges at other levels. These two operations ensure that L_1 will eventually become empty, and it can then be removed from the LSM-tree.

4.1.4 Partial Flush vs. Full Flush. As mentioned before, the new memory component structure enables partial flushes, i.e., flushing one SSTable at a time. While possible, partial flushes may not always be an optimal choice. Consider the case when the total write memory is large and flushes are only triggered by log truncation. Since the oldest entries can be distributed across all memory SSTables, most memory SSTables may have to be flushed in order to truncate the log. If a *full flush* is performed, which will merge-sort all memory SSTables across all levels, the flushed SSTables will have non-overlapping key ranges. In contrast, if partial flushes are used, the flushed SSTables may have overlapping key ranges, which will require subsequent merges to make these SSTables fully sorted and thus incur extra merge I/O cost. Thus, for a log-triggered flush, the optimal flush choice depends on the write memory size and the maximum transaction log length.

Developing an optimal flush solution is non-trivial since it also heavily depends on the key distribution of the write workload. Here we propose a simple heuristic to dynamically switch between partial and full flushes for log-triggered flushes. The basic idea is to use a window to keep track of how much write memory has been partially flushed before the log-triggered flush, where the window size is set as the maximum transaction log length. When log truncation is needed, if a large amount of write memory has already been flushed before, then only a small number of remaining SSTables will need to be flushed and thus partial flushes will be a better choice. Otherwise, full flushes should be performed. Implementation-wise, we introduce a threshold parameter β ranging from 0 to 1. When the total amount of previously flushed write memory is larger than β times the total write memory, partial flushes will be performed. Otherwise, the entire memory component will be flushed together using a full flush. Based on some preliminary simulation results, we set our default value for β to be 0.5 to minimize the overall write cost. (We leave the further exploration of the optimal choice of partial and full flushes as future work.)

4.2 Managing Multiple LSM-trees

When managing multiple LSM-trees, a fundamental question is how to allocate portions of the write memory to these LSM-trees. Since write memory is allocated on-demand, this question becomes how to select LSM-trees to flush. For log-triggered flushes, the LSM-tree with the minimum LSN should be flushed to perform log truncation. For memory-triggered flushes, existing LSM-tree implementations, such as RocksDB [6] and HBase [3], choose to flush the LSM-tree with the largest memory component. We call this policy the *max-memory* flush policy. The intuition is that flushing this LSM-tree can reclaim the most write memory, which can be used for subsequent writes. However, this policy may not be suitable for our partitioned memory components because flushing any LSM-tree will reclaim the same amount of write memory due to partial SSTable flushes.

Min-LSN Policy. One alternative flush policy is to always flush the LSM-tree with the minimum LSN for both log-triggered and memory-triggered flushes. We call this policy the *min-LSN* flush policy. The intuition is that the flush rate of an LSM-tree should be approximately proportional to its write rate. A hotter LSM-tree should be flushed more often than a colder one, but it still receives more write memory. This policy also facilitates log truncation, which can be beneficial if flushes are dominated by log truncation.

Optimal Policy. Given a collection of K LSM-trees, our ultimate goal is to find an optimal memory allocation that minimizes the overall write cost. For the i-th LSM-tree, we denote r_i as its the write rate (bytes/s). The optimal memory allocation can be obtained by solving the following optimization problem:

$$\min_{a_i} \sum_{i=1}^{K} \frac{r_i}{e_i} \cdot C_i, \text{ s.t. } \sum_{i=1}^{K} a_i = 1 \quad (2)$$

[4]Our preliminary solution [38] was to only increase the number of disk on-levels without ever decreasing it. However, our subsequent evaluation showed that this led to 5%-10% performance loss compared to an optimal LSM-tree.

By substituting Equation 1 from Section 2.1 into Equation 2 and using the Lagrange multiplier method, the optimal write memory ratio a_i^{opt} for the i-th LSM-tree is $a_i^{opt} = r_i / \sum_{j=1}^{K} r_j$. This shows that the write memory allocated to each LSM-tree should be proportional to its write rate. We call this policy the *optimal* flush policy. In terms of its implementation, we can use a window to keep track of the total number of writes to each LSM-tree, where the window size is set as the maximum transaction log length. When a memory-triggered flush is requested, each active LSM-tree is checked in turn and a flush is scheduled if its write memory ratio a_i is larger than its optimal write memory ratio a_i^{opt}.

5 MEMORY TUNER

After discussing how to efficiently manage the write memory, we now proceed to describe the memory tuner to tune the memory allocation between the write memory and the buffer cache. We first provide an overview of the tuning approach, which is followed by its design and implementation.

5.1 Tuning Approach

The goal of the memory tuner is to find an optimal memory allocation between the write memory and the buffer cache to minimize the I/O cost per operation. This should in turn maximize the system efficiency as well as the overall throughput. Suppose the total available memory is M. For ease of discussion, let us assume the write memory size is x, which implies that the buffer cache size is $M - x$. Let $write(x)$ and $read(x)$ be the write cost and read cost per operation when the write memory is x. Our tuning goal is to minimize the weighted I/O cost per operation (pages/op) $cost(x) = \omega \cdot write(x) + \gamma \cdot read(x)$. The non-negative weights ω and γ allow users to adjust the tuning goal for different use cases. For example, on hard disks, ω can be set smaller since LSM-trees mainly use sequential I/Os for writes, while on SSDs ω can be set larger since SSD writes are often more expensive than SSD reads.

In order to minimize the tuning goal $cost(x)$, we use an online gradient descent approach to adaptively tune the memory allocation based on $cost'(x)$, which is $\omega \cdot write'(x) + \gamma \cdot read'(x)$. Intuitively, $cost'(x)$ measures how the I/O cost changes if more write memory is allocated. Based on $cost'(x)$, the memory tuner can tune the memory allocation accordingly to reduce $cost(x)$. It should be noted that the optimality of the memory tuner depends on the shape of $cost(x)$. We will discuss this issue further in Section 5.5.

Based on this idea, the memory tuner uses a feedback-control loop to tune memory allocation, as depicted in Figure 5. The memory tuner continuously uses the collected statistics to tune the memory allocation between the write memory and the buffer cache without any user input nor training samples. Before describing the details of the memory tuner, we first introduce some notation used

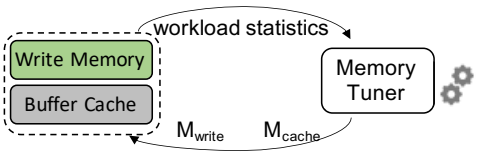

Figure 5: Workflow of Memory Tuner

Table 2: Memory Tuner Notation

Notation	Definition	Example
Global Notation		
K	number of LSM-trees	8
op	number of operations observed	10K ops
$saved_q$	saved query disk I/O by the simulated cache	0.01 page/op
$saved_m$	saved merge disk I/O by the simulated cache	0.002 page/op
sim	simulated cache size	32 MB
Local Notation		
w_i	number of entries written to an LSM-tree	50K entries
$flush_{log_i}$	write memory flushed by log truncation	1 GB
$flush_{mem_i}$	write memory flushed by high memory usage	8 GB

by the memory tuner (Table 2) in addition to the LSM-tree notation listed in Table 1. Note that with secondary indexes each operation may write multiple entries to multiple LSM-trees.

5.2 Estimating the Write Cost Derivative

For the i-th LSM-tree, recall that Equation 1 computes the per-entry write cost C_i. Since each operation writes $\frac{w_i}{op}$ entries to this LSM-tree, its write cost per operation $write_i(x)$ can computed as $\frac{w_i}{op} \cdot C_i$. By taking the derivative of $write_i(x)$, we have

$$write_i'(x) = \frac{w_i}{op} \cdot \frac{e_i}{P} \cdot (T+1) \cdot \frac{1}{x \cdot \ln T} \tag{3}$$

To reduce the estimation error, instead of collecting statistics for op, w_i, e_i and P, we simply collect the total number of merge writes per operation, $merge_i(x)$, in the last tuning cycle. By substituting $merge_i(x)$ into Equation 3, we have

$$write_i'(x) = -\frac{merge_i(x)}{x \cdot \ln \frac{|L_{N_i}|}{a_i \cdot x}} \tag{4}$$

Here we assume that the write memory of an LSM-tree is always smaller than its last level size. Thus, the estimated value of $write_i'(x)$ in Equation 4 is always negative as long as $merge_i(x)$ is not zero. This implies that adding more write memory can always reduce the write cost, which may not hold in practice. Once flushes are dominated by log truncation, adding more write memory will not further reduce the write cost. To account for the impact of log-triggered flushes, we further multiply Equation 4 by a scale factor $\frac{flush_{mem_i}}{flush_{mem_i} + flush_{log_i}}$ that we also keep statistics for. Intuitively, this scale factor will be close to 1 if flushes are mainly triggered by high memory usage and it will approach to 0 if flushes are mostly triggered by log truncation. Finally, $write'(x)$ is the sum of $write_i'(x)$ for all LSM-trees:

$$write'(x) = \sum_{i=1}^{K} -\frac{merge_i(x)}{x \cdot \ln \frac{|L_{N_i}|}{a_i \cdot x}} \cdot \frac{flush_{mem_i}}{flush_{mem_i} + flush_{log_i}} \tag{5}$$

5.3 Estimating the Read Cost Derivative

$read'(x)$ measures how the read cost per operation changes if more write memory is allocated. Since disk reads are performed by both queries and merges, we rewrite $read(x) = read_q(x) + read_m(x)$, where $read_q(x)$ is the number of query disk reads per operation and $read_m(x)$ is the number of merge disk reads per operation.

$read'_q(x)$ measures the impact of larger write memory on the query read cost, as larger write memory increases the buffer cache miss rate. To estimate $read'_q(x)$, we use a simulated cache as suggested by [55]. This simulated cache only stores page IDs. Whenever a page is evicted from the buffer cache, its page ID is added to the simulated cache. Whenever a page is about to be read from disk, a disk I/O could have been saved if the simulated cache contains that page ID. Suppose that the simulated cache size is sim and the saved read cost per operation is $saved_q$, then $read'_q(x) = \frac{saved_q}{sim}$.

$read'_m(x)$ measures the impact of larger write memory on the merge read cost. Intuitively, larger write memory reduces the disk merge cost, but also increases the buffer cache miss rate. Thus, to estimate $read'_m(x)$, we first rewrite $read_m(x) = pin_m(x) \cdot miss_m(x)$. $pin_m(x)$ is the number of page pins performed by disk merges per operation and it can be obtained by collecting runtime statistics. $miss_m(x)$ is the cache miss rate for merges and it can be computed as $misss_m(x) = \frac{read_m(x)}{pin_m(x)}$. Based on the derivative rule, we have $read'_m(x) = pin'_m(x) \cdot miss_m(x) + pin_m(x) \cdot miss'_m(x)$. $pin'_m(x)$ is the number of saved merge page pins per unit of write memory. Recall that we have computed $write'(x)$, which is the number of saved disk writes per unit of write memory. On average, each merge disk write requires $\frac{pin_m(x)}{merge(x)}$ page pins. As a result, $pin'_m(x) = write'(x) \cdot \frac{pin_m(x)}{merge(x)}$. To estimate $miss'_m(x)$, we again use the simulated cache to estimate the number of saved merge reads per operation $saved_m$. Thus, $miss'_m(x) = \frac{saved_m}{pin_m(x) \cdot sim}$. Putting everything together, $read'_m(x) = write'(x) \cdot \frac{read_m(x)}{merge(x)} + \frac{saved_m}{sim}$.

Finally, $read'(x)$ can be computed as

$$read'(x) = \frac{saved_q + saved_m}{sim} + write'(x) \cdot \frac{read_m(x)}{merge_m(x)} \quad (6)$$

5.4 Tuning Memory Allocation

Based on the computed $cost'(x)$, the memory allocation can then be tuned to reduce $cost(x)$. Intuitively, the write memory size x should be decreased if $cost'(x) > 0$ and it should be increased if $cost'(x) < 0$. To speed up the tuning process, we use the Newton–Raphson method to find the root of $cost'(x)$ directly, as $cost'(x) = 0$ is a necessary condition for minimizing $cost(x)$. $cost'(x)$ is approximated as a linear function $cost'(x) = Ax + B$ using the last K memory allocations, where K by default is set to 3. Given the current write memory size x_i, the next value is computed as $x_{i+1} = x_i - \frac{cost'(x_i)}{A}$.

Since the memory tuner deals with a complex system with constantly changing workloads and possible estimation errors, several heuristics are used to ensure the stability of the memory tuner. First, when the tuner does not have enough samples to construct the linear function or when the estimated memory allocation x_{i+1} does not reduce the total cost, we simply fall back to a fixed step size, e.g., 5% of the total memory. Second, the maximum step size is limited based on the memory region whose memory needs to be decreased. The intuition is that taking memory from a region may be harmful because both the write memory and the buffer cache are subject to diminishing returns. Thus, at each tuning step, we limit the maximum decreased memory size for either memory region to 10% of its currently allocated memory size. Finally, the memory tuner uses two stopping criteria to avoid oscillation. The memory allocation is not changed if the step size is too small, e.g., smaller than 32MB, or if the expected cost reduction is too small, e.g., smaller than 0.1% of the current I/O cost.

The last question for implementing the memory tuner is determining the appropriate tuning cycle length. Ideally, the tuning cycle should be long enough to capture the workload characteristics but be as short as possible for better responsiveness. To balance these two requirements, memory tuning is triggered whenever the accumulated log records exceed the maximum log length. This allows the memory tuner to capture the workload statistics more accurately by waiting for log-triggered flushes to complete. For read-heavy workloads, it may take a very long time to produce enough log records. To address this, the memory tuner also uses a timer-based tuning cycle, e.g., 10 minutes.

5.5 Optimality of Memory Tuner

The optimality of the memory tuner depends on the shape of $cost(x)$. To ensure that the memory tuner always finds the global minimum, $cost'(x)$ should have at most one root. However, after analyzing $cost''(x)$, i.e., the derivative of $cost'(x)$, we have found that this condition may not always hold. Intuitively, it is easy to see that $write(x)$ is monotonically decreasing and $read_q(x)$ is monotonically increasing. However, $read_m(x)$ is not monotonic because a larger x may both reduce the number of merge reads and increase the cache miss rate. Even though the memory tuner may not be able to always find an optimal memory allocation, this does not limit its applicability in practice. First, it can still find a better memory allocation to reduce the overall I/O cost. Moreover, we have found that $cost(x)$ for many practical workloads often allows the memory tuner to find the global minimum. For example, Figure 6 plots the I/O costs for the the YCSB [22] write-heavy workload and the TPC-C [7] workload. We used the YCSB write-heavy workload with 50% writes and 50% reads. The operations were distributed among 10 LSM-trees, each of which had 10 million records, following an 80-20 hotspot distribution. For TPC-C, the scale factor was set at 2000. (The detailed experimental setup is further described in Section 6.1.) For both workloads, the total I/O cost has one global minimum. The reason is that the merge read cost $read_m(x)$ is relatively small, even under the YCSB write-heavy workload, because many SSTables at small levels are often cached due to frequent merges and accesses. Thus, non-monotonicity of $read_m(x)$ does not affect the total I/O cost too much. We also evaluated other configurations by changing the total memory size and the read-write ratio and found that the total I/O cost always has the same general shape.

6 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the proposed techniques in the context of Apache AsterixDB [1]. Throughout the evaluation, we focus on the following two questions. First, what are the benefits

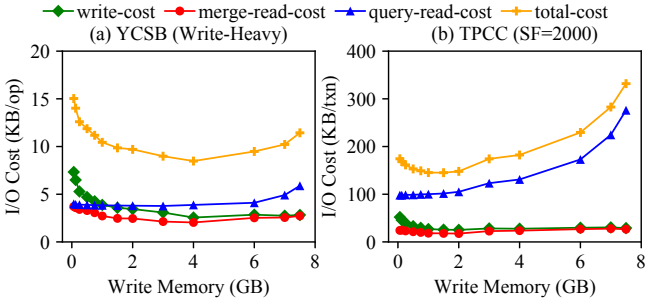

Figure 6: I/O Costs under Different Write Memory Sizes

of the partitioned memory component compared to alternative approaches? Second, what is the effectiveness of the memory tuner in terms of its accuracy and responsiveness? In the remainder of this section, we first describe the general experimental setup followed by the detailed evaluation results.

6.1 Experimental Setup

Hardware. All experiments were performed on a single node m5d.2xlarge on AWS. The node has an 8-core 2.50GHZ vCPUs, 32GB of memory, a 300GB NVMe SSD, and a 500GB elastic block store (EBS). We use the native NVMe for LSM storage and EBS for transactional logging. The NVMe SSD provides a write throughput of 250MB/s and a read throughput of 500MB/s. The EBS also provides a write throughput of 250MB/s. Asynchronous log flushing and group commit were further used to ensure that logging on the EBS is not the bottleneck. We allocated 26GB of memory for the AsterixDB instance. Unless otherwise noted, the total storage memory budget, including the buffer cache and the write memory, was set at 20GB. Both the disk page size and memory page size were set at 16KB. The maximum transaction log length was set at 10GB. Finally, we used 8 worker threads to execute workload operations.

LSM-tree Setup. All LSM-trees used a partitioned leveling merge policy with a size ratio of 10, which is a common setting in existing systems. Unless otherwise noted, the number of disk levels was dynamically determined based on the current write memory size. For the partitioned memory component, its active SSTable size was set at 32MB and the size ratio of the memory merge policy was also set at 10. We used 2 threads to execute flushes, 2 threads to execute memory merges, and 4 threads to execute disk merges. In each set of experiments, we first loaded the LSM storage based on the given workload. Each experiment always started with a fresh copy of the loaded LSM storage. For both memory and disk levels, we built a Bloom filter for each SSTable with a false positive rate of 1% to accelerate point lookups. Finally, both the memory flush threshold and the log truncation threshold were set at 95%.

Workloads. We used two popular benchmarks YCSB [22] and TPC-C [7]. YCSB is a popular and extensible benchmark for evaluating key-value stores. Due to its simplicity, we used YCSB to understand the basic performance of various techniques. In all experiments, we used the default YCSB record size, where each record has 10 fields with 1KB size in total, and the default Zipfian distribution. Since YCSB only supports a single LSM-tree, we further extended it to support multiple primary and secondary LSM-trees, which is described in Section 6.2. TPC-C is an industrial standard benchmark used to evaluate transaction processing systems. We chose TPC-C because it represents a more realistic workload with multiple datasets[5] and secondary indexes. It should be noted that AsterixDB only supports a basic record-level transaction model without full ACID transactions. Thus, all transactions in our evaluation were effectively running under the read-uncommitted isolation level from the TPC-C perspective. Because of this, we disabled the client-triggered aborts (1%) of the NewOrder transaction.

6.2 Evaluating Write Memory Management

We first evaluated the proposed techniques for managing the write memory with the following experiments. The first set of experiments uses a single LSM-tree to evaluate the basic performance of various memory component structures. The second set of experiments uses multiple datasets, each of which just has a primary LSM-tree. The third set of experiments focuses on LSM-based secondary indexes, all belonging to the same dataset. The last set of experiments uses a more realistic workload that contains multiple primary and secondary indexes. For the first three sets of experiments, we used the YCSB benchmark [22] due to its simplicity and customizability. For the last set of experiments, we used the TPC-C benchmark [7] since it represents a more realistic workload. Due to space limitations, we leave the third set of experiments to the extended version of this paper [41]. In general, we found that the performance trends in the secondary LSM-tree case were consistent with the results of the multiple primary LSM-tree case.

Evaluated Write Memory Management Schemes. First, we evaluated two variations of AsterixDB's static memory allocation scheme. The first variation, called *B+-static*, uses AsterixDB's default number of active datasets, which is 8. The second variation, called *B+-static-tuned*, configures the number of active datasets parameter setting based on each experiment. We further evaluated an optimized version of the write memory management scheme (called *B+-dynamic*) used in existing systems, e.g., RocksDB and HBase. This scheme uses a B+-tree to manage the memory component of each LSM-tree without any static size limit. We also evaluated two variations of Accordion [16]. Accordion separates keys from values by storing keys into an index structure while putting values into a log. The first variation, called *Accordion-index*, only merges the indexes without rewriting the logs. The second variation, called *Accordion-data* , merges both the indexes and logs. Finally, we evaluated the partitioned memory component structure, called *Partitioned*. For both *B+-dynamic* and *Partitioned*, we evaluated three variations based on the three flush policies described in Section 4.2, namely max-memory (called *MEM*), min-LSN (called *LSN*), and optimal (called *OPT*). It should be noted that *B+-dynamic* as implemented in existing systems always uses the max-memory policy to flush the LSM-tree with the largest memory component.

6.2.1 Single LSM-tree. In this experiment, the LSM-tree had 100 million records with a 110GB storage size. We evaluated four types of workloads, namely write-only (100% writes), write-heavy (50% writes and 50% lookups), read-heavy (5% writes and 95% lookups),

[5] A *dataset* in AsterixDB is equivalent to a *table* in the TPC-C benchmark.

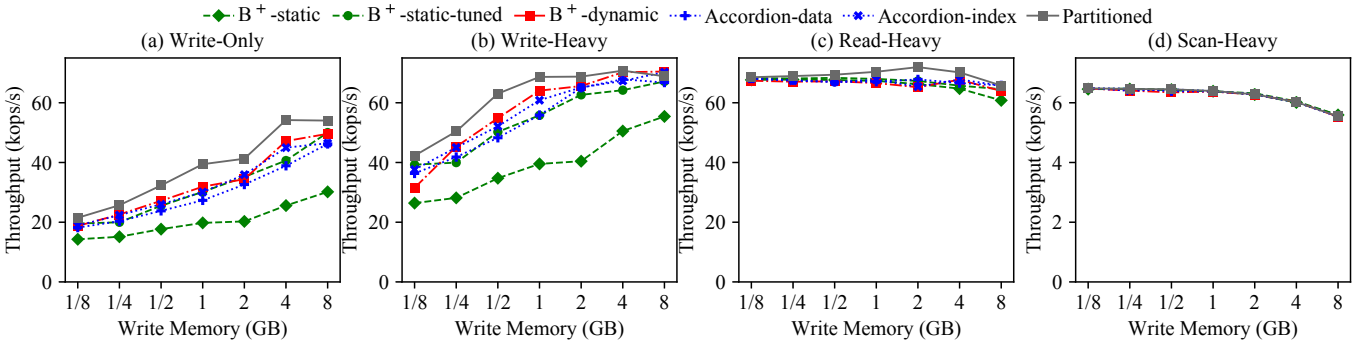

Figure 7: Experimental Results for a Single LSM-tree

and scan-heavy (5% writes and 95% scans). A write operation updates an existing key and each scan query accesses a range of 100 records. Each experiment ran for 30 minutes and the first 10-minute period was excluded when computing the throughput. It should be noted that in this experiment all flush policies have the identical behavior because there was only one LSM-tree.

Basic Performance. Figure 7 shows the throughput of each memory component scheme under different workloads and write memory sizes. In general, the write memory mainly impacts write-dominated workloads, such as write-only and write-heavy, and larger write memory improves the overall throughput by reducing the write cost. Among these structures, *B+-static* always performs the worst since any one LSM-tree is only allocated 1/8 of the write memory. *B+-dynamic* performs slightly better than *B+-static-tuned* because the former does not leave memory idle by preallocating two memory components for double buffering. *Partitioned* has the highest throughput under write-dominated workloads since it better utilizes the write memory. It also improves the overall throughput slightly under the read-heavy workload by reducing write amplification. For both *B+-dynamic* and *Partitioned*, the throughput stops increasing after the write memory exceeds 4GB because flushes are then dominated by log-truncation. Finally, Accordion does not provide any improvement compared to *B+-dynamic*. *Accordion-data* actually reduces the overall throughput because a large memory merge will temporarily double the memory usage, forcing memory components to be flushed. Moreover, Accordion was designed for reducing GC overhead since HBase [3] uses Java objects to manage memory components. Although AsterixDB is written in Java, it uses off-heap structures for memory management [17, 32]. In all experiments, its measured GC time was always less than 1% of the total run time. Based on these results, and because Accordion is mainly designed for a single LSM-tree, we excluded Accordion for further evaluation with multiple LSM-trees.

As suggested by [40], we further carried out an experiment to evaluate the 99th percentile write latencies of each scheme using a constant data arrival process, whose arrival rate was set at a high utilization level (95% of the measured maximum write throughput). We found out that the resulting 99th percentile latencies of all schemes were less than 1s, which suggests that all structures can provide a stable write throughput with a relatively small variance, even under a very high utilization level.

CPU Overhead of Memory Merges. We have seen that *Partitioned* outperforms other memory component structures by better utilizing the write memory. However, it may incur additional CPU overhead due to memory merges. To evaluate this overhead, we carried out an experiment focusing exclusively on the memory component performance. We used a smaller YCSB dataset with only 10 million records. We set the maximum write memory to be 20GB and disabled transaction logging so that the dataset always fits in the memory component. All operations were executed using a single thread and memory merges were always executed synchronously.

Figure 8 shows the resulting throughput under different workloads. To store the same experiment dataset, *Partitioned* only used 12GB of write memory while *B+-dynamic* used 15.5GB. In general, *Partitioned* reduces the in-memory throughput by 20%-40% as compared to *B+-dynamic* due to memory merges, where the write amplification was about 11.36. However, it should be noted that in-memory workloads are not the focus of this work. The partitioned memory component structure thus trades some CPU cycles to reduce the overall disk write amplification.

Benefits of Dynamically Adjusting Disk Levels. To evaluate the benefit of dynamically adjusting disk levels as the write memory changes, we conducted an experiment where the write memory size alternates between 1GB and 32MB every 30 minutes. Each experiment ran for two hours in total. We used the partitioned memory component structure but the disk levels were determined differently. In addition to the proposed approach that adjusts disk levels dynamically (called *dynamic*), we used two baselines where the number of disk levels is determined statically by assuming that the write memory is always 32MB (called *static-32MB*) or always

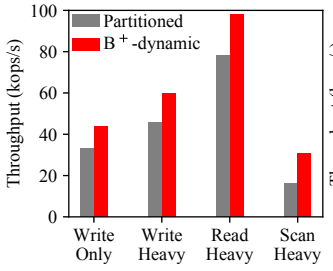

Figure 8: Evaluation of Memory Merge Overhead

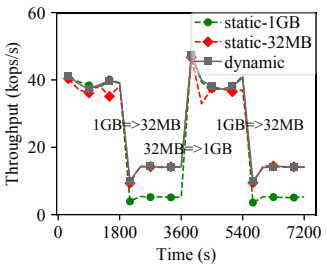

Figure 9: Write Throughput with Varying Write Memory

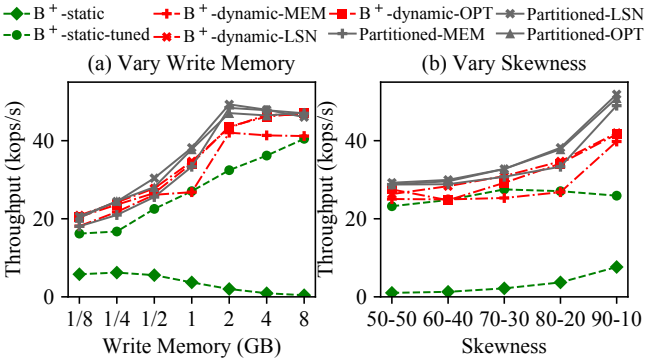

Figure 10: Evaluation of Multiple Primary LSM-trees

1GB (called *static-1GB*). The resulting write throughput, aggregated over 5-minute windows, is shown in Figure 9. The dynamic approach always has the highest throughput, which confirms the utility of adjusting disk levels as the write memory changes. Moreover, we see that having fewer levels when the write memory is small has a more negative impact than having more levels when the write memory is large since the write throughput for *static-1GB* is much lower under the small write memory.

6.2.2 Multiple Primary LSM-trees. In this set of experiments, we used 10 primary LSM-trees, each of which had 10 million records. Since the write memory mainly impacts write performance, a write-only workload was used in this experiment. Writes were distributed among the multiple LSM-trees following a hotspot distribution, where x% of the writes go to y% of the LSM-trees. For example, an 80-20 distribution means that 80% of the writes go to 20% of the LSM-trees, i.e., 2 hot LSM-trees, while the 20% of the writes go to 80% of the LSM-trees, yielding 8 cold LSM-trees. Within each LSM-tree, writes still followed YCSB's default Zipfian distribution.

Impact of Write Memory. We first evaluated the impact of the write memory size by fixing the skewness to be 80-20. The resulting write throughput is shown in Figure 10a. Note that B^+*-static* results in a much lower throughput because of thrashing. Since the default number of active datasets in AsterixDB is only 8, some LSM-trees have to be constantly activated and deactivated, resulting in many tiny flushes. B^+*-static-tuned* avoids the thrashing problem, but it still performs worse than the other baselines because it does not differentiate hot LSM-trees from cold ones. Both B^+*-dynamic* and *Partitioned* allocate the write memory dynamically, improving the write throughput. Moreover, we see that the min-LSN and optimal flush policies perform better than the max-memory flush policy for both structures. Since the max-memory policy always flushes the largest memory component, the memory components of the cold LSM-trees are not flushed until they are large enough or until the transaction log has to be truncated. The min-LSN policy also has a write throughput comparable to the optimal policy, which makes it a good approximation but with less implementation complexity. Finally, even under the same flush policy, we see that *Partitioned* still outperforms B^+*-dynamic* because it performs memory merges to further increase memory utilization.

Impact of Skewness. Next, we evaluated the impact of skewness by fixing the write memory to be 1GB. The resulting write throughput is shown in Figure 10b. All memory component structures except B^+*-static-tuned* benefit from skewed workloads. The problem of is that B^+*-static-tuned* always allocates the write memory evenly to the active datasets without differentiating hot LSM-trees from cold ones. For B^+*-static*, the thrashing problem is alleviated under skewed workloads since most writes go to a small number of LSM-trees. When the workload is more skewed, we see two interesting trends. First, under the min-LSN and optimal flush policies, the performance difference between *Partitioned* and B^+*-dynamic* becomes larger. This is because a small number of hot LSM-trees occupy most of the write memory, allowing more memory merges to be performed in these hot LSM-trees to reduce the write amplification. Moreover, the performance differences among the three flush policies also become larger since the min-LSN and optimal policies allocate more write memory to the hot LSM-trees.

6.2.3 TPC-C Results. Finally, we used the TPC-C benchmark to evaluate the alternative memory management schemes on a more realistic workload. We used two scale factors (SF) of TPC-C, i.e., 500, which results in a 50GB storage size, and 2000, which results in a 200GB storage size. Each experiment ran for one hour and the throughput was measured excluding the first 30 minutes.

The resulting throughput and the per-transaction disk writes (KB) under the two scale factors are shown in Figure 11. Note that B^+*-static-tuned* is omitted here, because the number of active datasets in TPC-C is 8, which is the same as the default value used in AsterixDB. B^+*-static* still has the highest I/O cost because it allocates write memory evenly to all datasets. TPC-C contains some hot datasets, such as order_line and stock, that receive most of the writes, as well as some cold datasets, such as warehouse and district, that only require a few megabytes of write memory. As we have seen in Figure 10, the min-LSN and optimal policies have reduced the write cost for both B^+*-dynamic* and *Partitioned*. *Partitioned-OPT* also led to the lowest write cost via extra memory merges, improving the system I/O efficiency. However, since TPC-C is a CPU-heavy workload, doing so may not always improve the overall transaction throughput due to the CPU overhead of memory merges as we have seen before. When the workload is CPU-bound at scale factor 500, the extra CPU overhead incurred by memory merges actually decreases the overall throughput as compared to B^+*-dynamic*. Thus, we observe that it is useful to design a memory management scheme to balance the CPU overhead and the I/O cost, which we leave as future work. Finally, the results also show that increasing the write memory may not always increase the overall transaction throughput. For example, when the scale factor is 2000, the optimal throughput is reached when the write memory is between 1GB and 2GB. This confirms the importance of memory tuning, which will be evaluated next.

6.2.4 Summary. As all experiments have illustrated, it is important to utilize a large write memory efficiently to reduce the I/O cost. Although the static memory allocation scheme is relatively simple and robust, it leads to sub-optimal performance because the write memory is always evenly allocated to active datasets. The optimized version of the memory management scheme used by existing systems reduces the I/O cost by dynamically allocating the write memory to active LSM-trees. This still does not achieve optimal performance, however, because it fails to manage large memory

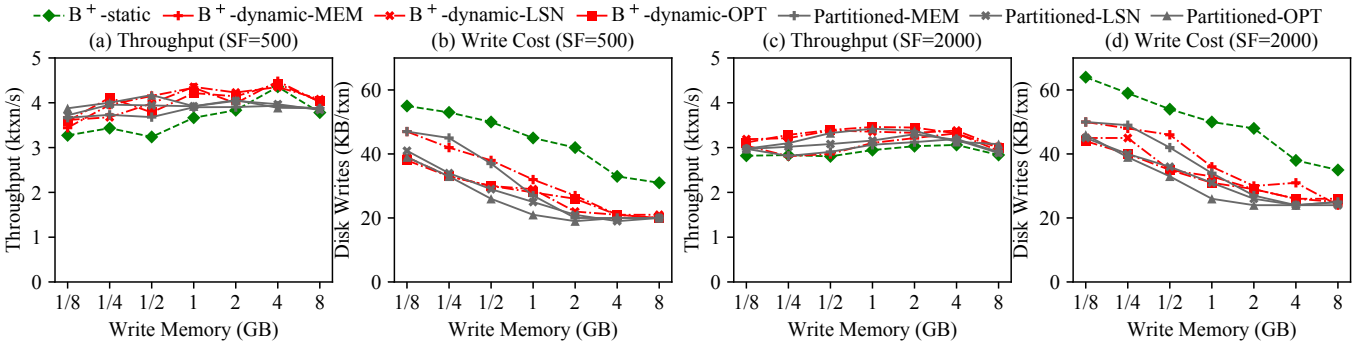

Figure 11: Experimental Results on TPC-C

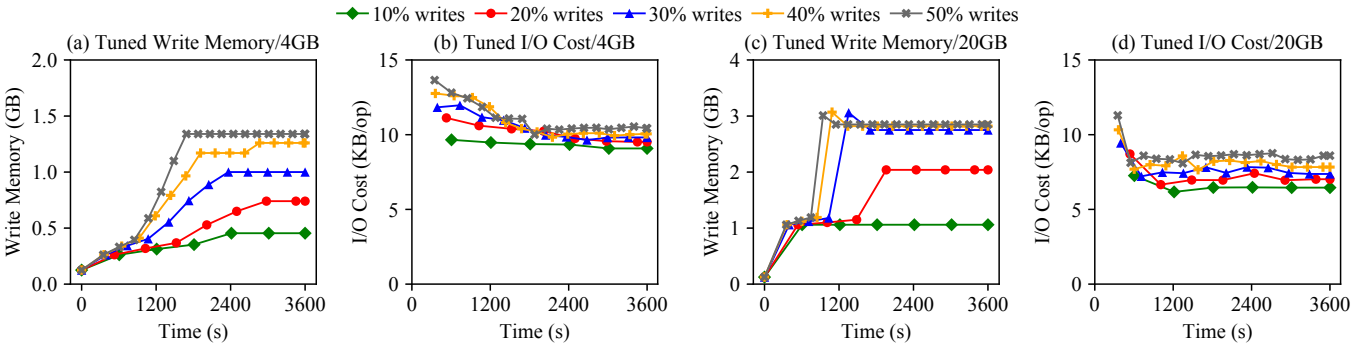

Figure 12: Evaluation of Memory Tuner on YCSB

components efficiently and its choice of flushes does not optimize the overall write cost. Finally, the proposed partitioned memory component structure and the optimal flush policy minimize the write cost for all workloads. The use of partitioned memory components manages the large write memory more effectively to reduce the write amplification of a single LSM-tree. The optimal flush policy allocates the write memory to multiple LSM-trees based on their write rates to minimize the overall write cost. However, the partitioned memory component structure may incur extra CPU overhead, which makes it less suitable for CPU-heavy workloads. Finally, we have observed that the min-LSN policy achieves comparable performance to the optimal policy, which makes it a good approximation but with less implementation complexity.

6.3 Evaluating the Memory Tuner

We now proceed to evaluate the memory tuner with the focus on the following questions: First, what are the basic mechanics of the memory tuner in terms of how it tunes the memory allocation for different workloads? Second, what is the accuracy of the memory tuner as compared to manually tuned memory allocation? Finally, how responsive is the memory tuner when the workload changes?

In all experiments below, the initial write memory size was set at 64MB and the simulated cache size was set to 128MB. Unless otherwise noted, other settings of the memory tuner, such as the number of samples for fitting the linear function, the stopping threshold, and the maximum step size, all used the default values given in Section 5.4.

6.3.1 Basic Mechanics. To understand how the memory tuner performs memory tuning to reduce the I/O cost for different workloads, we carried out a set of experiments using YCSB [22] with a single LSM-tree. We set both weights ω and γ to 1 since we focus on the I/O cost in this experiment. The LSM-tree had 100 million records with 110GB in total. We used a mixed read/write workload where the write ratio varied from 10% to 50%. The total memory budget was set at 4GB or 20GB. Each experiment ran for 1 hour.

The tuned write memory size and the corresponding I/O costs over time are shown in Figure 12. Note that each point denotes one tuning step performed by the memory tuner. We see that the memory tuner balances the relative gain of allocating more memory to the write memory and the buffer cache to reduce the overall I/O cost. As shown in Figures 12a and 12c, when the overall memory budget is fixed, the memory tuner allocates more write memory when the write ratio is increased because the benefit of having a large write memory increases. Moreover, by comparing the allocated write memory sizes in Figures 12a and 12c, we can see that when the write ratio is fixed, the memory tuner also allocates more write memory when the total memory becomes larger. This is because the benefit of having more buffer cache memory plateaus. Finally, as shown in Figures 12b and 12d, the overall I/O cost also decreases after the memory allocation is tuned over time.

6.3.2 Accuracy. To evaluate the accuracy of the memory tuner, we carried out a set of experiments on TPC-C to compare the tuned performance versus the optimal performance. Here we used TPC-C because it represents a more complex and more realistic workload than YCSB. The scale factor was set at 2000. Since for our SSD writes

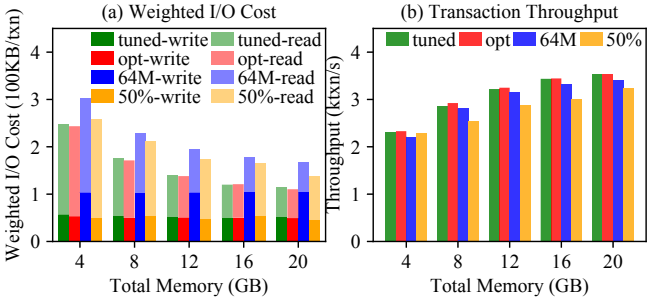

Figure 13: Memory Tuner's Accuracy on TPC-C

are twice as expensive as reads, we set the write weight ω to be 2 and the read weight γ to be 1 in the remaining experiments to balance these two costs. To find the optimal memory allocation (called *opt*), we used an exhaustive search to evaluate different memory allocations with an increment of 128MB. To show the effectiveness of the memory tuner, we included two additional baselines. The first baseline (called *64M*) always set the write memory at 64MB, which was the starting point of the memory tuner. The second baseline (called *50%*) divided the total memory budget evenly between the buffer cache and the write memory. We further varied the total memory budget from 4GB to 20GB. Each experiment ran for 1 hour and the initial 30 minutes were excluded from the measurement.

Figure 13 shows the weighted I/O cost per transaction and the transaction throughput for the different memory allocations. Using exhaustive search, we found that minimizing the weighted I/O cost also maximized the transaction throughput. The auto-tuned I/O cost and throughput (called *tuned*) are very close to the optimal ones found via exhaustive search, which shows the effectiveness of our memory tuner. Moreover, the memory tuner performs notably better than the two heuristic-based baselines. Allocating a small write memory minimizes the read cost but leads to a higher write cost. In contrast, allocating a large write memory minimizes the write cost but the read cost becomes much higher. As a result, both allocations also fail to maximize the overall transaction throughput.

6.3.3 Responsiveness. Finally, we used a variation of TPC-C to evaluate the responsiveness of the memory tuner. This experiment started with the default TPC-C transaction mix and the workload changed into a read-mostly variation, one which contains 5% write transactions, i.e., new_order, payment, and delivery, and 95% read

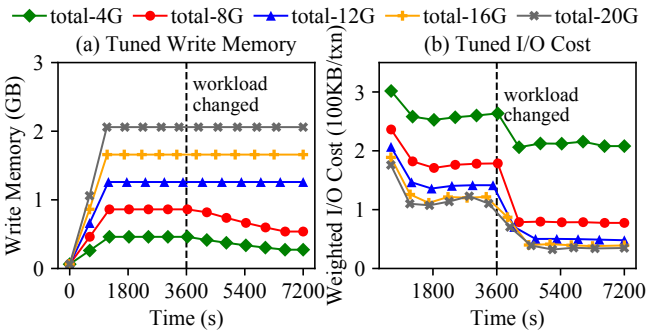

Figure 14: Memory Tuner's Responsiveness on TPC-C

transactions, i.e., order_status and stock_level. Each experiment ran for two hours and the workload was changed after the first hour. The resulting allocated write memory and weighted I/O cost over time are shown in Figure 14. After the workload changes, the memory tuner immediately starts to allocate more memory to the buffer cache. Note that the write memory decreases relatively slowly because the memory tuner limits its step size to 10% of the current write memory size to ensure stability. However, this does not impact the overall I/O cost too much because the buffer cache already occupies most of the memory. Also note that the write memory size does not change in response to the workload shift when the total memory is larger than 8GB. This is because the buffer cache already occupies most of the memory and allocating more memory would not change the I/O cost too much.

In [41], we further evaluated the impact of the maximum step size on the responsiveness and stability of the memory tuner. Even though a large step size allows the memory tuner to change the memory allocation more rapidly, it also leads to instability and oscillation. Thus, the memory tuner's default maximum step size is set at 10% to ensure stability while providing reasonable responsiveness.

6.3.4 Summary. We have evaluated the memory tuner in terms of its mechanics, accuracy, and responsiveness. Our tuner uses a white-box to minimize the overall I/O cost based on the relative gains of allocating more memory to the buffer cache or to the write memory. The experimental results show that this white-box approach enables the memory tuner to achieve both high accuracy with reasonable responsiveness, making it suitable for online tuning.

7 CONCLUSION

In this paper, we have described and evaluated a number of techniques to break down the memory walls in LSM-based storage systems. We first presented an LSM memory management architecture that facilitates adaptive memory management. We further proposed a partitioned memory component structure with new flush polices to better utilize the write memory in order to minimize the overall write cost. To break down the memory wall between the write memory and the buffer cache, we further introduced a memory tuner that uses a white-box approach to continuously tune the memory allocation. We have empirically demonstrated that these techniques together enable adaptive memory management to minimize the I/O cost for LSM-based storage systems. In the future, we plan to extend the memory tuner to incorporate other memory regions, such as query operator memory, to perform global memory tuning.

ACKNOWLEDGMENTS

We thank Mark Callaghan and anonymous reviewers for their helpful comments. This work has been supported by NSF awards CNS-1305430, IIS-1447720, IIS-1838248, and CNS-1925610 along with industrial support from Amazon, Google, and Microsoft and support from the Donald Bren Foundation (via a Bren Chair).

REFERENCES

[1] 2020. AsterixDB. https://asterixdb.apache.org/.
[2] 2020. Cassandra. http://cassandra.apache.org/.
[3] 2020. HBase. https://hbase.apache.org/.
[4] 2020. LevelDB. http://leveldb.org/.
[5] 2020. MyRocks. https://http://myrocks.io/.

[6] 2020. RocksDB. http://rocksdb.org/.

[7] 2020. TPC-C. http://www.tpc.org/tpcc/.

[8] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for Microsoft SQL Server 2005. In *ACM International Conference on Management of Data (SIGMOD)*. ACM, 930–932.

[9] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment (PVLDB)* 7, 14 (2014), 1905–1916.

[10] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proceedings of the VLDB Endowment (PVLDB)* 7, 10 (2014), 841–852.

[11] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX Annual Technical Conference (ATC)*. 363–375.

[12] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *USENIX Annual Technical Conference (ATC)*). 753–766.

[13] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *European Conference on Computer Systems (EuroSys)*. 80–94.

[14] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 301–316.

[15] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM (CACM)* 13, 7 (July 1970), 422–426.

[16] 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 (PVLDB)* 11, 12 (2018), 1863–1875.

[17] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A Bloat-Aware Design for Big Data Applications. In *International Symposium on Memory Management (ISMM)*. 119–130.

[18] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (2018), 1849–1862.

[19] Michael J. Carey. 2019. AsterixDB Mid-Flight: A Case Study in Building Systems in Academia. In *International Conference on Data Engineering (ICDE)*. 1–12.

[20] 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* 26, 2 (2008), 4:1–4:26.

[21] Hong Tai Chou and David J. DeWitt. 1986. An evaluation of buffer management strategies for relational database systems. *Algorithmica* 1, 1 (01 Nov 1986), 311–336.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*. 143–154.

[23] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *ACM International Conference on Management of Data (SIGMOD)*. 79–94.

[24] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)* 43, 4, Article 16 (2018), 16:1–16:48 pages.

[25] 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 *ACM International Conference on Management of Data (SIGMOD)*. 505–520.

[26] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *ACM International Conference on Management of Data (SIGMOD)*. 449–466.

[27] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Conference on Innovative Data Systems Research (CIDR)*.

[28] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment (PVLDB)* 2, 1 (2009), 1246–1257.

[29] Raman Grover and Michael J. Carey. 2015. Data Ingestion in AsterixDB. In *International Conference on Extending Database Technology (EDBT)*. 605–616.

[30] 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 *ACM International Conference on Management of Data (SIGMOD)*. 651–665.

[31] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *International Conference on Very Large Data Bases (VLDB)*. 439–450.

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

[33] Yongkun Li, Helen H. W. Chan, Patrick P. C. Lee, and Yinlong Xu. 2019. Enabling Efficient Updates in KV Storage via Hashing: Design and Performance Evaluation. *ACM Transactions on Storage (TOS)* 15, 3, Article 20 (2019).

[34] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *USENIX Annual Technical Conference (ATC)*. 739–752.

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

[36] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. 2019. Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems. *Proceedings of the VLDB Endowment (PVLDB)* 12, 12 (2019), 1970––1973.

[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies (FAST)*. 133–148.

[38] Chen Luo. 2020. Breaking Down Memory Walls in LSM-based Storage Systems. In *ACM International Conference on Management of Data (SIGMOD)*. 2817–2819.

[39] Chen Luo and Michael J. Carey. 2019. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proceedings of the VLDB Endowment (PVLDB)* 12, 5 (2019), 531–543.

[40] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-based Storage Systems. *Proceedings of the VLDB Endowment (PVLDB)* 13, 4 (2019), 449–462.

[41] Chen Luo and Michael J. Carey. 2020. Breaking Down Memory Walls: Adaptive Memory Management in LSM-based Storage Systems (Extended Version). *CoRR* abs/2004.10360 (2020).

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

[43] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *International Conference on Extending Database Technology (EDBT)*. 1–12.

[44] 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 *ACM International Conference on Management of Data (SIGMOD)*. 2071–2086.

[45] 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 *IEEE International Conference on Big Data*. 523–532.

[46] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *ACM Symposium on Cloud Computing (SoCC)*. 477–489.

[47] 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.

[48] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *SIGMOD Record* 22, 2 (June 1993), 297–306.

[49] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases. In *ACM International Conference on Management of Data (SIGMOD)*. 551–566.

[50] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Symposium on Operating Systems Principles (SOSP)*. 497–514.

[51] 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 (PVLDB)* 10, 13 (2017), 2037–2048.

[52] Giovanni Maria Sacco and Mario Schkolnick. 1982. A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model. In *International Conference on Very Large Data Bases (VLDB)*. 257–262.

[53] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *ACM International Conference on Management of Data (SIGMOD)*. 893–908.

[54] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *ACM International Conference on Management of Data (SIGMOD)*. 217–228.

[55] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *International Conference*

on Very Large Data Bases (VLDB). 1081–1092.

[56] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. IBTune: Individualized Buffer Tuning for Large-Scale Cloud Databases. *Proceedings of the VLDB Endowment (PVLDB)* 12, 10 (2019), 1221—-1234.

[57] Dinh Nguyen Tran, Phung Chinh Huynh, Yong C Tay, and Anthony KH Tung. 2008. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage (TOS)* 4, 1 (2008), 1–25.

[58] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *ACM International Conference on Management of Data (SIGMOD)*. 1009–1024.

[59] Xikui Wang and Michael J. Carey. 2019. An IDEA: An Ingestion Framework for Data Enrichment in AsterixDB. *Proceedings of the VLDB Endowment (PVLDB)* 12, 11 (2019), 1485–1498.

[60] Andrew Chi-Chih Yao. 1978. On random 2–3 trees. *Acta Informatica* 9, 2 (1978), 159–170.

[61] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *ACM International Conference on Management of Data (SIGMOD)*. 415–432.

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