UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
On Optimizing LSM-based Storage for Big Data Management Systems

Permalink
https://escholarship.org/uc/item/5x65p2dr

Author
Luo, Chen

Publication Date
2020

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

On Optimizing LSM-based Storage for Big Data Management Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Chen Luo

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Vassilis J. Tsotras

2020

Portions of Chapters 2 and 3 © 2020 Springer doi 10.1007/s00778-019-00555-y
Portions of Chapter 4 © 2019 VLDB Endowment doi 10.14778/3303753.3303759
Portions of Chapter 5 © 2019 VLDB Endowment doi 10.14778/3372716.3372719
Portions of Chapter 6 © 2021 VLDB Endowment doi 10.14778/3430915.3430916
All other materials © 2020 Chen Luo

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

First, I would like to thank my PhD advisor, Professor Michael Carey, for guiding and supporting me during my PhD studies. Professor Carey has been an excellent advisor. He gave enough freedom for me to explore interesting research problems, while providing me knowledgeable guidance so that I can always stay on track. His high research standards have deeply impacted me and shaped me into an independent researcher. I am also grateful for his promotion of my research work to the database community. Looking back, I feel very fortunate to join UCI four years ago – this decision has changed my life in a very positive way. I am deeply indebted for Professor Carey!

I would like to thank Professor Chen Li for joining my dissertation committee. My first database course taken at UCI was taught by Professor Li, and I am very fortunate to work with Professor Li as a TA for this course several times. The experience of taking and participating this course prepared me with solid foundations for my later research. I would also like to thank Professor Vassilis Tsotras for joining my dissertation committee. Professors Li and Tsotras have provided a lot of constructive feedback to this dissertation, which has greatly improved this work.

I would like to thank my mentors Pinar Tözün and Yuanyuan Tian and my manager Fatma Özcan for hosting my internship at IBM Almaden. It was a great opportunity to intern at IBM Alamden to work on a cutting-edge database project, especially during the early stage of my PhD studies. This internship not only deepened my research and technical skills, but more importantly, it broadened my horizon and inspired my later PhD work.

I would like to thank my mentor David Lomet for hosting my internship at Microsoft Research as well as my collaborator Jaeyoung Do. David is a highly recognized database researcher, yet his is very detail oriented. I learned a lot from his technical feedback on my internship work. I was also deeply impressed by his applying strong mathematical knowledge to solving database system problems. It was great experience to work with David at Microsoft Research.

I would like to thank Pat Helland, Neal Young, and Mark Callaghan for their helpful discussion and constructive feedback to this dissertation. Pat Helland provided a lot of useful comments and feedback to the first two topics of this dissertation. Neal Young gave me useful feedback to the proofs in this dissertation. Mark Callaghan offered me many useful comments to the survey of LSM storage techniques and the last topic of this dissertation. I also thank Mark Callaghan for his precious support during my job finding. Their comments and feedback have greatly improved the quality of this dissertation.

I would also like to thank my friends and colleagues while I was at UCI, Xikui Wang, Jianfeng Jia, Taewoo Kim, Yingyi Bu, Shiva Jahangiri, Gift Sinthong, Wail Alkowaileet, Qiushi Bai, Zuozhi Wang, Ian Maxon, Abdullah Alamoudi, Murtadha Hubail, Dmitry Lychagin, Michael Blow, and Till Westmann for their help and support during my PhD journey.

Most importantly, I would like to thank my wife, Dongxu, for her previous support, encouragement, and love during the past four years. My PhD journey would be incomplete without her support and accompanying.

The work reported in this dissertation 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).

VITA

Chen Luo

EDUCATION

Doctor of Philosophy in Computer Science **2020**
University of California, Irvine *Irvine, CA*

Master of Engineering in Software Engineering **2016**
Tsinghua University *Beijing, China*

Bachelor of Engineering in Software Engineering **2013**
Tongji University *Shanghai, China*

PUBLICATIONS

Breaking down memory walls: Adaptive memory management in LSM-based storage systems **2021**
Proceedings of the VLDB Endowment (PVLDB)

Programming an SSD controller to support batched writes for variable-size pages **2021**
IEEE International Conference on Data Engineering (ICDE)

LSM-based storage techniques: a survey **2020**
The VLDB Journal (VLDBJ)

Robust and efficient memory management in Apache AsterixDB **2020**
Software: Practice and Experience

On performance stability in LSM-based storage systems **2019**
Proceedings of the VLDB Endowment (PVLDB)

Efficient data ingestion and query processing for LSM-based storage systems **2019**
Proceedings of the VLDB Endowment (PVLDB)

Umzi: Unified multi-zone indexing for large-scale HTAP **2019**
International Conference on Extending Database Technology (EDBT)

PSpec-SQL: Enabling fine-grained control for distributed data analytics **2019**
IEEE Transactions on Dependable and Secure Computing (TDSC)

Inferring software behavioral models with MapReduce **2017**
Science of Computer Programming

ABSTRACT OF THE DISSERTATION

On Optimizing LSM-based Storage for Big Data Management Systems

By

Chen Luo

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Michael J. Carey, Chair

In recent years, the Log-Structured Merge-tree (LSM-tree) has been widely used in the storage layer in modern NoSQL systems. Different from traditional index structures that apply updates in-place, an LSM-tree first buffers all writes in memory and subsequently flushes them to disk and merges them using sequential I/Os. This out-of-place update design brings a number of advantages, including superior write performance, high space utilization, tunability, and simplification of concurrency control and recovery. These advantages have enabled LSM-trees to serve a large variety of workloads in production systems.

Despite the popularity of LSM-trees, the existing research efforts have been primarily focusing on improving the maximum throughput of LSM-trees in simple key-value store settings. This leads to several outages when adopting LSM-based storage techniques in a Big Data Management System (BDMS) with multiple heterogeneous LSM-trees and requiring performance metrics beyond just the maximum throughput.

In this dissertation, we focus on optimizing LSM-trees for BDMSs. We first propose a set of techniques to efficiently maintain and exploit LSM-based auxiliary structures, including secondary indexes and filters. These techniques include a series of optimizations for efficient batched point lookups, significantly improving the range of applicability of LSM-based secondary indexes, and several new and efficient maintenance strategies to maintain LSM-based auxiliary structures to

accommodate the out-of-place update nature of LSM-trees.

In addition to maximum throughput, performance stability measures, such as percentile latency, are another important performance metric for storage systems. However, LSM-trees often exhibit large performance variance due to periodic write stalls. To tackle this problem, we propose a simple yet effective two-phase approach to evaluate write stalls of LSM-trees. We further explore the design choices of merge schedulers for various LSM-tree designs to minimize write stalls given a disk bandwidth budget.

Finally, we present adaptive memory management techniques to break down the memory walls in LSM-based storage systems, enabling the shared management of memory components of multiple datasets and adaptive memory allocation between memory components and the disk buffer cache. These techniques together have successfully reduced the disk I/O cost of LSM-based storage systems, improving the system performance and efficiency.

Chapter 1

Introduction

The Log-Structured Merge-tree (LSM-tree) has been widely adopted in the storage layers of modern NoSQL systems, including BigTable [35], Dynamo [46], HBase [5], Cassandra [2], LevelDB [6], RocksDB [10], and AsterixDB [18]. Different from traditional index structures that apply in-place updates, the LSM-tree uses an out-of-place update scheme by first buffering all writes in memory, which are subsequently flushed and merged using sequential I/Os. This design brings a number of advantages, including superior write performance, high space utilization, tunability, and simplification of concurrency control and recovery. These advantages have enabled LSM-trees to serve a large variety of workloads, including real-time data processing [38], graph processing [4], stream processing [38], OLTP workloads [7, 109], and HTAP workloads [58, 81, 125].

Due to the popularity of LSM-trees among modern data stores, a large number of LSM-tree improvements have been proposed by the research community. However, most of these improvements have focused on a key-value store setting to improve the maximum throughput of a single LSM-tree. This leads to several outages when adopting LSM-trees in a Big Data Management System (BDMS) with multiple heterogeneous LSM-trees from multiple datasets and indexes. Moreover, a BDMS also requires performance metrics beyond maximum throughput, such as performance

stability and efficiency. To address these challenges, in this dissertation we optimize LSM-trees for BDMSs in the context of AsterixDB [1] based on the following topics.

LSM-based Auxiliary Structures. Two types of LSM-based auxiliary structures have been used in LSM-based storage systems to facilitate query processing, namely secondary indexes and filters. However, we have found two general problems related to these auxiliary structures. First, no particularly efficient point lookup algorithms have been proposed for the efficient fetching of the records identified by a secondary index search. This limits the range of applicability of LSM-based secondary indexes, requiring the query optimizer to maintain accurate statistics to make correct decisions for processing ad-hoc queries efficiently. Second, existing systems typically employ an eager strategy to maintain these structures during data ingestion by prefacing each incoming write with a point lookup, which leads to significant overhead during ingestion.

Performance Stability. Despite the popularity of LSM-trees, they have been criticized for suffering from write stalls and large performance variances [3, 103, 131] because of their out-of-place update design. Due to the inherent mismatch between fast in-memory writes and slower background I/O operations, in-memory writes must be slowed down or stopped if background flushes or merges cannot catch up, which, however, may cause large tail latencies. It remains an open question that whether an LSM-tree can provide a high and stable write throughput with low write latencies.

Adaptive Memory Management. Efficient memory management is critical for storage systems to achieve good performance. Compared to update-in-place systems where all pages are managed within shared buffer pools, LSM-trees have introduced additional memory walls. First, due to the LSM-tree's out-of-place update design, the write memory is isolated from the buffer cache. Moreover, LSM-based storage systems must deal with multiple heterogeneous LSM-trees from multiple datasets and indexes, which requires the write memory to be efficiently shared among multiple LSM-trees. Existing LSM-tree implementations, such as RocksDB [10] and AsterixDB [1, 66], have opted for simplicity and robustness over optimal performance by adopting static memory al-

location schemes. Since the optimal memory allocation heavily depends on the workload, memory management should be workload-adaptive to maximize the system performance.

The rest of this dissertation is organized as follows. Chapter 2 discusses the background information of AsterixDB and LSM-trees. Chapter 3 surveys existing research on optimizing LSM-trees. Chapter 4 presents novel strategies for efficiently maintaining and exploiting LSM-based auxiliary structures. Chapter 5 studies the performance stability problem of LSM-trees. Chapter 6 describes the design and implementation of adaptive memory management techniques for LSM-trees. Finally, Chapter 7 concludes this dissertation and discusses future research directions.

Chapter 2

Background

2.1 Apache AsterixDB

Apache AsterixDB [1, 17] is an open-source Big Data Management System (BDMS) that aims to manage massive amounts of semi-structured (e.g., JSON) data efficiently. Here we focus on the storage management [18] and memory management [66] aspects of AsterixDB.

AsterixDB uses a shared-nothing parallel database style architecture. The records of each dataset are hash-partitioned based on their primary keys across multiple nodes. Each partition of a dataset is managed by an LSM-based storage engine, with a primary index, a primary key index, and multiple local secondary indexes. AsterixDB uses a record-level transaction model to ensure that all of the indexes are kept consistent within each partition. The primary index stores records indexed by primary keys, and the primary key index stores primary keys only. The primary key index is built to support COUNT(*) style queries and uniqueness checks efficiently [77] since it is much smaller than the primary index.

Secondary indexes use the composition of the secondary key and the primary key as their index

keys. AsterixDB supports LSM-based B^+-trees, R-trees, and inverted indexes using a generic LSM-ification framework that can convert an in-place index into an LSM-based index. For LSM-based R-trees, a linear order, such as a Hilbert curve for point data and a Z-order curve for non-point data, is used to sort the entries in disk components, while in the memory component, deleted keys are recorded in a separate B^+-tree to avoid multi-path traversals during deletes. By default, each LSM index's components are merged independently using a tiering-like merge policy. AsterixDB also supports a correlated merge policy that synchronizes the merges of all of a dataset's indexes together to improve query performance with filters. The basic idea of this policy is to delegate merge scheduling to the primary index. When a sequence of primary index components are merged, all corresponding components from other indexes will be merged as well.

AsterixDB uses a static memory allocation scheme for simplicity and robustness [66]. 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.

2.2 LSM-tree

2.2.1 Basic Structure

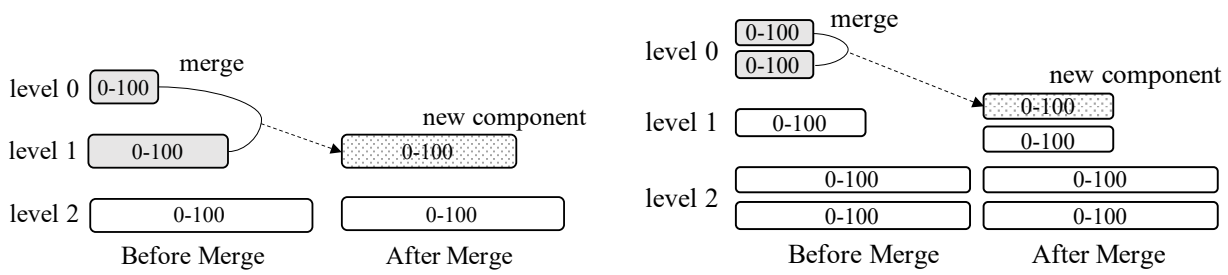
The log-structured merge-tree (LSM-tree) [90] is a write-optimized index structure applying out-of-place updates. In an LSM-tree, all incoming writes are first appended into a memory component. An insert or update operation simply adds a new entry, while a delete operation adds an anti-matter entry indicating that a key has been deleted. Whenever a memory component is full, it is flushed to disk to form an immutable disk component. Multiple disk components can be merged together

into a new one without modifying the existing disk components.

Internally, an LSM-tree component can be implemented using any index structure. Today's LSM-tree implementations typically organize their memory components using a concurrent data structure such as a skip-list or a B^+-tree, while they organize their disk components using B^+-trees or sorted-string tables (SSTables). An SSTable contains a list of data blocks and an index block; a data block stores key-value pairs ordered by keys, and the index block stores the key ranges of all data blocks.

A query over an LSM-tree has to search multiple components to perform reconciliation, that is, to find the latest version of each key. A *point lookup query*, which fetches the value for a specific key, can simply search all components one by one, from newest to oldest, and stop immediately after the first match is found. A *range query* can search all components at the same time, feeding the search results into a priority queue to perform reconciliation.

As disk components accumulate over time, the query performance of an LSM-tree tends to degrade since more components must be examined. To address this, disk components are gradually merged to reduce the total number of components. Two types of merge policies are typically used in practice [79]. As shown in Figure 2.1, both policies organize disk components into logical levels (or tiers) and are controlled by a size ratio T. Each component is labeled with its potential key range in the figure. In the leveling merge policy (Figure 2.1a), each level only maintains one component, but the component at level L is T times larger than the component at level $L-1$. As a result, the component at level L will be merged multiple times with incoming components at level $L-1$ until it fills up, and it will then be merged into level $L+1$. For example in the figure, the component at level 0 is merged with the component at level 1, which will result in a bigger component at level 1. In contrast, the tiering merge policy (Figure 2.1b) maintains up to T components per level. When level L is full, its T components are merged together into a new component at level $L+1$. In the figure, the two components at level 0 are merged together to form a new component at level 1. It should be noted that if level L is already the configured maximum level, then the resulting



(a) Leveling Merge Policy

(b) Tiering Merge Policy

Figure 2.1: LSM-tree Merge Policies

component remains at level *L*. In practice, for a stable workload where the volume of inserts equal the volume of deletes, the total number of levels remains static[1]. In general, the leveling merge policy optimizes for query performance since there are fewer components to search in the LSM-tree. The tiering merge policy is more write optimized since it reduces the merge frequency. We will discuss the performance of these two merge policies further in Section 2.2.3.

2.2.2 Well-Known Optimizations

Bloom Filter. A Bloom filter [29] is a space-efficient probabilistic data structure designed to aid in answering set membership queries. It supports two operations, i.e., inserting a key and testing the membership of a given key. To insert a key, it applies multiple hash functions to map the key into multiple locations in a bit vector and sets the bits at these locations to 1. To check the existence of a given key, the key is again hashed to multiple locations. If all of the bits are 1, then the Bloom filter reports that the key probably exists. By design, the Bloom filter can report false positives but not false negatives.

Bloom filters can be built on top of disk components to greatly improve point lookup performance. To search a disk component, a point lookup query can first check its Bloom filter and then proceed

[1]Even for an append-mostly workload, the total number of levels will grow extremely slowly since the maximum number of entries that an LSM-tree can store increases exponentially with a factor of T as the number of levels increases.

to search its B^+-tree only if its associated Bloom filter reports a positive answer. Alternatively, a Bloom filter can be built for each leaf page of a disk component. In this design, a point lookup query can first search the non-leaf pages of a B^+-tree to locate the leaf page, where the non-leaf pages are assumed to be small enough to be cached, and then check the associated Bloom filter before fetching the leaf page to reduce disk I/Os. Note that the false positives reported by a Bloom filter do not impact the correctness of a query, but a query may waste some I/O searching for non-existent keys. The false positive rate of a Bloom filter can be computed as $(1 - e^{-kn/m})^k$, where k is the number of hash functions, n is the number of keys, and m is the total number of bits [29]. Furthermore, the optimal number of hash functions that minimizes the false positive rate is $k = \frac{m}{n} ln2$. In practice, most systems typically use 10 bits/key as a default configuration, which gives a 1% false positive rate. Since Bloom filters are very small and can often be cached in memory, the number of disk I/Os for point lookups is greatly reduced by their use.

Partitioning. Another commonly adopted optimization is to range-partition the disk components of LSM-trees into multiple (usually fixed-size) small partitions. To minimize the potential confusion caused by different terminologies, we use the term *SSTable* to denote such a partition, following the terminology from LevelDB [6]. This optimization has several advantages. First, partitioning breaks a large component merge operation into multiple smaller ones, bounding the processing time of each merge operation as well as the temporary disk space needed to create new components. Moreover, partitioning can optimize for workloads with sequentially created keys or skewed updates by only merging components with overlapping key ranges. For sequentially created keys, essentially no merge is performed since there are no components with overlapping key ranges. For skewed updates, the merge frequency of the components with cold update ranges can be greatly reduced.

It should be noted that partitioning is orthogonal to merge policies; both leveling and tiering (as well as other emerging merge policies) can be adapted to support partitioning. To the best of our knowledge, only the partitioned leveling policy has been fully implemented by industrial LSM-

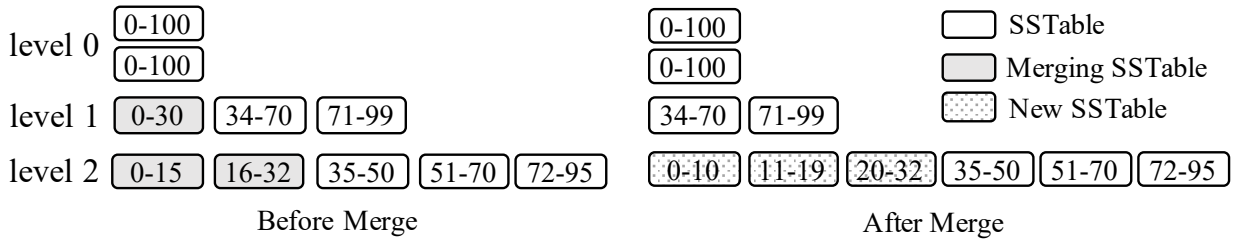


Figure 2.2: Partitioned Leveling Merge Policy

based storage systems, such as LevelDB [6] and RocksDB [10]. Some recent papers [20, 86, 95, 124, 128] have proposed various forms of a partitioned tiering merge policy to achieve better write performance[2].

In the partitioned leveling merge policy, pioneered by LevelDB [6], the disk component at each level is range-partitioned into multiple fixed-size SSTables, as shown in Figure 2.2. Each SSTable is labeled with its key range in the figure. Note that the disk components at level 0 are not partitioned since they are directly flushed from memory. This design can also help the system to absorb write bursts since it can tolerate multiple unpartitioned components at level 0. To merge an SSTable from level L into level $L+1$, all of its overlapping SSTables at level $L+1$ are selected, and these SSTables are merged with it to produce new SSTables still at level $L+1$. For example, in the figure, the SSTable labeled 0-30 at level 1 is merged with the SSTables labeled 0-15 and 16-32 at level 2. This merge operation produces new SSTables labeled 0-10, 11-19, and 20-32 at level 2, and the old SSTables will then be garbage-collected. Different policies can be used to select which SSTable to merge next at each level. For example, LevelDB uses a round-robin policy (to minimize the total write cost).

The partitioning optimization can also be applied to the tiering merge policy. However, one issue in doing so is that each level can contain multiple SSTables with overlapping key ranges. These SSTables must be ordered properly based on their recency to ensure correctness. Two possible schemes can be used to organize the SSTables at each level, namely vertical grouping and horizon-

[2]RocksDB supports a limited form of a partitioned tiering merge policy to bound the maximum size of each SSTable due to its internal restrictions. However, the disk space may still be doubled temporarily during large merges.

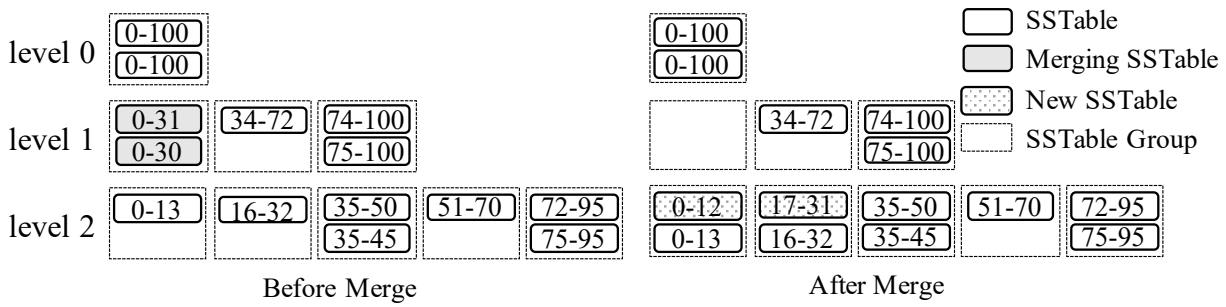


Figure 2.3: Partitioned Tiering with Vertical Grouping

tal grouping. In both schemes, the SSTables at each level are organized into groups. The vertical grouping scheme groups SSTables with overlapping key ranges together so that the groups have disjoint key ranges. Thus, it can be viewed as an extension of partitioned leveling to support tiering. Alternatively, under the horizontal grouping scheme, each logical disk component, which is range-partitioned into a set of SSTables, serves as a group directly. This allows a disk component to be formed incrementally based on the unit of SSTables. We will discuss these two schemes in detail below.

An example of the vertical grouping scheme is shown in Figure 2.3. In this scheme, SSTables with overlapping key ranges are grouped together so that the groups have disjoint key ranges. During a merge operation, all of the SSTables in a group are merged together to produce the resulting SSTables based on the key ranges of the overlapping groups at the next level, which are then added to these overlapping groups. For example in the figure, the SSTables labeled 0-30 and 0-31 at level 1 are merged together to produce the SSTables labeled 0-12 and 17-31, which are then added to the overlapping groups at level 2. Note the difference between the SSTables before and after this merge operation. Before the merge operation, the SSTables labeled 0-30 and 0-31 have overlapping key ranges and both must be examined together by a point lookup query. However, after the merge operation, the SSTables labeled 0-12 and 17-31 have disjoint key ranges and only one of them needs to be examined by a point lookup query. It should also be noted that under this scheme SSTables are no longer fixed-size since they are produced based on the key ranges of the overlapping groups at the next level.

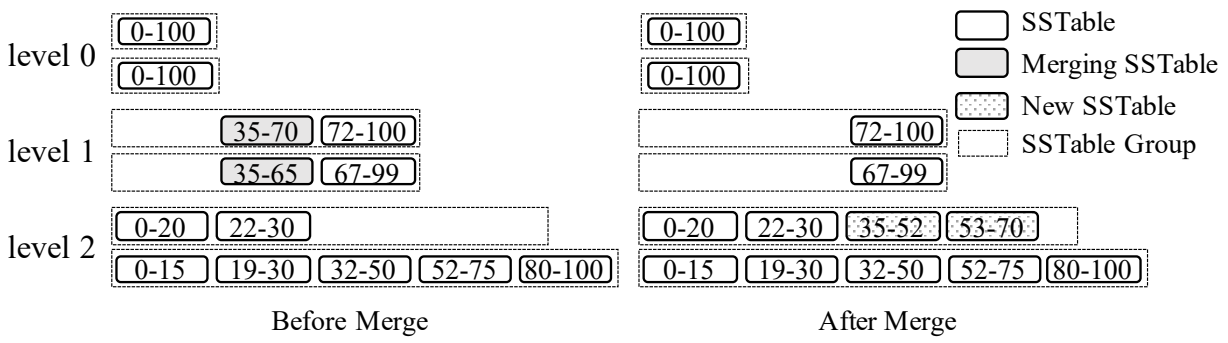


Figure 2.4: Partitioned Tiering with Horizontal Grouping

Figure 2.4 shows an example of the horizontal grouping scheme. In this scheme, each component, which is range-partitioned into a set of fixed-size SSTables, serves as a logical group directly. Each level L further maintains an active group, which is also the first group, to receive new SSTables merged from the previous level. This active group can be viewed as a partial component being formed by merging the components at level $L-1$ in the unpartitioned case. A merge operation selects the SSTables with overlapping key ranges from all of the groups at a level, and the resulting SSTables are added to the active group at the next level. For example in the figure, the SSTables labeled 35-70 and 35-65 at level 1 are merged together, and the resulting SSTables labeled 35-52 and 53-70 are added to the first group at level 2. However, although SSTables are fixed-size under the horizontal grouping scheme, it is still possible that one SSTable from a group may overlap a large number of SSTables in the remaining groups.

Concurrency Control and Recovery

We now briefly discuss the concurrency control and recovery techniques used by today's LSM-tree implementations. For concurrency control, an LSM-tree needs to handle concurrent reads and writes and to take care of concurrent flush and merge operations. Ensuring correctness for concurrent reads and writes is a general requirement for access methods in a database system. Depending on the transactional isolation requirement, today's LSM-tree implementations either use a locking scheme [17] or a multi-version scheme [2, 5, 10]. A multi-version scheme works well with an

LSM-tree since obsolete versions of a key can be naturally garbage-collected during merges. Concurrent flush and merge operations, however, are unique to LSM-trees. These operations modify the metadata of an LSM-tree, e.g., the list of active components. Thus, accesses to the component metadata must be properly synchronized. To prevent a component in use from being deleted, each component can maintain a reference counter. Before accessing the components of an LSM-tree, a query can first obtain a snapshot of active components and increment their in-use counters.

Since all writes are first appended into memory, write-ahead logging (WAL) can be performed to ensure their durability. To simplify the recovery process, existing systems typically employ a no-steal buffer management policy [56]. That is, a memory component can only be flushed when all active write transactions have terminated. During recovery for an LSM-tree, the transaction log is replayed to redo all successful transactions, but no undo is needed due to the no-steal policy. Meanwhile, the list of active disk components must also be recovered in the event of a crash. For unpartitioned LSM-trees, this can be accomplished by adding a pair of timestamps to each disk component that indicate the range of timestamps of the stored entries. This timestamp can be simply generated using local wall-clock time or a monotonic sequence number. To reconstruct the component list, the recovery process can simply find all components with disjoint timestamps. In the event that multiple components have overlapping timestamps, the component with the largest timestamp range is chosen and the rest can simply be deleted since they will have been merged to form the selected component. For partitioned LSM-trees, this timestamp-based approach does not work anymore since each component is further range-partitioned. To address this, a typical approach, used in LevelDB [6] and RocksDB [10], is to maintain a separate metadata log to store all changes to the structural metadata, such as adding or deleting SSTables. The state of the LSM-tree structure can then be reconstructed by replaying the metadata log during recovery.

2.2.3 Cost Analysis

To help understand the performance trade-offs of LSM-trees, we can turn to the cost analysis of writes, point lookups, range queries, and space amplification presented in [79]. The cost of writes and queries is measured by counting the number of disk I/Os per operation. This analysis considers an unpartitioned LSM-tree and represents a worst-case cost.

Let the size ratio of a given LSM-tree be T, and suppose that the LSM-tree contains L levels. In practice, for a stable LSM-tree where the volume of inserts equals the volume of deletes, L remains static. Let B denote the page size, that is, the number of entries that each data page can store, and let P denote the number of pages of a memory component. As a result, a memory component will contain at most $B \cdot P$ entries, and level i ($i \geq 0$)will contain at most $T^{i+1} \cdot B \cdot P$ entries. Given N total entries, the largest level contains approximately $N \cdot \frac{T}{T+1}$ entries since it is T times larger than the previous level. Thus, the number of levels for N entries can be approximated as $L = \lceil \log_T (\frac{N}{B \cdot P} \cdot \frac{T}{T+1}) \rceil$.

The write cost, which is also referred to as write amplification in the literature, measures the amortized I/O cost of inserting an entry into an LSM-tree. It should be noted that this cost measures the overall I/O cost for this entry to be merged into the largest level since inserting an entry into memory does not incur any disk I/O. For leveling, a component at each level will be merged $T-1$ times until it fills up and is pushed to the next level. For tiering, multiple components at each level are merged only once and are pushed to the next level directly. Since each disk page contains B entries, the write cost for each entry will be $O(T \cdot \frac{L}{B})$ for leveling and $O(\frac{L}{B})$ for tiering.

The I/O cost of a query depends on the number of components in an LSM-tree. Without Bloom filters, the I/O cost of a point lookup will be $O(L)$ for leveling and $O(T \cdot L)$ for tiering. However, Bloom filters can greatly improve the point lookup cost. For a zero-result point lookup, i.e., for a search for a non-existent key, all disk I/Os are caused by Bloom filter false positives. Suppose all Bloom filters have M bits in total and have the same false positive rate across all levels. With

N total keys, each Bloom filter has a false positive rate of $O(e^{-\frac{M}{N}})$ [29]. Thus, the I/O cost of a zero-result point lookup will be $O(L \cdot e^{-\frac{M}{N}})$ for leveling and $O(T \cdot L \cdot e^{-\frac{M}{N}})$ for tiering. To search for an existing unique key, at least one I/O must be performed to fetch the entry. Given that in practice the Bloom filter false positive rate is much smaller than 1, the successful point lookup I/O cost for both leveling and tiering will be $O(1)$.

The I/O cost of a range query depends on the query selectivity. Let s be the number of unique keys accessed by a range query. A range query can be considered to be *long* if $\frac{s}{B} > 2 \cdot L$, otherwise it is *short* [41, 43]. The distinction is that the I/O cost of a long range query will be dominated by the largest level since the largest level contains most of the data. In contrast, the I/O cost of a short range query will derive (almost) equally from all levels since the query must issue one I/O to each disk component. Thus, the I/O cost of a long range query will be $O(\frac{s}{B})$ for leveling and $O(T \cdot \frac{s}{B})$ for tiering. For a short range query, the I/O cost will be $O(L)$ for leveling and $O(T \cdot L)$ for tiering.

Finally, let us examine the space amplification of an LSM-tree, which is defined as the overall number of entries divided by the number of unique entries. For leveling, the worst case occurs when all of the data at the first $L-1$ levels, which contain approximately $\frac{1}{T}$ of the total data, are updates to the entries at the largest level. Thus, the worst case space amplification for leveling is $O(\frac{T+1}{T})$. For tiering, the worst case happens when all of the components at the largest level contain exactly the same set of keys. As a result, the worst case space amplification for tiering will be $O(T)$. In practice, the space amplification is an important factor to consider when deploying storage systems [49], as it directly impacts the storage cost for a given workload.

The cost complexity of LSM-trees is summarized in Table 2.1. Note how the size ratio T impacts

Table 2.1: Summary of Cost Complexity of LSM-trees

Merge Policy	Write	Point Lookup (Zero-Result / Non-Zero-Result)	Short Range Query	Long Range Query	Space Amplification
Leveling	$O(T \cdot \frac{L}{B})$	$O(L \cdot e^{-\frac{M}{N}})$ / $O(1)$	$O(L)$	$O(\frac{s}{B})$	$O(\frac{T+1}{T})$
Tiering	$O(\frac{L}{B})$	$O(T \cdot L \cdot e^{-\frac{M}{N}})$ / $O(1)$	$O(T \cdot L)$	$O(T \cdot \frac{s}{B})$	$O(T)$

the performance of leveling and tiering differently. In general, leveling is optimized for query performance and space utilization by maintaining one component per level. However, components must be merged more frequently, which will incur a higher write cost by a factor of T. In contrast, tiering is optimized for write performance by maintaining up to T components at each level. This, however, will decrease query performance and worsen space utilization by a factor of T. As one can see, the LSM-tree is highly tunable. For example, by changing the merge policy from leveling to tiering, one can greatly improve write performance with only a small negative impact on point lookup queries due to the Bloom filters. However, range queries and space utilization will be significantly impacted.

Chapter 3

Related Work

In this chapter, we present a comprehensive survey of existing LSM-tree improvements. We first present a taxonomy for use in classifying the existing literature on improving LSM-trees. We then provide an in-depth survey of the LSM-tree literature that follows the structure of the proposed taxonomy.

3.1 A Taxonomy of LSM-tree Improvements

Despite the popularity of LSM-trees in modern NoSQL systems, the basic LSM-tree design suffers from various drawbacks and insufficiencies. We now identify the major issues of the basic LSM-tree design, and further present a taxonomy of LSM-tree improvements based on these drawbacks.

Write Amplification. Even though LSM-trees can provide much better write throughput than in-place update structures such as B^+-trees by reducing random I/Os, the leveling merge policy, which has been adopted by modern key-value stores such as LevelDB [6] and RocksDB [10], still incurs relatively high write amplification. High write amplification not only limits the write performance of an LSM-tree but also reduces the lifespan of SSDs due to frequent disk writes. A large body of

research has been conducted to reduce the write amplification of LSM-trees.

Merge Operations. Merge operations are critical to the performance of LSM-trees and must therefore be carefully implemented. Moreover, merge operations can have negative impacts on the system, including buffer cache misses after merges and write stalls during large merges. Several improvements have been proposed to optimize merge operations to address these problems.

Hardware. In order to maximize performance, LSM-trees must be carefully implemented to fully utilize the underling hardware platforms. The original LSM-tree has been designed for hard disks, with the goal being reducing random I/Os. In recent years, new hardware platforms have presented new opportunities for database systems to achieve better performance. A significant body of recent research has been devoted to improving LSM-trees to fully exploit the underling hardware platforms, including large memory, multi-core, SSD, and native storage.

Special Workloads. In addition to hardware opportunities, certain special workloads can also be considered to achieve better performance in those use cases. In this case, the basic LSM-tree implementation must be adapted and customized to exploit the unique characteristics exhibited by these special workloads.

Auto-Tuning. Based on the RUM conjecture [24], no access method can be read-optimal, write-optimal, and space-optimal at the same time. The tunability of LSM-trees is a promising solution to achieve optimal trade-offs for a given workload. However, LSM-trees can be hard to tune because of too many tuning knobs, such as memory allocation, merge policy, and size ratio etc. To address this issue, several auto-tuning techniques have been proposed in the literature.

Query Processing. Even though the LSM-tree is a write-optimized structure, efficient query processing is also an important consideration for database systems. Thus, various research efforts have been made to optimize the query processing aspect of LSM-trees, including secondary indexing, range filtering, and exploiting LSM lifecycles to perform additional work for better query performance.

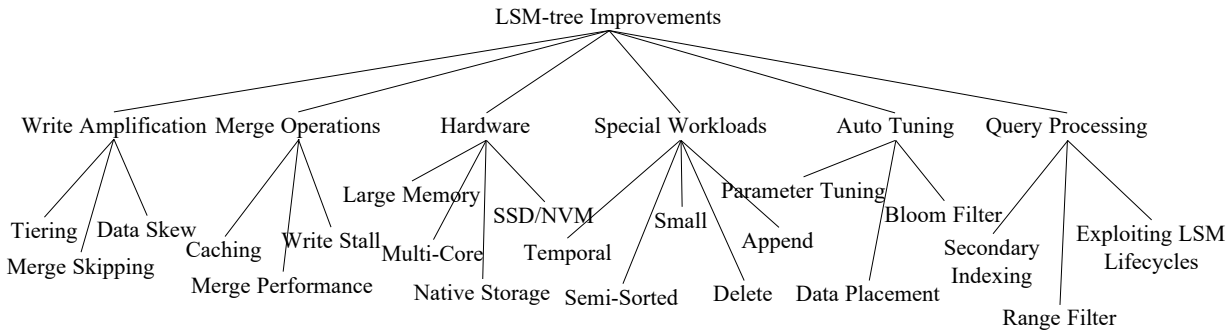


Figure 3.1: Taxonomy of LSM-tree Improvements

Based on these major issues of the basic LSM-tree design, we present a taxonomy of LSM-tree improvements, shown in Figure 3.1, to highlight the specific aspects that the existing research efforts try to optimize. Given this taxonomy, Table 3.1 further classifies the LSM-tree improvements in terms of each improvement's primary and secondary concerns (♣ denotes the primary category and △ denotes the secondary categories). With this taxonomy and classification in hand, we now proceed to examine each improvement in more depth.

3.2 Reducing Write Amplification

In this section, we review the improvements in the literature that aim to reduce the write amplification of LSM-trees. Most of these improvements are based on tiering since it has much better write performance than leveling. Other proposed improvements have developed new techniques to perform merge skipping or to exploit data skews.

3.2.1 Tiering

One way to optimize write amplification is to apply tiering since it has much lower write amplification than leveling. However, recall from Section 2.2.3 that this will lead to worse query performance and space utilization. The improvements in this category can all be viewed as some variants

Table 3.1: Classification of LSM-tree Improvements

Publication	Write Amplification	Merge Operations	Hardware	Special Workloads	Auto Tuning	Secondary Indexing
WB-tree [20]	♣					
LWC-tree [127]	♣					
PebblesDB [95]	♣					
SifrDB [86]	♣	△	△			
Skip-tree [130]	♣					
TRIAD [26]	♣					
VT-tree [106]		♣				
Zhang et al. [133]		♣				
Ahmad et al. [15]		♣				
LSbM-tree [113, 114]		♣				
bLSM [103]		♣				
FloDB [25]	△		♣			
Accordion [30]	△		♣			
cLSM [53]			♣			
FD-tree [69]			♣			
FD+tree [115]		△	♣			
MaSM [23]	△		♣	△		
WiscKey [75]	△		♣			
HashKV [71]	△		♣			
Kreon [91]	△		♣			
NoveLSM [64]			♣			
LDS [85]			♣			
LOCS [121]			♣			
NoFTL-KV [120]			♣			
LHAM [88]				♣		
LSM-trie [124]	△			♣		
SlimDB [98]	△			♣		
Mathieu et al. [84]	△			♣		
Lethe [101]				♣	△	
Lim et al. [72]					♣	
Monkey [41, 42]					♣	
Dostoevsky [43]	△				♣	
LSM-Bush [44]				△	♣	
Thonangi and Yang [116]	△				♣	
ElasticBF [70]					♣	
Mutant [129]					♣	
Kim et al. [67]						♣
Qader et al. [94]						♣
Diff-Index [111]						♣
DELI [112]						♣
Rosetta [82]						♣
Alsubaiee et al. [19]					△	♣
Statistics Collection [13]						♣
Tuple Compaction [16]						♣

of the partitioned tiering design with vertical or horizontal grouping discussed in Section 2.2.2. Here we will mainly discuss the modifications made by these improvements.

The WriteBuffer (WB) Tree [20] can be viewed as a variant of the partitioned tiering design with vertical grouping. It has made the following modifications. First, it relies on hash-partitioning to achieve workload balance so that each SSTable group roughly stores the same amount of data. Furthermore, it organizes SSTable groups into a B^+-tree-like structure to enable self-balancing to minimize the total number of levels. Specifically, each SSTable group is treated like a node in a B^+-tree. When a non-leaf node becomes full with T SSTables, these T SSTables are merged together to form new SSTables that are added into its child nodes. When a leaf node becomes full with T SSTables, it is split into two leaf nodes by merging all of its SSTables into two leaf nodes with smaller key ranges so that each new node receives about $T/2$ SSTables.

The light-weight compaction tree (LWC-tree) [127] adopts a similar partitioned tiering design with vertical grouping. It further presents a method to achieve workload balancing of SSTable groups. Recall that under the vertical grouping scheme, SSTables are no longer strictly fixed-size since they are produced based on the key ranges of the overlapping groups at the next level instead of based on their sizes. In the LWC-tree, if a group contains too many entries, it will shrink the key range of this group after the group has been merged (now temporarily empty) and will widen the key ranges of its sibling groups accordingly.

PebblesDB [95] also adopts a partitioned tiering design with vertical grouping. The major difference is that it determines the key ranges of SSTable groups using the idea of guards as inspired by the skip-list [92]. Guards, which are the key ranges of SSTable groups, are selected probabilistically based on inserted keys to achieve workload balance. Once a guard is selected, it is applied lazily during the next merge. PebblesDB further performs parallel seeks of SSTables to improve range query performance.

As one can see, the structures described above all share a similar high-level design based on par-

titioned tiering with vertical grouping. They mainly differ in how workload balancing of SSTable groups is performed. For example, the WB-tree [20] relies on hashing, but doing so gives up the ability of supporting range queries. The LWC-tree [127] dynamically shrinks the key ranges of dense SSTable groups, while PebblesDB [95] relies on probabilistically selected guards. It is not clear how skewed SSTable groups would impact the performance of these structures, and future research is needed to understand this problem and evaluate these workload balancing strategies.

The partitioned tiering design with horizontal grouping has been adopted by SifrDB [86]. SifrDB also proposes an early-cleaning technique to reduce disk space utilization during merges. During a merge operation, SifrDB incrementally activates newly produced SSTables and deactivates the old SSTables. SifrDB further exploits I/O parallelism to speedup query performance by examining multiple SSTables in parallel.

3.2.2 Merge Skipping

The skip-tree [130] proposes a merge skipping idea to improve write performance. The observation is that each entry must be merged from level 0 down to the largest level. If some entries can be directly pushed to a higher level by skipping some level-by-level merges, then the total write cost will be reduced. As shown in Figure 3.2, during a merge at level L, the skip-tree directly pushes some keys to a mutable buffer at level $L+K$ so that some level-by-level merges can be skipped. Meanwhile, the skipped entries in the mutable buffer will be merged with the SSTables at level $L+K$ during subsequent merges. To ensure correctness, a key from level L can be pushed to level $L+K$ only if this key does not appear in any of the intermediate levels $L+1,\cdots,L+K-1$. This condition can be tested efficiently by checking the Bloom filters of the intermediate levels. The skip-tree further performs write-ahead logging to ensure durability of the entries stored in the mutable buffer. To reduce the logging overhead, the skip-tree only logs the key plus the ID of the original SSTable and prevents an SSTable from being deleted if it is referenced by any

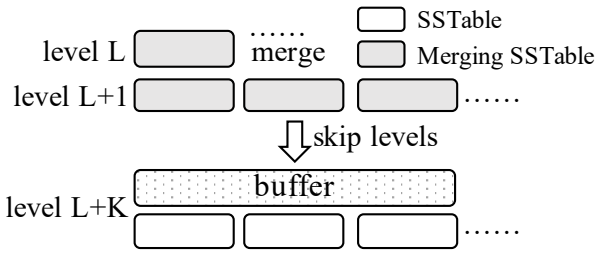


Figure 3.2: Skip-tree Example

key in the buffer. Although merge skipping is an interesting idea to reduce write amplification, it introduces non-trivial implementation complexity to manage the mutable buffers. Moreover, since merge skipping essentially reduces some merges at the intermediate levels, it is not clear how the skip-tree would compare against a well-tuned LSM-tree by reducing the size ratio.

3.2.3 Exploiting Data Skew

TRIAD [26] reduces write amplification for skewed update workloads where some hot keys are updated frequently. The basic idea is to separate hot keys from cold keys in the memory component so that only cold keys are flushed to disk. As a result, when hot keys are updated, old versions can be discarded directly without writing them to disk. Even though hot keys are not flushed to disk, they are periodically copied to a new transaction log so that the old transaction log can be reclaimed. TRIAD also reduces write amplification by delaying merges at level 0 until level 0 contains multiple SSTables. Finally, it presents an optimization that avoids creating new disk components after flushes. Instead, the transaction log itself is used as a disk component and an index structure is built on top of it to improve lookup performance. However, range query performance will still be negatively impacted since entries are not sorted in the log.

3.2.4 Summary

Tiering has been widely used to improve the write performance of LSM-trees, but this will decrease query performance and space utilization, as discussed in Section 2.2.3. The existing tiering-based improvements mainly differ in how SSTables are managed, either by vertical grouping [20, 95, 127] or horizontal grouping [86]. It is not clear how these different grouping schemes impact system performance and it would be useful as future work to study and evaluate their impact. The skip-tree [130] and TRIAD [26] propose several new ideas to improve write performance, ideas that are orthogonal to tiering. However, these optimizations bring non-trivial implementation complexity to real systems, such as the mutable buffers introduced by the skip-tree and the use of transaction logs as flushed components by TRIAD.

All of the improvements in this category, as well as some improvements in the later sections, have claimed that they can greatly improve the write performance of LSM-trees, but their performance evaluations have often failed to consider the tunability of LSM-trees. That is, these improvements have mainly been evaluated against a default (untuned) configuration of LevelDB or RocksDB, which use the leveling merge policy with a size ratio of 10. It is not clear how these improvements would compare against well-tuned LSM-trees. To address this, one possible solution would be to tune RocksDB to achieve a similar write throughput to the proposed improvements by changing the size ratio or by adopting the tiering merge policy and then evaluating how these improvements can improve query performance and space amplification. Moreover, these improvements have primarily focused on query performance; space amplification has often been neglected. It would be a useful experimental study to fully evaluate these improvements against well-tuned baseline LSM-trees to evaluate their actual usefulness. We also hope that this situation can be avoided in future research by considering the tunability of LSM-trees when evaluating the proposed improvements.

3.3 Optimizing Merge Operations

Next we review some existing work that improves the implementation of merge operations, including improving merge performance, minimizing buffer cache misses, and eliminating write stalls.

3.3.1 Improving Merge Performance

The VT-tree [106] presents a stitching operation to improve merge performance. The basic idea is that when merging multiple SSTables, if the key range of a page from an input SSTable does not overlap the key ranges of any pages from other SSTables, then this page can be simply pointed to by the resulting SSTable without reading and copying it again. Even though stitching improves merge performance for certain workloads, it has a number of drawbacks. First, it can cause fragmentation since pages are no longer continuously stored on disk. To alleviate this problem, the VT-tree introduces a stitching threshold K so that a stitching operation is triggered only when there are at least K continuous pages from an input SSTable. Moreover, since the keys in stitched pages are not scanned during a merge operation, a Bloom filter cannot be produced. To address this issue, the VT-tree uses quotient filters [27] since multiple quotient filters can be combined directly without accessing the original keys.

Zhang et al. [133] proposed a pipelined merge implementation to better utilize CPU and I/O parallelism to improve merge performance. The key observation is that a merge operation contains multiple phases, including the read phase, merge-sort phase, and write phase. The read phase reads pages from input SSTables, which will then be merge-sorted to produce new pages during the merge-sort phase. Finally, the new pages will be written to disk during the write phase. Thus, the read phase and write phase are I/O heavy while the merge-sort phase is CPU heavy. To better utilize CPU and I/O parallelism, the proposed approach pipelines the execution of these three phases, as illustrated by Figure 3.3. In this example, after the first input page has been read, this ap-

proach continues reading the second input page (using disk) and the first page can be merge-sorted (using CPU).

3.3.2 Reducing Buffer Cache Misses

Merge operations can interfere with the caching behavior of a system. After a new component is enabled, queries may experience a large number of buffer cache misses since the new component has not been cached yet. A simple write-through cache maintenance policy cannot solve this problem. If all of the pages of the new component were cached during a merge operation, a lot of other working pages would be evicted, which will again cause buffer cache misses.

Ahmad et al. [15] conducted an experimental study of the impact of merge operations on system performance. They found that merge operations consume a large number of CPU and I/O resources and impose a high overhead on query response times. To address this, this work proposed to offload large merges to remote servers to minimize their impact. After a merge operation is completed, a smart cache warmup algorithm is used to fetch the new component incrementally to minimize buffer cache misses. The idea is to switch to the new component incrementally, chunk by chunk, to smoothly redirect incoming queries from the old components to the new component. As a result, the burst of buffer cache misses is decomposed into a large number of smaller ones, minimizing the negative impact of component switching on query performance.

One limitation of the approach proposed by Ahmad et al. [15] is that merge operations must be

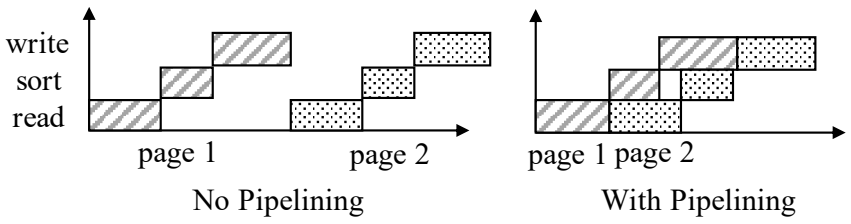


Figure 3.3: Pipelined Merge Example

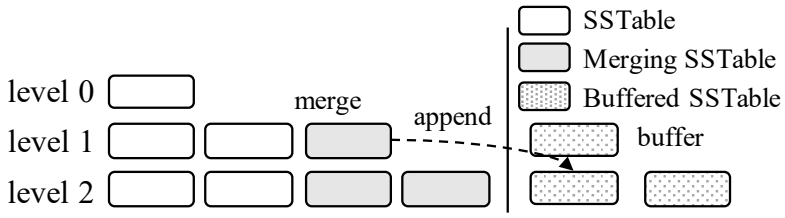


Figure 3.4: LSbM-tree Example

offloaded to separate servers. The incremental warmup algorithm alone was subsequently found to be insufficient due to contention between the newly produced pages and the existing hot pages [113, 114]. To address this limitation, the Log-Structured buffered Merge tree (LSbM-tree) [113, 114] proposes an alternative approach. As illustrated by Figure 3.4, after an SSTable at level L is merged into level $L+1$, the old SSTables at level L is appended to a buffer associated with level $L+1$ instead of being deleted immediately. Note that there is no need to add old SSTables at level $L+1$ into the buffer, as the SSTables at $L+1$ all come from level L and the entries of these old SSTables will have already been added to the buffer before. The buffered SSTables are searched by queries as well to minimize buffer cache misses, and they are deleted gradually based on their access frequency. This approach does not incur any extra disk I/O during a merge operation since it only delays the deletion of the old SSTables. However, this approach is mainly effective for skewed workloads where only a small range of keys are frequently accessed. It can introduce extra overhead for queries accessing cold data that are not cached, especially for range queries since they cannot benefit from Bloom filters.

3.3.3 Minimizing Write Stalls

Although the LSM-tree offers a much higher write throughput compared to traditional B^+-trees, it often exhibits write stalls and unpredictable write latencies since heavy operations such as flushes and merges run in the background.

bLSM [103] proposes a spring-and-gear merge scheduler to minimize write stalls for the unparti-

tioned leveling merge policy. Its basic idea is to tolerate an extra component at each level so that merges at different levels can proceed in parallel. Furthermore, the merge scheduler controls the progress of merge operations to ensure that level L produces a new component at level $L+1$ only after the previous merge operation at level $L+1$ has completed. This eventually cascades to limit the maximum write speed at the memory component and eliminates large write stalls. However, bLSM only bounds the maximum latency of writing to memory components while the queuing latency, which is often a major source of performance variability, is ignored.

3.3.4 Summary

The improvements in this category optimize the implementation of merge operations in terms of performance, buffer cache misses, and write stalls. To speedup merge operations, the VT-tree [106] introduces the stitching operation that avoids copying input pages if applicable. However, this may cause fragmentation, which is undesirable for hard disks. Moreover, this optimization is incompatible with Bloom filters, which are widely used in modern LSM-tree implementations. The pipelined merge implementation [133] improves merge performance by exploiting CPU and I/O parallelism. It should be noted that many LSM-based storage systems have already implemented some form of pipelining by exploiting disk read-ahead and write-behind.

Ahmed et al. [15] and the LSbM-tree [113, 114] present two alternative methods to alleviating buffer cache misses caused by merges. However, both approaches appear to have certain limitations. The approach proposed by Ahmed et al. [15] requires dedicated servers to perform merges, while the LSbM-tree [113, 114] delays the deletion of old components that could negatively impact queries accessing cold data. Write stalls are a unique problem of LSM-trees due to its out-of-place update nature. There has been several research effort [103] to address this problem, but it remains an open problem of bounding tail latencies for LSM-trees.

3.4 Hardware Opportunities

We now review the LSM-tree improvements proposed for different hardware platforms, including large memory, multi-core, SSD/NVM, and native storage. A general paradigm of these improvements is to modify the basic design of LSM-trees to fully exploit the unique features provided by the target hardware platform to achieve better performance.

3.4.1 Large Memory

It is beneficial for LSM-trees to have large memory components to reduce the total number of levels, as this will improve both write performance and query performance [80]. Thus, it is important to utilize large memory efficiently to achieve better performance.

FloDB [25] presents a two-layer design to manage large memory components. The top level is a small concurrent hash table to support fast writes, and the bottom level is a large skip-list to support range queries efficiently. When the hash table is full, its entries are efficiently migrated into the skip-list using a batched algorithm. By limiting random writes to a small memory area, this design significantly improves the in-memory write throughput. To support range queries, FloDB requires that a range query must wait for the hash table to be drained so that the skip-list alone can be searched to answer the query. However, FloDB suffers from two major problems. First, it is not efficient for workloads containing both writes and range queries due to their contention. Second, the skip-list may have a large memory footprint and lead to lower memory utilization.

To address the drawbacks of FloDB, Accordion [30] uses a multi-layer approach to manage its large memory components. In this design (Figure 3.5), there is a small mutable memory component in the top level to process writes. When the mutable memory component is full, instead of being flushed to disk, it is simply flushed into a (more compact) immutable memory component via an in-memory flush operation. Similarly, such immutable memory components can be merged

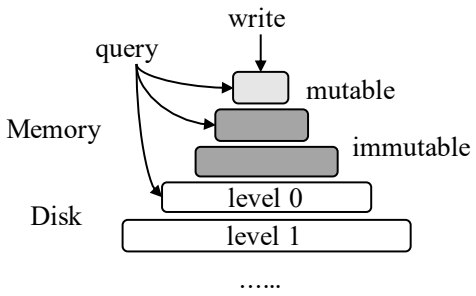


Figure 3.5: Accordion's Multi-Layer Structure

via in-memory merge operations to improve query performance and reclaim space occupied by obsolete entries. Note that in-memory flush and merge operations do not involve any disk I/O, which reduces the overall disk I/O cost by leveraging large memory.

3.4.2 Multi-Core

cLSM [53] optimizes for multi-core machines and presents new concurrency control algorithms for various LSM-tree operations. It organizes LSM components into a concurrent linked list to minimize blocking caused by synchronization. Flush and merge operations are carefully designed so that they only result in atomic modifications to the linked list that will never block queries. When a memory component becomes full, a new memory component is allocated while the old one will be flushed. To avoid writers inserting into the old memory component, a writer acquires a shared lock before modifications and the flush thread acquires an exclusive lock before flushes. cLSM also supports snapshot scans via multi-versioning and atomic read-modify-write operations using an optimistic concurrency control approach that exploits the fact that all writes, and thus all conflicts, involve the memory component.

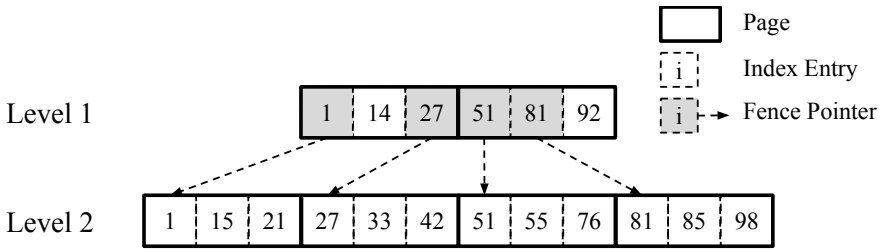


Figure 3.6: Example FD-tree Structure

3.4.3 SSD/NVM

Different from traditional hard disks, which only support efficient sequential I/Os, new storage devices such as solid-state drives (SSDs) and non-volatile memories (NVMs) support efficient random I/Os as well. NVMs further provide efficient byte-addressable random accesses with persistence guarantees.

The FD-tree [69] uses a similar design to LSM-trees to reduce random writes on SSDs. One major difference is that the FD-tree exploits fractional cascading [37] to improve query performance instead of Bloom filters. For the component at each level, the FD-tree additionally stores fence pointers that point to each page at the next level. For example in Figure 3.6, the pages at level 2 are pointed at by fence pointers with keys 1, 27, 51, 81 at level 1. After performing a binary search at level 0, a query can follow these fence pointers to traverse all of the levels. However, this design introduces additional complexity to merges. When the component at level L is merged into level $L+1$, all of the previous levels 0 to $L-1$ must be merged as well to rebuild the fence pointers. Moreover, a point lookup still needs to perform disk I/Os when searching for non-existent keys, which can be mostly avoided by using Bloom filters. For these reasons, modern LSM-tree implementations prefer Bloom filters rather than fractional cascading[1].

The FD+tree [115] improves the merge process of the FD-tree [69]. In the FD-tree, when a merge happens from level 0 to level L, new components at levels 0 to L must be created, which will

[1]RocksDB [10] supports a limited form of fractional cascading by maintaining the set of overlapping SSTables at the adjacent next level for each SSTable. These pointers are used to narrow down the search range when locating specific SSTables during point lookups.

temporarily double the disk space. To address this, during a merge operation, the FD+tree incrementally activates the new components and reclaims pages from the old components that are not used by any active queries.

MaSM (materialized sort-merge) [23] is designed for supporting efficient updates for data warehousing workloads by exploiting SSDs. MaSM first buffers all updates into an SSD. It uses the tiering merge policy to merge intermediate components with low write amplification. The updates are then merged back to the base data, which resides in the hard disk. MaSM can be viewed as a simplified form of the lazy leveling merge policy proposed by Dostoevsky [43], as we will see later in this survey. Moreover, since MaSM mainly targets long range queries to support data warehousing workloads, the overhead introduced by intermediate components stored in SSDs is negligible compared to the cost of accessing the base data. This enables MaSM to only incur a small overhead on queries with concurrent updates.

Since SSDs support efficient random reads, separating values from keys becomes a viable solution to improve the write performance of LSM-trees. By transforming an LSM-tree into a log-structured store, this design can significantly improve write performance since the write cost of a log-structured store is much smaller than that of an LSM-tree [73, 99]. However, this design will significantly impact range query performance because values are not sorted anymore. This approach was first implemented by WiscKey [75] and subsequently adopted by HashKV [71] and SifrDB [86]. As shown in Figure 3.7, WiscKey [75] stores key-value pairs into an append-only log and the LSM-tree simply serves as a primary index that maps each key to its location in the log. Only keys are merged while the value log is garbage-collected to reclaim the storage space. In WiscKey, garbage-collection is performed in three steps. First, WiscKey scans the log tail and validates each entry by performing point lookups against the LSM-tree to find out whether the location of each key has changed or not. Second, valid entries, whose locations have not changed, are then appended to the log and their locations are updated in the LSM-tree as well. Finally, the log tail is truncated to reclaim the storage space.

HashKV [71] introduces a more efficient approach to garbage-collect obsolete values. The basic idea is to hash-partition the value log into multiple partitions based on keys and to garbage-collect each partition independently. In order to garbage-collect a partition, HashKV performs a group-by operation on the keys to find the latest value for each key. Valid key-value pairs are added to a new log and their locations are then updated in the LSM-tree. HashKV further stores cold entries separately so that they can be garbage-collected less frequently.

Kreon [91] exploits memory-mapped I/O to reduce CPU overhead by avoiding unnecessary data copying. It implements a customized memory-mapped I/O manager in the Linux kernel to control cache replacement and to enable blind writes. To improve range query performance, Kreon reorganizes data during query processing by storing the accessed key-value pairs together in a new place.

NoveLSM [64] is an implementation of LSM-trees on NVMs. NoveLSM adds an NVM-based memory component to serve writes when the DRAM memory component is full so that writes can still proceed without being stalled. It further optimizes the write performance of the NVM memory component by skipping logging since NVM itself provides persistence. Finally, it exploits I/O parallelism to search multiple levels concurrently to reduce lookup latency.

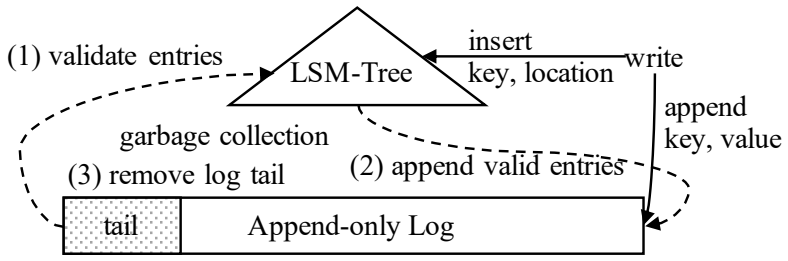


Figure 3.7: Example WiscKey Structure

3.4.4 Native Storage

Performing native management of storage devices has attracted the attention from the research community [47, 48, 105] due to the recent advance in the storage technologies. Here we review some research work that performs native storage management to optimize the performance of LSM-tree implementations.

The LSM-tree-based Direct Storage system (LDS) [85] bypasses the file system to better exploit the sequential and aggregated I/O patterns exhibited by LSM-trees. The on-disk layout of LDS contains three parts: chunks, a version log, and a backup log. Chunks store the disk components of the LSM-tree. The version log stores the metadata changes of the LSM-tree after each flush and merge. For example, a version log record can record the obsolete chunks and the new chunks resulting from a merge. The version log is regularly checkpointed to aggregate all changes so that the log can be truncated. Finally, the backup log provides durability for in-memory writes by write-ahead logging.

LOCS [121] is an implementation of the LSM-tree on open-channel SSDs. Open-channel SSDs expose internal I/O parallelism via an interface called channels, where each channel functions independently as a logical disk device. This allows applications to flexibly schedule disk writes to leverage the available I/O parallelism, but disk reads must be served by the same channel where the data is stored. To exploit this feature, LOCS dispatches disk writes due to flushes and merges to all channels using a least-weighted-queue-length policy to balance the total amount of work allocated to each channel. To further improve the I/O parallelism for partitioned LSM-trees, LOCS places SSTables from different levels with similar key ranges into different channels so that these SSTables can be read in parallel.

NoFTL-KV [120] proposes to extract the flash translation layer (FTL) from the storage device into the key-value store to gain direct control over the storage device. Traditionally, the FTL translates the logical block address to the physical block address to implement wear leveling, which improves

the lifespan of SSDs by distributing writes evenly to all blocks. NoFTL-KV argues for a number of advantages of extracting FTL, such as pushing tasks down to the storage device, performing more efficient data placement to exploit I/O parallelism, and integrating the garbage-collection process of the storage device with the merge process of LSM-trees to reduce write amplification.

3.4.5 Summary

In this subsection, we have reviewed the LSM-tree improvements exploiting hardware platforms, including large memory [25, 30], multi-core [53], SSD/NVM [23, 64, 69, 71, 75, 91, 115], and native storage [85, 121, 120]. To manage large memory components, both FloDB [25] and Accordion [30] take a multi-layer approach to limit random writes to a small memory area. The difference is that FloDB [25] only uses two layers, while Accordion [30] uses multiple layers to provide better concurrency and memory utilization. For multi-core machines, cLSM [53] presents a set of new concurrency control algorithms to improve concurrency.

A general theme of the improvements for SSD/NVM is to exploit the high random read throughput while reducing the write amplification of LSM-trees to improve the lifespan of these storage devices. The FD-tree [69] and its successor FD+tree [115] propose to use fractional cascading [37] to improve point lookup performance so that only one random I/O is needed for searching each component. However, today's implementations generally prefer Bloom filters since unnecessary I/Os can be mostly avoided by point lookups. Separating keys from values [71, 75, 91] can significantly improve the write performance of LSM-trees since only keys are merged. However, this leads to lower query performance and space utilization. Meanwhile, values must be garbage-collected separately to reclaim disk space, which is similar to the traditional log-structured file system design [99]. Finally, some recent work has proposed to perform native management of storage devices, including HDDs [85] and SSDs [120, 121], which can often bring large performance gains by exploiting the sequential and non-overwriting I/O patterns exhibited by LSM-trees.

3.5 Handling Special Workloads

We now review some existing LSM-tree improvements that target special workloads to achieve better performance. The considered special workloads include temporal data, small data, semi-sorted data, append-mostly workloads, and delete-heavy workloads.

The log-structured history access method (LHAM) [88] improves the original LSM-tree to more efficiently support temporal workloads. The key improvement made by LHAM is to attach a range of timestamps to each component to facilitate the processing of temporal queries by pruning irrelevant components. It further guarantees that the timestamp ranges of components are disjoint from one another. This is accomplished by modifying the rolling merge process to always merge the records with the oldest timestamps from a component C_i into C_{i+1}.

The LSM-trie [124] is an LSM-based hash index for managing a large number of key-value pairs where each key-value pair is small. It proposes a number of optimizations to reduce the metadata overhead. The LSM-trie adopts a partitioned tiering design to reduce write amplification. Instead of storing the key ranges of each SSTable directly, the LSM-trie organizes its SSTables using the prefix of their hash values to reduce the metadata overhead, as shown in Figure 3.8. The LSM-trie further eliminates the index page, instead assigning key-value pairs into fixed-size buckets based on their hash values. Overflow key-value pairs are assigned to underflow buckets and this information is recorded in a migration metadata table. The LSM-trie also builds a Bloom filter for each bucket. Since there are multiple SSTables in each group at a level, the LSM-trie clusters all Bloom filters of the same logical bucket of these SSTables together so that they can be fetched using a single I/O by a point lookup query. In general, the LSM-trie is mainly effective when the number of key-value pairs is so large that even the metadata, e.g., index pages and Bloom filters, cannot be totally cached. However, the LSM-trie only supports point lookups since its optimizations heavily depend on hashing.

SlimDB [98] targets semi-sorted data in which each key contains a prefix x and a suffix y. It

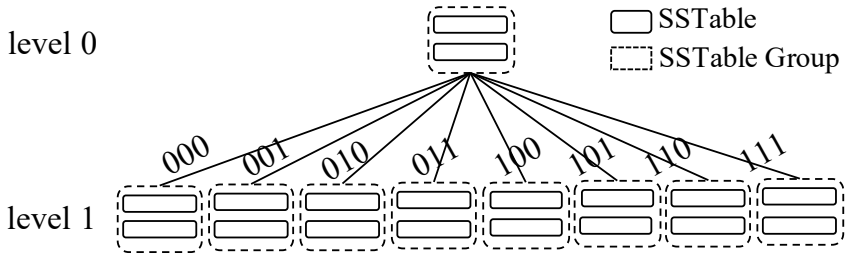


Figure 3.8: Example LSM-trie Structure

supports normal point lookups, given both the prefix and the suffix, as well as retrieving all the key-values pairs sharing the same prefix key x. To reduce write amplification, SlimDB adopts a hybrid structure with tiering on the lower levels and leveling on the higher levels. SlimDB further uses multi-level cuckoo filters [51] to improve point lookup performance for levels that use the tiering merge policy. At each level, a multi-level cuckoo filter maps each key to the ID of the SSTable where the latest version of the key is stored so that only one filter check is needed by a point lookup. To reduce the metadata overhead of SSTables, SlimDB uses a multi-level index structure as follows: It first maps each prefix key into a list of pages that contain this prefix key so that the key-value pairs can be retrieved efficiently given a prefix key. It then stores the range of suffix keys for each page to efficiently support point lookup queries based on both prefix and suffix keys.

Mathieu et al. [84] proposed two new merge policies optimized for append-mostly workloads with a bounded number of components. One problem of both leveling and tiering is that the number of levels depends on the total number of entries. Thus, with an append-mostly workload, where the amount of data keeps increasing, the total number of levels will be unbounded in order to achieve the write cost described in Section 2.2.3. To address this, this work studied the theoretical lower bound of the write cost of an online merge policy for an append-mostly workload given at most K components. It further proposed two merge policies, MinLatency and Binomial, to achieve this lower bound. These two policies have been experimentally evaluated by [83], which shows that the write cost can be greatly reduced under append-mostly workloads.

Lethe [101] presents two optimizations for delete-heavy workloads. First, it proposes a delete-aware merge policy to proactively merge anti-matter entries into the last level. This design not only bounds the maximum delete persistence latency, but also helps to quickly reclaim the disk space occupied by deleted entries. Lethe further proposes a key weaving storage layout to support efficient range deletes on a pre-defined secondary key. The basic idea is to divide each SSTable into a set of tiles, each of which contains a set of disk pages. Within each tile, records are sorted based on the specified secondary key. While within each page, records are still sorted based on the primary key. When performing a range delete on the secondary key, it is likely that many disk pages can be dropped in full, which will greatly improve the delete performance and allow the deleted pages to be reclaimed by the operating system immediately.

The improvements presented here each target a specialized workload. It should be noted that their optimizations may be useless or even inapplicable for general purpose workloads. For example, the LSM-trie [124] only supports point lookups, while SlimDB [98] only supports a limited form of range queries by fetching all values for a prefix key. The adoption of these optimizations should be chosen carefully based on the given workload.

3.6 Auto-Tuning

We now review some research efforts to develop auto-tuning techniques for the LSM-tree to reduce the tuning burden for the end-user. Some techniques perform co-tuning of all parameters to find an optimal design, while others focus on some specific aspect such as merge policies, Bloom filters, or data placement.

3.6.1 Parameter Tuning

Lim et al. [72] presented an analytical model that incorporates the key distribution to improve the cost estimation of LSM-tree operations and further used this model to tune the parameters of LSM-trees. The key insight is that the conventional worse-case analysis (Section 2.2.3) fails to take the key distribution into consideration. If a key is found to be deleted or updated during an early merge, it will not participate in future merges and thus its overall write cost will be reduced. The proposed model assumes a priori knowledge of the key distribution using a probability mass function $f_X(k)$ that measures the probability that a specific key k is written by a write request. Given p total write requests, the number of unique keys is estimated using its expectation as $Unique(p) = N - \sum_{k \in K} (1 - f_X(k))^p$, where N is the total number of unique keys and K is the total key space. Based on this formula, the total write cost for p writes can be computed by summing up the cost of all flushes and merges, except that duplicates keys, if any, are excluded from future merges. Finally, the cost model is used to find the optimal system parameters by minimizing the total write cost.

Monkey [41, 42] co-tunes the merge policy, size ratio, and memory allocation between memory components and Bloom filters to find an optimal LSM-tree design for a given workload. The first contribution of Monkey is to show that the usual Bloom filter memory allocation scheme, which allocates the same number of bits per key for all Bloom filters, results in sub-optimal performance. The intuition is that the T components at the last level, which contain most of the data, consume most of the Bloom filter memory but their Bloom filters can only save at most T disk I/Os for a point lookup. To minimize the overall false positive rates across all of the Bloom filters, Monkey analytically shows that more bits should be allocated to the components at the lower levels so that the Bloom filter false positive rates will be exponentially increasing. Under this scheme, the I/O cost of zero-result point lookup queries will be dominated by the last level, and the new I/O cost becomes $O(e^{-\frac{M}{N}})$ for leveling and $O(T \cdot e^{-\frac{M}{N}})$ for tiering. Monkey then finds an optimal LSM-tree design by maximizing the overall throughput using a cost model similar to the one in Section 2.2.3 considering the workload's mix of the various operations.

3.6.2 Tuning Merge Policies

Dostoevsky [43] shows that the existing merge policies, that is, tiering and leveling, are sub-optimal for certain workloads. The intuition is that for leveling, the cost of zero-result point lookups, long range queries, and space amplification are dominated by the largest level, but the write cost derives equally from all of the levels. To address this, Dostoevsky introduces a lazy-leveling merge policy that performs tiering at the lower levels but leveling at the largest level. Lazy-leveling has much better write cost than leveling, but has similar point lookup cost, long range query cost, and space amplification to leveling. It only has a worse short range query cost than leveling since the number of components is increased. Dostoevsky also proposes a hybrid policy that has at most Z components in the largest level and at most K components at each of the smaller levels, where Z and K are tunable. It then finds an optimal LSM-tree design for a given workload using a similar method as Monkey [41]. It is worth noting that the performance evaluation of Dostoevsky [43] is very thorough; it was performed against well-tuned LSM-trees to show that Dostoevsky strictly dominates the existing LSM-tree designs under certain workloads.

LSM-Bush [44] extends the design space of LSM merge policies for write and point lookup heavy workloads by assigning larger size ratios for smaller levels. The key intuition is that larger levels contribute the most to the point lookup cost and the Bloom filter space requirement of an LSM-tree, but each level contributes equally to the write cost. By using exponentially larger size ratios for smaller levels, the overall write cost of an LSM-tree can be reduced from $O(logN)$ to $O(loglogN)$, where N is the number of entries, without impacting the point lookup cost and Bloom filter space requirement.

Thonangi and Yang [116] formally studied the impact of partitioning on the write cost of LSM-trees. This work first proposed a ChooseBest policy that always selects an SSTable with the fewest overlapping SSTables at the next level to merge to bound the worst case merge cost. Although the ChooseBest policy outperforms the unpartitioned merge policy in terms of the overall write

cost, there are certain periods when the unpartitioned merge policy has a lower write cost since the current level becomes empty after a full merge, which reduces the future merge cost. To exploit this advantage of full merges, this work further proposed a mixed merge policy that selectively performs full merges or partitioned merges based on the relative size between adjacent levels and that dynamically learns these size thresholds to minimize the overall write cost for a given workload.

3.6.3 Dynamic Bloom Filter Memory Allocation

All of the existing LSM-tree implementations, even Monkey [41], adopt a static scheme to manage Bloom filter memory allocation. That is, once the Bloom filter is created for a component, its false positive rate remains unchanged. Instead, ElasticBF [70] dynamically adjusts the Bloom filter false positive rates based on the data hotness and access frequency to optimize read performance. Given a budget of k Bloom filter bits per key, ElasticBF constructs multiple smaller Bloom filters with k_1, $\cdots$, k_n bits so that $k_1 + \cdots + k_n = k$. When all of these Bloom filters are used together, they provide the same false positive rate as the original monolithic Bloom filter. ElasticBF then dynamically activates and deactivates these Bloom filters based on the access frequency to minimize the total amount of extra I/O. Their experiments reveal that ElasticBF is mainly effective when the overall Bloom filter memory is very limited, such as only 4 bits per key on average. In this case, the disk I/Os caused by the Bloom filter false positives will be dominant. When memory is relatively large and can accommodate more bits per key, such as 10, the benefit of ElasticBF becomes limited since the number of disk I/Os caused by false positives is much smaller than the number of actual disk I/Os to locate the keys.

3.6.4 Optimizing Data Placement

Mutant [129] optimizes the data placement of the LSM-tree on cloud storage. Cloud vendors often provide a variety of storage options with different performance characteristics and monetary

costs. Given a monetary budget, it can be important to place SSTables on different storage devices properly to maximize system performance. Mutant solves this problem by monitoring the access frequency of each SSTable and finding a subset of SSTables to be placed in fast storage so that the total number of accesses to fast storage is maximized while the number of selected SSTables is bounded. This optimization problem is equivalent to a 0/1 knapsack problem, which is N/P hard, and can be approximated using a greedy algorithm.

3.6.5 Summary

The techniques presented in this category aim at automatically tuning LSM-trees for given workloads. Both Lim et al. [72] and Monkey [41, 42] attempt to find optimal designs for LSM-trees to maximize system performance. However, these two techniques are complimentary to each other. Lim et al. [72] uses a novel analytical model to improve the cost estimation but only focuses on tuning the maximum level sizes of the leveling merge policy. In contrast, Monkey [41, 42], as well as its follow-up work Dostoevsky [43], co-tune all parameters of LSM-trees to find an optimal design but only optimize for the worst-case I/O cost. It would be useful to combine these two techniques together to enable more accurate performance tuning and prediction.

Dostoevsky [43] and LSM-Bush [44] extend the design space of LSM-trees with new merge policies for certain workloads to make better performance trade-offs. Thonangi and Yang [116] proposed to combine full merges with partitioned merges to achieve better write performance. Other tuning techniques focus on some aspects of the LSM-tree implementation, such as tuning Bloom filters by ElasticBF [70] and optimizing data placement by Mutant [129].

3.7 Query Processing

In the last category, we will discuss the research efforts that optimize the query performance of LSM-trees, including secondary indexing, range filtering, and exploiting opportunities provided by LSM lifecycles.

3.7.1 Secondary Indexing

Kim et al. [67] conducted an experimental study of LSM-based spatial index structures for geo-tagged data, including LSM-tree versions of the R-tree [55], Dynamic Hilbert B^+-tree (DHB-tree) [68], Dynamic Hilbert Value B^+-tree (DHVB-tree) [68], Static Hilbert B^+-tree (SHB-tree) [52], and Spatial Inverted File (SIF) [65]. An R-tree is a balanced search tree that stores multi-dimensional spatial data using their minimum bounding rectangles. DHB-trees and DHVB-trees store spatial points directly into B^+-trees using space-filling curves. SHB-trees and SIFs exploit a grid-based approach by statically decomposing a two-dimensional space into a multi-level grid hierarchy. For each spatial object, the IDs of its overlapping cells are stored. The difference between these two structures is that an SHB-tree stores the pairs of cell IDs and primary keys in a B^+-tree, while a SIF stores a list of primary keys for each cell ID in an inverted index. The key conclusion of this study is that there is no clear winner among these index structures, but the LSM-based R-tree performs reasonably well for both ingestion and query workloads without requiring too much tuning. It also handles both point and non-point data well. Moreover, for non-index-only queries, the final primary key lookup step is generally dominant since it often requires a separate disk I/O for each primary key. This further diminished the differences between these spatial indexing methods.

Qadar et al. [94] conducted an experimental study of LSM-based secondary indexing techniques including filters and secondary indexes. For filters, they evaluated component-level range filters and Bloom filters on secondary keys. For secondary indexes, they evaluated two secondary index-

ing schemes based on composite keys and key lists. Depending on how the secondary index is maintained, the key list scheme can be further classified as being either eager or lazy. The eager key list scheme always reads the previous list to create a full new list with the new entry added and inserts the new list into the memory component. The lazy key list scheme simply maintains multiple partial lists at each component. The experimental results suggest that the eager inverted list scheme incurs a large overhead on data ingestion because of the point lookups and high write amplification. When the query selectivity becomes larger, that is, when the result set contains more entries, the performance difference between the lazy key list scheme and the composite key scheme diminishes, as the final point lookup step becomes dominant. Finally, filters were found to be very effective with small storage overhead for time-correlated workloads. However, the study did not consider cleaning up secondary indexes in the case of updates, which means that secondary indexes could return obsolete primary keys.

A key challenge of maintaining LSM-based secondary indexes is handling updates. For a primary LSM-tree, an update can blindly add the new entry (with the identical key) into the memory component so that the old entry is automatically deleted. However, this mechanism does not work for a secondary index since a secondary key value can change during an update. Extra work must be performed to clean up obsolete entries from secondary indexes during updates.

Diff-Index [111] presents four index maintenance schemes for LSM-based secondary indexes, namely sync-full, sync-insert, async-simple, and async-session. During an update, two steps must be performed to update a secondary index, namely inserting the new entry and cleaning up the old entry. Inserting the new entry is very efficient for LSM-trees, but cleaning up the old entry is generally expensive since it requires a point lookup to fetch the old record. Sync-full performs these two steps synchronously during the ingestion time. It optimizes for query performance since secondary indexes are always up-to-date, but incurs a high overhead during data ingestion because of the point lookups. Sync-insert only inserts new data into secondary indexes, while cleaning up obsolete entries lazily by queries. Async-simple performs index maintenance asynchronously

but guarantees its eventual execution by appending updates into an asynchronous update queue. Finally, async-session enhances async-simple with session consistency for applications by storing new updates temporarily into a local cache on the client-side.

Deferred Lightweight Indexing (DELI) [112] enhances the sync-insert update scheme of Diff-Index [111] with a new method to cleanup secondary indexes by scanning the primary index components. Specifically, when multiple records with identical keys are encountered when scanning primary index components, the obsolete records are used to produce anti-matter entries to clean up the secondary indexes. Note that this procedure can be naturally integrated with the merge process of the primary index to reduce the extra overhead. Meanwhile, since secondary indexes are not always up-to-date, queries must always validate search results by fetching records from the primary index. Because of this, DELI cannot support index-only queries efficiently since point lookups must be performed for validation.

3.7.2 Range Filters

Rosetta [82] extends Bloom filters to enable filtering for range queries. For each disk component, Rosetta builds multiple Bloom filters, each of which corresponds to a binary key prefix. A range query is then converted into multiple probes, one for each non-overlapping binary key prefix. If all probes report negative results, then the disk component can be safely ignored from the range query. It should be noted that Rosetta may incur a lot of CPU overhead due to multiple Bloom filter probes. Thus, Rosetta is more suitable for short range queries, where disk components are more likely to be filtered, in I/O-heavy workloads

Alsubaiee et al. [19] proposed to augment each component of the primary and secondary indexes with a filter to enable data pruning based on a (non-primary) filter key. A filter stores the minimum and maximum values of the chosen filter key for the entries in a component. Thus, a component can be pruned by a query if the search condition is disjoint with the minimum and maximum values of

its filter. Though a filter can be built on arbitrary fields, it is really only effective for time-correlated fields since components are naturally partitioned based on time and are likely to have disjoint filter ranges.

3.7.3 Exploiting LSM Lifecycles

LSM-trees provide well-defined lifecycle events, i.e., flushes and merges, to manage their data. These lifecycle events provide opportunities for database systems to perform extra tasks to optimize query performance.

Absalyamov et al. [13] proposed a lightweight statistics collection framework for LSM-based systems. The basic idea is to integrate the task of statistics collection into the flush and merge operations to minimize the statistics maintenance overhead. During flush and merge operations, statistical synopses, such as histograms and wavelets, are created on-the-fly and are sent back to the system catalog. Due to the multi-component nature of LSM-trees, the system catalog stores multiple statistics for a dataset. To reduce the overhead during query optimization, mergeable statistics, such as equi-width histograms, are merged beforehand. For statistics that are not mergeable, multiple synopses are kept to improve the accuracy of cardinality estimation.

Alkowaileet et al. [16] presented an LSM-based tuple compaction framework for storing schemaless JSON documents exploiting the immutability of LSM disk components. Its basic idea to automatically perform schema inference during flushes and merges so that the JSON records can be stored into a more efficient format, which greatly reduces the space overhead and improves query performance. The schema information is further propagated to a metadata node so that it can be utilized by the query optimizer to generate efficient query plans.

3.7.4 Summary

The techniques in this category all focus on improving the query processing capability of LSM-trees. To speedup query processing, two commonly used approaches are secondary indexing [67, 94, 111, 112] and range filtering [19, 82]. Moreover, the well-defined LSM lifecycle events provide additional opportunities for effective query processing, such as collecting data statistics [13] and compacting JSON records for efficient access [16].

3.8 Discussion of Overall Trade-offs

Based on the RUM conjecture [24], no access method can be read-optimal, write-optimal, and space-optimal at the same time. As we have seen in this chapter, many LSM-tree improvements that optimize for certain workload or system aspects will generally make trade-offs. To conclude this chapter, we provide a qualitative analysis and summary of the trade-offs made by those research efforts that seek to optimize various aspects of LSM-trees. We will consider the leveling merge policy as the baseline for this discussion.

The performance trade-offs of the various LSM-tree improvements are summarized in Table 3.2. As one can see, most of these improvements try to improve the write performance of the leveling merge policy since it has relatively high write amplification. A common approach taken by existing improvements is to apply the tiering merge policy [20, 86, 95, 124, 127], but this will negatively impact query performance and space utilization. Moreover, tiering has a larger negative impact on range queries than point lookups since range queries cannot benefit from Bloom filters.

Other proposed improvements, such as the skip-tree [130], TRIAD [26], and the VT-tree [106], propose several new ideas to improve the write performance of LSM-trees. However, in addition to extra overhead on query performance and space utilization, these optimizations may bring non-

Table 3.2: Summary of Trade-offs Made by LSM-tree Improvements

Publication	Write	Point Lookup	Short Range	Long Range	Space	Remark
WB-tree [20]	↑↑	↓	↓↓	↓↓	↓↓	Tiering
LWC-tree [127]	↑↑	↓	↓↓	↓↓	↓↓	Tiering
PebblesDB [95]	↑↑	↓	↓↓	↓↓	↓↓	Tiering
SifrDB [86]	↑↑	↓	↓↓	↓↓	↓↓	Tiering
Skip-tree [130]	↑	↓	↓	↓	–	Mutable skip buffers
TRIAD [26]	↑	↓	↓	↓	–	Separate cold entries from hot entries; delay merges at level 0; use logs as flushed components
VT-tree [106]	↑	–	↓	↓	↓	Stitching merge
MaSM [23]	↑↑	↓	↓↓	↓	↓	Lazy leveling
WiscKey [75]	↑↑↑	↓	↓↓↓	↓↓↓	↓↓↓	KV separation
HashKV [71]	↑↑↑	↓	↓↓↓	↓↓↓	↓↓↓	KV separation
Kreon [91]	↑↑↑	↓	↓↓↓	↓↓↓	↓↓↓	KV separation
LSM-trie [124]	↑↑	↑	×	×	↓↓	Tiering + hashing
SlimDB [98]	↑↑	↑	↓↓/×	↓/×	↓	Only support range queries for each key prefix group
Lim et al. [72]	↑	–	–	–	–	Exploit data redundancy
Monkey [41, 42]	–	↑	–	–	–	Better Bloom filter memory allocation
Dostoevsky [43]	↑↑	↓	↓↓	↓	↓	Lazy leveling
LSM-Bush [44]	↑↑	–	↓↓↓	↓	–	Exponentially larger size ratios for smaller levels

trivial implementation complexity to real systems. For example, the skip-tree introduces mutable buffers to store skipped keys, which contradicts the immutability of disk components. TRIAD proposes to use transaction logs as disk components to eliminate flushes, which is again highly non-trivial since transaction logs usually have very different storage formats and operation interfaces from disk components. Moreover, a common practice is to store transaction logs on a dedicated disk to minimize the negative impacts caused by log forces. The stitching operation proposed by the VT-tree [106] can cause fragmentation and is incompatible with Bloom filters.

The LSM-trie [124] and SlimDB [98] give up some query capabilities to improve performance. The LSM-trie exploits hashing to improve both read and write performance, but range queries cannot be supported. SlimDB only supports a limited form of range queries based on a common prefix key. These improvements would be desirable for certain workloads where complete range queries are not needed.

Separating keys from values [71, 75, 91] can drastically improve the write performance of LSM-trees since only keys are merged. However, a major problem is that range queries will be significantly impacted because values are not sorted anymore. Even though this problem can be mitigated by exploiting the I/O parallelism of SSDs [71, 75], this still leads to lower disk efficiency especially when values are relatively small. Moreover, storing values separately leads to lower space utilization since values are not garbage-collected during merges. A separate garbage-collection process must be designed to reclaim disk space occupied by obsolete values.

Given that trade-offs are inevitable, it is valuable to explore the design space of LSM-trees so that one can make better or optimal trade-offs. For example, Lim et al. [72] exploits data redundancy to tune the maximum sizes for each level to optimize write performance. This has little or no impact on other performance metrics since the number of levels remains the same. Another example is Monkey [41, 42], which unifies the design space of LSM-trees in terms of merge policies, size ratios, and memory allocation between memory components and Bloom filters. It further identifies a better memory allocation scheme for Bloom filters that improves point lookup performance without any negative impact on other metrics. Finally, Dostoevsky [43] and LSM-Bush [44] extend the design space of LSM-trees with new merge policies that lead to better performance trade-offs for certain workloads.

Chapter 4

Efficient Maintenance and Exploitation of LSM-based Auxiliary Structures

4.1 Introduction

A wide range of applications, such as risk management, online recommendations, and location-based advertising, demand the capability of performing real-time analytics on high-speed, continuously generated data coming from sources such as social networks, mobile devices and IoT applications. As a result, modern Big Data systems need to efficiently support both fast data ingestion and real-time queries.

Auxiliary structures, such as secondary indexes, are critical to enable the efficient processing of ad-hoc queries. Two types of LSM-based auxiliary structures have been used in practice to facilitate query processing, namely secondary indexes and filters. A secondary index is an LSM-tree that maps secondary key values to their corresponding primary keys. A filter, such as a Bloom filter [29] or a range filter [19] on secondary keys, is directly built into LSM-trees to enable data skipping for faster scans. While needed for queries, maintaining these structures during data inges-

tion comes with extra cost. Especially in the case of updates, both types of structures require accessing old records so that they can be properly maintained. Existing LSM-based systems, such as AsterixDB [1, 17], MyRocks [7], and Phoenix [8], employ an eager strategy to maintain auxiliary structures by prefacing each incoming write with a point lookup. This strategy is straightforward to implement and optimizes for query performance since auxiliary structures are always up-to-date. However, it leads to significant overhead during ingestion because of the point lookups.

An outage with respect to the general-purpose use of LSM indexing is that no particularly efficient point lookup algorithms have been proposed for the efficient fetching of the records identified by a secondary index search. While sorting fetch lists based on primary keys is a well-known optimization [54], performing the subsequent point lookups independently still incurs high overhead. This limits the range of applicability of LSM-based secondary indexes, requiring the query optimizer to maintain accurate statistics to make correct decisions for processing ad-hoc queries efficiently.

This chapter focuses on efficient data ingestion and query processing techniques for general-purpose LSM-based storage systems. The first contribution of this chapter is to propose and evaluate a series of optimizations for efficient index-to-index navigation for LSM-based indexes. We show how to leverage the internal structure of LSM to efficiently process a large number of point lookups. We further conduct a detailed empirical analysis to evaluate the effectiveness of these optimizations. The experimental results show that the proposed optimizations greatly improve the range of applicability of LSM-based secondary indexes. Even with relatively large selectivities, such as 10 - 20%, LSM-based secondary indexes can still provide performance better than or (worst case) comparable to that of a full scan.

The second contribution of this chapter is to study alternative maintenance strategies for LSM-based auxiliary structures. The key insight here is the applicability of a *primary key index*, which only stores primary keys, for use in index maintenance. For secondary indexes, we present a *validation* strategy that cleans up obsolete entries lazily using a primary key index. For filters, we introduce a *mutable-bitmap* strategy that allows deleted keys to be directly reflected for immutable

data through mutable bitmaps by accessing primary keys instead of full records. Since primary keys are much smaller than full records, we show that the exploitation of a primary key index can greatly reduce required I/Os and significantly increase the overall ingestion throughput.

Finally, we have implemented all of the proposed techniques inside Apache AsterixDB [1, 17], an open-source LSM-based Big Data Management System. We have conducted extensive experiments to evaluate their impacts on ingestion performance and query performance.

The remainder of the chapter is organized as follows: Section 4.2 discusses the work related to this chapter. Section 4.3 presents a general architecture for LSM-based storage systems and introduces various optimizations for efficient point lookups. Sections 4.4 and 4.5 describe in detail the proposed Validation strategy and the Mutable-bitmap strategy. Section 4.6 experimentally evaluates the proposed techniques. Finally, Section 4.7 concludes the chapter.

4.2 Related Work

In Chapter 3, we have comprehensively discussed the recent research of optimizing LSM-trees. Here we mainly focus on some recent work related to LSM-based auxiliary structures.

DELI [112] is a lazy secondary index maintenance strategy that repairs secondary indexes during the merge of primary index components. In addition to its eager strategy, AsterixDB [18] supports a deleted-key B^+-tree strategy that attaches a B^+-tree to each secondary index component that records the deleted keys in this component. Since DELI and the deleted-key B^+-tree strategy are closely related to our work, we will further discuss them in detail in Section 4.4.1. Qadar et al. [94] conducted an experimental study of LSM-based secondary indexes. However, their study did not consider cleaning up secondary indexes in the case of updates.

Alsubaiee et al. [19] added the option of a range filter on LSM-based primary and secondary in-

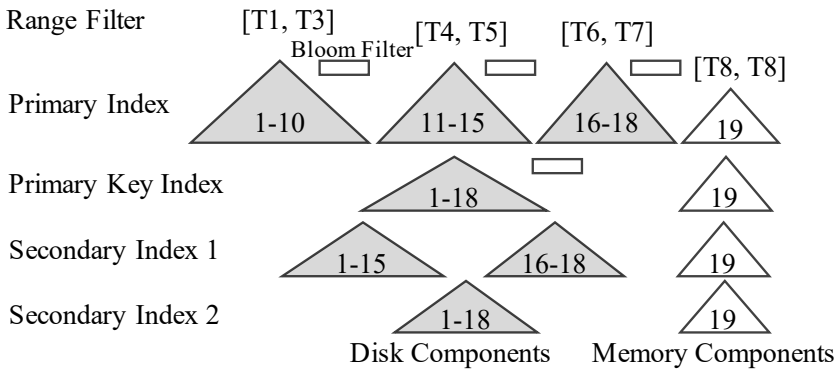


Figure 4.1: LSM Storage Architecture

dexes for the efficient processing of time-correlated queries. Jia [62] exploited LSM range filters to accelerate primary key lookups for append-only and time-correlated workloads. However, the eager strategy for maintaining filters incurs high lookup cost during ingestion and reduces their pruning capabilities in the presence of updates. Several commercial database systems have similarly supported range filter-like structures, such as zone maps in Oracle [134] and synopses in DB2 BLU [97], to enable data skipping during scans. However, these systems are not based on LSM-trees.

Several related write-optimized indexes [104, 122, 123] have been proposed to efficiently index append-only observational streams and time-series data. Though they provide high ingestion performance, updates and deletes are not supported. Our work targets general workloads that include updates and deletes as well as appends.

4.3 LSM Storage Architecture

In this section, we present a general LSM-based storage architecture that will be the foundation for the rest of this chapter. We also present the Eager strategy used in existing systems as well as a series of optimizations for index-to-index navigation given LSM-based indexes.

The storage architecture is depicted in Figure 4.1. Each dataset has a primary index, a primary key index, and a set of secondary indexes, which are all based on LSM-trees. All indexes in a dataset share a common memory budget for memory components, so they are always flushed together (as in AsterixDB). Each component in the figure is labeled with its *component ID*, which is represented as a pair of timestamps (minTS - maxTS). The component ID is simply maintained as the minimum and maximum timestamps of the index entries stored in the component, where the timestamp is generated using the local wall clock time when a record is ingested. Through component IDs, one can infer the recency ordering among components of different indexes, which can be useful for secondary index maintenance operations (as discussed later). For example, component IDs indicate that component 1-15 of Secondary Index 1 is older than component 16-18 of the primary index and that it overlaps component 1-10 of the primary index.

The primary index stores the records indexed by their primary keys. To reduce point lookups during data ingestion, as we will see later, we further build a primary key index that stores primary keys only. Both of these indexes internally use a B^+-tree to organize the data within each component. Each primary or primary key disk component also has a Bloom filter [29] on the stored primary keys to speed up point lookups. A point lookup query can first check the Bloom filter of a disk component and search the actual B^+-tree only when the Bloom filter reports that the key may exist. Secondary indexes use a composition of the secondary key and the primary key as their index key in order to efficiently handle duplicate secondary keys. A secondary index query can first search the secondary index to return a list of matching primary keys and then perform point lookups to fetch records from the primary index.

In general, the primary index may have a set of filters for efficient pruning during scans[1]. Without loss of generality, we assume that each primary index component may have a range filter that stores the minimum and maximum values of the component's secondary filter key (denoted as [Ti, Tj] in Figure 4.1). During a scan, a component can be pruned if its filter is disjoint with the search

[1] As suggested by [19], secondary indexes could have filters as well. However, in this chapter we only focus on the use of filters on the primary index to support efficient scans.

condition of a query.

4.3.1 Data Ingestion with the Eager Strategy

During data ingestion, all storage structures must be properly maintained. Here we briefly review the Eager strategy commonly used by existing LSM-based systems such as AsterixDB [1, 17], MyRocks [7], and Phoenix [8].

To *insert* a record, its key uniqueness is first checked by performing a point lookup. As an optimization, the primary key index can be searched instead for efficiency. If the given primary key already exists, the record is ignored; otherwise, the record is recorded in the memory components of all of the dataset's indexes. Any filters of the memory components must be maintained based on the new record as well.

To *delete* a record given its key, a point lookup is first performed to fetch the record. If the record does not exist, the key is simply ignored. Otherwise, anti-matter entries must be inserted into all of the dataset's LSM indexes to delete the record. Any filters of the memory components must also be maintained based on the deleted record; otherwise, a query could erroneously prune a memory component and thus access deleted records.

To *upsert* a record, a point lookup is first performed to locate the old record with the same key. If the old record exists, anti-matter entires are generated to delete the old record from all of the dataset's secondary indexes. The new record is then inserted into the memory components of all of the dataset's LSM indexes. As an optimization, if the value of some secondary key did not change, the corresponding secondary index can be simply skipped for maintenance. Any filters of the memory components must be maintained based on both the old record (if it exists) and the new record.

As a running example, consider the initial LSM indexes depicted in Figure 4.2 for a UserLocation

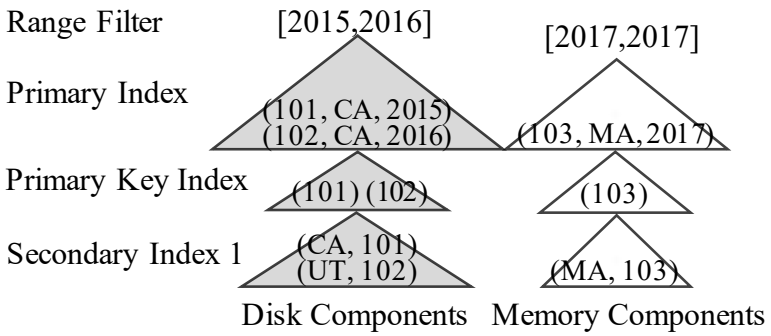


Figure 4.2: Running Example

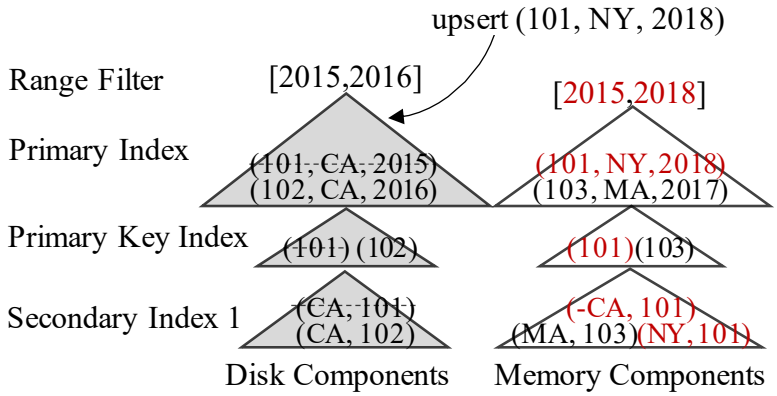


Figure 4.3: Upsert Example with Eager Strategy

dataset with three attributes: UserID (the primary key), Location, and Time. The Location attribute stores the last known location of the user (in terms of states) and the Time attribute stores the time of the last known location (in terms of years). We have a secondary index on Location and a range filter on Time. Figure 4.3 shows the resulting LSM indexes after upserting a new record (101, NY, 2018) with an existing key 101. A point lookup is first performed to locate the old record (101, CA, 2015). In addition to adding the new record to all memory components, an anti-matter entry (-CA, 101) is added to the secondary index to eliminate the obsolete entry (which is dashed in the figure). Also, notice that the memory component's range filter is maintained based on both 2015 and 2018 so that future queries will properly see that the old record with UserID 101 has been deleted.

We further use Figure 4.3's example to illustrate how secondary indexes and filters are used in

query processing. First consider a query Q1 that finds all user records with Location CA. Q1 first searches the secondary index to return a list of UserIDs. In this example only UserID 102 is returned since (CA, 101) is deleted by the anti-matter entry (-CA, 101). The primary index is then searched to fetch the record using UserID 102, which returns (102, CA, 2016). Note that without the anti-matter entry, the obsolete UserID 101 would be erroneously searched by Q1 as well. Consider another query Q2 that finds all records with Time $<$ 2017. Q2 scans the primary index by first collecting a set of candidate components whose range filters overlap with the search condition Time $<$ 2017. In this example, both components will be scanned by Q2 and only one record (102, CA, 2016) is returned. However, suppose that when upserting record (101, NY, 2018), the range filter of the memory component had only been maintained based on the new value 2018. In this case, the memory component would have been pruned and (101, CA, 2015) would be erroneously returned as well.

4.3.2 Efficient Index-to-index Navigation

Navigating from secondary indexes to the primary index is a fundamental operation for query processing. Traditionally, primary keys are sorted to ensure that the pages of the primary index will be accessed sequentially [54]. Here we discuss further optimizations to improve point lookup performance for LSM-trees. Note that some of the optimizations below are not new. Our contribution is to evaluate their effectiveness and integrate them to improve the range of applicability of LSM-based secondary indexes.

Batched Point Lookup. Even though primary keys are sorted, when searching multiple LSM components, it is still possible that index pages will be fetched via random I/Os since the sorted keys can be scattered across different components. To avoid this, we propose here a *batched point lookup* algorithm that works as follows: Sorted primary keys are first divided into batches. For each batch, all of the LSM components are accessed one by one, from newest to oldest. Specifically, for

each key in the current batch that has not been found yet, it is searched against a primary component by first checking the Bloom filter and then the B^+-tree. A given batch terminates either when all components have been searched or all keys have been found. The batch search algorithm ensures that components' pages are accessed sequentially, avoiding random I/Os when fetching their leaf pages. However, a downside is that the returned data records will no longer still be ordered on primary keys. We will experimentally evaluate this trade-off in Section 4.6.

Stateful B^+-tree Lookup. To reduce the in-memory B^+-tree search overhead, one can use a stateful B^+-tree search cursor that remembers the search history from root to leaf. Instead of always traversing from the root for each key, the cursor starts from the last leaf page to reduce the tree traversal cost. One can further use exponential search [28] instead of binary search to reduce the search cost within each page. To search a key, this algorithm starts from the last search position and searches for the key using exponentially increasing steps to locate the range which this key resides in. The key can then be located by performing a binary search within this range.

Blocked Bloom Filter. Finally, to reduce the overhead of checking the components' Bloom filters, a cache-friendly approach called blocked Bloom filter [93] can be used. The basic idea is to divide the bit space into fixed-length blocks whose size is the same as the CPU cache line size. The first hash function maps a key to a block, while the rest of the hash functions perform the usual bit tests but within this block. This ensures that each Bloom filter test will only lead to one cache miss, at the cost of requiring an extra bit per key to achieve the same false positive rate [93].

4.4 Validation Strategy

In this section, we propose the Validation strategy for maintaining secondary indexes efficiently. We first present an overview followed by detailed discussions of this strategy.

4.4.1 Overview

In a primary LSM-tree, an update can blindly place a new entry (with the identical key) into memory to mark the old entry as obsolete. However, this mechanism does not work with secondary indexes, as the index key could change after an update. Similarly, one cannot efficiently determine whether a given primary key is still valid based on a secondary index alone, as entries in a secondary index are ordered based on secondary keys. Extra work must be performed to maintain secondary indexes during data ingestion.

The Eager strategy maintains secondary indexes by producing anti-matter entries for each old record, which incurs a large point lookup overhead during data ingestion. An alternative strategy is to only insert new entries into secondary indexes during data ingestion while cleanup obsolete entries lazily so that the expensive point lookups can be avoided. This design further ensures that secondary indexes can only return false positives (obsolete primary keys) but not false negatives, which simplifies query processing. Although the idea of lazy maintenance is straightforward, two challenges must be addressed: (1) how to support queries efficiently, including both non-index-only and index-only queries; (2) how to repair secondary indexes efficiently to cleanup obsolete entries while avoiding making cleanup a new bottleneck.

There have been several proposals for lazy secondary index maintenance on LSM-based storage systems. DELI [112] maintains secondary indexes lazily, while merging the primary index components, without introducing any additional structures. If multiple records with the same primary key are encountered during a merge, DELI produces anti-matter entries for obsolete records to clean up secondary indexes. However, this design does not completely address the above two challenges. First, non-index-only queries cannot be supported efficiently, as queries must fetch records to validate the search results. Second, DELI lacks the flexibility to repair secondary indexes efficiently. For update-heavy workload, it is desirable to repair secondary indexes more frequently to improve query performance. However, in DELI this requires constantly merging or scanning all

components of the primary index, incurring a high I/O cost.

To support index-only queries, extra structures on top of secondary indexes should be maintained. AsterixDB supports a deleted-key B^+-tree strategy that attaches a B^+-tree to each secondary index component that records the deleted keys in this component. For index-only queries, validation can be performed by searching these deleted-key B^+-trees without accessing full records. However, since these B^+-trees are duplicated for each secondary index, and no efficient repairing algorithms have been described, cleaning up secondary indexes incurs a high overhead during merges.

To address the aforementioned challenges, we choose to use the primary key index to maintain all secondary indexes efficiently. The primary key index can be used for validating index-only queries by storing an extra timestamp for each index entry in all secondary indexes as well as in the primary key index. This timestamp is generated using the node-local clock time when a record is ingested and is stored as an integer with 8 bytes. We further propose an efficient index repair algorithm based on the primary key index, greatly reducing the I/O cost by avoiding accessing full records.

4.4.2 Data Ingestion

Under the Validation strategy, an *insert* is handled exactly as in the Eager strategy except that timestamps are added to the primary key index entries and secondary index entries. To *delete* a record given its key, an anti-matter entry is simply inserted into both the primary index and the primary key index. To *upsert* a record, the new record is simply inserted into all of the dataset's LSM indexes. (Notice that when deleting or upserting a record, the memory component of the primary index has to be searched to find the location for the entry being added. As an optimization, then, if the old record happens to reside in the memory component, it can be used to produce local anti-matter entries to clean up the secondary indexes without additional cost.)

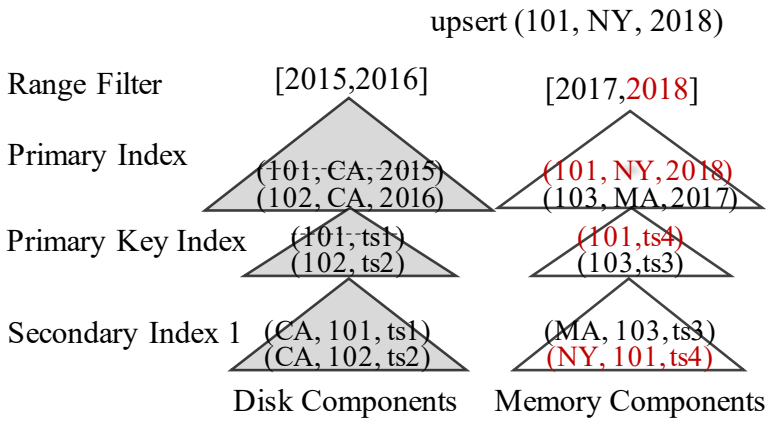


Figure 4.4: Upsert Example with Validation Strategy

Consider the running example that began in Figure 4.2. Figure 4.4 shows the resulting LSM indexes after upserting the record (101, NY, 2018) under the Validation strategy. The primary key index and the secondary index each now contain an extra timestamp field (denoted as "ts"). To upsert the new record, we add the new record to all memory components without any point lookups. As a result, the obsolete secondary index entry (CA, 101, ts1) still appears to be valid even though it points to a deleted record. The Validation strategy can be naturally extended to support filters: Since no pre-operation point lookup is performed, the filters of the memory components are only maintained based on new records. As in the example, then, the range filter is only maintained based on 2018. As a result, a query that accesses an older component has to access all newer components in order not to miss any newer overriding updates.

4.4.3 Query Processing

In the Validation strategy, secondary indexes are not always guaranteed to be up-to-date and can thus return obsolete entries to queries. For correctness, queries have to perform an extra validation step to ensure that only valid keys are eventually accessed. Here we present two validation method variations suitable for different queries (Figure 4.5).

The Direct Validation method (Figure 4.5a) directly performs point lookups to fetch all of the

candidate records and re-checks the search condition. A sort-distinct step is performed first to remove any duplicate primary keys. After checking the search condition, only the valid records are returned to the query. However, this method rules out the possibility of supporting index-only queries efficiently.

To address these drawbacks, the Timestamp Validation method (Figure 4.5b) uses the primary key index to perform validation. A secondary index search returns the primary keys plus their timestamps. Point lookups against the primary key index are then performed to validate the fetched primary keys. Specifically, a key is invalid if the same key exists in the primary key index but with a larger timestamp. The valid keys are then used to fetch records if necessary.

Consider the example in Figure 4.4, and a query that wants to find all records with Location CA. The secondary index search returns primary keys with their timestamps (101, ts1) and (102, ts2). Direct Validation performs point lookups to locate records (101, NY, 2018) and (102, CA, 2016). The first record will be filtered out because its Location is not CA anymore. Timestamp Validation would perform a point lookup against the primary key index to filter out UserID 101 since its timestamp ts1 is older than ts4.

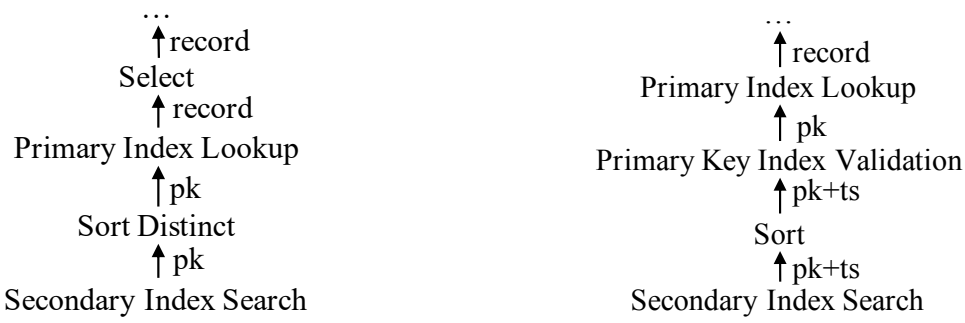


Figure 4.5: Query Validation Methods

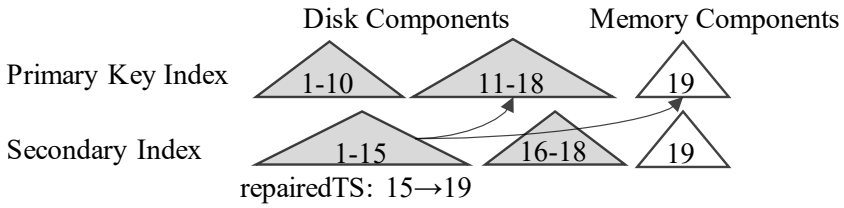


Figure 4.6: Repaired Timestamp Example

4.4.4 Secondary Index Repair

Since secondary indexes are not cleaned up during ingestion under the Validation strategy, obsolete entries could accumulate and degrade query performance. To address this, we propose performing index repair operations in the background to clean up obsolete index entries. Index repair can either be performed during merge time, which we will call *merge repair*, or scheduled independently from merges, which we will call *standalone repair*.

The basic idea of index repair is to validate each primary key in a component by searching the primary key index. For efficiency, the primary keys in a given component of a secondary index should only be validated against newly ingested keys. To keep track of the repair progress, we associate a *repaired timestamp* (repairedTS) with each disk component of a secondary index. During a repair operation, all primary key index components with maxTS no larger than the repairedTS can be pruned[2]. A repaired component receives a new repairedTS computed as the maximum timestamp of the unpruned primary key index components. To clarify this, consider the example in Figure 4.6. Component 1-15 of the secondary index has an initial repairedTS of 15. To repair this component, we only need to search components 11-18 and 19 of the primary key index, while component 1-10 can be pruned. After the repair operation, the new repairedTS of this component would be 19.

A naive implementation of merge repair would simply validate each primary key by performing a point lookup against the primary key index. For standalone repair, one could simply produce a new component with only valid entries without merging. However, this implementation would be

[2]This optimization applies to Timestamp Validation as well.

highly inefficient because of the expensive random point lookups.

To handle this, we propose a more efficient merge repair algorithm (Algorithm 1). The basic idea is to first sort the primary keys to improve point lookup performance, and further use an immutable bitmap to avoid having to sort entries back into secondary key order. The immutable bitmap of a disk component indicates whether each of the component's index entries is (still) valid or not, and thus it stores one bit per entry. Without loss of generality, we assume that a bit being 1 indicates that the corresponding index entry is invalid.

Algorithm 1 Pseudo Code for Merge Repair

```
1:  Create search cursor on merging components
2:  position ← 0
3:  while cursor.hasNext() do
4:      entry ← cursor.getNextEntry()
5:      add entry to new component
6:      add (pkey, ts, position) to sorter
7:      position ← position + 1
8:  initialize bitmap with all 0s
9:  sorter.sort()
10: for sorted entries (pkey, ts, position) do
11:     validate pkey against primary key index
12:     if pkey is invalid then
13:         mark position of bitmap to 1
```

Initially, we create a scan cursor over all merging components to obtain all valid index entries, i.e., entries where their immutable bitmap bits are 0. These entries are directly added to the new component (line 5). Meanwhile, the primary keys, with their timestamps and positions in the new component, are streamed to a sorter (line 6). These sorted primary keys are then validated by searching the primary key index. As an optimization, if the number of primary keys to be validated is larger than the number of recently ingested keys in the primary key index, we can simply merge scan the sorted primary keys and the primary key index. If a key is found to be invalid, that is, if the same key exists in the primary key index with a larger timestamp, we simply set the corresponding position of the new component's bitmap to 1 (lines 12-13). Standalone repair can be implemented similarly, except that only a new bitmap is created.

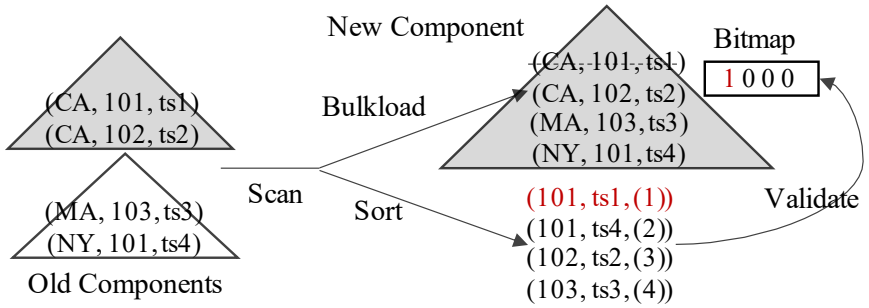


Figure 4.7: Merge Repair Example

To illustrate the immutable bitmap and the repair process, consider the example secondary index in Figure 4.4. Suppose we want to merge and repair all components of the secondary index. The process is shown in Figure 4.7. The index entries scanned from its old components are directly added to the new component, and in the meanwhile they are sorted into primary key order. The sorted primary keys are validated using the primary key index. During validation, the key 101 with timestamp ts1 is found to be invalid since the same key exists with a larger timestamp ts4. This index entry has ordinal position 1 (denoted "(1)" in the figure), so first bit in the component's bitmap is set to 1. Note that this invalid entry will be physically removed during the next merge.

Bloom Filter Optimization. It is tempting to use Bloom filters to further optimize the index repair operation. The idea is that if the Bloom filters of the primary key index components do not contain a key, which implies that the key has not been updated, then the key can be excluded from sorting and further validation. However, if implemented directly, this would provide little help since a dataset's various LSM-trees are merged independently. To see this problem, consider the example LSM indexes back in Figure 4.1. Suppose we want to merge and repair the two disk components of Secondary Index 1. Since the disk components of the primary key index have already been merged into a single component beforehand, its Bloom filter would always report positives, which would provide no help and actually cause some extra overhead.

To maximize the effectiveness of the Bloom filter optimization, one must ensure that during each repair operation the unpruned primary key index components are always strictly newer than the

keys in the repairing component(s). For this purpose, we can use a correlated merge policy [19] to synchronize the merge of all secondary indexes with the primary key index in order to ensure that their components are always merged together. Furthermore, all secondary indexes must be repaired during every merge. As a side-effect of this optimization, the timestamps of index entries can be discarded since validations can be performed by using the relative ordering among components. For example, in Figure 4.4 the secondary index entry (CA, 101) is invalid since the same key 101 exists in a newer (memory) component of the primary key index.

The Bloom filter optimization improves repair efficiency and is thus suitable for update-heavy workloads that require secondary indexes to be frequently repaired. However, for workloads that contain few updates, it might be better to just schedule repair operations during off-peak hours, and thus the Bloom filter optimization may not be suitable. To alleviate the tuning effort required from the end-user, we plan to develop auto-tuning techniques in the future.

4.5 Mutable-Bitmap Strategy

In this section, we present the Mutable-bitmap strategy designed for maintaining a primary index with filters.

4.5.1 Overview

The key difficulty of applying filters to the LSM-tree is its out-of-place update nature, that is, updates are added to the new component. In the case of updates, the filter of the new component must be maintained using old records so that queries would not miss any new updates. The Eager strategy performs point lookups to maintain filters using old records, incurring a high point lookup cost during data ingestion. The Validation strategy skips point lookups but requires queries to access all newer components for validation, halving the pruning capabilities of filters.

The Mutable-bitmap strategy presented below aims at both maximizing the pruning capabilities of filters and reducing the point lookup cost during data ingestion. The first goal can be achieved if old records from disk components can be deleted directly. However, if we were to place new updates directly into the disk components where old records are stored, a lot of complexity would be introduced on concurrency control and recovery. Instead, our solution is to add a mutable bitmap to each disk component to indicate the validity of each entry using a very limited degree of mutability. We present efficient solutions to address concurrency control and recovery issues, exploiting the simple semantics of mutable bitmaps, that is, writers only change bits from 0 to 1 to mark records as deleted[3]. To minimize the point lookup cost, the maintenance of mutable bitmaps is performed by searching the primary key index instead of accessing full records. To achieve this, we synchronize the merges of the primary index and the primary key index using the correlated merge policy (as in Section 4.4.4). An alternative implementation is to add a key-only B^+-tree to each primary index component, combining two indexes together.

4.5.2 Data Ingestion

For ease of discussion, let us first assume that there are no concurrent flush and merge operations, the handling of which are postponed to Section 4.5.3. An *insert* is handled exactly as in the Eager/Validation strategy, and no bitmaps are updated because no record is deleted. To *delete* a record given its key, we first search the primary key index to locate the position of the deleted key. If the key is found and is in a disk component, then its corresponding bit in that component's bitmap is set (mutated) to 1. An anti-matter key is also added to the memory component, for two reasons. First, we view the mutable bitmap as an auxiliary structure built on top of LSM that should not change the semantics of LSM itself. Second, if the Validation strategy is used for secondary indexes, inserting anti-matter entries ensures that validation can be performed with only recently ingested keys. To *upsert* a record, the primary key index is first searched to mark the old record as

[3]Aborts internally change bits from 1 to 0.

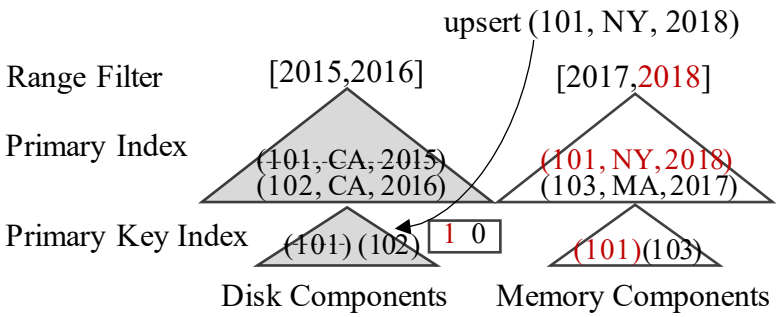


Figure 4.8: Upsert Example with Mutable-bitmap Strategy

deleted if necessary, and the new record is added to the index memory components. Any filters of the memory components are only maintained based on the new record, not the old one.

Consider the running example from Figure 4.2. Figure 4.8 shows the resulting LSM-trees after upserting a new record (101, NY, 2018). The secondary index is not shown since it can be maintained using either the Eager or Validation strategy. In this case, the primary index and the primary key index are synchronized and their components share a mutable bitmap to indicate the validity of their records. To upsert the new record, the primary key index is searched to locate the position of the old record and the bitmap is mutated to mark the old record as deleted. The range filter of the memory component is maintained based only on the new record, avoiding unnecessary widening based on 2015.

We now discuss concurrency control and recovery issues for the mutable bitmaps. We assume that each writer acquires an exclusive (X) lock on a primary key throughout the (record-level) transaction. To prevent two writers from modifying the same bitmap byte, one can use latching or compare-and-swap instructions. For recovery, we use an additional update bit in the log record for each delete or upsert operation to indicate whether the key existed in a disk component. To abort a transaction, if the update bit is 1, we simply perform a primary key index lookup (without bitmaps) to unset the bit from 1 to 0. To unify the recovery of bitmaps with LSM-trees, we use a no-steal and no-force policy for bitmaps as well. A modified bitmap page is pinned until the transaction terminates to prevent dirty pages from being flushed. Regular checkpointing can be performed to

flush dirty pages of bitmaps. Upon recovery, committed transactions are simply replayed to bring bitmaps up-to-date based on the last checkpointed LSN. Again, a log record is replayed on the bitmaps only when its update bit is 1.

4.5.3 Concurrency Control for Flush/Merge

Mutable bitmaps also introduce concurrency control issues for LSM flush and merge operations. This is because concurrent writers may need to modify the bitmaps of the components that are being formed by flush or merge operations. This problem bears some similarities with previous work on online index construction [87, 107], which builds new indexes concurrently with updates. Here we propose two possible concurrency control methods with different flavors in terms of how new updates are applied to the new component. The *Lock* method directly applies new updates to the new component at the cost of extra locking overhead. The *Side-file* method buffers new updates in a side-file and then applies them after the new component has been fully built.

Algorithm 2 Pseudo Code for Lock Method

```
1:  function BUILD(new component)
2:      set old component(s) point to new component C'
3:      create cursor on old component(s)
4:      while cursor.hasNext() do
5:          (key, record) ← cursor.getNext()
6:          S lock key
7:          if key is still valid by checking bitmap then
8:              add (key, record) to new component
9:          C'.ScannedKey ← key
10:         unlock key
11: function DELETE(key)
12:     X lock on key
13:     search key from the primary key index
14:     if key exists in immutable component C then
15:         mark key deleted in C
16:         if C points to C' AND key ≤ C'.ScannedKey then
17:             mark key deleted in C'
18:     unlock key
```

Lock Method. The pseudo code for the Lock method is shown in Algorithm 2. The component builder uses the BUILD function to build the new component during a flush or merge operation. It first sets the old component(s) to point to the new component (line 2) so that the new component is visible to writers. A full scan over the old component(s) is then performed to build the new component. For each scanned key, the component builder acquires a shared lock (line 6) to prevent the key from being deleted by writers. Otherwise, if a transaction were to, the deleted key may have already been skipped by the component builder, making it impossible to rollback in the middle of the bulkloading process. After the lock has been acquired, the bitmap is re-checked (line 7) to ensure that the key is still valid. The writer uses the DELETE function to delete key. To do so, the primary key index is first searched. If the key is found in a disk component C, it is marked as deleted (lines 14-15). If C further points to a new component C' and the deleted key has already been added to C', the key is further marked as deleted in C' by performing another point lookup (lines 16-17).

Side-file Method. The pseudo code for the Side-file method is shown in Algorithm 3. For ease of exposition, we divide the component building process into three phases, i.e., INITIALIZE, BUILD, and CATCHUP. During INITIALIZE (lines 1-5), the component builder acquires a shared lock on the dataset to drain ongoing transactions (explained later) and then creates immutable bitmap snapshots of old components. During BUILD (lines 6-10), the old components are scanned with the bitmap snapshots to avoid interference from concurrent updates. Finally, during CATCHUP (lines 11-16), a shared lock on the dataset is acquired again to close the side-file. The component builder sorts the side-file as suggested in [107] and then applies the deleted keys to the new component.

Shared locks on the dataset are acquired to ensure correctness in the case of transaction rollbacks. Otherwise, if a transaction were to abort afterwards, it may not be able to undo its changes to the new components. If a transaction deletes a key before the initialization phase, but aborts after the bitmap snapshot has been created, the deleted key may have already been skipped by the component builder and cannot be re-added to the new component. Similarly, if a transaction

Algorithm 3 Pseudo for Side-file Method

```
1:  function INITIALIZE(new component)
2:      S lock dataset
3:      create bitmap snapshots of old component(s)
4:      set old component(s) point to new component C'
5:      unlock dataset
6:  function BUILD(new component)
7:      create cursor on old component(s) with bitmap snapshots
8:      while cursor.hasNext() do
9:          (key, record) ← cursor.getNext()
10:         add (key, record) to new component C'
11: function CATCHUP(new component)
12:     S lock dataset
13:     mark side-file closed
14:     unlock dataset
15:     sort side-file
16:     apply updates in side-file to new component C'
17: function DELETE(key)
18:     X lock key
19:     search key from the primary key index
20:     if key exists in immutable component C then
21:         mark key deleted in C
22:         if C points to C' then
23:             try to append key to side-file
24:             if append fails then
25:                 mark key deleted in C'
26:     unlock key
```

deletes a key during the build phase, but aborts after the side-file has been closed, the component builder may re-delete this key based on the side-file.

The writer, which deletes a key using the DELETE function, is similar to that of the lock method, except that the deleted key is first appended to the side-file (line 23). If this fails, which implies that the side-file has been closed, then the deleted key is applied to the new component directly. In the case of rollback, the transaction simply appends an anti-matter key to the side-file if the side-file is still open. Otherwise, the transaction simply unsets the bitmap of the new component to 0.

4.6 Experimental Evaluation

In this section, we evaluate the proposed techniques in the context of Apache AsterixDB [1]. We are primarily interested in evaluating the effectiveness of the various point lookup optimizations (Section 4.6.2) and the ingestion and query performance of the Eager, Validation, and Mutable-bitmap strategies discussed in this chapter (Sections 4.6.3 - 4.6.4). We also evaluate the proposed secondary index repair method in detail (Section 4.6.5). Finally, we examine the performance of the two proposed concurrency control methods for the Mutable-bitmap strategy (Section 4.6.6).

4.6.1 Experimental Setup

All experiments were performed on a single node with a single partition. We used two types of AWS nodes to evaluate the proposed techniques on both hard disks and SSDs. The hard disk node, h1.2xlarge, has a 8-core CPU, 32GB of memory, and a 2TB hard disk. The SSD node, m5d.xlarge, has a 8-core CPU, 32GB of memory, and a 300GB SSD. For both nodes, we used the elastic block store (EBS) to store transaction logs and their native disks for the LSM storage. We allocated 20GB of memory for the AsterixDB instance. Within that allocation, the disk buffer cache size was set at 6GB and the memory component budget was set at 1GB. The OS file system caching was disabled. Each LSM-tree had two memory components to minimize stalls during flushes. We used a tiering merge policy with a size ratio of 1.2 throughout the experiments, similar to the one used in other systems [2, 5]. This policy merges a sequence of components when the total size of the younger components is 1.2 times larger than that of the oldest component in the sequence. The Bloom filter false positive rate setting was 1%. The disk page size was set at 32KB for the hard disk and 16KB for the SSD. For hard disks, whenever a scan was performed, a 4MB read-ahead size was used to minimize random I/Os.

For the experimental workload, although YCSB [40] is a popular benchmark for key-value store

systems, it is not suitable for our evaluation since it does not have secondary keys nor secondary index queries. As a result, we implemented a synthetic tweet generator for our evaluation, as in other LSM secondary indexing work [18, 94]. Each tweet has several attributes such as ID, message_text, user_id, location, and creation_time etc., and its size is about 500 bytes with variations due to the variable length (450 to 550 bytes) of the randomly generated tweet messages. Among these attributes, the following three are related to our evaluation. First, each tweet has an ID as its primary key, which is a randomly generated 64-bit integer. Second, the user_id attribute, which is a randomly generated integer in the range 0 to 100K, is used for formulating secondary index queries with various controlled selectivities. Finally, each tweet has a creation_time attribute, which is a monotonically increasing timestamp used to test the range filter.

4.6.2 Point Lookup Optimizations

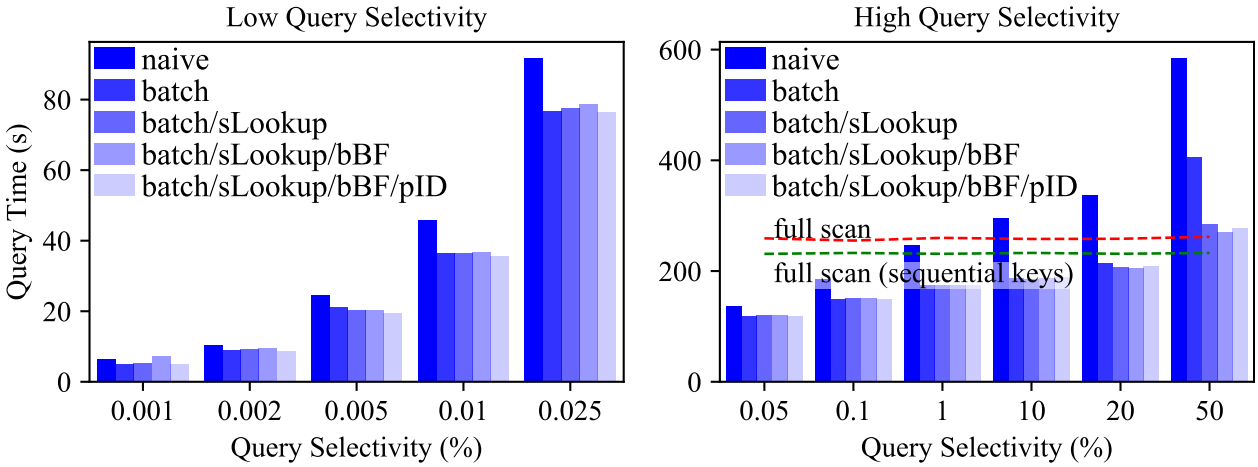
We first studied the effectiveness of the various point lookup optimizations discussed in Section 4.3.2. To this end, we performed a detailed experimental analysis as follows: On top of a naive lookup implementation (denoted as "naive") which only sorts the primary keys, we enabled the batched point lookup with a batch size of 16MB (denoted as "batch"), stateful B^+-tree lookup (denoted as "sLookup"), and blocked Bloom filter (denoted as "bBF") optimizations one by one. We also studied another optimization proposed by Jia [62] (denoted as "pID"). Its basic idea is to propagate and use the component IDs of the secondary index components in which keys are found to prune primary index components during point lookups[4].

To prepare the experiment dataset, we inserted 80 million tweet records with no updates. The resulting primary index has about 30GB of data, while the secondary index on the user_id attribute only has 3GB of data. We then evaluated the query performance for different controlled selectivities based on the user_id attribute. For each query selectivity, queries with different range predicates

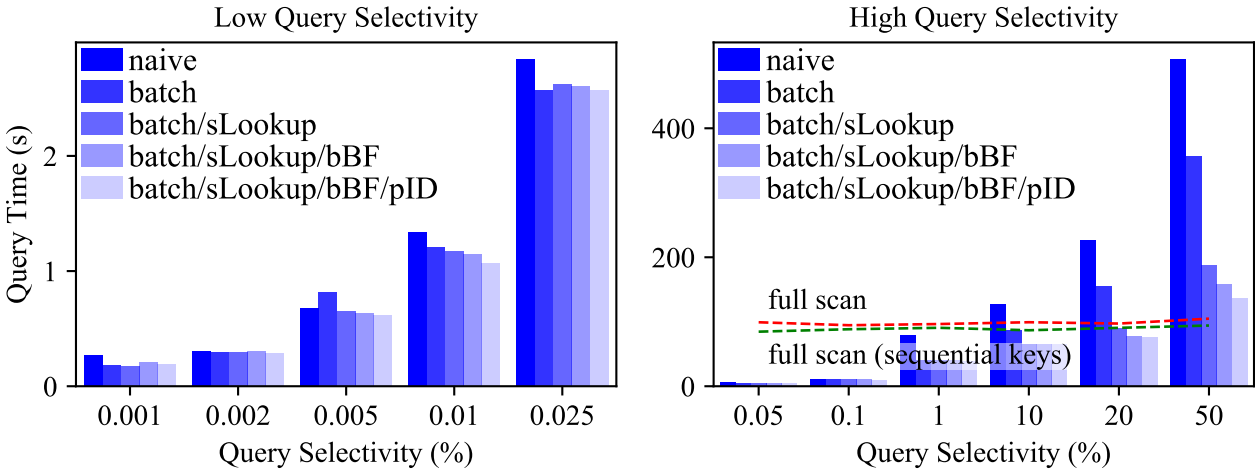
[4]Jia's original technique actually propagated the ranges of the range filter built on a time-correlated attribute; this is equivalent to propagating component IDs in our setting.

were executed until the cache was warmed and the average stable running time is reported.

The running time for different selectivities on the hard disk and the SSD is reported in Figure 4.9. Note that for high query selectivities, we also included the full scan time as a baseline. The upper line represents the full scan time on the experimental dataset. Since disk read-ahead caused long I/O wait times on the hard disk, we further evaluated an optimized case on a dataset prepared with sequential primary keys (lower line). In that case, sequential I/Os and the OS's asynchronous read-ahead were fully exploited to minimize the I/O wait time.



(a) Hard Disk

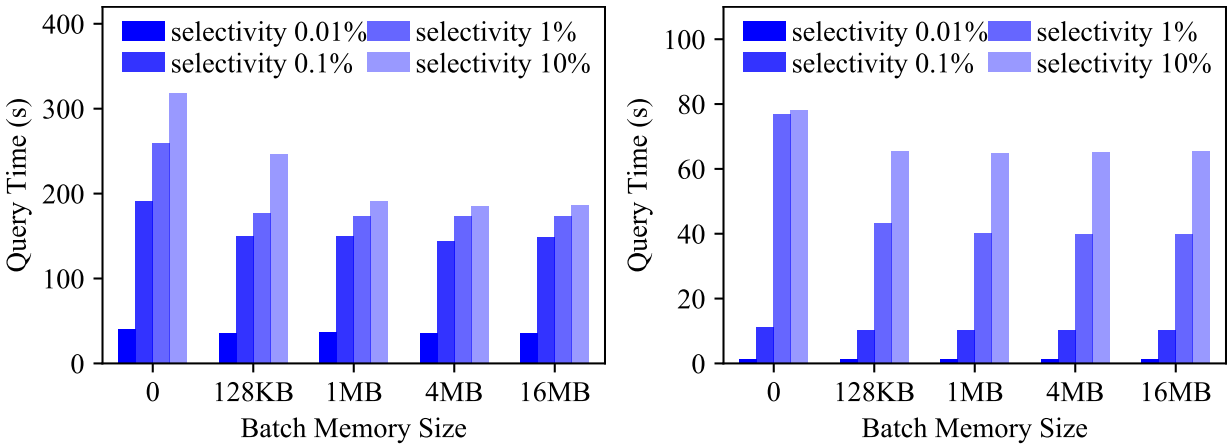


(b) SSD

Figure 4.9: Effectiveness of Point Lookup Optimizations

In general, the results show similar performance trends on the hard disk (Figure 4.9a) and the SSD (Figure 4.9b). When the query selectivity is low, where only a small number of records

are returned, batching slightly improves query performance by reducing random I/Os, while other optimizations are less effective since the query time is dominated by disk I/Os. As the query selectivity increases, the running time of the naive point lookup implementation grows quickly since accessing multiple LSM components leads to random I/Os. The batched point lookup is the most effective at avoiding random I/Os. Even though random I/Os are no longer a problem on SSDs, avoiding random I/Os via batching still improves query performance by better exploiting the OS read-ahead mechanism. The stateful B^+-tree lookup and blocked Bloom filter optimizations further reduce the in-memory search cost since the disk I/O cost is bounded as most pages must be accessed. Note that for large selectivities, a full scan starts to outperform a secondary index search since most pages of the primary index must be accessed while a secondary index search incurs the extra cost of accessing the secondary index and sorting primary keys. However, these optimizations together have greatly improved the range of applicability of LSM-based secondary indexes, allowing query optimizers, especially rule-based ones, to have more confidence when choosing secondary indexes. Contrary to Jia [62], we found here that propagating component IDs provides little benefit. The reason is that Jia [62] considered an append-only temporal workload with hundreds of (filtered) LSM components in a dataset, so the propagation of component IDs led to the skipping of a large number of Bloom filter tests. However, in a more general setting, skipping a smaller number of Bloom filter tests per key would not make so big a difference.



(a) Hard Disk (b) SSD

Figure 4.10: Impact of Batch Size

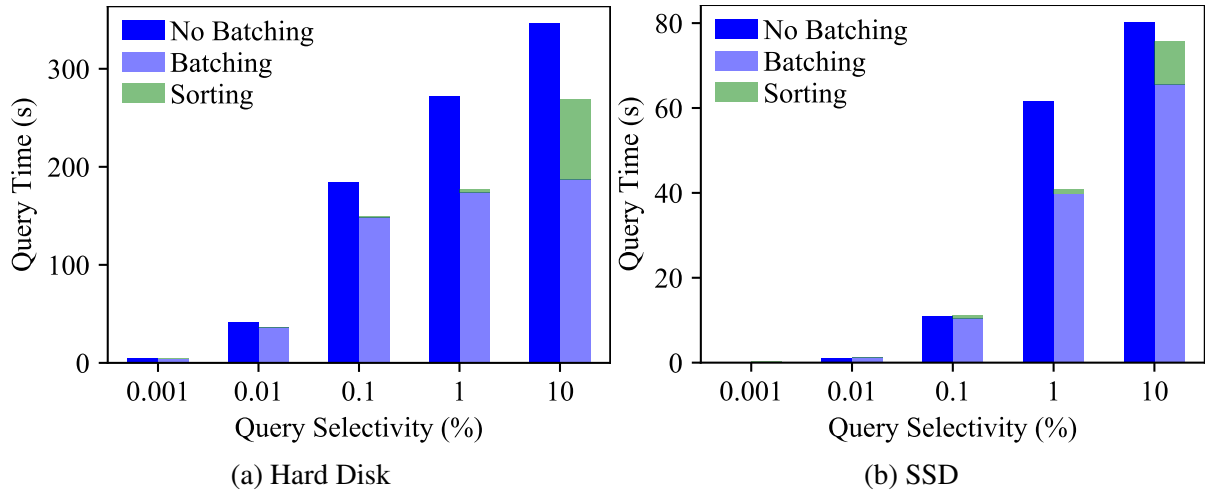


Figure 4.11: Impact of Sorting

We further evaluated batched point lookup in terms of the available batching memory and the sorting overhead. In each experiment that follows, stateful B$^+$-tree lookup and blocked Bloom filter were enabled by default. The running time of different query selectivities under different batch sizes is shown in Figure 4.10. On both the hard disk and the SSD, a small batch size such as 128KB already provides optimal performance when the query selectivity is small, as a batch of keys are often distributed over a large number of pages. Even when the query selectivity becomes larger, a few MBs of the batching memory is sufficient to provide optimal performance. Finally, since batching destroys the primary key ordering of the final results, we evaluated the sorting overhead by either not using batching or using batching plus sorting. The running time of different query selectivities with these two query plans is shown in Figure 4.11. Even though sorting must be performed, batching still improves the overall query performance. This is because the point lookup step needs to fetch records distributed across a large number of pages, while the resulting records can often fit into a small number of pages and can be sorted efficiently.

To summarize, batched point lookup is the most effective optimization for reducing random I/Os when accessing LSM components. Stateful B$^+$-tree lookup and blocked Bloom filter are mainly effective for non-selective queries at further reducing the in-memory search cost.

4.6.3 Ingestion Performance

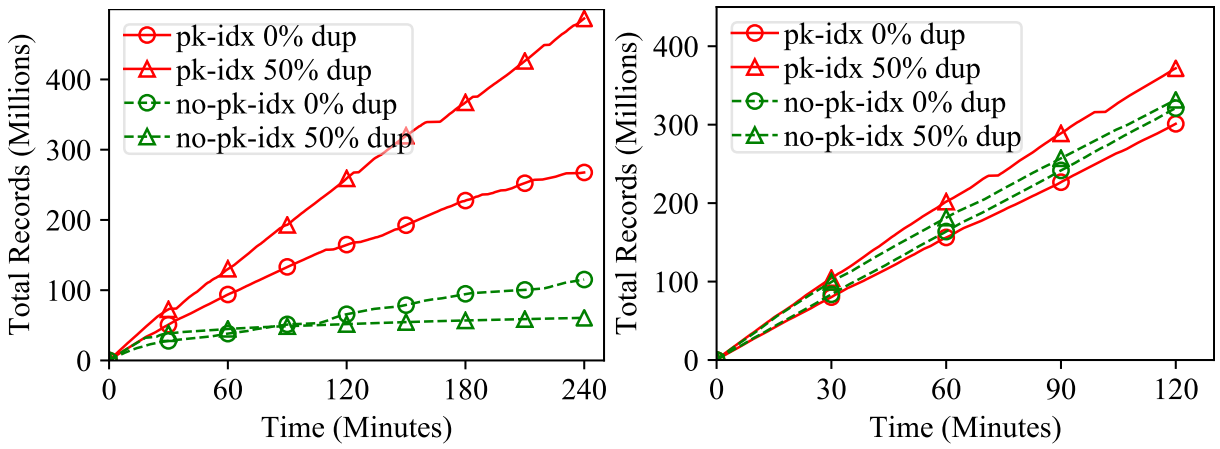
We next evaluated the ingestion performance of the different maintenance strategies, focusing on the following key questions: (1) What is the effectiveness of building a primary key index, which reduces the point lookup cost, for improving the overall ingestion throughput, since maintaining the primary key index itself incurs extra cost? (2) Does repairing secondary indexes become a new bottleneck, especially for a dataset with multiple secondary indexes? (3) What is the effectiveness of building a single primary key index and the proposed repair algorithm (Section 4.4.4) for improving the ingestion throughput as compared to the deleted-key B^+-tree strategy supported by AsterixDB?

To answer these questions, we used insert and upsert workloads in the following evaluation. Delete workloads were omitted since the cost of deletes would be similar to upserts; that is, an upsert is logically equivalent to a delete plus an insert. In each experiment below, we ingested as much data as possible without concurrent queries. Each dataset had a primary index with a component-level range filter on the creation_time attribute, a primary key index, and a secondary index on the user_id attribute.

Insert Workload

For inserts, we evaluated two methods for enforcing key uniqueness, using either the primary index or the primary key index. In the former case, the primary key index was omitted to eliminate its overhead. This workload was controlled by a duplicate ratio, which is the ratio of duplicates among all records. Duplicates were randomly generated following a uniform distribution over all the past keys.

Figure 4.12 shows the insert throughput under different duplicate ratios on the hard disk and the SSD. On the hard disk (Figure 4.12a), the ingestion throughput degrades quickly without the pri-



(a) Hard Disk (b) SSD

Figure 4.12: Insert Ingestion Performance

mary key index once the dataset could not be totally cached. Building a primary key index greatly improves the ingestion throughput since the keys are much smaller and can be better cached to reduce disk I/Os caused by point lookups. Even though the ingestion throughput drops when the primary key index cannot be totally cached, building a primary key index is still helpful by increasing the cache hit ratio. The duplicate-heavy workload results in higher ingestion throughput when using the primary key index since duplicate keys are simply excluded from insertion into the storage. However, when the primary key index is not built, the duplicate-heavy workload reduces the throughput because of a large number of random I/Os for the uniqueness check.

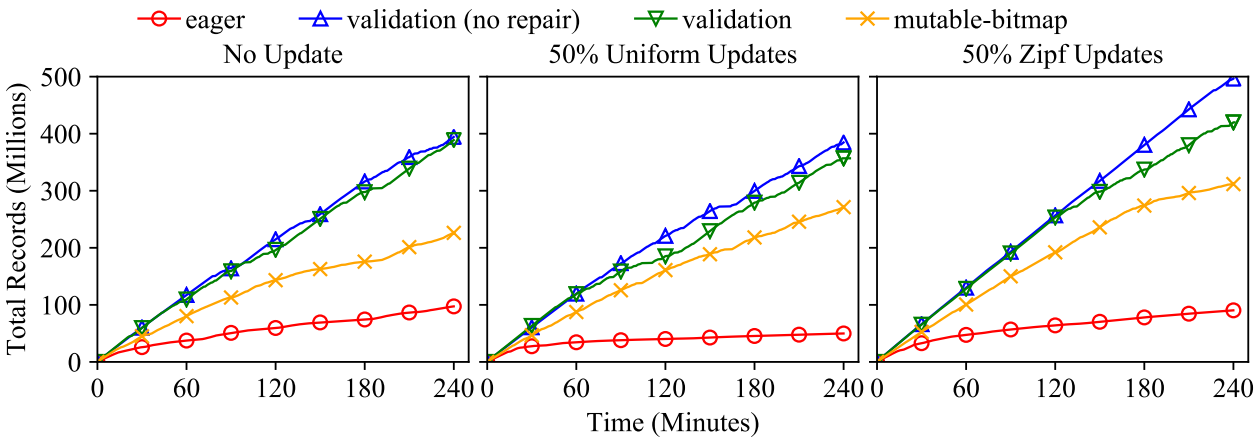
In contrast, on the SSD (Figure 4.12b), the benefit of maintaining a primary key index becomes smaller since SSDs can support efficient random I/Os. Moreover, under the append only workload, building a primary key index slightly reduces the ingestion throughput because of the overhead required to maintain the primary key index. However, this overhead is quite negligible and using a primary key index still improves the ingestion throughput for the duplicate-heavy workload by reducing disk I/Os.

Upsert Workload

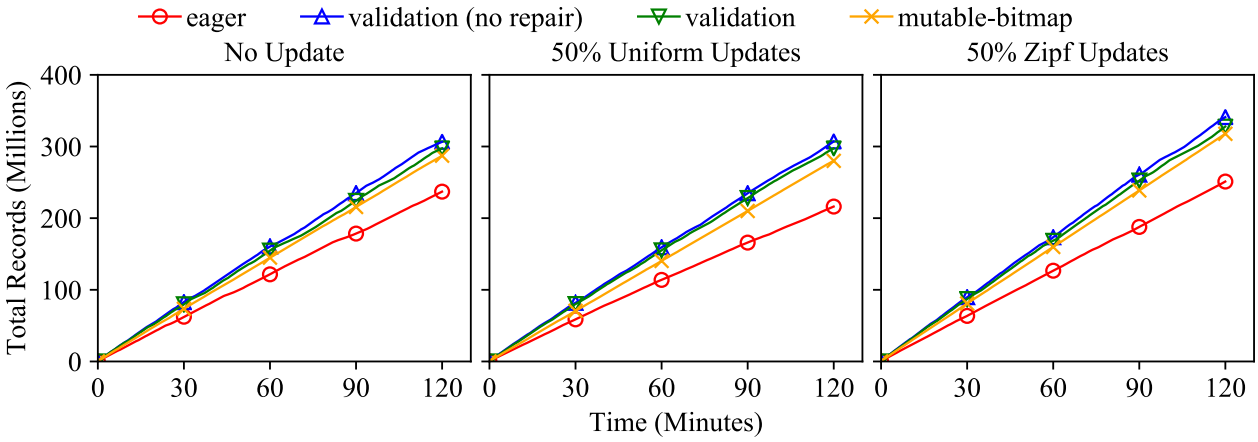
The upsert workload is the main focus of our evaluation since the three maintenance strategies discussed in this chapter mainly differ in how upserts are handled. This workload was controlled by an update ratio, which is the ratio of updates (records with past ingested keys) among total records. Updates were randomly generated by following either a uniform distribution, that is, all past keys were updated equally, or a Zipf distribution with a theta value 0.99 as in the YCSB benchmark [40], that is, recently ingested keys were updated more frequently. Unless otherwise specified, the update ratio was chosen as 10% and the updates followed a uniform distribution.

For the Validation strategy, we evaluated two variations to measure the overhead of repairing secondary indexes. In the first variation, repair was totally disabled to maximize ingestion performance. In the second variation, merge repair was enabled together with the Bloom filter optimization (Section 4.4.4). For the Mutable-bitmap strategy, the secondary index was maintained using the Validation strategy without repair to minimize the ingestion overhead due to secondary indexes. The Side-file method was used for concurrency control to minimize the locking overhead.

Basic Upsert Ingestion Performance. In this experiment, we compared the proposed Validation and Mutable-bitmap strategies with the Eager strategy in terms of the upsert ingestion performance under different update ratios, ranging from no updates to 50% updates following either the uniform or Zipf distribution. The experimental results are shown in Figure 4.13. The Eager strategy, which ensures that secondary indexes are always up-to-date, has the worst ingestion performance on both the hard disk and the SSD because of the point lookups to maintain secondary indexes using the old records. On the other hand, the Validation strategy without repairing has the best ingestion performance since secondary indexes are not cleaned up at all. Note that the benefit of the Validation strategy becomes smaller on the SSD because the point lookup overhead becomes much smaller. The result also shows that repairing secondary indexes incurs only a small amount of extra overhead. Of course, since secondary indexes are not always up-to-date, the Validation strategy



(a) Hard Disk



(b) SSD

Figure 4.13: Upsert Ingestion Performance

sacrifices query performance, which we will evaluate later in Section 4.6.4. The Mutable-bitmap strategy also has much better ingestion performance than the Eager strategy since it only searches the primary key index instead of accessing full records. Updates generally have a small impact on the overall ingestion throughput since updated records must be inserted into memory and subsequently flushed and merged (as for inserted records). The Eager and Mutable-bitmap strategies both benefit from skewed update workloads since most of the updates only touch recent keys, reducing disk I/Os incurred by the point lookups. These results again confirm that it is helpful to build a primary key index to reduce the point lookup cost.

Ingestion Impact of Secondary Indexes. We further evaluated the scalability of the strategies

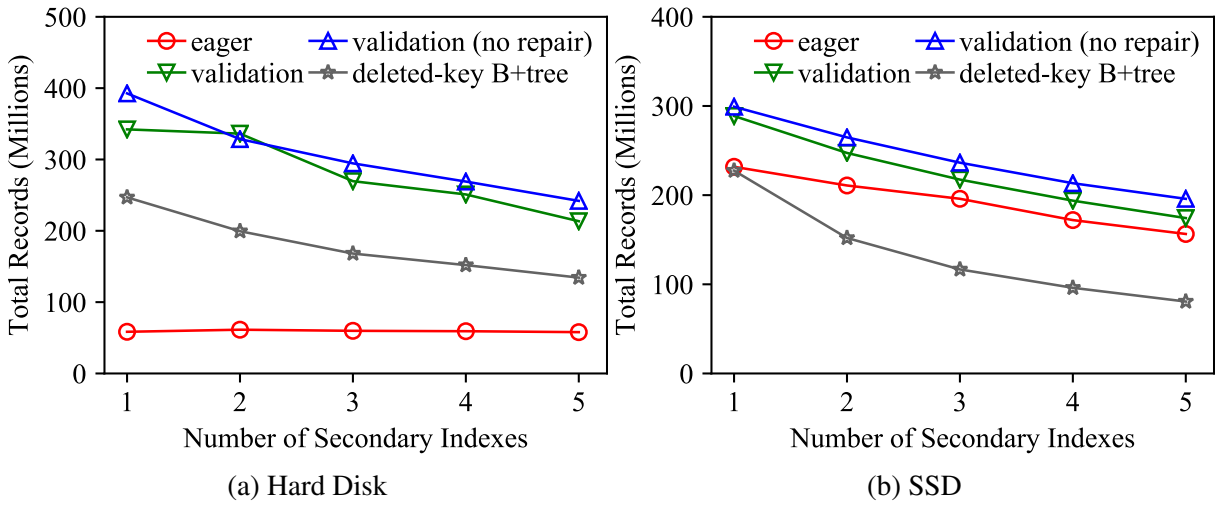


Figure 4.14: Impact of Secondary Indexes on Upsert Ingestion Performance

by adding more secondary indexes. We excluded the Mutable-bitmap strategy since it is unaffected by secondary indexes, but instead included the deleted-key B$^+$-tree strategy for comparison. Figure 4.14 shows the ingestion throughput for different number of secondary indexes. On both the hard disk and the SSD, having more secondary indexes negatively impacts the ingestion performance of all the strategies since more LSM-trees must be maintained. On the hard disk (Figure 4.14a), the Eager strategy preforms poorly due to the expensive point lookups. By deferring point lookups to the background into large batches, the deleted-key B$^+$-tree strategy improves the ingestion throughput. However, the relative performance trend flips on the SSD (Figure 4.14b), where the Eager strategy performs better than the deleted-key B$^+$-tree strategy. This is because point lookups become much cheaper on SSDs, but the merge operation of the deleted-key B$^+$-tree strategy, which cleans up secondary indexes, becomes a new bottleneck. The proposed Validation strategy improves the ingestion throughput on both the hard disk and the SSD. Moreover, the experiment also shows the scalability of the proposed index repair operations since they introduce just a small overhead on data ingestion. Comparing to the current AsterixDB deleted-key B$^+$-tree strategy, the negative impact of index repair operations has been greatly reduced by using a single primary key index and the efficient repair algorithm presented in Section 4.4.4.

4.6.4 Query Performance

We next evaluated the query performance of the different maintenance strategies, focusing on the following two aspects. First, we evaluated the overhead of the Validation strategy with both non-index-only and index-only queries compared to the read-optimized Eager strategy and the benefit of repairing secondary indexes. Second, we evaluated the pruning capabilities of filters resulting from different maintenance strategies. As in Section 4.6.2, each dataset that follows was prepared by upserting 80 million records with different actual update ratios (0% or 50%).

Secondary Index Query Performance

We first evaluated the overhead of the Validation strategy on secondary index queries. We again considered two variations of the Validation strategy, depending on whether merge repair was enabled, to evaluate the benefit of repairing secondary indexes. Each query for a given selectivity was repeated with different range predicates until the cache was warmed and the average stable time is reported.

Non-Index-Only Query Performance. For non-index-only queries, we enabled batching (with 16MB memory), stateful B^+-tree search, and blocked Bloom filter to optimize the subsequent point lookups. For the Validation strategy, we evaluated both Direct Validation (denoted as "direct") and Timestamp Validation (denoted as "ts"). The running time of non-index-only queries is shown in Figure 4.15. In general, all evaluated strategies have similar performance trends on the hard disk (Figure 4.15a) and the SSD (Figure 4.15b). For the append-only workload (left side), the Direct Validation method has similar performance to the Eager strategy since secondary indexes do not contain any obsolete entries. However, the Timestamp Validation method leads to some extra overhead because of the validation step. The results are different for the update-heavy workload (right side). When the secondary index is not repaired, the Direct Validation method leads to some overhead for selective queries, as some disk I/Os are wasted searching for obsolete keys. The

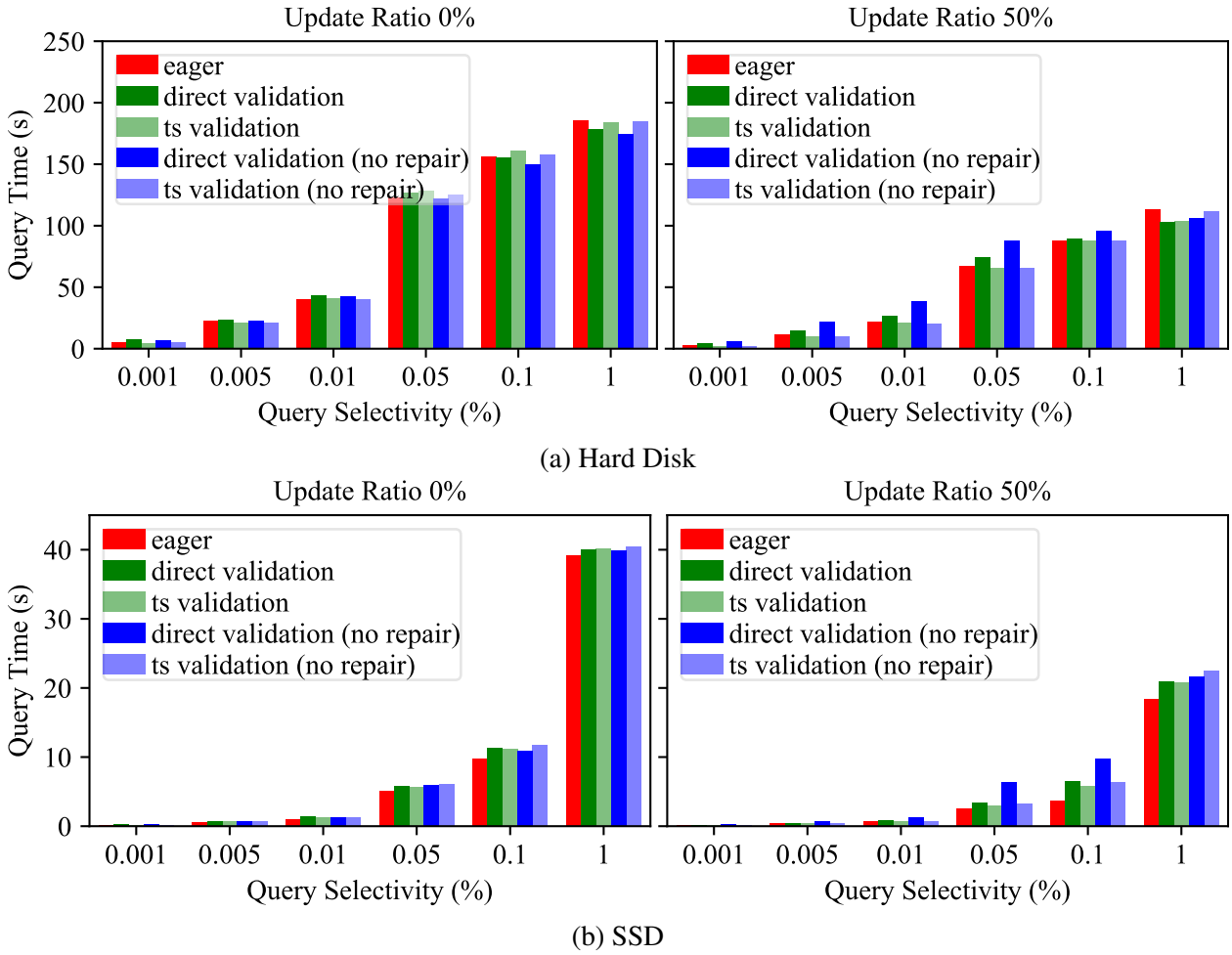
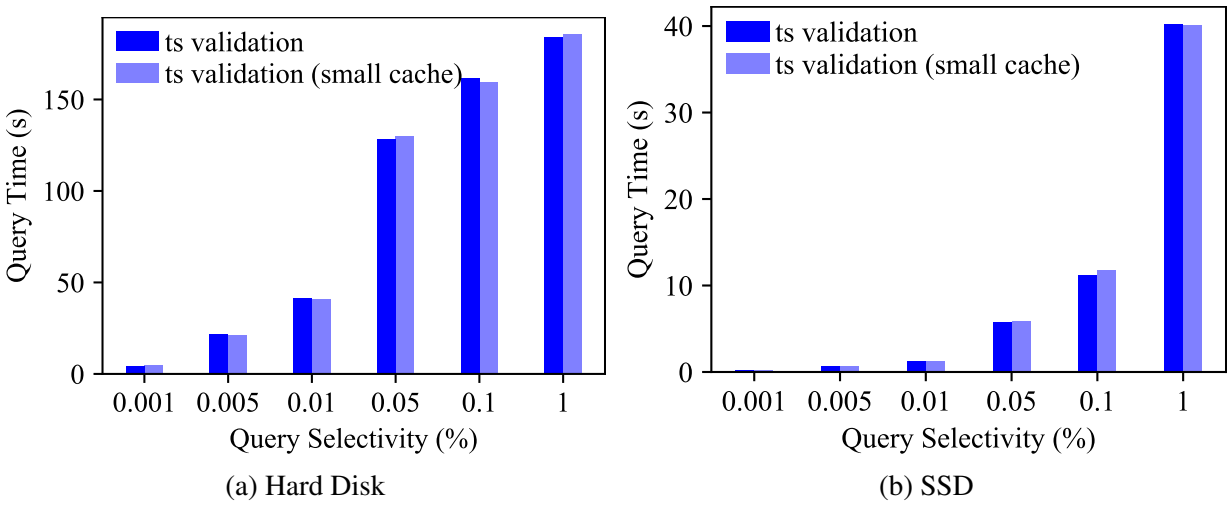


Figure 4.15: Non-Index-Only Query Performance

Timestamp Validation method is helpful for reducing wasted I/Os by filtering out the obsolete keys using the primary key index when the query selectivity is small. When the selectivity becomes larger, such as 1%, the extra overhead of both validation methods diminishes since even searches for valid keys must access almost all pages of the primary index. With merge repair, most of the obsolete keys would be removed, which improves the query performance of both validation methods.

We further evaluated the performance of the Timestamp Validation method under a smaller cache memory setting (only 512MB) so that the primary key index cannot totally fit into memory. The dataset used in this experiment contains no updates. Figure 4.16 depicts the query time of this experiment on both the hard disk and the SSD. In general, we found that a small cache setting has

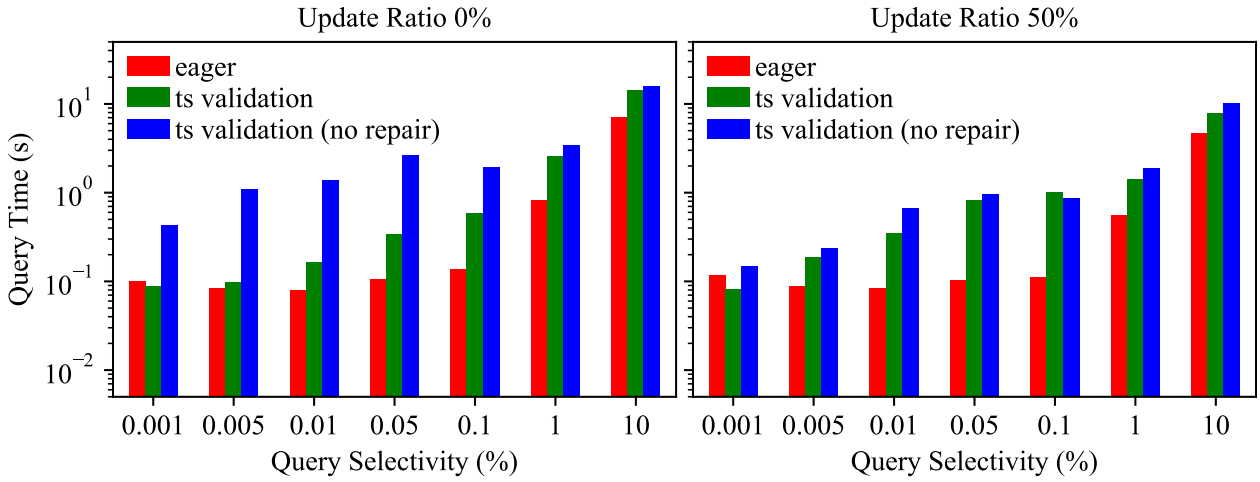
limited impact on the Timestamp Validation method since the primary key index is much smaller than the primary index, resulting in only a small amount of extra I/Os during validation.



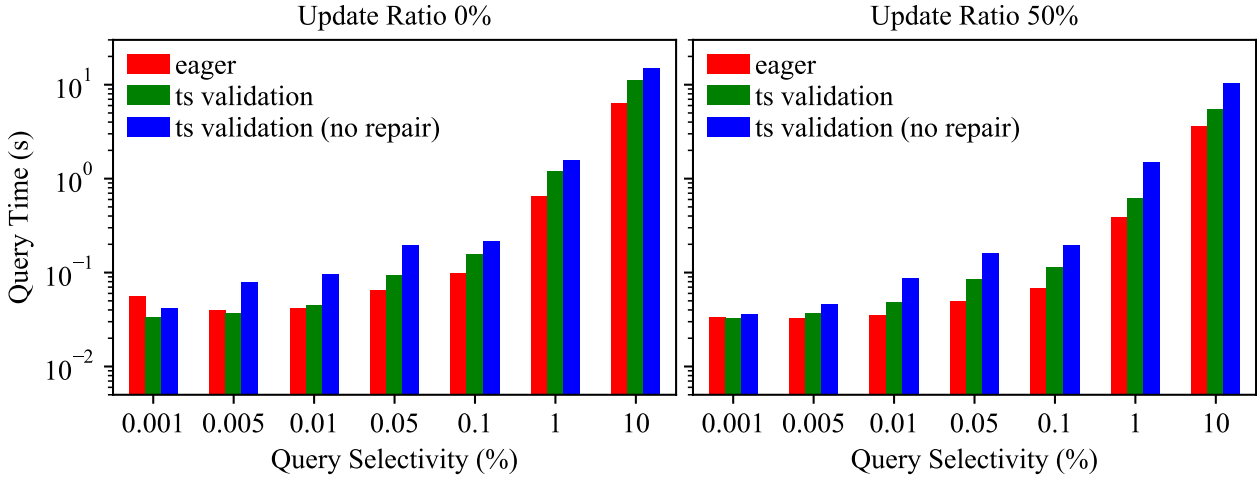
(a) Hard Disk (b) SSD

Figure 4.16: Impact of Small Cache on Timestamp Validation

Index-Only Query Performance. Figure 4.17 shows the query time (log scale) for index-only queries. We omit the Direct Validation method since it must fetch records for validation and thus has same performance as for non-index-only queries. In general, the Validation strategy performs worse than the Eager strategy on both the hard disk (Figure 4.17a) and the SSD (Figure 4.17b) because of its extra sorting and validation steps. Even without obsolete entries (left side), the validation step still leads to extra overhead. Still, Timestamp Validation provides much better performance than Direct Validation would by accessing the primary key index. Moreover, under small query selectivities, Timestamp Validation incurs a larger query overhead on the hard disk than on the SSD. This is because Timestamp Validation incurs additional disk I/Os when accessing obsolete secondary index entries and the primary key index, and the disk I/O cost becomes much smaller on the SSD. Finally, merge repair is helpful for index-only queries by increasing repaired timestamps to prune more primary key index components during validation, as shown in the append-only case (left side), and by cleaning up obsolete entries for the update-heavy case (right side).



(a) Hard Disk



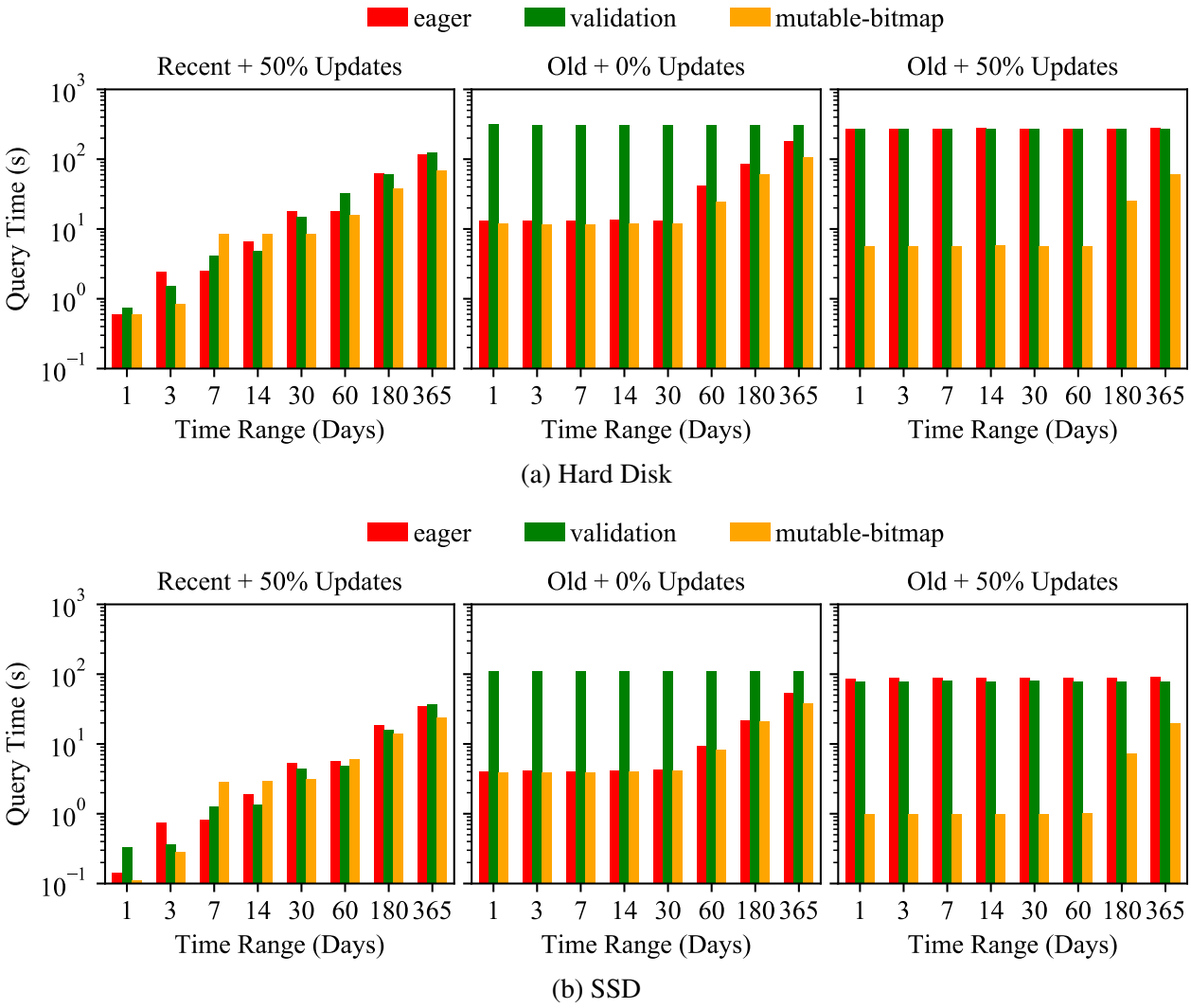
(b) SSD

Figure 4.17: Index-Only Query Performance

Range Filter Query Performance

We next evaluated the effectiveness of the maintenance strategies for filters. In particular, we used the range filter in our evaluation. Recall that the range filter was built on the creation_time attribute, which is a monotonically increasing timestamp. The range of the creation_time attribute of all tweet records in the experiment dataset spanned about 2 years. Each query in this experiment has a range predicate on the creation_time attribute, and it is processed by scanning the primary index with the component-level pruning provided by the range filters. We evaluated two types of queries, queries that access recent data (with creation_time $>$ T) and that access old data (with creation_time $<$ T).

Each query was repeated 5 times with a clean cache for each run, and the average query time is reported. In addition to the Eager and Mutable-bitmap strategies, we included a variation of the Validation strategy that simply maintains the filters using new records in which case queries must check in all newer components for correctness.



(a) Hard Disk

(b) SSD

Figure 4.18: Query Performance of Range Filters

The query times (log scale) of range filters are summarized in Figure 4.18. In general, all the strategies exhibit similar performance trends on both the hard disk (Figure 4.18a) and the SSD (Figure 4.18b) . For the queries that access recent data[5], all the strategies provide effective pruning

[5]Here we omitted the results for the append-only case since all the strategies have the identical performance trends as in the update-heavy case.

capabilities. The Mutable-bitmap strategy further improves scan performance since LSM components are accessed one by one and the reconciliation step is no longer needed. In contrast, when accessing old data, the Validation strategy provides no pruning capability since all newer components must be accessed to ensure correctness. The Eager strategy is only effective for the append-only case, but recall that its point lookups lead to high cost during data ingestion. The Mutable-bitmap strategy provides effective pruning capabilities via the use of Mutable bitmaps under all settings, and does so with only a small amount of overhead on data ingestion.

4.6.5 Index Repair Performance

We then evaluated the index repair performance of the proposed Validation strategy (referred as *secondary repair*) in detail, as well as the proposed Bloom filter optimization (denoted as "bf"). For comparison, we also evaluated the index repair method proposed by DELI [112] (referred as *primary repair*). Recall that DELI repairs secondary indexes by merging or scanning primary index components to identify obsolete records, while our approach uses a primary key index to avoid accessing full records.

In each of the following experiments, we upserted 100 million tweet records with merge repair enabled. Since the two methods trigger repair operations differently, the resulting secondary indexes may have different amounts of obsolete entries during data ingestion. To enable a fair comparison, instead of measuring the overall ingestion throughput, we measured the index repair performance directly as follows. For every 10 million records ingested, we stopped the ingestion and triggered a full repair operation to bring all secondary indexes up-to-date. This shows the performance trend of index repair operations as data accumulates. For our secondary repair method, standalone repair was performed to exclude the extra overhead due to merges.

Basic Repair Performance. We first evaluated the index repair performance under different update ratios (0% and 50%). For primary repair [112], we evaluated two variations depending

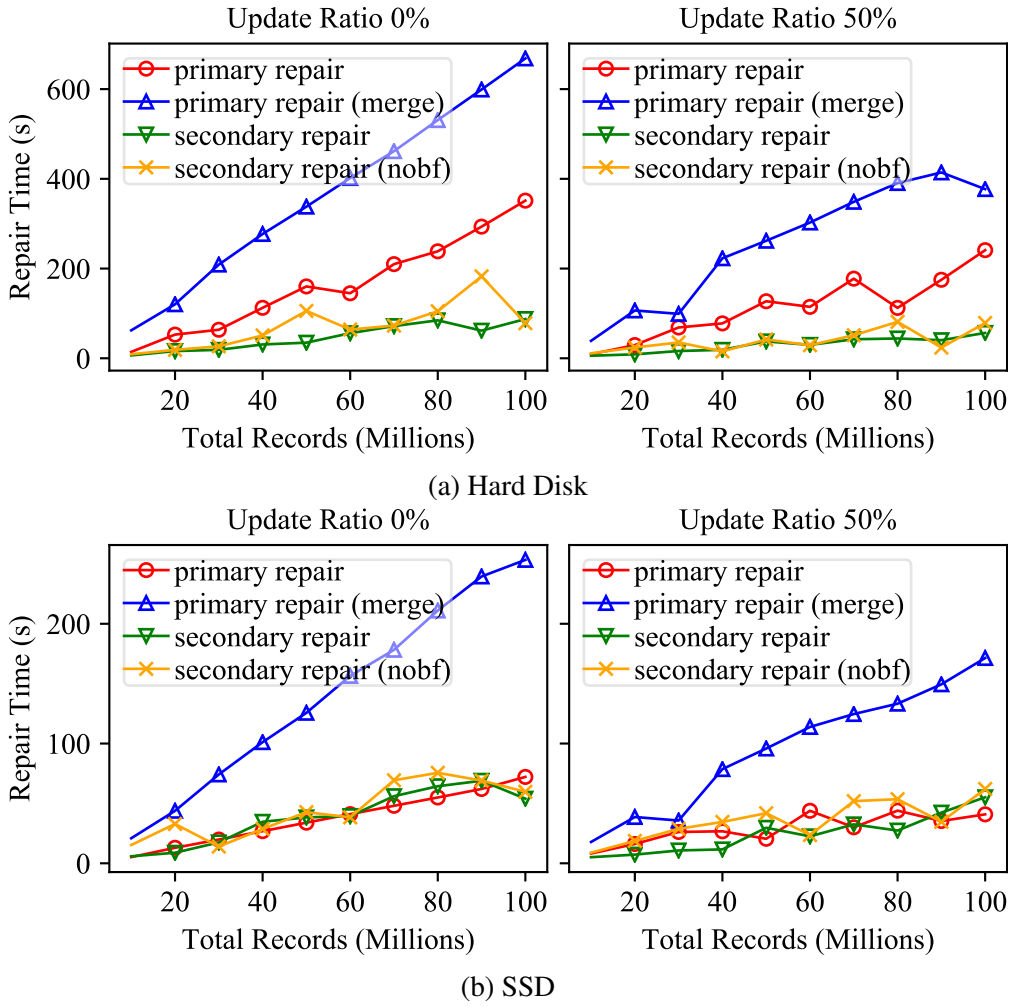
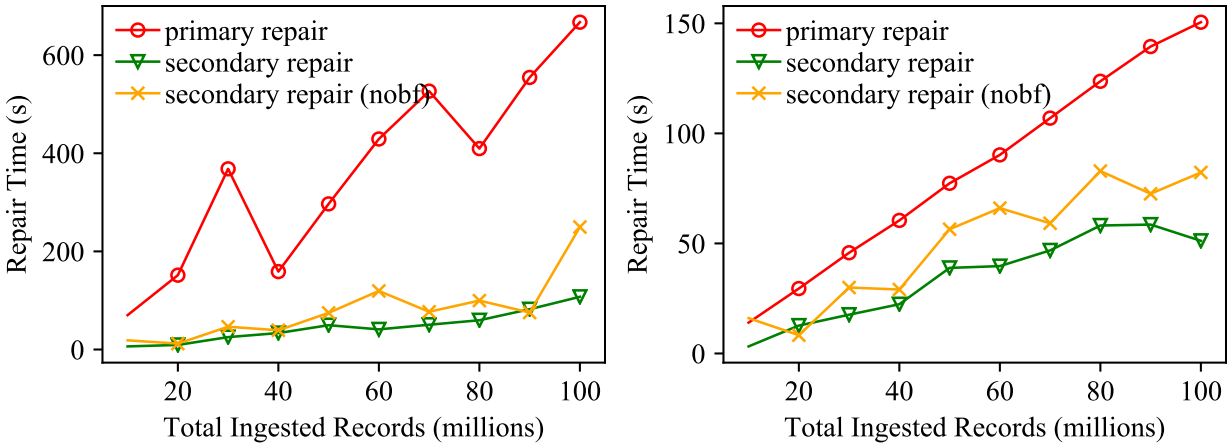


Figure 4.19: Repair Performance with Varying Update Ratio

whether a full merge is performed as a by-product of the repair operation. The index repair performance over time is depicted in Figure 4.19, where each data point represents the time to complete a repair operation. For primary repair, a full merge always leads to extra overhead due to the extra merges on both the hard disk and the SSD. The secondary repair method proposed here always outperforms the primary repair methods on the hard disk (Figure 4.19a) because only it only accesses the primary key index, significantly reducing disk I/Os. However, on the SSD (Figure 4.19a), both the primary and secondary repair methods have similar repair performance because the disk I/O cost has been greatly reduced. Finally, the Bloom filter optimization further improves repair performance on both the hard disk and the SSD by reducing the volume of primary keys to be sorted and validated.

Impact of Large Records. In this set of experiments, we used large record size 1KB to evaluate the impact of large records. The result is depicted in Figure 4.20, where the update ratio is set to 10%. On both the hard disk (Figure 4.20a) and the SSD (Figure 4.20b), large records negatively impact the primary repair method since more disk I/Os must be performed. However, large records have little impact over the secondary repair method, as it only accesses the primary key index.



(a) Hard Disk (b) SSD

Figure 4.20: Repair Impact of Large Records

Impact of Secondary Indexes. We next used 5 secondary indexes to evaluate the scalability of the index repair methods. The update ratio was set to 10%. For the secondary repair method, each secondary index was repaired in parallel. The index repair time is depicted in Figure 4.21. Having more secondary indexes negatively impacts the performance of both methods on both the hard disk and the SSD. For the primary repair method, more anti-matter entries must be inserted into more secondary indexes. For the secondary repair method, more secondary index entries must be scanned and sorted for validation. The result does show that the proposed secondary index repair method is easily parallelizable, as it only performs a small amount of I/Os and most operations are CPU-heavy. Furthermore, the Bloom filter optimization reduces the negative impact of having more secondary indexes, as it sorts fewer keys, significantly reducing the I/O overhead during the sort step.

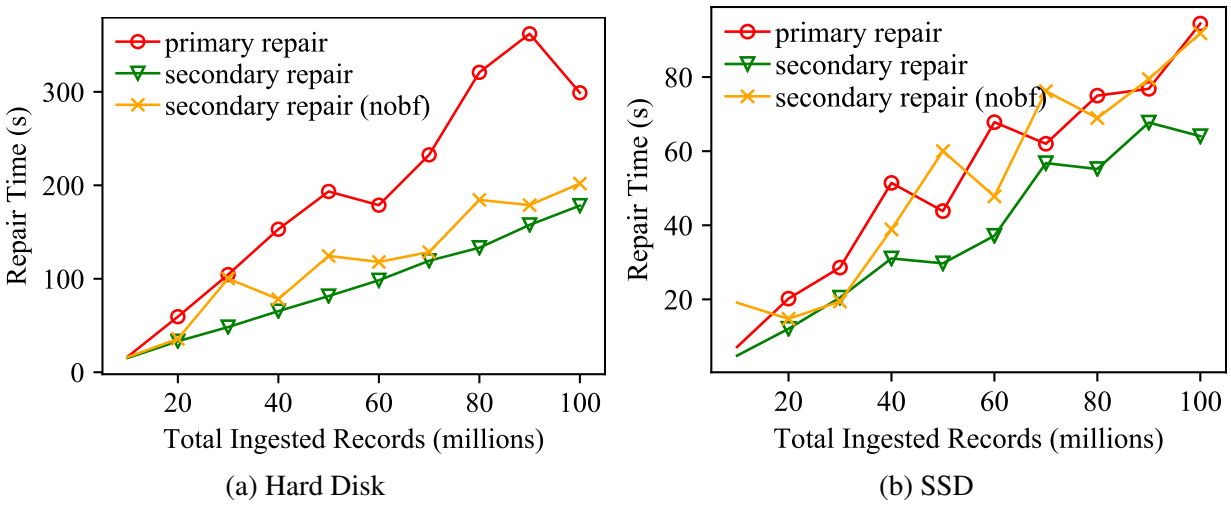
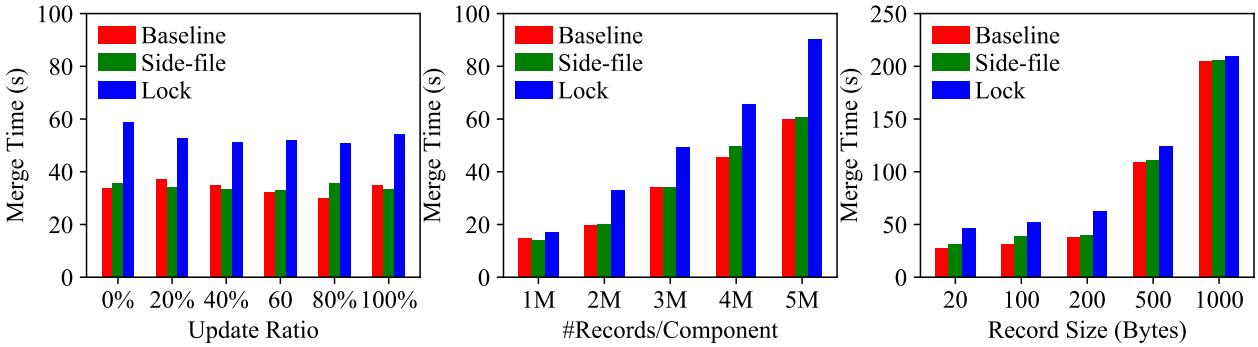


Figure 4.21: Repair Impact of Secondary Indexes

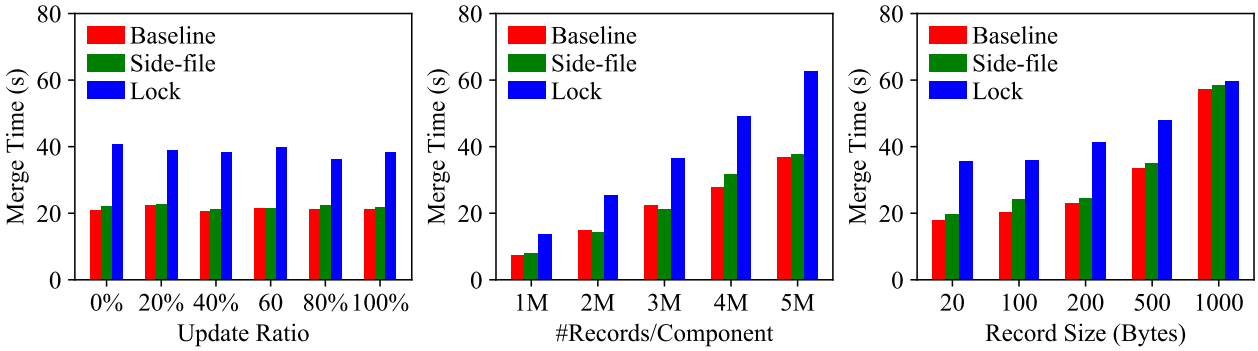
4.6.6 Concurrency Control for Mutable-Bitmap

In the final set of experiments, we evaluated the overhead of the alternative concurrency control methods proposed for the Mutable-bitmap strategy, i.e., the Lock method and the Side-file method. We chose the merge time without any concurrency control as the baseline. In each experiment, we merged 4 components with concurrent data ingestion at the maximum speed. Unless otherwise noted, each component had 3 million records, each record had 100 bytes, and the update ratio of the newly ingested records was set to 50%.

We evaluated the impact of update ratio, the size of merged components, and the record size on the performance of the proposed concurrency control methods. The experiment results are summarized in Figure 4.22. In general, the Side-file method incurs negligible overhead on both the hard disk (Figure 4.22a) and the SSD (Figure 4.22b) against the baseline since no locks are acquired on a record-basis. In contrast, the Lock method performs worse in all the settings because of the locking overhead, even though the locking overhead is marginalized as the record size grows larger. The Lock method also benefits from updates since the updated entries are simply skipped during the merge, while the Side-file method has to apply updates later.



(a) Hard Disk



(b) SSD

Figure 4.22: Overhead of Mutable-Bitmap Concurrency Control Methods

4.7 Conclusions

In this chapter, we have presented techniques for efficient maintenance and exploitation of LSM-based auxiliary structures. We described and evaluated a series of optimizations for efficient point lookups, greatly improving the range of applicability for LSM-based secondary indexes. We further presented new and efficient strategies for maintaining LSM-based auxiliary structures, including secondary indexes and filters. The Validation strategy defers secondary index maintenance to the background using a primary key index, improving ingestion performance by eliminating ingestion-time point lookups. This leads to a small overhead for non-index-only queries, but a relatively high overhead for index-only queries because of the extra validation step. The Mutable-bitmap strategy maximizes the pruning capabilities of filters and improves ingestion performance by only accessing the primary key index, instead of full records, during data ingestion. Our experimental results show that both strategies can significantly improve ingestion performance on hard

disks by eliminating expensive random I/Os. Even on SSDs, where random I/Os are no longer a problem, the proposed strategies can still improve ingestion performance by reducing ingestion-time point lookups.

Chapter 5

Performance Stability of LSM-based Storage Systems

5.1 Introduction

Despite their popularity, LSM-trees have been criticized for suffering from write stalls and large performance variances [3, 103, 131]. To illustrate this problem, we conducted a micro-experiment on RocksDB [10], a state-of-the-art LSM-based key-value store, to evaluate its write throughput on SSDs using the YCSB benchmark [40]. The instantaneous write throughput over time is depicted in Figure 5.1. As one can see, the write throughput of RocksDB periodically slows down after the first 300 seconds, which is when the system has to wait for background merges to catch up. Write stalls can significantly impact percentile write latencies and must be minimized to improve the end-user experience or to meet strict service-level agreements [61].

In this chapter, we study the impact of write stalls and how to minimize write stalls for various LSM-tree designs. It should first be noted that some write stalls are inevitable. Due to the inherent mismatch between fast in-memory writes and slower background I/O operations, in-memory writes

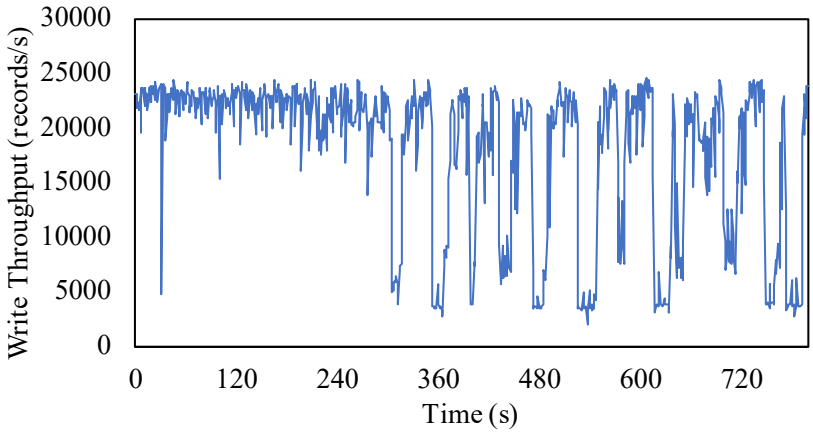


Figure 5.1: Instantaneous Write Throughput of RocksDB

must be slowed down or stopped if background flushes or merges cannot catch up. Without such a flow control mechanism, the system will eventually run out of memory (due to slow flushes) or disk space (due to slow merges). Thus, it is not a surprise that an LSM-tree can exhibit large write stalls if one measures its maximum write throughput by writing as quickly data as possible, such as we did in Figure 5.1.

This inevitability of write stalls does not necessarily limit the applicability of LSM-trees since in practice writes do not arrive as quickly as possible, but rather are controlled by the expected data arrival rate. The data arrival rate directly impacts the write stall behavior and resulting write latencies of an LSM-tree. If the data arrival rate is relatively low, then write stalls are unlikely to happen. However, it is also desirable to maximize the supported data arrival rate so that the system's resources can be fully utilized. Moreover, the expected data arrival rate is subject to an important constraint - it must be smaller than the processing capacity of the target LSM-tree. Otherwise, the LSM-tree will never be able to process writes as they arrive, causing infinite write latencies. Thus, to evaluate the write stalls of an LSM-tree, the first step is to choose a proper data arrival rate.

As the first contribution, we propose a simple yet effective approach to evaluate the write stalls of various LSM-tree designs by answering the following question: If we set the data arrival rate close

to (e.g., 95% of) the maximum write throughput of an LSM-tree, will that cause write stalls? In other words, can a given LSM-tree design provide both a high write throughput and a low write latency? Briefly, the proposed approach consists of two phases: a *testing* phase and a *running* phase. During the testing phase, we experimentally measure the maximum write throughput of an LSM-tree by simply writing as much data as possible. During the running phase, we then set the data arrival rate close to the measured maximum write throughput as the limiting data arrival rate to evaluate its write stall behavior based on write latencies. If write stalls happen, the measured write throughput is not *sustainable* since it cannot be used in the long-term due to the large latencies. However, if write stalls do not happen, then write stalls are no longer a problem since the given LSM-tree can provide a high write throughput with small performance variance.

Although this approach seems to be straightforward at first glance, there exist two challenges that must be addressed. First, how can we accurately measure the maximum sustainable write rate of an LSM-tree experimentally? Second, how can we best schedule LSM I/O operations so as to minimize write stalls at runtime? In the remainder of this chapter, we will see that the merge scheduler of an LSM-tree can have a large impact on write stalls. As the second contribution, we identify and explore the design choices for LSM merge schedulers and present a new merge scheduler to address these two challenges.

As the chapter's final contribution, we have implemented the proposed techniques and various LSM-tree designs inside Apache AsterixDB [17]. This enabled us to carry out extensive experiments to evaluate the write stalls of LSM-trees and the effectiveness of the proposed techniques using our two-phase evaluation approach. We argue that with proper tuning and configuration, LSM-trees can achieve both a high write throughput and small performance variance.

The remainder of this chapter is organized as follows: Section 5.2 discusses some work related to this chapter. Section 5.3 describes the general experimental setup used throughout this chapter. Section 5.4 identifies the design choices for LSM merge schedulers and evaluates bLSM's spring-and-gear scheduler [103]. Sections 5.5 and 5.6 present our techniques for minimizing write

stalls for full merges and partitioned merges respectively. Section 5.8 summarizes the lessons and insights from our evaluation. Finally, Section 5.9 concludes the chapter.

5.2 Related Work

In Chapter 3, we have extensively surveyed the existing research on optimizing LSM-trees. In this section, we will focus on the recent research related to the performance stability aspect of LSM-trees and database systems.

Write Stalls in LSM-trees. Several LSM-tree implementations seek to bound the write processing latency to alleviate the negative impact of write stalls [69, 131]. Similarly, modern key-value stores, such as LevelDB [6] and RocksDB [10], add small artificial delays to write requests before writes are forced to completely stop. bLSM [103], the most closely related idea to this work, proposes a spring-and-gear merge scheduler to bound the write latency. As shown in Figure 5.2, bLSM has one memory component, C_0, and two disk components, C_1 and C_2. The memory component C_0 is continuously flushed and merged with C_1. When C_1 becomes full, a new C_1 component is created while the old C_1, which now becomes C_1', will be merged with C_2. bLSM ensures that for each Level i, the progress of merging C_i' into C_{i+1} (denoted as "out_i") will be roughly identical to the progress of the formation of a new C_i (denoted as "in_i"). This eventually limits the write rate for the memory component (in_0) and avoids blocking writes. However, as we will see later in this chapter, simply bounding the maximum write processing latency alone is insufficient, because a large variance in the write throughput can still cause large queuing delays for subsequent writes.

Performance Stability. Performance stability has long been recognized as a critical performance metric. The TPC-C benchmark [11] not only measures absolute throughput, but also specifies the acceptable upper bounds for the percentile latencies of the transactions. Huang et al. [61] applied VProfiler [60] to identify and eliminate major sources of variance in database transactions

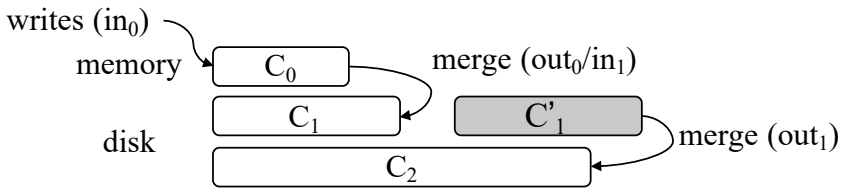


Figure 5.2: bLSM's Spring-and-Gear Merge Scheduler

and proposed a variance-aware transaction scheduling algorithm. Chaudhuri et al. [36] proposed techniques to optimize parameterized queries while balancing the average and variance of query cost. To reduce the variance of query processing, most existing proposals have either emphasized the use of table scans [32, 96, 118] or stuck to worst-case query plans [21, 22]. Cao et al. [34] conducted an experimental study of the performance variance of modern storage stacks; they found that variance is common in storage stacks and heavily depends on configurations and workloads. Dean and Barroso [45] discussed several engineering techniques to reduce performance variance in large-scale distributed systems at Google. Different from these efforts, in this work we focus on evaluating and minimizing the performance variances of LSM-trees due to their inherent out-of-place update design.

5.3 Experimental Methodology

For ease of presentation, we will mix our techniques with a detailed performance analysis for each LSM-tree design. We now describe the general experimental setup and methodology for all experiments to follow.

5.3.1 Experimental Setup

All experiments in this chapter were run on a single node with an 8-core Intel i7-7567U 3.5GHZ CPU, 16 GB of memory, a 500GB SSD, and a 1TB 7200 rpm hard disk. We used the SSD for LSM

storage and configured the hard disk for transaction logging due to its sufficiently high sequential throughput. We allocated 10GB of memory for the AsterixDB instance. Within that allocation, the buffer cache size was set at 2GB. Each LSM memory component had a 128MB budget, and each LSM-tree had two memory components to minimize stalls during flushes. Each disk component had a Bloom filter with a false positive rate setting of 1%. The data page size was set at 4KB to align with the SSD page size.

It is important to note that not all sources of performance variance can be eliminated [61]. For example, writing a key-value pair with a 1MB value inherently requires more work than writing one that only has 1KB. Moreover, short time periods with quickly occurring writes (workload bursts) will be much more likely to cause write stalls than a long period of slow writes, even though their long-term write rate may be the same. In this chapter, we will focus on the *avoidable variance* [61] caused by the internal implementation of LSM-trees instead of variances in the workloads.

To evaluate the internal variances of LSM-trees, we adopt YCSB [40] as the basis for our experimental workload. Instead of using the pre-defined YCSB workloads, we designed our own workloads to better study the performance stability of LSM-trees. Each experiment first loads an LSM-tree with 100 million records, in random key order, where each record has size 1KB. It then runs for 2 hours to update the previously loaded LSM-tree. This ensures that the measured write throughput of an LSM-tree is stable over time. Unless otherwise noted, we used one writer thread for writing data to the LSM memory components. The specific workload setups will be discussed in the subsequent sections.

We used two commonly used I/O optimizations when implementing LSM-trees, namely I/O throttling and periodic disk forces. In all experiments, we throttled the SSD write speed of all LSM flush and merge operations to 100MB/s. This was implemented by using a rate limiter to inject artificial sleeps into SSD writes. This mechanism bounds the negative impact of the SSD writes on query performance and allows us to more fairly compare the performance differences of various LSM merge schedulers. We further had each flush or merge operation force its SSD writes after

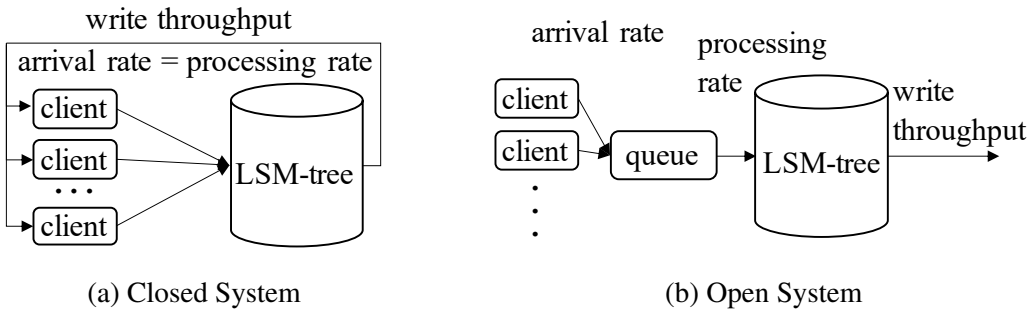
each 16MB of data. This is a common technique use in existing systems to limit the OS I/O queue depth, reducing the negative impact of SSD writes on queries. We have verified that disabling this optimization would not impact the performance trends of writes; however, large forces at the end of each flush and merge operation, which are required for durability, can significantly interfere with queries.

5.3.2 Performance Metrics

To quantify the impact of write stalls, we will not only present the write throughput of LSM-trees but also their write latencies. However, there are different models for measuring write latencies. Throughout the chapter, we will use *arrival rate* to denote the rate at which writes are submitted by clients, *processing rate* to denote the rate at which writes can be processed by an LSM-tree, and *write throughput* to denote the number of writes processed by an LSM-tree per unit of time. The difference between the write throughput and arrival/processing rates is discussed further below.

The bLSM paper [103], as well as most of the existing LSM research, used the experimental setup depicted in Figure 5.3a to write as much data as possible and measure the latency of each write. In this *closed system* setup [57], the processing rate essentially controls the arrival rate, which further equals the write throughput. Although this model is sufficient for measuring the maximum write throughput of LSM-trees, it is not suitable for characterizing their write latencies for several reasons. First, writing to memory is inherently faster than background I/Os, so an LSM-tree will always have to stall writes in order to wait for lagged flushes and merges. Moreover, under this model, a client cannot submit its next write until its current write is completed. Thus, when the LSM-tree is stalled, only a small number of ongoing writes will actually experience a large latency since the remaining writes have not been submitted yet[1].

[1]The original release of the YCSB benchmark [40] mistakenly used this model; this was corrected later in 2015 [12].



(a) Closed System (b) Open System

Figure 5.3: Models for Measuring Write Latency

In practice, a DBMS generally cannot control how quickly writes are submitted by external clients, nor will their writes always arrive as fast as possible. Instead, the arrival rate is usually independent from the processing rate, and when the system is not able to process writes as fast as they arrive, the newly arriving writes must be temporarily queued. In such an *open system* (Figure 5.3b), the measured write latency includes both the queuing latency and processing latency. Moreover, an important constraint is that the arrival rate must be smaller than the processing rate since otherwise the queue length will be unbounded. Thus, the (overall) write throughput is actually determined by the arrival rate.

A simple example will illustrate the important difference between these two models. Suppose that 5 clients are used to generate an intended arrival rate of 1000 writes/s and that the LSM-tree stalls for 1 second. Under the closed system model (Figure 5.3a), only 5 delayed writes will experience a write latency of 1s since the remaining (intended) 995 writes simply will not occur. However, under the open system model (Figure 5.3b), all 1000 writes will be queued and their average latency will be at least 0.5s.

To evaluate write latencies in an open system, one must first set the arrival rate properly since the write latency heavily depends on the arrival rate. It is also important to maximize the arrival rate to maximize the system's utilization. For these reasons, we propose a two-phase evaluation approach with a *testing* phase and a *running* phase. During the *testing* phase, we use the closed system model (Figure 5.3a) to measure the maximum write throughput of an LSM-tree, which

is also its processing rate. When measuring the maximum write throughput, we excluded the initial 20-minute period (out of 2 hours) of the testing phase since the initially loaded LSM-tree has a relatively small number of disk components at first. During the *running* phase, we use the open system model (Figure 5.3b) to evaluate the *write latencies* under a constant arrival rate set at 95% of the measured maximum write throughput. Based on queuing theory [57], the queuing time approaches infinity when the utilization, which is the ratio between the arrival rate and the processing rate, approaches 100%. We thus empirically determine a high utilization load (95%) while leaving some room for the system to absorb variance. If the running phase then reports large write latencies, the maximum write throughput as determined in the testing phase is not sustainable; we must improve the implementation of the LSM-tree or reduce the expected arrival rate to reduce the latencies. In contrast, if the measured write latency is small, then the given LSM-tree can provide a high write throughput with a small performance variance.

5.4 LSM Merge Scheduler

Different from a merge policy, which decides which components to merge, a *merge scheduler* is responsible for executing the merge operations created by the merge policy. In this section, we discuss the design choices for a merge scheduler and evaluate bLSM's spring-and-gear merge scheduler.

5.4.1 Scheduling Choices

The write cost of an LSM-tree, which is the number of I/Os per write, is determined by the LSM-tree design itself and the workload characteristics but not by how merges are executed [79]. Thus, a merge scheduler will have little impact on the overall write throughput of an LSM-tree as long as the allocated I/O bandwidth budget can be fully utilized. However, different scheduling choices

can significantly impact the write stalls of an LSM-tree, and merge schedulers must be carefully designed to minimize write stalls. We have identified the following design choices for a merge scheduler.

Component Constraint: A merge scheduler usually specifies an upper-bound constraint on the total number of components allowed to accumulate before incoming writes to the LSM memory components should be stalled. We call this the *component constraint*. For example, bLSM [103] allows at most two disk components per level, while other systems like HBase [5] or Cassandra [2] specify the total number of disk components across all levels.

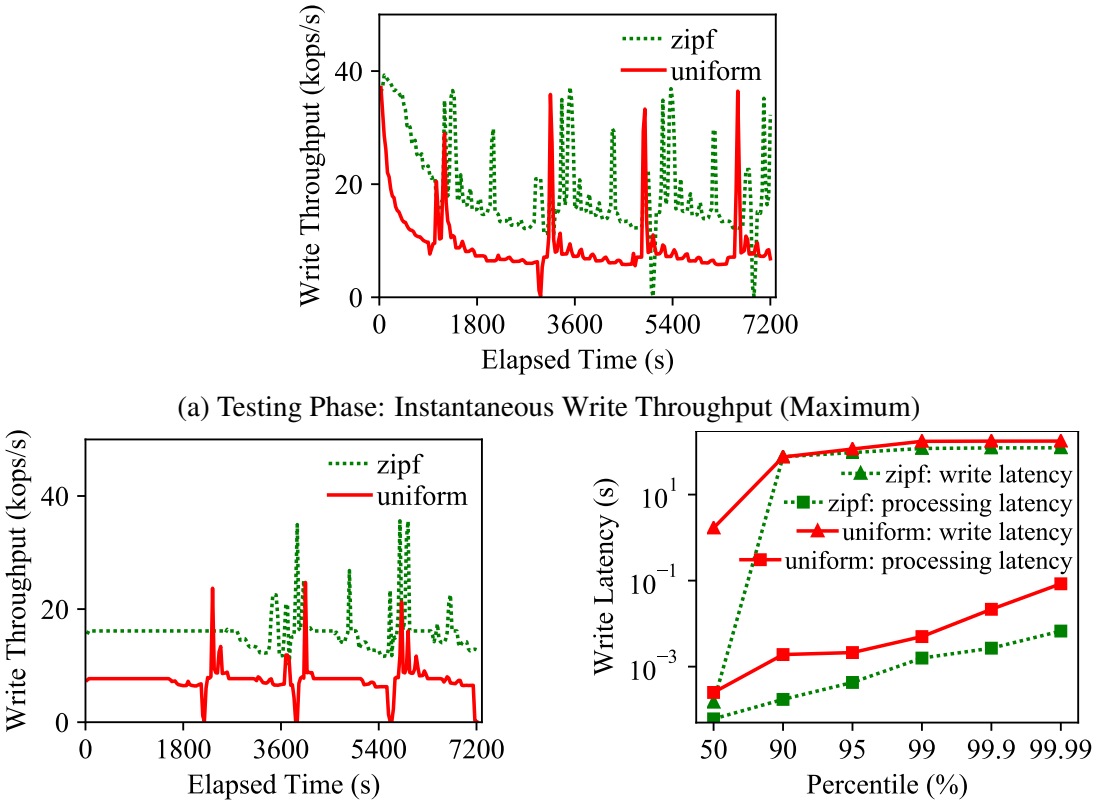
Interaction with Writes: There exist different strategies to enforce a given component constraint. One strategy is to simply stop processing writes once the component constraint is violated. Alternatively, the processing of writes can be degraded gracefully based on the merge pressure [103].

Degree of Concurrency: In general, an LSM-tree can often create multiple merge operations in the same time period. A merge scheduler should decide how these merge operations should be scheduled. Allowing concurrent merges will enable merges at multiple levels to proceed concurrently, but they will also compete for CPU and I/O resources, which can negatively impact query performance [15]. As two examples, bLSM [103] allows one merge operation per level, while LevelDB [6] uses just one single background thread to execute all merges one by one.

I/O Bandwidth Allocation: Given multiple concurrent merge operations, the merge scheduler should further decide how to allocate the available I/O bandwidth among these merge operations. A commonly used heuristic is to allocate I/O bandwidth "fairly" (evenly) to all active merge operations. Alternatively, bLSM [103] allocates I/O bandwidth based on the relative progress of the merge operations to ensure that merges at each level all make steady progress.

5.4.2 Evaluation of bLSM

Due to the implementation complexity of bLSM and its dependency on a particular storage system, Stasis [102], we chose to directly evaluate the released version of bLSM [9]. bLSM uses the leveling merge policy with two on-disk levels. We set its memory component size to 1GB and size ratio to 10 so that the experimental dataset with 100 million records can fit into the last level. We used 8 write threads to maximize the write throughput of bLSM.



(a) Testing Phase: Instantaneous Write Throughput (Maximum)

(b) Running Phase: Instantaneous Write Throughput (95% Load)

(c) Running Phase: Percentile Write Latencies (95% Load)

Figure 5.4: Two-Phase Evaluation of bLSM

Testing Phase. During the testing phase, we measured the maximum write throughput of bLSM by writing as much data as possible using both the uniform and Zipf update workloads. The instantaneous write throughput of bLSM under these two workloads is shown in Figure 5.4a. For readability, the write throughput is averaged over 30-second windows. (Unless otherwise noted, the same aggregation applies to all later experiments as well.)

Even though bLSM's merge scheduler prevents writes from being stalled, the instantaneous write throughput still exhibits a large variance. Recall that bLSM uses the merge progress at each level to control its in-memory write speed. After the component C_1 is full and becomes C_1', the original C_1 will be empty and will have much shorter merge times. This will temporarily increase the in-memory write speed of bLSM, which then quickly drops as C_1 grows larger and larger. Moreover, the Zipf update workload increases the write throughput only because updated entries can be reclaimed earlier, but the overall variance performance trends are still the same.

Running Phase. Based on the maximum write throughput measured in the testing phase, we then used a constant data arrival process (95% of the maximum) in the running phase to evaluate bLSM's behavior. Figure 5.4b shows the instantaneous write throughput of bLSM under the uniform and Zipf update workloads. bLSM maintains a sustained write throughput during the initial period of the experiment, but later has to slow down its in-memory write rate periodically due to background merge pressure. Figure 5.4c further shows the resulting percentile write and processing latencies. The processing latency measures only the time for the LSM-tree to process a write, while the write latency includes both the write's queuing time and processing time. By slowing down the in-memory write rate, bLSM indeed bounds the processing latency. However, the write latency is much larger because writes must be queued when they cannot be processed immediately. This suggests that simply bounding the maximum processing latency is far from sufficient; it is important to minimize the variance in an LSM-tree's processing rate to minimize write latencies.

5.5 Full Merges

In this section, we explore the scheduling choices of LSM-trees with full merges and then evaluate the impact of merge scheduling on write stalls using our two-phase approach. Finally, we examine other variations of the tiering merge policy that are used in practical systems.

Table 5.1: List of Notation

Term	Definition	Unit
T	size ratio of the merge policy	
L	the number of levels in an LSM-tree	
M	memory component size	entries
B	I/O bandwidth	entries/s
μ	write arrival rate	entries/s
W	write throughput of an LSM-tree	entries/s

5.5.1 Merge Scheduling for Full Merges

We first introduce some useful notation for use throughout our analysis in Table 5.1. To simplify the analysis, we will ignore the I/O cost of flushes since merges consume most of the I/O bandwidth.

Component Constraint

To provide acceptable query performance and space utilization, the total number of disk components of an LSM-tree must be bounded. We call this upper bound the *component constraint*, and it can be enforced either *locally* or *globally*. A local constraint specifies the maximum number of disk components per level. For example, bLSM [103] uses a local constraint to allow at most two components per level. A global constraint instead specifies the maximum number of disk components across all levels. Here we argue that global component constraints will better minimize write stalls. In addition to external factors, such as deletes or shifts in write patterns, the merge time at each level inherently varies for leveling since the size of the component at Level i varies from 0 to $(T-1) \cdot M \cdot T^{i-1}$. Because of this, bLSM cannot provide a high yet stable write throughput over time. Global component constraints will better absorb this variance and minimize the write stalls .

It remains a question how to determine the maximum number of disk components for the component constraint. In general, tolerating more disk components will increase the LSM-tree's ability to reduce write stalls and absorb write bursts, but it will decrease query performance and space

utilization. Given the negative impact of stalls on write latencies, one solution is to tolerate a sufficient number of disk components to avoid write stalls while the worst-case query performance and space utilization are still bounded. For example, one conservative constraint would be to tolerate twice the expected number of disk components, e.g., $2 \cdot L$ components for leveling and $2 \cdot T \cdot L$ components for tiering.

Interaction with Writes

When the component constraint is violated, the processing of writes by an LSM-tree has to be slowed down or stopped. Existing LSM-tree implementations [6, 10, 103] prefer to gracefully slow down the in-memory write rate by adding delays to some writes. This approach reduces the maximum processing latency, as large pauses are broken down into many smaller ones, but the overall processing rate of an LSM-tree, which depends on the I/O cost of each write, is not affected. Moreover, this approach will result in an even larger queuing latency. There may be additional considerations for gracefully slowing down writes, but we argue that processing writes as quickly as possible minimizes the overall write latency, as stated by the following theorem.

THEOREM 5.1. *Given any data arrival process and any LSM-tree, processing writes as quickly as possible minimizes the latency of each write.*

Proof. Given an LSM-tree, consider two merge schedulers S and S' which only differ in that S may add arbitrary delays to writes to avoid write stalls while S' processes writes as quickly as possible. Denote the total number of writes processed by S and S' at time instant T as W_T and W'_T respectively. Since S' processes writes as quickly as possible, we have $W_T \leq W'_T$. In other words, given the same numbers of writes, S' processes these writes no later than S.

Consider the i-th write request that arrives at time instant T_{a_i}. Suppose this write is processed by S and S' at time instants T_{p_i} and T'_{p_i} respectively. Based on the analysis above, it is straightforward

that $T_{p_i} \geq T'_{p_i}$. Thus, we have $T_{p_i} - T_{a_i} \geq T'_{p_i} - T_{a_i}$, which implies that S' minimizes the latency of each write. □

It should be noted that Theorem 1 only considers write latencies. By processing writes as quickly as possible, disk components can stack up more quickly (up to the component constraint), which may negatively impact query performance. Thus, a better approach may be to increase the write processing rate, e.g., by changing the structure of the LSM-tree. We leave the exploration of this direction as future work.

Degree of Concurrency

A merge policy can often create multiple merge operations simultaneously. For full merges, we can show that a single-threaded scheduler that executes one merge at a time is not sufficient for minimizing write stalls. Consider a merge operation at Level i. For leveling, the merge time varies from 0 to $\frac{M \cdot T^i}{B}$ because the size of the component at Level i varies from 0 to $(T-1) \cdot M \cdot T^{i-1}$. For tiering, each component has size $M \cdot T^{i-1}$ and merging T components thus takes time $\frac{M \cdot T^i}{B}$. Suppose the arrival rate is μ. Without concurrent merges, there would be $\frac{\mu}{M} \cdot \frac{M \cdot T^i}{B} = \frac{\mu \cdot T^i}{B}$ newly flushed components added while this merge operation is being executed, assuming that flushes can still proceed.

Our two-phase evaluation approach chooses the maximum write throughput of an LSM-tree as the arrival rate μ. For leveling, the maximum write throughput is approximately $W_{level} = \frac{2 \cdot B}{T \cdot L}$, as each entry is merged $\frac{T}{2}$ times per level. For tiering, the maximum write throughput is approximately $W_{tier} = \frac{B}{L}$, as each entry is merged only once per level. By substituting W_{level} and W_{tier} for μ, one needs to tolerate at least $\frac{2 \cdot T^{i-1}}{L}$ flushed components for leveling and $\frac{T^i}{L}$ flushed components for tiering to avoid write stalls. Since the term T^i grows exponentially, a large number of flushed components will have to be tolerated when a large disk component is being merged. Consider the leveling merge policy with a size ratio of 10. To merge a disk component at Level 5, approximately

$\frac{2 \cdot 10^4}{5} = 4000$ flushed components would need to be tolerated, which is highly unacceptable.

Clearly, concurrent merges must be performed to minimize write stalls. When a large merge is being processed, smaller merges can still be completed to reduce the number of components. By the definition of the tiering and leveling merge policies, there can be at most one active merge operation per level. Thus, given an LSM-tree with L levels, at most L merge operations can be scheduled concurrently.

I/O Bandwidth Allocation

Given multiple active merge operations, the merge scheduler must further decide how to allocate I/O bandwidth to these operations. A heuristic used by existing systems [2, 5, 10] is to allocate I/O bandwidth fairly (evenly) to all ongoing merges. We call this the *fair* scheduler. The fair scheduler ensures that all merges at different levels can proceed, thus eliminating potential starvation. Recall that write stalls occur when an LSM-tree has too many disk components, thus violating the component constraint. It is unclear whether or not the fair scheduler can minimize write stalls by minimizing the number of disk components over time.

Recall that both the leveling and tiering merge policies always merge the same number of disk components at once. We propose a novel *greedy* scheduler that always allocates the full I/O bandwidth to the merge operation with the smallest remaining number of bytes. The greedy scheduler has a useful property that it minimizes the number of disk components over time for a given set of merge operations.

THEOREM 5.2. *Given any set of merge operations that process the same number of disk components and any I/O bandwidth budget, the greedy scheduler minimizes the number of disk components at any time instant.*

Proof. Let S be an arbitrary merge scheduler and S' be the greedy scheduler. Suppose there are

N merge operations in total and the initial time instant is t_0. Denote by t_i and t_i' the time instants when S and S' complete their i-th merge operation, respectively. Since all merge operations always process the same number of disk components, we only need to show that for any $i \in [1,N]$, $t_i \geq t_i'$ always holds. In other words, S' completes each merge operation no later than S.

Suppose there exists $i \in [1,N]$ s.t. $t_i < t_i'$. Denote by $|S_{\leq i}|$ and $|S'_{\leq i}|$ the total number of bytes read and written by S and S' up to the completion of the i-th merge operation. By the definition of the greedy scheduler S', we have $|S_{\leq i}| \geq |S'_{\leq i}|$. Since $t_i < t_i'$, we further have $\frac{|S_{\leq i}|}{t_i - t_0} > \frac{|S_{\prime \leq i}|}{t_i' - t_0}$. This implies that the merge scheduler S requires a larger I/O bandwidth budget than S', which leads to a contradiction. Thus, for any $i \in [1,N]$, $t_i \leq t_i'$ always holds, which proves that S' minimizes the number of disk components over time. □

Theorem 2 only considers a set of statically created merge operations. This conclusion may not hold in general because sometimes completing a large merge may enable the merge policy to create smaller merges, which can then reduce the number of disk components more quickly. Because of this, there actually exists no merge scheduler that can always minimize the number of disk components over time, as stated by the following theorem. However, as we will see in our later evaluation, the greedy scheduler is still a very effective heuristic for minimizing write stalls.

THEOREM 5.3. *Given any I/O bandwidth budget, no merge scheduler can minimize the number of disk components at any time instant for any data arrival process and any LSM-tree for a deterministic merge policy where all merge operations process the same number of disk components.*

Proof. In this proof, we will construct an example showing that no such merge scheduler can be designed. Consider a two-level LSM-tree with a tiering merge policy. The size ratio of this merge policy is set at 2. Suppose Level 1, which is the last level, contains three disk components D_1, D_2, D_3 and Level 0 contains two disk components, D_4 and D_5. For simplicity, assume that no more writes will arrive. Initially, the merge policy creates two merge operations, namely the merge operation M_{1-2} that processes D_1 and D_2 and the merge operation M_{4-5} that processes D_4 and D_5.

Upon the completion of M_{1-2}, which produces a new disk component D_{1-2}, the merge policy will create a new merge operation M_{1-3} that processes D_{1-2} and D_3. We further denote the amount of I/O bandwidth required by each merge operation M_{1-2}, M_{4-5}, and M_{1-3} as $|M_{1-2}|$, $|M_{4-5}|$, and $|M_{1-3}|$. Finally, we assume that $|M_{1-3}| < |M_{4-5}| < |M_{1-2}|$. This can happen if D_2 contains a large number of deleted keys against D_1 so that the merged disk component D_{1-2} is very small.

Suppose that the initial time instant is t_0 and let the given I/O bandwidth budget be B. Consider a merge scheduler S that first executes M_{4-5} and then M_{1-2}. At time instant $t_1 = t_0 + \frac{|M_{4-5}|}{B}$, S completes M_{4-5} and reduces the number of disk components by 1. At time instant $t_2 = t_0 + \frac{|M_{4-5}|}{B} + \frac{|M_{1-2}|}{B}$, S completes M_{1-2}. Consider another merge scheduler S' that first executes M_{1-2}. At time instant $t'_1 = t_0 + \frac{|M_{1-2}|}{B}$, S' completes M_{1-2}. Now the merge policy creates a new merge operation M_{1-3}, which is then executed by S'. At time instant $t'_2 = t_0 + \frac{|M_{1-2}|}{B} + \frac{|M_{1-3}|}{B}$, S' completes M_{1-3}. Based on the assumption $|M_{1-3}| < |M_{4-5}| < |M_{1-2}|$, it follows that $t_1 < t'_1$ and $t'_2 < t_2$. Suppose there exists a merge scheduler S^* that minimizes the number of disk components over time. Then, S^* must satisfy the following two constraints: (1) complete one merge operation no later than t_1; (2) complete two merge operations no later than t'_2.

To satisfy constraint (1), S^* must execute M_{4-5} first. Then, S^* must complete the second merge operation within time interval $t'_2 - t_1 = \frac{|M_{1-2}|}{B} + \frac{|M_{1-3}|}{B} - \frac{|M_{4-5}|}{B}$. Since $|M_{1-3}| < |M_{4-5}|$, we have $t'_2 - t_1 < \frac{|M_{1-2}|}{B}$. Thus, S^* cannot satisfy constraint (2) by completing the second merge operation no later than t'_2 because the only remaining merge operation M_{1-2} takes time $\frac{|M_{1-2}|}{B}$ to finish. This leads to a contradiction that S^* minimizes the number of disk components over time. Thus, we have constructed an example for which no such merge scheduler can be designed, which proves the theorem. □

Putting Everything Together

Based on the discussion of each scheduling choice, we now summarize the proposed greedy scheduler. The greedy scheduler enforces a global component constraint with a sufficient number of disk components, e.g., twice the expected number of components of an LSM-tree, to minimize write stalls while ensuring the stability of the LSM-tree. It processes writes as quickly as possible and only stops the processing of writes when the component constraint is violated. The greedy scheduler performs concurrent merges but allocates the full I/O bandwidth to the merge operation with the smallest remaining bytes.

Algorithm 4 Pseudocode for Greedy Scheduling Algorithm

```
1:  mergeOps ← the list of scheduled merge operations
2:  activeOp ← the active merge operation
3:  function SCHEDULEMERGE(newOp)
4:      mergeOps.ADD(newOp)
5:      notify GreedyScheduler
6:  function COMPLETEMERGE
7:      mergeOps.REMOVE(activeOp)
8:      activeOp ← NULL
9:      notify GreedyScheduler
10: function GREEDYSCHEDULER
11:     while mergeOps changes do
12:         newOp ← find the merge operation with the fewest remaining input pages in mergeOps
13:         if newOp ≠ NULL AND newOp ≠ activeOp then
14:             pause activeOp
15:             resume newOp
16:             activeOp ← newOp
```

The pseudocode for the greedy scheduling algorithm is shown in Algorithm 4. It stores the list of scheduled merge operations in *mergeOps*. At any time, there is at most one merge operation being executed by the merge thread, which is denoted by *activeOp*. The merge policy calls SCHEDULEMERGE when a new merge operation is scheduled, and the merge thread calls COMPLETEMERGE when a merge operation is completed. In both functions, *mergeOps* is updated accordingly and the merge scheduler is notified to check whether a new merge operation needs to be executed. It should be noted that in general one cannot exactly know which merge operation requires the least

amount of I/O bandwidth until the new component has been fully produced. Thus, line 12 uses the number of remaining input pages as an approximation to determine the smallest merge operation. Finally, if the newly selected merge operation is inactive, i.e., not being executed, the scheduler pauses the previous active merge operation and activates the new one.

Under the greedy scheduler, larger merges may be starved at times since they receive lower priority. This has a few implications. First, during normal user workloads, such starvation can only occur if the arrival rate is temporarily faster than the processing rate of an LSM-tree. Given the negative impact of write stalls on write latencies, it can actually be beneficial to temporarily delay large merges so that the system can better absorb write bursts. Second, the greedy scheduler should not be used in the testing phase because it would report a higher but unsustainable write throughput due to such starved large merges.

Finally, our discussions of the greedy scheduler as well as the single-threaded scheduler are based on an important assumption that a single merge operation is able to fully utilize the available I/O bandwidth budget. Otherwise, multiple merges must be executed at the same time. It is straightforward to extend the greedy scheduler to execute the smallest k merge operations, where k is the degree of concurrency needed to fully utilize the I/O bandwidth budget.

5.5.2 Experimental Evaluation

We now experimentally evaluate the write stalls of LSM-trees using our two-phase approach. We discuss the specific experimental setup followed by the detailed evaluation, including the impact of merge schedulers on write stalls, the benefit of enforcing the component constraint globally and of processing writes as quickly as possible, and the impact of merge scheduling on query performance.

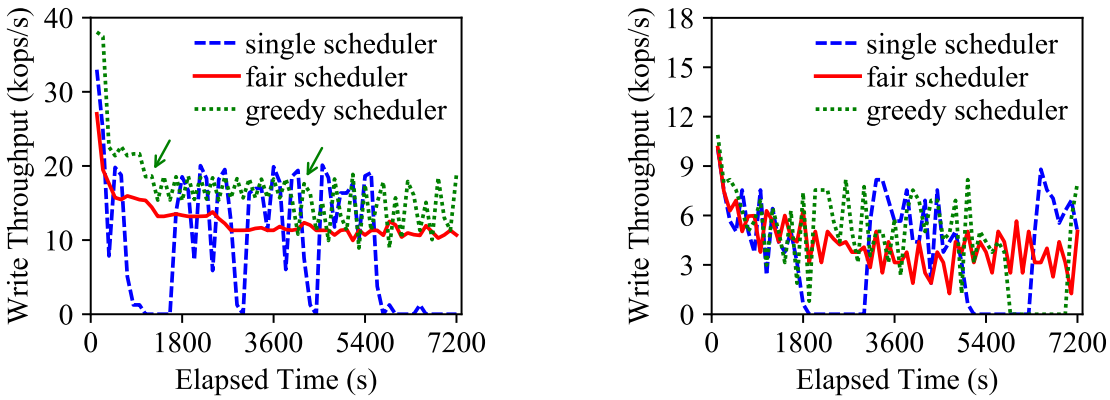
Experimental Setup

All experiments in this section were performed using AsterixDB with the general setup described in Section 5.3. Unless otherwise noted, the size ratio of leveling was set at 10, which is a commonly used configuration in practice [6, 10]. For the experimental dataset with 100 million unique records, this results in a three-level LSM-tree, where the last level is nearly full. For tiering, the size ratio was set at 3, which leads to better write performance than leveling without sacrificing too much on query performance. This ratio results in an eight-level LSM-tree.

We evaluated the single-threaded scheduler, the fair scheduler, and the proposed greedy scheduler. The single-threaded scheduler only executes one merge at a time using a single thread. Both the fair and greedy schedulers are concurrent schedulers that execute each merge using a separate thread. The difference is that the fair scheduler allocates the I/O bandwidth to all ongoing merges evenly, while the greedy scheduler always allocates the full I/O bandwidth to the smallest merge. To minimize flush stalls, a flush operation is always executed in a separate thread and receives higher I/O priority. Unless otherwise noted, all three schedulers enforce global component constraints and process writes as quickly as possible. The maximum number of disk components is set at twice the expected number of disk components for each merge policy. Each experiment was performed under the uniform update workload. The choice of update distribution, e.g., Zipf, may impact the overall write throughput, but does not impact the performance stability of LSM-trees.

Testing Phase

During the testing phase, we measured the maximum write throughput of an LSM-tree by writing as much data as possible. In general, alternative merge schedulers have little impact on the maximum write throughput since the I/O bandwidth budget is fixed, but their measured write throughput may be different due to the finite experimental period.



(a) Tiering Merge Policy

(b) Leveling Merge Policy

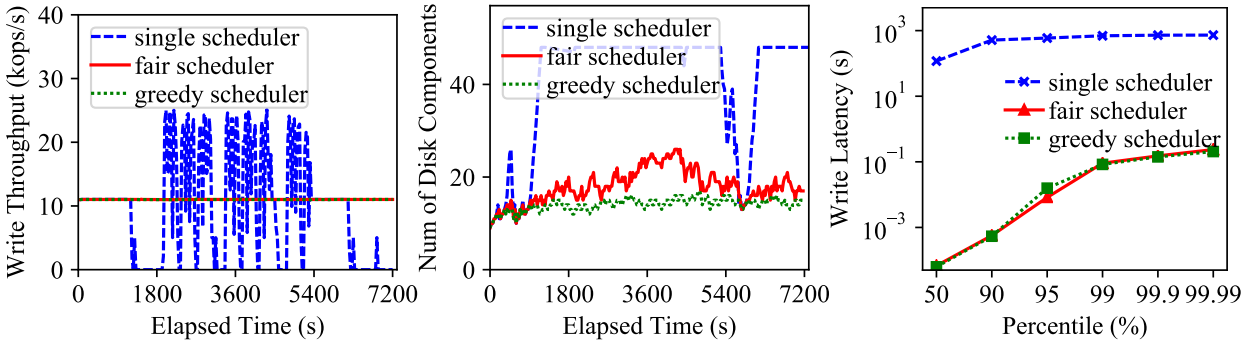
Figure 5.5: Testing Phase: Instantaneous Write Throughput

Figures 5.5a and 5.5b shows the instantaneous write throughout of LSM-trees using different merge schedulers for tiering and leveling. Under both merge policies, the single-threaded scheduler regularly exhibits long pauses, making its write throughput vary over time. The fair scheduler exhibits a relatively stable write throughput over time since all merge levels can proceed at the same rate. With leveling, its write throughput still varies slightly over time since the component size at each level varies. The greedy scheduler appears to achieve a higher write throughput than the fair scheduler by starving large merges. However, this higher write throughput eventually drops when no small merges can be scheduled. For example, the write throughput with tiering drops slightly at 1100s and 4000s, and there is a long pause from 6000s to 7000s with leveling. This result confirms that the fair scheduler is more suitable for testing the maximum write throughput of an LSM-tree, as merges at all levels can proceed at the same rate. In contrast, the single-threaded scheduler incurs many long pauses, causing a large variance in the measured write throughput. The greedy scheduler provides a higher write throughput by starving large merges, which would be undesirable at runtime.

Running Phase

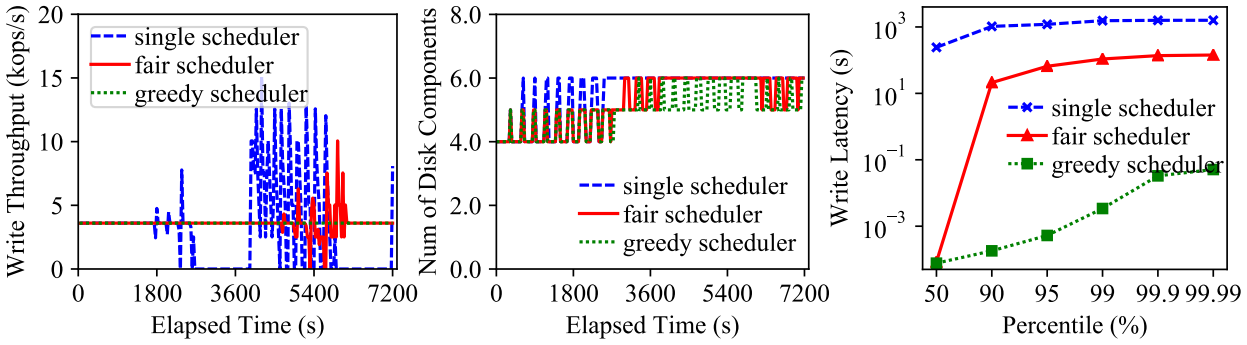
Turning to the running phase, we used a constant data arrival process, configured based on 95% of the maximum write throughput measured by the fair scheduler, to evaluate the write stalls of LSM-trees.

LSM-trees can provide a stable write throughput. We first evaluated whether LSM-trees with different merge schedulers can support a high write throughput with low write latencies. For each experiment, we measured the instantaneous write throughput and the number of disk components over time as well as percentile write latencies.



(a) Instantaneous Write Throughput (b) Number of Disk Components (c) Percentile Write Latencies

Figure 5.6: Running Phase of Tiering Merge Policy (95% Load)



(a) Instantaneous Write Throughput (b) Number of Disk Components (c) Percentile Write Latencies

Figure 5.7: Running Phase of Leveling Merge Policy (95% Load)

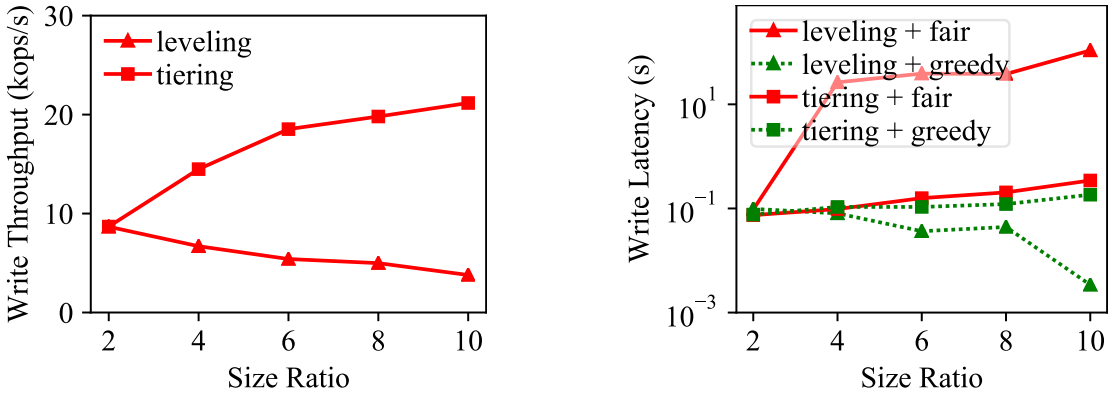
The results for tiering are shown in Figure 5.6. Both the fair and greedy schedulers are able to

provide stable write throughputs and the total number of disk components never reaches the configured threshold. The greedy scheduler also minimizes the number of disk components over time. The single-threaded scheduler, however, causes a large number of write stalls due to the blocking of large merges, which confirms our previous analysis. Because of this, the single-threaded scheduler incurs large percentile write latencies. In contrast, both the fair and greedy schedulers provide small write latencies because of their stable write throughput. Figure 5.7 shows the corresponding results for leveling. The single-threaded scheduler again performs poorly, causing a lot of stalls and thus large write latencies. Due to the inherent variance of merge times, the fair scheduler alone cannot provide a stable write throughput; this results in relatively large write latencies. In contrast, the greedy scheduler avoids write stalls by always minimizing the number of components, which results in small write latencies.

This experiment confirms that LSM-trees can achieve a stable write throughput with a relatively small performance variance. Moreover, the write stalls of an LSM-tree heavily depend on the design of the merge scheduler.

Impact of Size Ratio. To verify our findings on LSM-trees with different shapes, we further carried out a set of experiments by varying the size ratio from 2 to 10 for both tiering and leveling. For leveling, we applied the dynamic level size optimization [49] so that the largest level remains almost full by slightly modifying the size ratio between Levels 0 and 1. This optimization maximizes space utilization without impacting write or query performance.

During the testing phase, we measured the maximum write throughput for each LSM-tree configuration using the fair scheduler, which is shown in Figure 5.8a. In general, a larger size ratio increases write throughput for tiering but decreases write throughput for leveling because it decreases the merge frequency of tiering but increases that of leveling. During the running phase, we evaluated the 99% percentile write latency for each LSM-tree configuration using constant data arrivals, which is shown in Figure 5.8b. With tiering, both the fair and greedy schedulers are able to provide a stable write throughput with small write latencies. With leveling, the fair scheduler

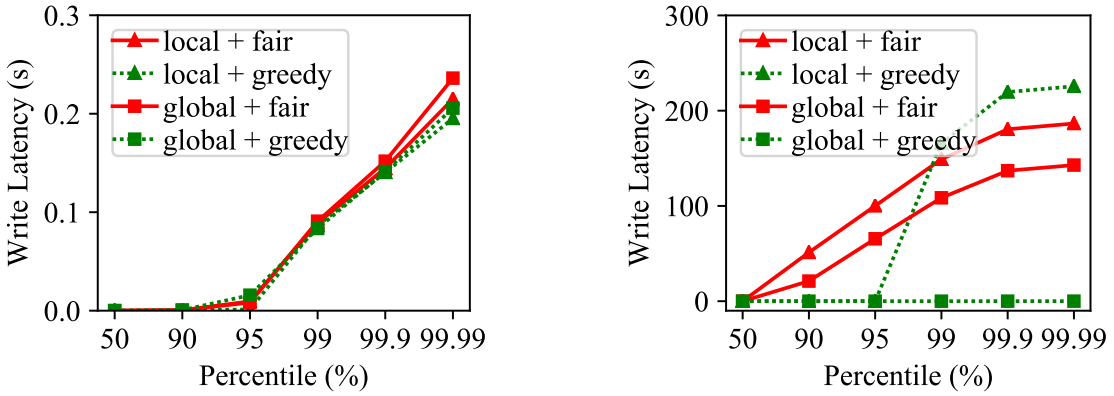


(a) Testing Phase: Maximum Write Throughput (b) Running Phase: 99% Percentile Write Latency

Figure 5.8: Impact of Size Ratio on Write Stalls

causes large write latencies when the size ratio becomes larger, as we have seen before. In contrast, the greedy scheduler is always able to provide a stable write throughput along with small write latencies. This again confirms that LSM-trees, despite their size ratios, can provide a high write throughput with a small variance with an appropriately chosen merge scheduler.

Benefit of Global Component Constraints. We next evaluated the benefit of global component constraints in terms of minimizing write stalls. We additionally included a variation of the fair and greedy schedulers that enforces local component constraints, that is, 2 components per level for leveling and $2 \cdot T$ components per level for tiering.



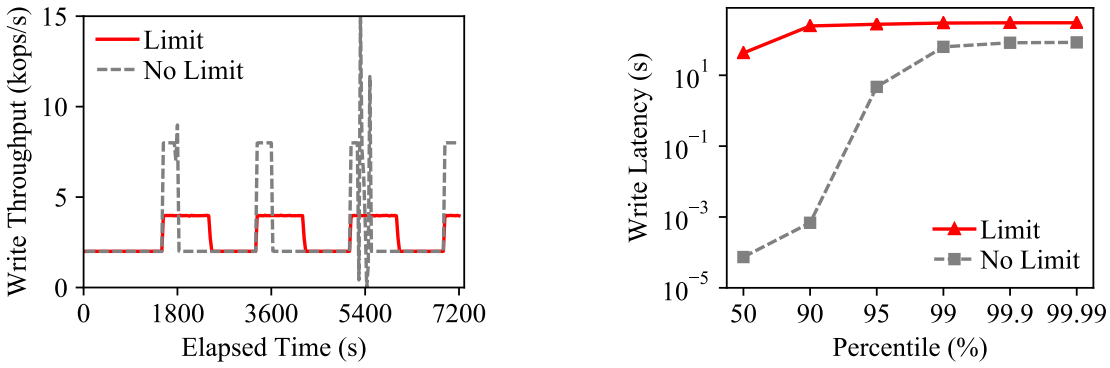
(a) Tiering Merge Policy (b) Leveling Merge Policy

Figure 5.9: Impact of Enforcing Component Constraints on Percentile Write Latencies

The resulting write latencies are shown in Figure 5.9. In general, local component constraints have

little impact on tiering since its merge time per level is relatively stable. However, the resulting write latencies for leveling become much large due to the inherent variance of its merge times. Moreover, local component constraints have a larger negative impact on the greedy scheduler. The greedy scheduler prefers small merges, which may not be able to complete due to possible violations of the constraint at the next level. This in turn causes longer stalls and thus larger percentile write latencies. In contrast, global component constraints better absorb these variances, reducing the write latencies.

Benefits of Processing Writes As Quickly As Possible. We further evaluated the benefit of processing writes as quickly as possible. We used the leveling merge policy with a burst data arrival process that alternates between a normal arrival rate of 2000 records/s for 25 minutes and a high arrival rate of 8000 records/s for 5 minutes. We evaluated two variations of the greedy scheduler. The first variation processes writes as quickly as possible (denoted as "No Limit"), as we did before. The second variation enforces a maximum in-memory write rate of 4000 records/s (denoted as "Limit") to avoid write stalls.



(a) Instantaneous Write Throughput

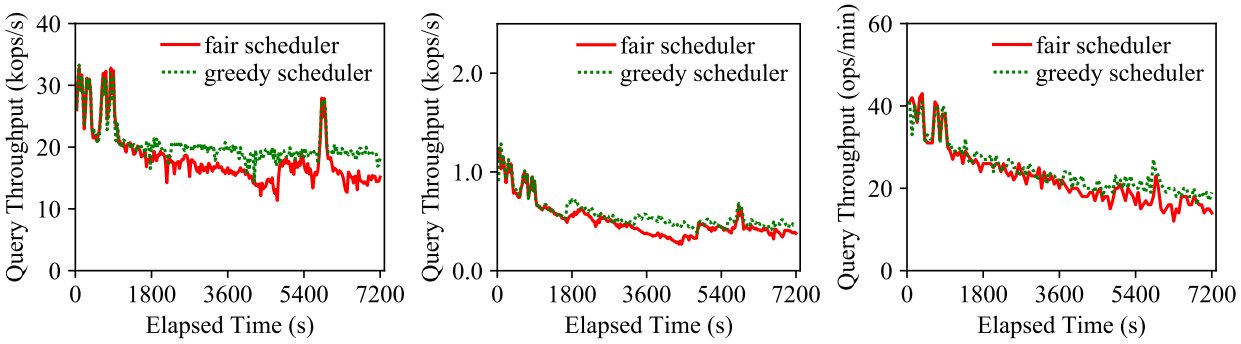
(b) Percentile Write Latencies

Figure 5.10: Running Phase with Burst Data Arrivals

The instantaneous write throughput and the percentile write latencies of the two variations are shown in Figures 5.10a and 5.10b respectively. As Figure 5.10a shows, delaying writes avoids write stalls and the resulting write throughput is more stable over time. However, this causes larger write latencies (Figure 5.10b) since delayed writes must be queued. In contrary, writing as quickly

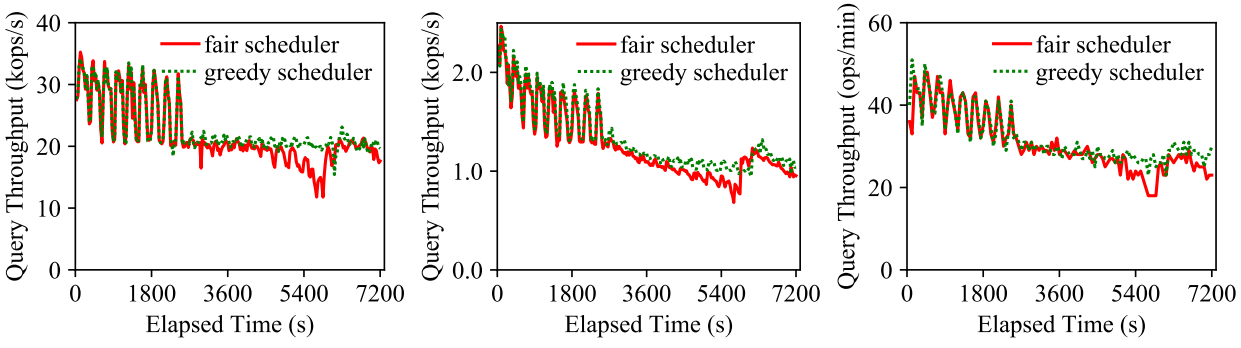
as possible causes occasional write stalls but still minimizes overall write latencies. This confirms our previous analysis that processing writes as quickly as possible minimizes write latencies.

Impact on Query Performance. Finally, since the point of having data is to query it, we evaluated the impact of the fair and greedy schedulers on concurrent query performance. We evaluated three types of queries, namely point lookups, short scans, and long scans. A point lookup accesses 1 record given a primary key. A short scan query accesses 100 records and a long scan query accesses 1 million records. In each experiment, we executed one type of query concurrently with concurrent updates with constant arrival rates as before. To maximize query performance while ensuring that LSM flush and merge operations receive enough I/O bandwidth, we used 8 query threads for point lookups and short scans and used 4 query threads for long scans.



(a) Point Lookup (b) Short Range Query (c) Long Range Query

Figure 5.11: Instantaneous Query Throughput of Tiering Merge Policy



(a) Point Lookup (b) Short Range Query (c) Long Range Query

Figure 5.12: Instantaneous Query Throughput of Leveling Merge Policy

The instantaneous query throughput under the tiering and leveling merge policies is depicted in

Figure 5.11 and Figure 5.12 respectively. For point lookups and short scans, the query throughput is averaged over 30-second windows. For long scans, the query throughput is averaged over 1-minute windows. As the results show, leveling has similar point lookup throughput to tiering because Bloom filters can filter out most unnecessary I/Os, but it has much better range query throughput than tiering. Moreover, the greedy scheduler always improves query performance by minimizing the number of components. Among the three types of queries, point lookups and short scans benefit more from the greedy scheduler since these two types of queries are more sensitive to the number of disk components. In contrast, long scans incur most of their I/O cost at the largest level. Moreover, the tiering merge policy benefits more from the greedy scheduler than the leveling merge policy since the performance difference between the greedy and fair schedulers is larger under the tiering merge policy. This is because the tiering merge policy has more disk components than the leveling merge policy. Note that under the leveling merge policy, there is a drop in query throughput under the fair scheduler at around 5400s, even though there is little difference in the number of disk components between the fair and greedy scheduler. This drop is caused by write stalls during that period, as indicated by the instantaneous write throughput of Figure 5.7. After the LSM-tree recovers from write stalls, it attempts to write as much data as possible in order to catch up, which results in a lower query throughput.

5.5.3 Tiering in Practice

Existing LSM-based systems, such as BigTable [35] and HBase [5], use a slight variation of the tiering merge policy discussed in the literature. This variation, often referred as the *size-tiered* merge policy, does not organize components into levels explicitly but simply schedules merges based on the sizes of disk components. This policy has three important parameters, namely the size ratio T, the minimum number of components to merge *min*, and the maximum number of components to merge *max*. It merges a sequence of components, whose length is at least *min*, when the total size of the sequence's the younger components is T times larger than that of the

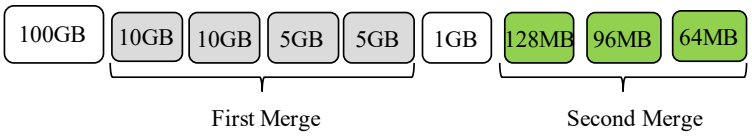


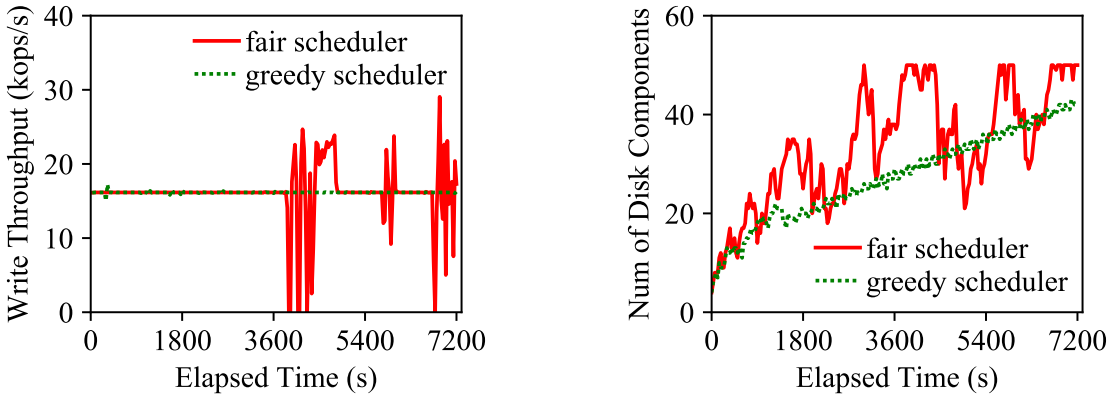
Figure 5.13: Example of Size-Tiered Merge Policy

oldest component in the sequence. It also seeks to merge as many components as possible at once until *max* is reached. Concurrent merges can also be performed. For example, in HBase [5], each execution of the size-tiered merge policy will always examine the longest prefix of the component sequence in which no component is being merged.

An example of the size-tiered merge policy is shown in Figure 5.13, where each disk component is labeled with its size. Let the size ratio be 1.2 and the minimum and maximum number of components per merge be 2 and 4 respectively. Suppose initially that no component is being merged. The first of execution the size-tiered merge policy starts from the oldest component, labeled 100GB. However, no merge is scheduled since this component is too large. It then examines the next component, labeled 10GB, and schedules a merge operation for the 4 components labeled from 10GB to 5GB. The next execution of the size-tiered merge policy starts from the component labeled 1GB, and it schedules a merge for the 3 components labeled from 128MB to 64MB.

To evaluate the write stalls of the size-tiered merge policy, we repeated the experiments using our two-phase approach. In our evaluation, the size ratio was set at 1.2, which is the default value in HBase [5], and the minimum and maximum mergeable components were set at 2 and 10 respectively. The maximum tolerated disk components parameter was set at 50.

During the testing phase, the maximum write throughput measured by using the fair scheduler was 17,008 records/s. Then during the running phase, we used a constant data arrival process based on 95% of this maximum throughput to evaluate write stalls. The instantaneous write throughput of the LSM-tree and the number of disk components over time are shown in Figures 5.14a and 5.14b respectively. As one can see, write stalls have occurred under the fair scheduler. Moreover, even though the greedy scheduler avoids write stalls, its number of disk components keeps increasing



(a) Instantaneous Write Throughput

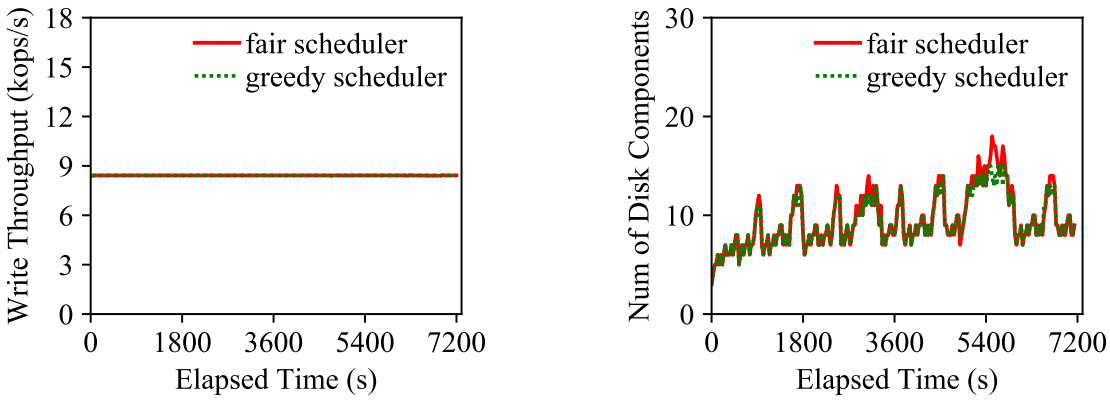
(b) Number of Disk Components

Figure 5.14: Running Phase of Size-Tiered Merge Policy (95% Load)

over time. This result indicates that the maximum write throughput measured during the testing phase is not sustainable.

This problem is caused by the non-determinism of the size-tiered merge policy since it tries to merge as many disk components as possible. This behavior impacts the maximum write throughput of the LSM-tree. During the testing phase, when writes are often blocked because of too many disk components, this merge policy tends to merge more disk components at once, which then leads to a higher write throughput. However, during the running phase, when writes arrive steadily, this merge policy tends to schedule smaller merges as flushed components accumulate. For example, during the running phase of this experiment, 55 large merges that involved 10 components were scheduled, but only 24 large merges were scheduled under the fair scheduler during the running phase. Even worse, 99.76% of the scheduled merges under the greedy scheduler involved no more than 4 components since large merges were starved.

To address problem and to minimize write stalls, the arrival rate must be reduced. However, finding the maximum "stall free" arrival rate is non-trivial due to the non-determinism of the size-tiered merge policy. Instead, we propose a simple and conservative solution to avoid write stalls. During the testing phase, we propose to measure the lower bound write throughput by always merging the minimum number of disk components. This write throughput will serve as a baseline of the



(a) Instantaneous Write Throughput

(b) Number of Disk Components

Figure 5.15: Running Phase of Size-Tiered Merge Policy with the Proposed Solution

arrival rate. During runtime, the size-tiered merge policy can merge more disk components to dynamically increase its write throughput to minimize stalls.

We repeated the previously experiments based on this solution. During the testing phase, the merge policy always merged 2 disk components, which resulted in a lower maximum write throughput of 8,863 records/s. We then repeated the running phase based on this throughput. Figures 5.15a and 5.15b show the instantaneous write throughput and the number of disk components over time respectively during the running phase. In this case, both schedulers exhibit no write stalls and the number of disk components is more stable over time. Moreover, the greedy scheduler still slightly reduces the number of disk components.

5.6 Partitioned Merges

We now examine the write stall behavior of partitioned LSM-trees using our two-phase approach. In a partitioned LSM-tree, a large disk component is range-partitioned into multiple small SSTables and each merge operation only processes a small number of SSTables with overlapping ranges. Since merges always happen immediately once a level is full, a single-threaded scheduler could be sufficient to minimize write stalls. In the remainder of this section, we will evaluate LevelDB's

single-threaded merge scheduler.

5.6.1 LevelDB's Merge Scheduler

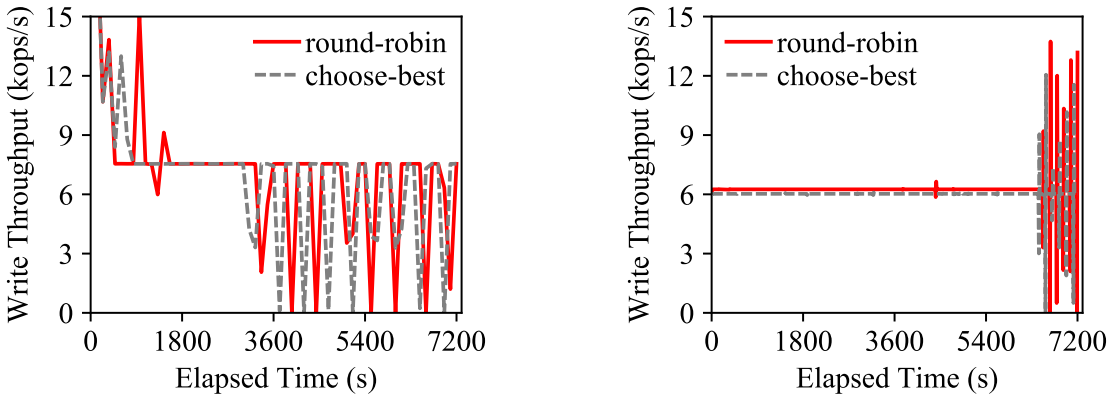
LevelDB's merge scheduler is single-threaded. It computes a score for each level and selects the level with the largest score to merge. Specifically, the score for Level 0 is computed as the total number of flushed components divided by the minimum number of flushed components to merge. For a partitioned level (1 and above), its score is defined as the total size of all SSTables at this level divided by the configured maximum size. A merge operation is scheduled if the largest score is at least 1, which means that the selected level is full. If a partitioned level is chosen to merge, LevelDB selects the next SSTable to merge in a round-robin way.

LevelDB only restricts the number of flushed components at Level 0. By default, the minimum number of flushed components to merge is 4. The processing of writes will be slowed down or stopped of the number of flushed component reaches 8 and 12 respectively. Since we have already shown in Section 5.5.1 that processing writes as quickly as possible reduces write latencies, we will only use the stop threshold (12) in our evaluation.

Experimental Evaluation. We have implemented LevelDB's partitioned leveling merge policy and its merge scheduler inside AsterixDB for evaluation. Similar to LevelDB, the minimum number of flushed components to merge was set at 4 and the stop threshold was set at 12 components. Unless otherwise noted, the maximum size of each SSTable was set at 64MB. The memory component size was set at 128MB and the base size of Level 1 was set at 1280MB. The size ratio was set at 10. For the experimental dataset with 100 million records, this results in a 4-level LSM-tree where the largest level is nearly full. To minimize write stalls caused by flushes, we used two memory components and a separate flush thread. We further evaluated the impact of two widely used merge selection strategies on write stalls. The round-robin strategy chooses the next SSTable to merge in a round-robin way. The choose-best strategy [116] chooses the SSTable with the fewest

overlapping SSTables at the next level.

We used our two-phase approach to evaluate this partitioned LSM-tree design. The instantaneous write throughput during the testing phase is shown in Figure 5.16a, where the write throughput of both strategies decreases over time due to more frequent stalls. Moreover, under the uniform update workload, the alternative selection strategies have little impact on the overall write throughput, as reported in [72]. During the testing phase, we used a constant arrival process to evaluate write stalls. The instantaneous write throughput of both strategies is shown in Figure 5.16b. As the result shows, in both cases write stalls start to occur after time 6000s. This suggests that the measured write throughput during the testing phase is not sustainable.



(a) Testing Phase

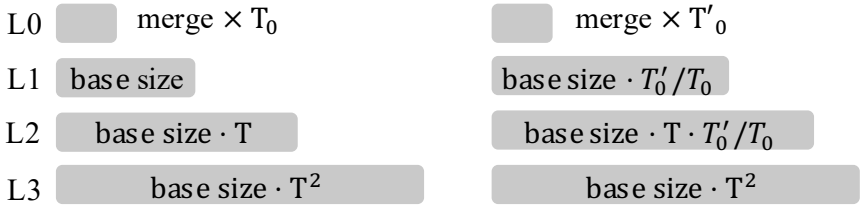
(b) Running Phase

Figure 5.16: Instantaneous Write Throughput of Partitioned LSM-tree

5.6.2 Measuring Sustainable Write Throughput

One problem with LevelDB's score-based merge scheduler is that it merges as many components at Level 0 as possible at once. To see this, suppose that the minimum number of mergeable components at Level L_0 is T_0 and that the maximum number of components at Level 0 is T_0'. During the testing phase, where writes pile up as quickly as possible, the merge scheduler tends to merge the maximum possible number of components T_0' instead of just T_0 at once. Because of this, the LSM-tree will eventually transit from the expected shape (Figure 5.17a) to the actual shape (Fig-

ure 5.17b), where T is the size ratio of the partitioned levels. Note that the largest level is not affected since its size is determined by the number of unique entries, which is relatively stable. Even though this elastic design dynamically increases the processing rate as needed, it has the following problems.



(a) Expected LSM-tree (b) Actual LSM-tree

Figure 5.17: Problem of Score-Based Merge Scheduler

Unsustainable Write Throughput. The measured maximum write throughput is based on merging T_0' flushed components at Level 0 at once. However, this is likely to cause write stalls during the running phase since flushes cannot further proceed.

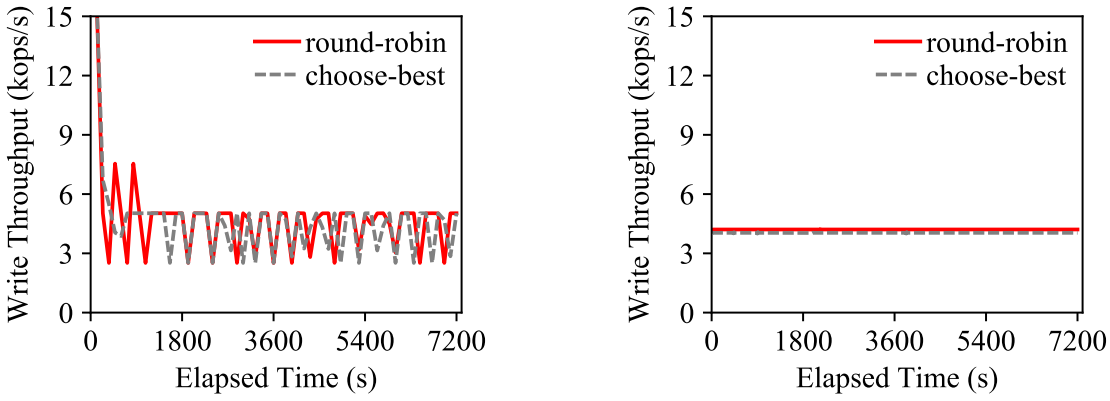
Suboptimal Trade-Offs. The LSM-tree structure in Figure 5.17b is no longer making optimal performance trade-offs since the size ratios between its adjacent levels are not the same anymore [90]. By adjusting the sizes of intermediate levels so that adjacent levels have the same size ratio, one can improve both write throughput and space utilization without affecting query performance.

Lower Space Utilization. One motivation for industrial systems to adopt partitioned LSM-trees is their higher space utilization [49]. However, the LSM-tree depicted in Figure 5.17b violates this performance guarantee because the ratio of wasted space increases from $1/T$ to $T_0'/T_0 \cdot 1/T$.

Because of these problems, the measured maximum write throughput cannot be used in the long-term. We propose a simple solution to address these problems. During the testing phase, we always merge exactly T_0 components at Level 0. This ensures that merge preferences will be given equally to all levels so that the LSM-tree will stay in the expected shape (Figure 5.17a). Then, during the running phase, the LSM-tree can elastically merge more components at Level 0 as needed to absorb

write bursts.

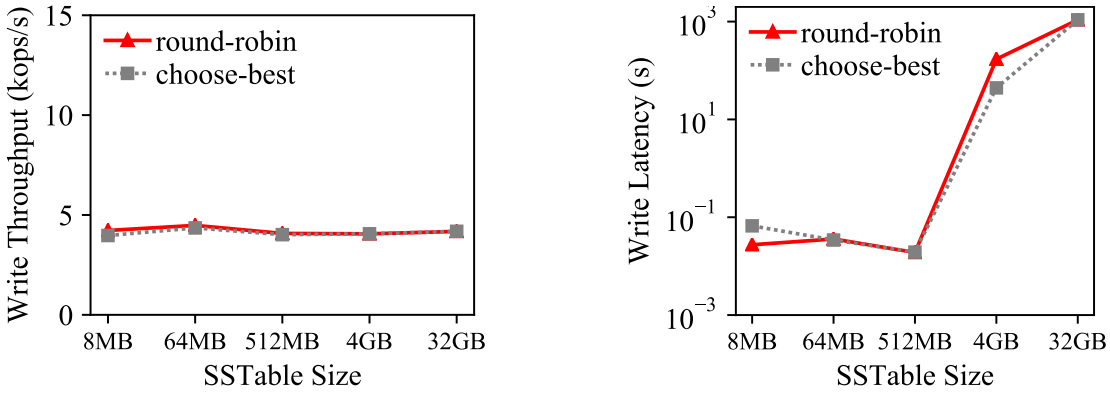
To verify the effectiveness of the proposed solution, we repeated the previous experiments on the partitioned LSM-tree. During the testing phase, the LSM-tree always merged 4 components at Level 0 at once. The measured instantaneous write throughput is shown in Figure 5.18a, which is 30% lower than that of the previous experiment. During the running phase, we used a constant arrival process based on this lower write throughput. The resulting instantaneous write throughput is shown in Figure 5.18b, where the LSM-tree successfully maintains a sustainable write throughput without any write stalls, which in turn results in low write latencies (not shown in the figure). This confirms that LevelDB's single-threaded scheduler is sufficient to minimize write stalls, given that a single merge thread can fully utilize the I/O bandwidth budget.



(a) Testing Phase (b) Running Phase

Figure 5.18: Instantaneous Write Throughput of Partitioned LSM-tree with the Proposed Solution

After fixing the unsustainable write throughput problem of LevelDB, we further evaluated the impact of the SSTable size on the write stalls of partitioned LSM-trees. In this experiment, we varied the maximum size of each SSTable from 8MB to 32GB so that partitioned merges effectively transit into full merges. The maximum write throughput during the running phase and the 99th percentile write latencies during the testing phase are shown in Figures 5.19a and 5.19b respectively. Even though the SSTable size has little impact on the overall write throughput, a large SSTable size can cause large write latencies since we have shown in Section 5.5 that a single-threaded scheduler is insufficient to minimize write stalls for full merges. Most implementations



(a) Testing Phase: Maximum Write Throughput (b) Running Phase: 99% Percentile Write Latency

Figure 5.19: Impact of SSTable Size on Write Throughput and Write Stalls

of partitioned LSM-trees today already choose a small SSTable size to bound the temporary space occupied by merges. We see here that one more reason to do so is to minimize write stalls under a single-threaded scheduler.

5.7 Extension: Secondary Indexes

We now extend our two-phase approach to evaluate LSM-based datasets in the presence of secondary indexes. We first discuss two secondary index maintenance strategies used in practical systems, followed by the experimental evaluation and analysis.

5.7.1 Secondary Index Maintenance

An LSM-based storage system often contains a primary index plus multiple secondary indexes for a given dataset [77]. The primary index stores the records indexed by their keys, while each secondary index stores the mapping from secondary keys to primary keys. During data ingestion, secondary indexes must be properly maintained to ensure correctness. In the primary LSM-tree, writes (inserts, deletes, and updates) can be added blindly to memory since entries with identical

keys will be reconciled by queries automatically. However, this mechanism does not work for secondary indexes since the value of a secondary index key might change. Thus, in addition to adding the new entry to the secondary index, the old entry (if any) must be cleaned as well. We now discuss two secondary index maintenance strategies used in practice [77].

The *eager* index maintenance strategy performs a point lookup to fetch the old record during the ingestion time. If the old record exists, anti-matter entries are produced to cleanup its secondary indexes. The new record is then added to the primary index and all secondary indexes. In an update-heavy workload, these point lookups can become the ingestion bottleneck instead of the LSM-tree write operations.

The *lazy* index maintenance strategy does not cleanup secondary indexes during the ingestion time. Instead, it only adds the new entry into secondary indexes without any point lookups. Secondary indexes are then cleaned up in the background either when merging the primary index components [112] or when merging the secondary index components [77]. Evaluating different secondary index cleanup methods is beyond the scope of this chapter. Instead, we choose to evaluate the lazy strategy without cleaning up secondary indexes.

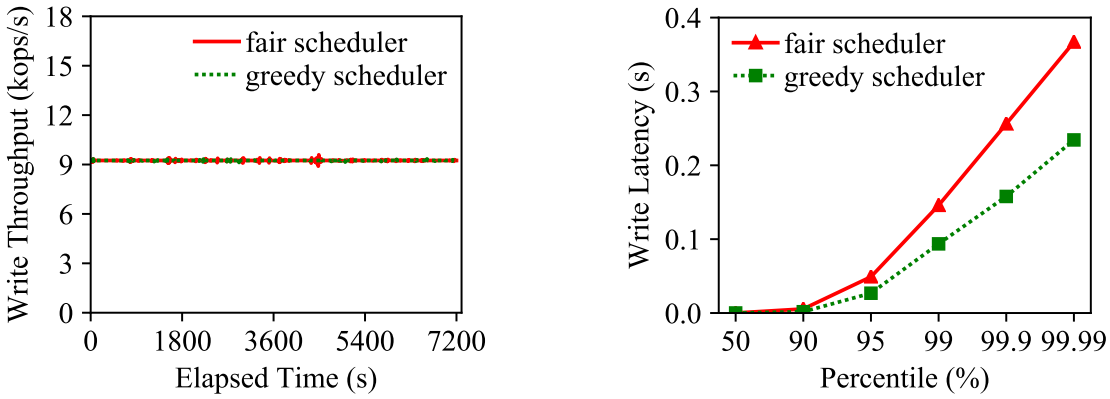
5.7.2 Experimental Evaluation

Experiment Setup. In this set of experiments, we modified the YCSB benchmark to allow us incorporate secondary indexes and formulate secondary index queries. Specifically, we generated records with multiple fields with each secondary field value randomly following a uniform distribution based on the total number of base records. We built two secondary indexes in our experiment. All indexes used the tiering merge policy with size ratio 3.

In this set of experiments, we evaluated two merge schedulers, namely fair and greedy. Each LSM-tree was merged independently with a separate merge scheduler instance. However, these

LSM-trees shared the same memory budget 128MB for each memory component and the I/O bandwidth budget of 100MB/s. We also evaluated two index maintenance strategies, namely eager and lazy. For the eager strategy, we used 8 writer threads to maximize the point lookup throughput. For the lazy strategy, 1 writer thread was sufficient to reach the maximum write throughput since there were no point lookups during data ingestion.

Testing Phase. We first measured the maximum write throughput of the lazy and eager strategies using the fair scheduler during the testing phase. The maximum write throughput was 9,731 records/s for the lazy strategy and 7,601 records/s for the eager strategy. (The eager strategy results in a slightly lower write throughput because it has to cleanup secondary indexes using point lookups.)

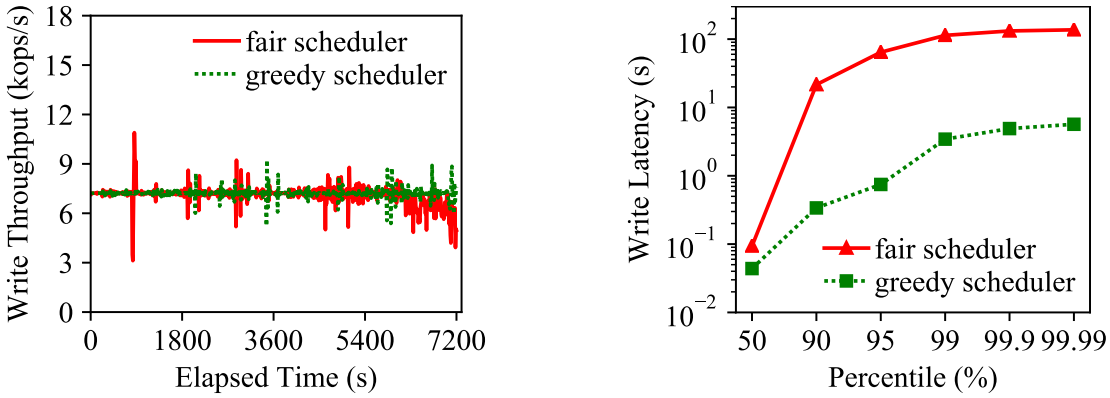


(a) Instantaneous Write Throughput

(b) Percentile Write Latencies

Figure 5.20: Running Phase of Lazy Strategy

Running Phase. During the running phase, we used constant data arrivals to evaluate write stalls. The instantaneous write throughput and percentile write latencies for the lazy and eager strategies are shown in Figures 5.20 and 5.21 respectively. The lazy strategy exhibits a relatively stable write throughput (Figure 5.20a) and lower write latencies (Figure 5.20b), which is similar to the single LSM-tree case. However, under the eager strategy, there are regular fluctuations in the write throughput (Figure 5.21a), results in larger write latencies (Figure 5.21b). This is because the write throughput of the eager strategy is bounded by point lookups in this experiment, and the point lookup throughput inherently varies due to ongoing disk activities and the number of disk



(a) Instantaneous Write Throughput

(b) Percentile Write Latencies

Figure 5.21: Running Phase of Eager Strategy

components. Based on queuing theory [57], the system utilization must be reduced to minimize the write latency. Moreover, the greedy scheduler still has lower write latencies due to its minimizing the number of disk components to improve point lookup performance.

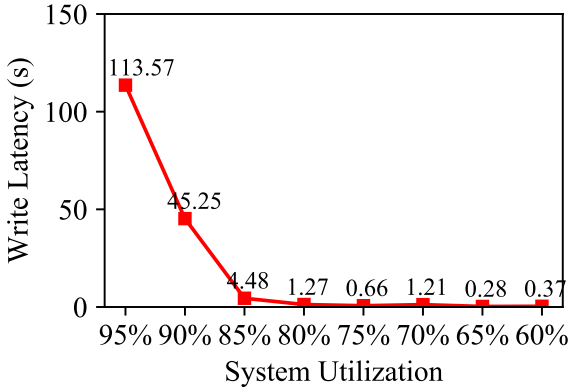
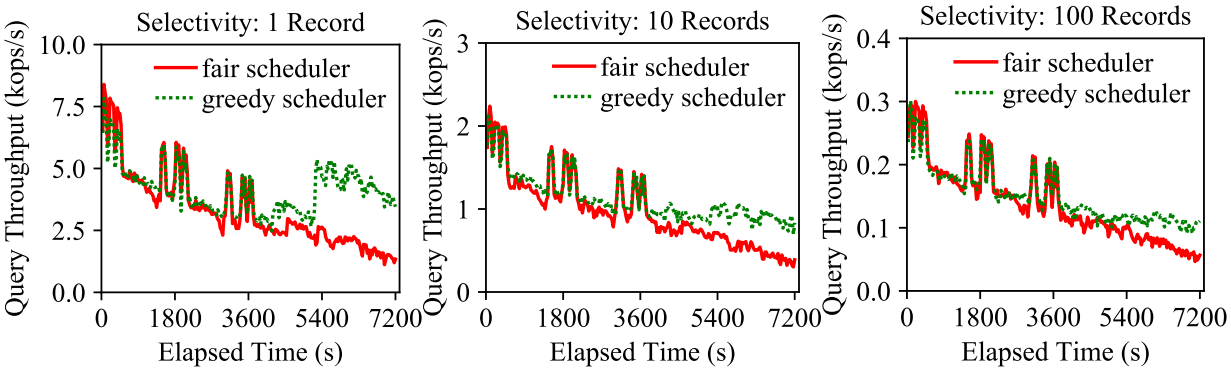


Figure 5.22: 99% Percentile Write Latencies under Eager Strategy with Varying Utilization

Since the eager strategy results in large percentile write latencies under a high data arrival rate, we further carried out another experiment to evaluate the percentile write latencies under different system utilizations, that is, different data arrival rates. The resulting 99% percentile write latencies under various utilizations are shown in Figure 5.22. As the result shows, the write latency becomes relatively small once the utilization is below 80%. This is much smaller than the utilization used in our previous experiments, which was 95%. This result also confirms that because of the inherent variance of the point lookup throughput, one must reduce the data arrival rate, that is, the system

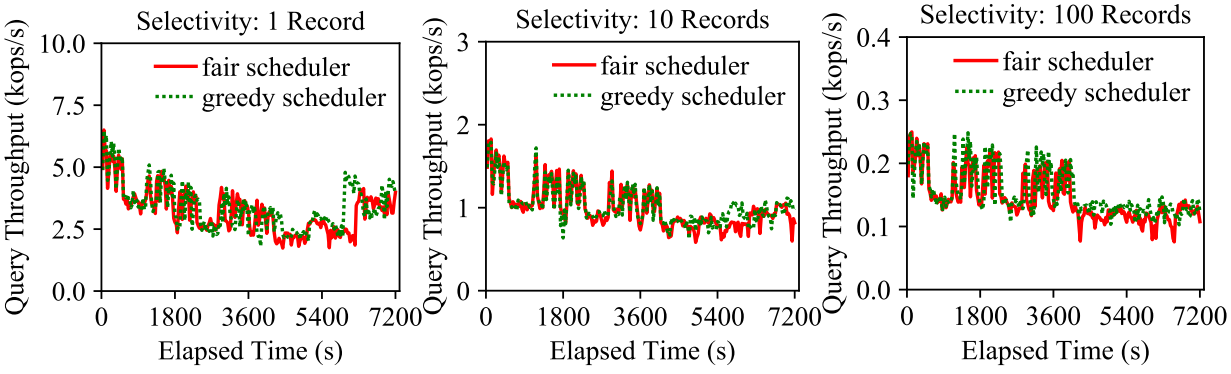
utilization, to achieve smaller write latencies.

Secondary Index Queries. Finally, we evaluated the impact of different merge schedulers and maintenance strategies on the performance of secondary index queries. We used 8 query threads to maximize query throughput. Each secondary index query first scans the secondary index to fetch primary keys, which are then sorted and used to fetch records from the primary index. We varied the query selectivity from 1 record to 100 records so that the performance bottleneck eventually shifts from secondary index scans to primary index lookups.



(a) Selectivity: 1 Record (b) Selectivity: 10 Records (c) Selectivity: 100 Records

Figure 5.23: Instantaneous Query Throughput under Lazy Strategy



(a) Selectivity: 1 Record (b) Selectivity: 10 Records (c) Selectivity: 100 Records

Figure 5.24: Instantaneous Query Throughput under Eager Strategy

The instantaneous query throughput for various query selectivities under the lazy and eager strategy is shown in Figures 5.23 and 5.24 respectively. The query throughput is averaged over each 30-second windows. When the query selectivity becomes larger, the query throughput decreases

because each query has to perform more point lookups to fetch records. The greedy scheduler improves secondary index query performance under the lazy strategy since it reduces the number of disk components. However, the improvement becomes smaller when the query selectivity becomes larger since the query throughput becomes dominated by point lookups. Recall that a secondary index search scans all disk components, while a primary index lookup usually needs to check only one disk component because of Bloom filters. Moreover, the benefit of the greedy scheduler is also less significant under the eager strategy since the write arrival rate is lower.

To summarize, under the lazy strategy, an LSM-based dataset with multiple secondary indexes has similar performance characteristics to the single LSM-tree case, because this can be viewed as a simple extension to multiple LSM-trees. The greedy scheduler also improves query performance by minimizing the number of disk components as before. However, under the eager strategy, the point lookups actually become the ingestion bottleneck instead of LSM-tree write operations. This not only reduces the overall write throughput, but further causes larger write latencies due to the inherent variance of the point lookup throughput.

5.8 Lessons and Insights

Having studied and evaluated the write stall problem for various LSM-tree designs, here we summarize the lessons and insights observed from our evaluation.

The LSM-tree's write latency must be measured properly. The out-of-place update nature of LSM-trees has introduced the write stall problem. Throughout our evaluation, we have seen cases where one can obtain a higher but unsustainable write throughput. For example, the greedy scheduler would report a higher write throughput by starving large merges, and LevelDB's merge scheduler would report a higher but unsustainable write throughput by dynamically adjusting the shape of the LSM-tree. Based on our findings, we argue that in addition to the testing phase, used

by existing LSM research, an extra running phase must be performed to evaluate the usability of the measured maximum write throughput. Moreover, the write latency must be measured properly due to queuing. One solution is to use the proposed two-phase evaluation approach to evaluate the resulting write latencies under high utilization, where the arrival rate is close to the processing rate.

Merge scheduling is critical to minimizing write stalls. Throughout our evaluation of various LSM-tree designs, including bLSM [103], full merges, and partitioned merges, we have seen that merge scheduling has a critical impact on write stalls. Comparing these LSM-tree designs in general depends on many factors and is beyond the scope of this chapter; here we have focused on how to minimize write stalls for each LSM-tree design.

bLSM [103], an instance of full merges, introduces a sophisticated spring-and-gear merge scheduler to bound the processing latency of LSM-trees. However, we found that bLSM still has large variances in its processing rate, leading to large write latencies under high arrival rates. Among the three evaluated schedulers, namely single-threaded, fair, and greedy, the single-threaded scheduler should not be used in practical systems due to the long stalls caused by large merges. The fair scheduler should be used when measuring the maximum throughput because it provides fairness to all merges. The greedy scheduler should be used at runtime since it better minimizes the number of disk components, both reducing write stalls and improving query performance. Moreover, as an important design choice, global component constraints better minimizes write stalls.

Partitioned merges simplify merge scheduling by breaking large merges into many smaller ones. However, we found a new problem that the measured maximum write throughput of LevelDB is unsustainable because it dynamically adjusts the size ratios under write-intensive workloads. After fixing this problem, a single-threaded scheduler with a small SSTable size, as used by LevelDB, is sufficient for delivering low write latencies under high utilization. However, fixing this problem reduced the maximum write throughput of LevelDB by roughly one-third in our evaluation.

For both full and partitioned merges, processing writes as quickly as possible better minimizes

write latencies. Finally, with proper merge scheduling, all LSM-tree designs can indeed minimize write stalls by delivering low write latencies under high utilizations.

5.9 Conclusions

In this chapter, we have studied and evaluated the write stall problem for various LSM-tree designs. We first proposed a two-phase approach to use in evaluating the impact of write stalls on percentile write latencies using a combination of closed and open system testing models. We then identified and explored the design choices for LSM merge schedulers. For full merges, we proposed a greedy scheduler that minimizes write stalls. For partitioned merges, we found that a single-threaded scheduler is sufficient to provide a stable write throughput but that the maximum write throughput must be measured properly. Based on these findings, we have shown that performance variance must be considered together with write throughput to ensure the actual usability of the measured throughput.

Chapter 6

Adaptive Memory Management for LSM-based Storage Systems

6.1 Introduction

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 [7] on RocksDB [10], PolarDB [33] on X-Engine [59], and AsterixDB [17], 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.

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

Unfortunately, adaptivity is non-trivial, as it is highly workload-dependent. Existing LSM-tree implementations, such as RocksDB [10] and AsterixDB [66], 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 chapter, we seek to break down these memory walls in LSM-based storage systems to enable adaptive memory memory management and maximize performance and efficiency. We will focus on the partitioned leveling design (Figure 2.2) due to its wide adoption in today's LSM-based storage systems.

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.

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 [40] and the TPC-C benchmark [11] 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 chapter is organized as follows. Section 6.2 discusses some background information and related work of this chapter. Section 6.3 presents our adaptive memory management architecture for LSM-trees. Section 6.4 describes the new memory component structure for managing the write memory. Section 6.5 presents the design and implementation of the memory tuner. Section 6.6 experimentally evaluates the proposed techniques. Finally, Section 6.7 concludes the chapter.

6.2 Background

6.2.1 Write Memory of LSM-trees

We have discussed the background information of LSM-trees in Section 2.2. In order to show the importance of the write memory, here we provide a cost analysis that shows the relationship between the write memory size and the per-entry write I/O cost. Our notation is shown in Table 6.1. Note that since we consider multiple LSM-trees, Table 6.1 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 chapter, 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 $\sum_{i=1}^{K} a_i = 1$.

Table 6.1: 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

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

$$C = \frac{e}{P} + \frac{e}{P} \cdot (T+1) \cdot \log_T \frac{|L_N|}{a \cdot M_w} \tag{6.1}$$

As Equation 6.1 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.

6.2.2 Related Work

In Chapter 3, we have surveyed the existing research efforts for optimizing LSM-trees. Here we focus on the memory management aspect of LSM-trees and database systems.

Memory Management of LSM-trees. Efficient memory management is important for LSM-trees to achieve good performance. FloDB [25] 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 [30] 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 6.6. Monkey [41] uses analytical models to tune the memory allocation between memory components and Bloom filters. ElasticBF [70] 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. Except [66], which describes memory management in AsterixDB, we are not aware of any previous work that considers memory management of multiple heterogeneous LSM-trees.

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 [39], 2Q [63], LRU-K [89], and Hot-Set [100], 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 [14, 108]. Depending on the tuning goals, the memory tuning techniques can be classified as maximiz-

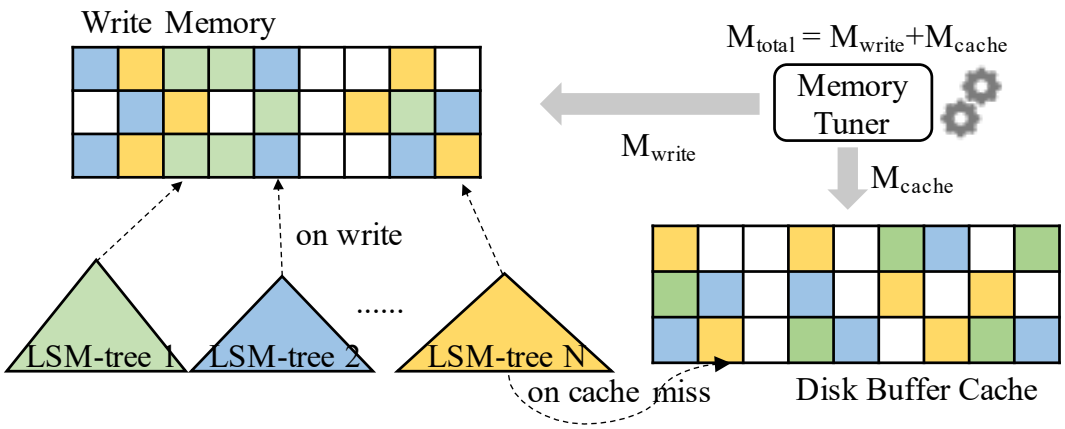


Figure 6.1: Memory Management Architecture

ing the overall throughput or meeting latency requirements. DB2's self-tuning memory manager (STMM) [108] 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 [117] or machine learning approaches [110].

There has been recent interest in exploiting machine learning to tune database configurations [50, 74, 119, 132], 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.

6.3 Memory Management Architecture

In this section, we present our memory management architecture to enable adaptive memory management. In this architecture, depicted in Figure 6.1, 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 6.4.

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 6.5.

6.4 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 the write memory of multiple LSM-trees.

6.4.1 Partitioned Memory Component

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 [126]. 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[2].

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 6.2, 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 6.2) are flushed to disk in a round-robin

[2]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%.

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.

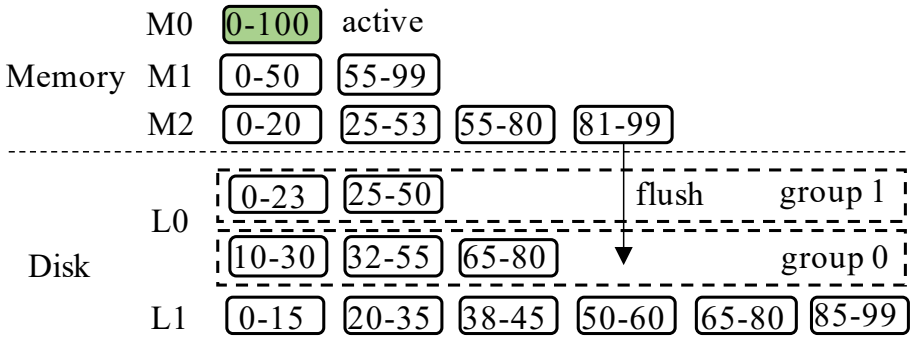


Figure 6.2: LSM-tree with a Partitioned Memory Component

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 [72]. 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.

Grouped L_0

In the original LSM-tree design (Figure 2.2), 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. Consider the two groups in Figure 6.2, where group 0 is older than group 1. When flushing the SSTable labeled 81-99, the resulting SSTable will be inserted into the older group 0. If the SSTable labeled 25-53 is flushed, a new group will be created because group 1 contains an overlapping SSTable 25-50. 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 the ratio between the total size of the overlapping SSTables at L_1 and the total size of the merging SSTables at L_0. Consider the example LSM-tree in Figure 6.2. Group 1 will be selected for the merge because it has fewer SSTables than group 0. The SSTable labeled 0-23 could be merged with the SSTable labeled 10-30 at group 0 and the SSTables labeled 0-15 and 20-35 at L_1, whose overlapping ratio would be 1. The SSTable labeled 25-50 could be merged with the SSTables labeled 10-30 and 32-55 in group 0 and the SSTables labeled from 0-15 to 50-60 at L_1, whose overlapping ratio would be $4/3$. Thus, to reduce write amplification, the SSTable labeled 0-23 will be selected for the merge.

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[3].

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 6.6, 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 6.3. 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

[3]Our preliminary solution [76] 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.

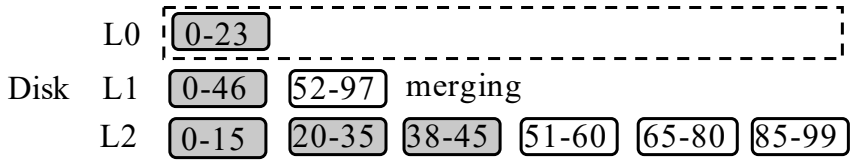


Figure 6.3: Example Merge for Removing L_1

removed from the LSM-tree.

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.)

6.4.2 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 [10] and HBase [5], 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:

$$\min_{a_i} \sum_{i=1}^{K} \frac{r_i}{e_i} \cdot C_i, \text{ s.t.} \sum_{i=1}^{K} a_i = 1 \tag{6.2}$$

By substituting Equation 6.1 from Section 6.2.1 into Equation 6.2 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}.

6.5 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, followed by its design and implementation.

6.5.1 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 6.5.5.

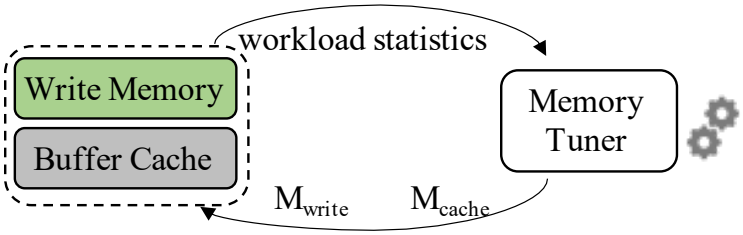


Figure 6.4: Workflow of Memory Tuner

Based on this idea, the memory tuner uses a feedback-control loop to tune memory allocation, as depicted in Figure 6.4. 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 by the memory tuner (Table 6.2) in addition to the LSM-tree notation listed in Table 6.1. Note that with secondary indexes each operation may write multiple entries to multiple LSM-trees.

Table 6.2: 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

6.5.2 Estimating the Write Cost Derivative

For the i-th LSM-tree, recall that Equation 6.1 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

$$write_i'(x) = \frac{w_i}{op} \cdot \frac{e_i}{P} \cdot (T+1) \cdot \frac{1}{x \cdot \ln T} \tag{6.3}$$

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 6.3, we have

$$write_i'(x) = -\frac{merge_i(x)}{x \cdot \ln \frac{|L_{N_i}|}{a_i \cdot x}} \tag{6.4}$$

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 6.4 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 6.4 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:

$$write'(x) = \sum_{i=1}^{K} -\frac{merge_i(x)}{x \cdot \ln \frac{|L_{N_i}|}{a_i \cdot x}} \cdot \frac{flush_{mem_i}}{flush_{mem_i} + flush_{log_i}} \tag{6.5}$$

Example 5.1. Consider an example with two LSM-trees. Suppose that the total write memory x is 128MB. Suppose that the first LSM-tree receives 80% of the write memory ($a_1 = 0.8$) with a last level size of 100GB ($|L_{N_1}| = 100GB$) and that its merge cost per operation is 1 page/op ($merge_1(128MB) = 1$ page/op). Similarly, for the second LSM-tree, suppose that $a_2 = 0.2$, $|L_{N_2}| = 50GB$, and $merge_2(128MB) = 0.8$ page/op. For simplicity, suppose that all flushes are memory-triggered. Based on Equation 6.4, $write'_1(128MB) \approx -1.08e^{-9}$ page/op and $write'_2(128MB) \approx -0.78e^{-9}$ page/op. Thus, $write'(128MB) \approx -1.86e^{-9}$ page/op. This implies that if we allocate one more byte of write memory, the write cost can be reduced by $1.86e^{-9}$ page/op.

6.5.3 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 [108]. 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

$$read'(x) = \frac{saved_q + saved_m}{sim} + write'(x) \cdot \frac{read_m(x)}{merge_m(x)} \tag{6.6}$$

Example 5.2. Continuing from Example 5.1, suppose that the simulated cache size is 32MB. Suppose that the simulated cache reports that the saved query disk reads per operation is $saved_q = 0.01$ page/op and that the saved merge disk reads per operation is $saved_m = 0.008$ page/op. Moreover, suppose that the total number of merge disk reads per operation is $read_m(x) = 2.4$ page/op. Thus, we can compute $read'(x) = -1.94e^{-9}$ page/op. This means that allocating 1 more byte of write memory can decrease the disk read cost per operation by $1.94e^{-9}$ page/op, as disk reads are mainly

performed by merges in this case.

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 $x_{i+1} = x_i - \frac{cost'(x_i)}{A}$.

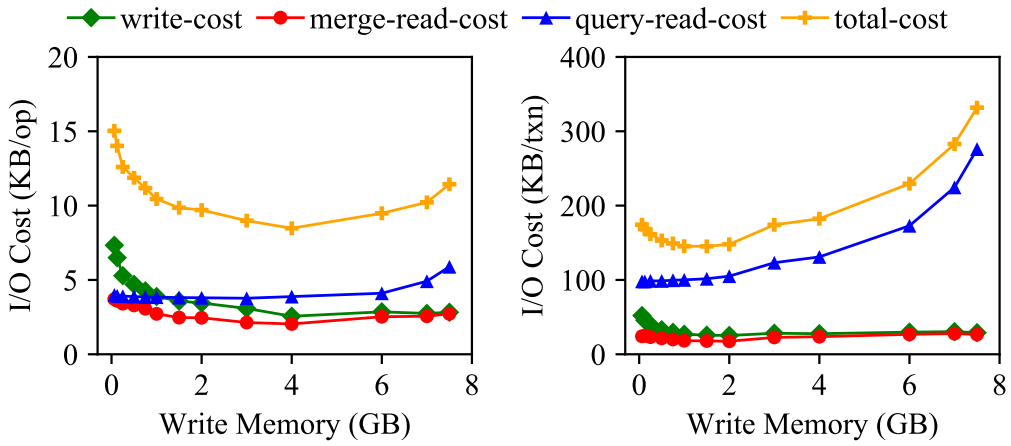
Since the memory tuner deals with a complex system with constantly changing workloads and possible estimation errors, several heuristics are used to ensure the stability of the memory tuner. First, when the tuner does not have enough samples to construct the linear function or when the estimated memory 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.

6.5.5 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 6.5 plots the I/O costs for the the YCSB [40] write-heavy workload and the TPC-C [11] 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 6.6.1.) 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.



(a) YCSB (Write-Heavy) (b) TPCC (SF=2000)

Figure 6.5: I/O Costs under Different Write Memory Sizes

6.6 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 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.

6.6.1 Experimental Setup

Hardware. All experiments in this chapter 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 [40] and TPC-C [11]. 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 6.6.2. 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[4] 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

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

client-triggered aborts (1%) of the NewOrder transaction.

6.6.2 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 [40] due to its simplicity and customizability. For the last set of experiments, we used the TPC-C benchmark [11] since it represents a more realistic workload.

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 [30]. 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 6.4.2, namely max-memory (called *MEM*), min-LSN (called *LSN*), and opti-

mal (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.

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), 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 6.6 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 [5] uses Java objects to manage memory components. Although AsterixDB is written in

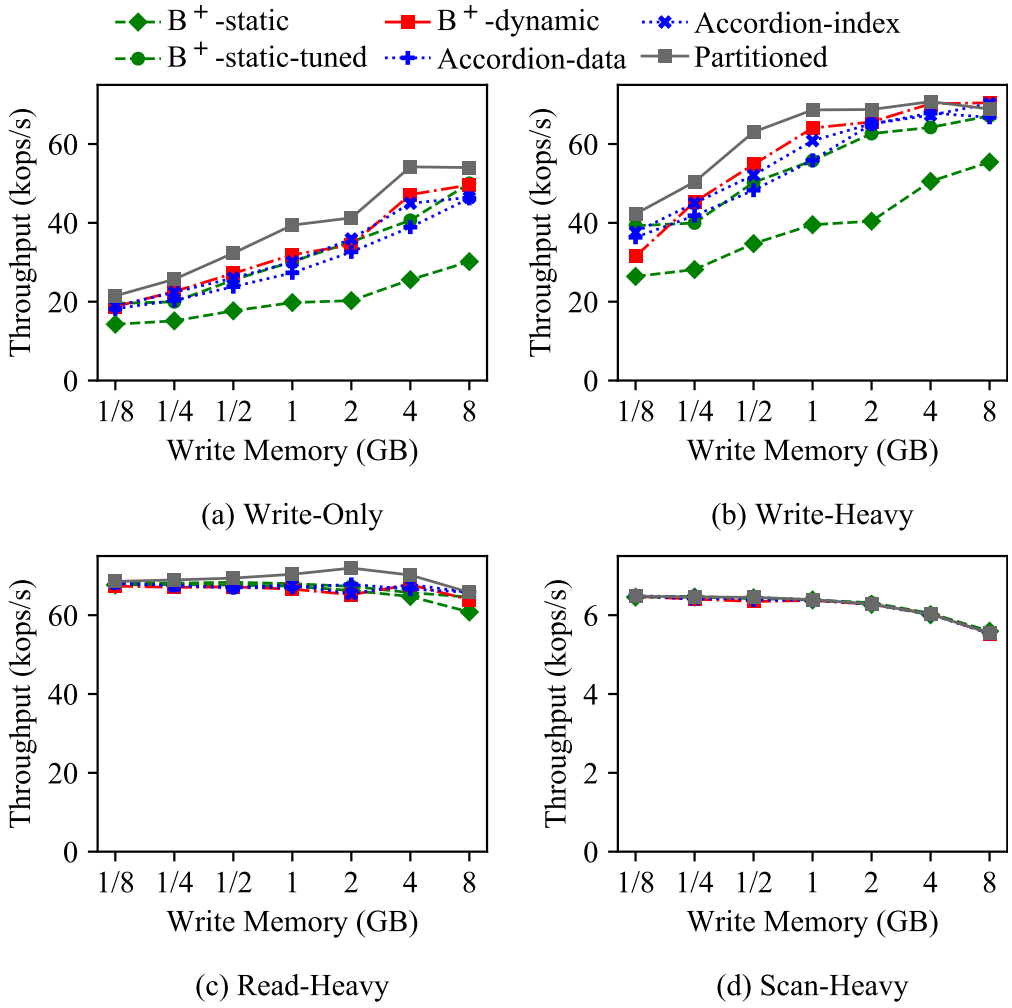


Figure 6.6: Experimental Results for a Single LSM-tree

Java, it uses off-heap structures for memory management [31, 66]. 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 [78], 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 6.7 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.

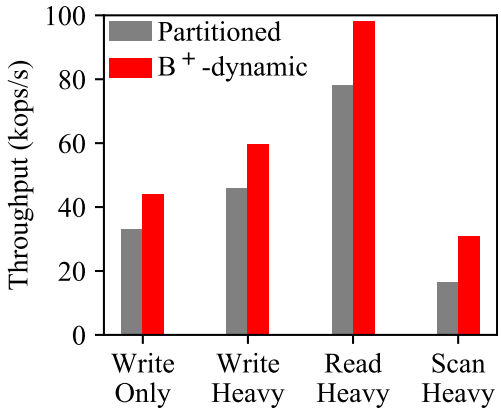


Figure 6.7: Evaluation of Memory Merge Overhead

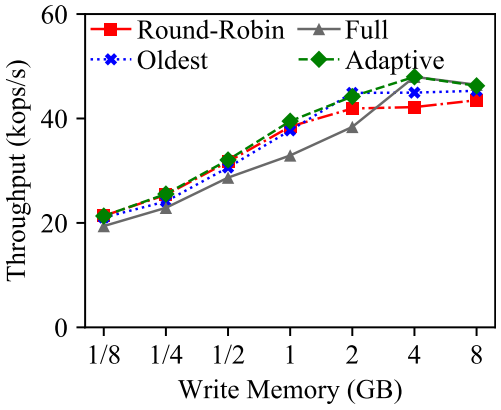


Figure 6.8: Evaluation of Flush Heuristics

Evaluation of Flush Heuristics. In Section 6.4.1, we described a number of heuristics to adaptive select SSTables to flush by distinguishing memory-triggered flushes, log-triggered flushes, and full flushes. To evaluate the benefit of the proposed heuristics, we performed an experiment with

the following flush strategies. The first strategy, called *Round-Robin*, always flushes SSTables at the last level in a round-robin way. The second strategy, called *Oldest*, always flushes the oldest SSTable along with all overlapping SSTables at higher levels. The third strategy, called *Full*, always performs full flushes by flushing all SSTables together. Finally, the proposed strategy, called *Adaptive*, combines different flush strategies as described in Section 6.4.1. We used the write-only workload since different flush strategies mainly impact the write performance.

Figure 6.8 shows the resulting write throughput under different write memory sizes. As one can see, there is no clear winner among the first three strategies. Under small write memory (smaller than 1GB) with many memory-triggered flushes, *Round-Robin* achieves the highest throughput. *Oldest* performs better when the write memory size becomes larger with many log-triggered flushes, as flushing the oldest SSTable better facilitates log truncation. When the write memory size becomes even larger (larger than 4GB), log-triggered flushes become dominating and thus *Full* achieves the best throughput. This experiment also shows that by combining these three flush strategies together, the proposed flush strategy always achieves the best possible write throughput.

Evaluation of Grouped L_0 Structure. To evaluate the effectiveness of the grouped L_0 structure that organizes non-overlapping SSTables into groups, we performed an experiment with the following alternative structures. First, we evaluated the L_0 structure in the original LSM-tree design shown in Figure 2.2, called *Original*, which simply tolerates N SSTables. The second structure, called *Grouped*, organizes SSTables into groups but does not use the proposed heuristics to minimize the write amplification. Specifically, this structure always picks the leftmost SSTable from the oldest group to merge, along with overlapping SSTables from newer groups. The last structure, called *Greedy-Grouped*, further uses the proposed heuristics to minimize the write amplification. Since all these L_0 structures have the same worst-case query performance, we used a write-only workload to evaluate their impact on writes. To show the impact of the alternative L_0 structures, we set the maximum number of groups at L_0 as 4.

Figure 6.9 shows the resulting write throughput of alternative L_0 structures. The L_0 structure

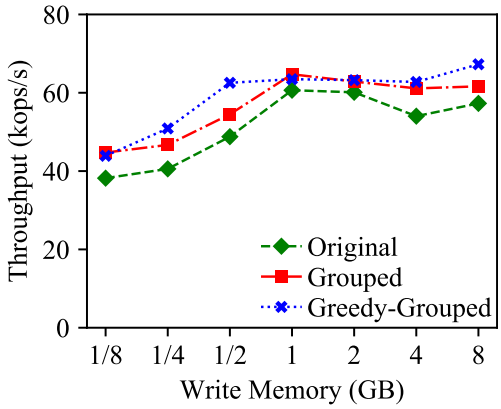


Figure 6.9: Evaluation of L0 Structure

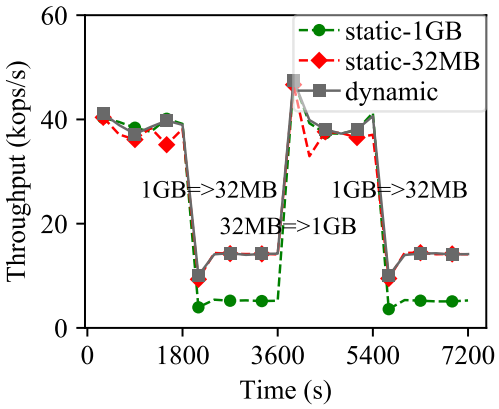


Figure 6.10: Write Throughput with Varying Write Memory

used in the original LSM-tree design led to the worst write throughput because it did not exploit the disjoint SSTables to reduce the write cost. By organizing disjoint SSTables into groups, the grouped L_0 structure increased the write throughput since multiple overlapping SSTables can be merged at once. Finally, the proposed greedy heuristics further increased the write throughput by greedily selecting SSTables to merge to minimize the write amplification.

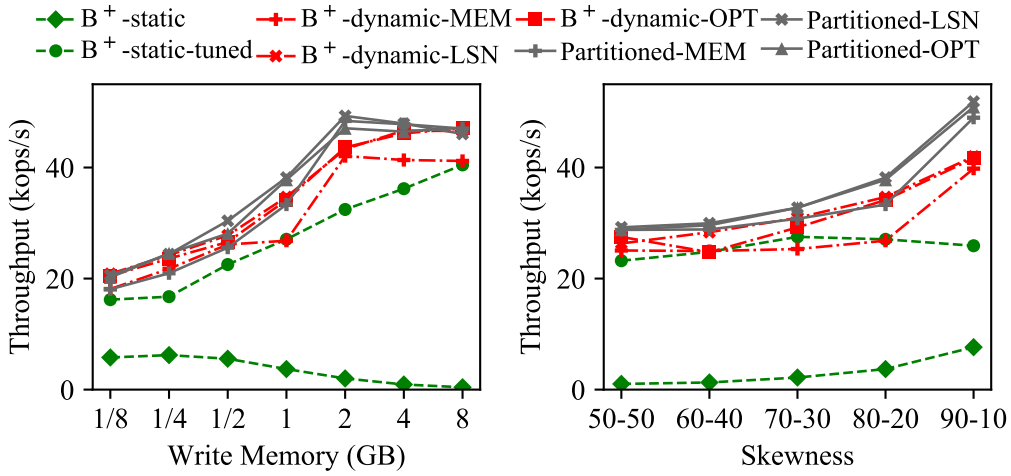
Benefits of Dynamically Adjusting Disk Levels. To evaluate the benefit of dynamically adjusting disk levels as the write memory changes, we conducted an experiment where the write memory size alternates between 1GB and 32MB every 30 minutes. Each experiment ran for two hours in total. We used the partitioned 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 1GB (called *static-1GB*). The resulting write throughput, aggregated over 5-minute windows, is shown in Figure 6.10. 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.

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 6.11a. 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 6.11b. All memory component structures



(a) Vary Write Memory

(b) Vary Skewness

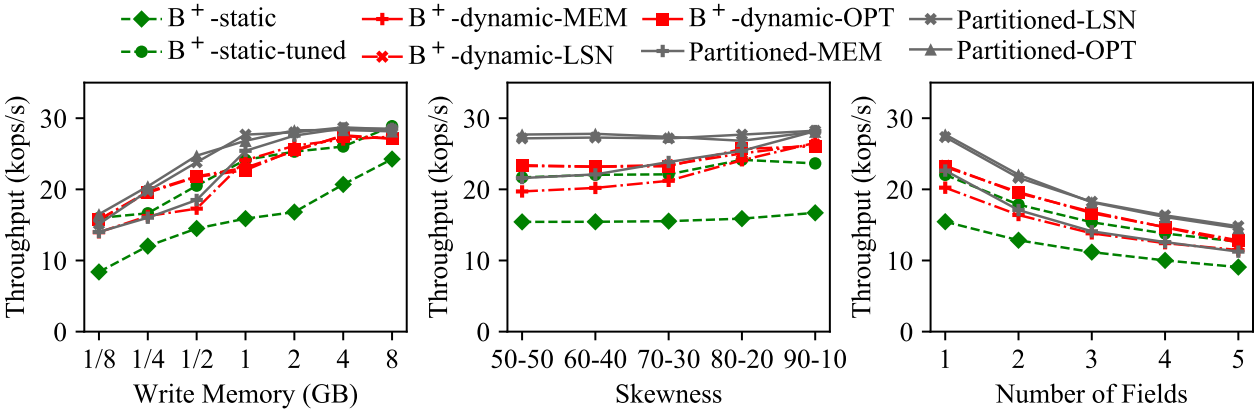
Figure 6.11: Evaluation of Multiple Primary LSM-trees

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.

Multiple Secondary LSM-trees

We further evaluated the alternative memory component structures using multiple secondary LSM-trees for one dataset. The dataset had one primary LSM-tree and 10 secondary LSM-trees, with one secondary LSM-tree per field. The primary LSM-tree had 50 million records with 55GB storage size, and each secondary LSM-tree was about 5GB. As before, we used the write-only workload

to focus on write performance. It should be noted that each write must also performs a primary index lookup to cleanup secondary indexes [77]. Unless otherwise noted, each write only updates one secondary field, but the choice of updated fields followed the same hotspot distribution as in Section 6.6.2. For each field, its values followed the default Zipfian distribution used in YCSB.



(a) Vary Write Memory (b) Vary Skewness (c) Vary Updated Fields

Figure 6.12: Evaluation of Multiple Secondary LSM-trees

Impact of Write Memory. First, we varied the total write memory to evaluate its impact on the different memory component structures with secondary indexes. The resulting write throughput is shown in Figure 6.12a. In general, the results are consistent with the multiple primary LSM-tree case in Figure 6.11a. Note that B^+-*static-tuned* achieves better performance in this case because B^+-*static-tuned* allocates the write memory at a dataset level. Thus, the primary LSM-tree and all secondary LSM-trees share the same memory budget, which is similar to other schemes.

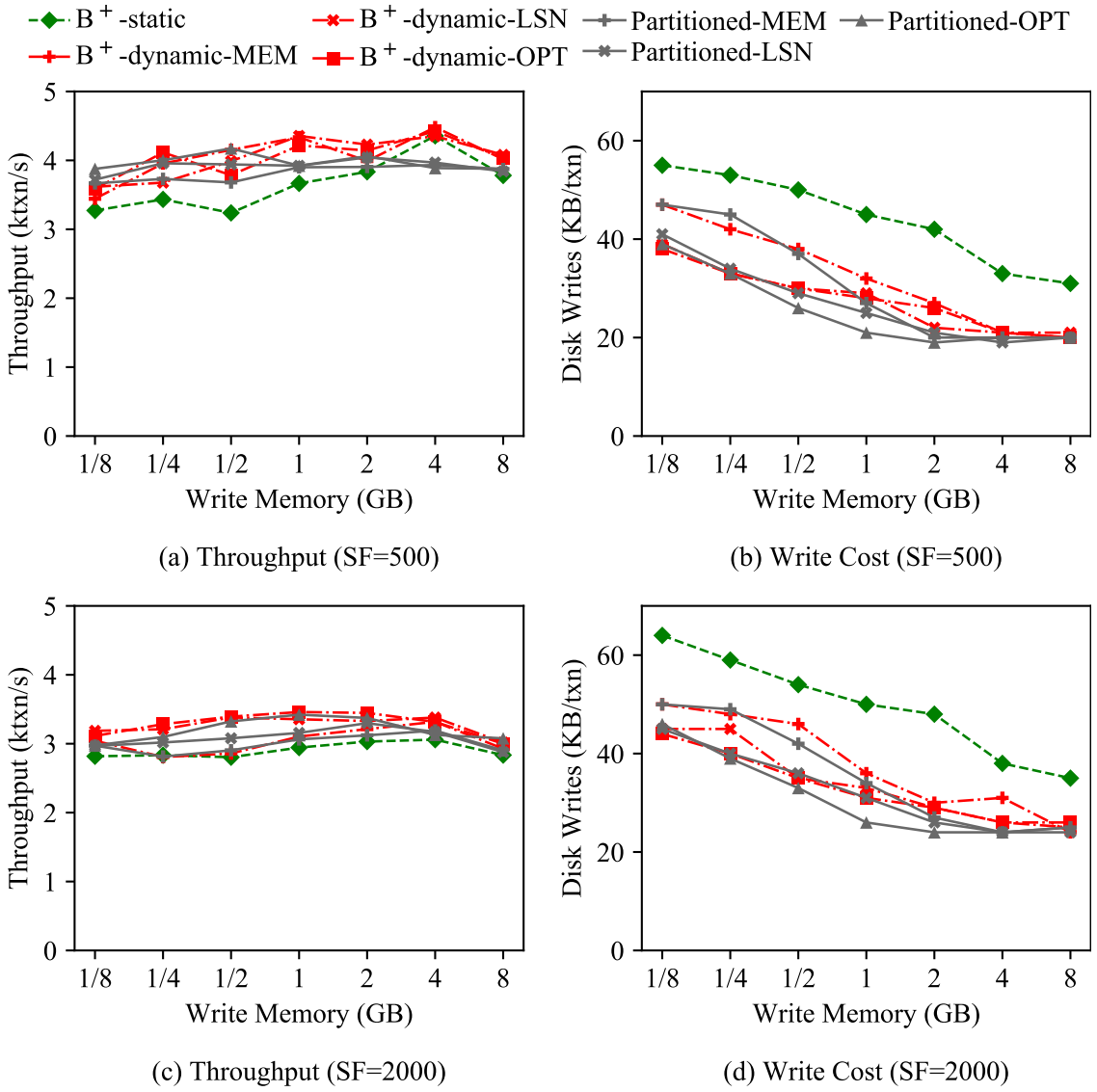
Impact of Updated Field Skewness. We further varied the skewness of updated fields to study its impact on write throughput. Figure 6.12b shows the resulting write throughput. As one can see, the skewness of updated fields has a smaller performance impact here compared to the multiple primary LSM-tree case in Figure 6.11b because the size of a secondary LSM-tree is much smaller than the primary one. Moreover, B^+-*dynamic-MEM* and *Partitioned-MEM*, both of which used the max-memory flush policy, still benefit from a more skewed workload. The reason is that when most writes access a small number of hot secondary indexes, the size of their memory components will grow faster and they will be selected by the max-memory policy to flush.

Impact of Number of Updated Fields. Finally, we studied the performance impact of the number of updated fields per write, ranging from 1 to 5. The resulting throughput is shown in Figure 6.12c. Increasing the number of updated fields per write negatively impacts the write throughput because each logical write produces more physical writes. Because of this, the write throughput of all memory component structures decreases in the same way when each write updates more fields.

TPC-C Results

Finally, we used the TPC-C benchmark [11] 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 6.13. 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 6.11, 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



(a) Throughput (SF=500)

(b) Write Cost (SF=500)

(c) Throughput (SF=2000)

(d) Write Cost (SF=2000)

Figure 6.13: Experimental Results on TPC-C

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.

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 ro-

bust, 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 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.

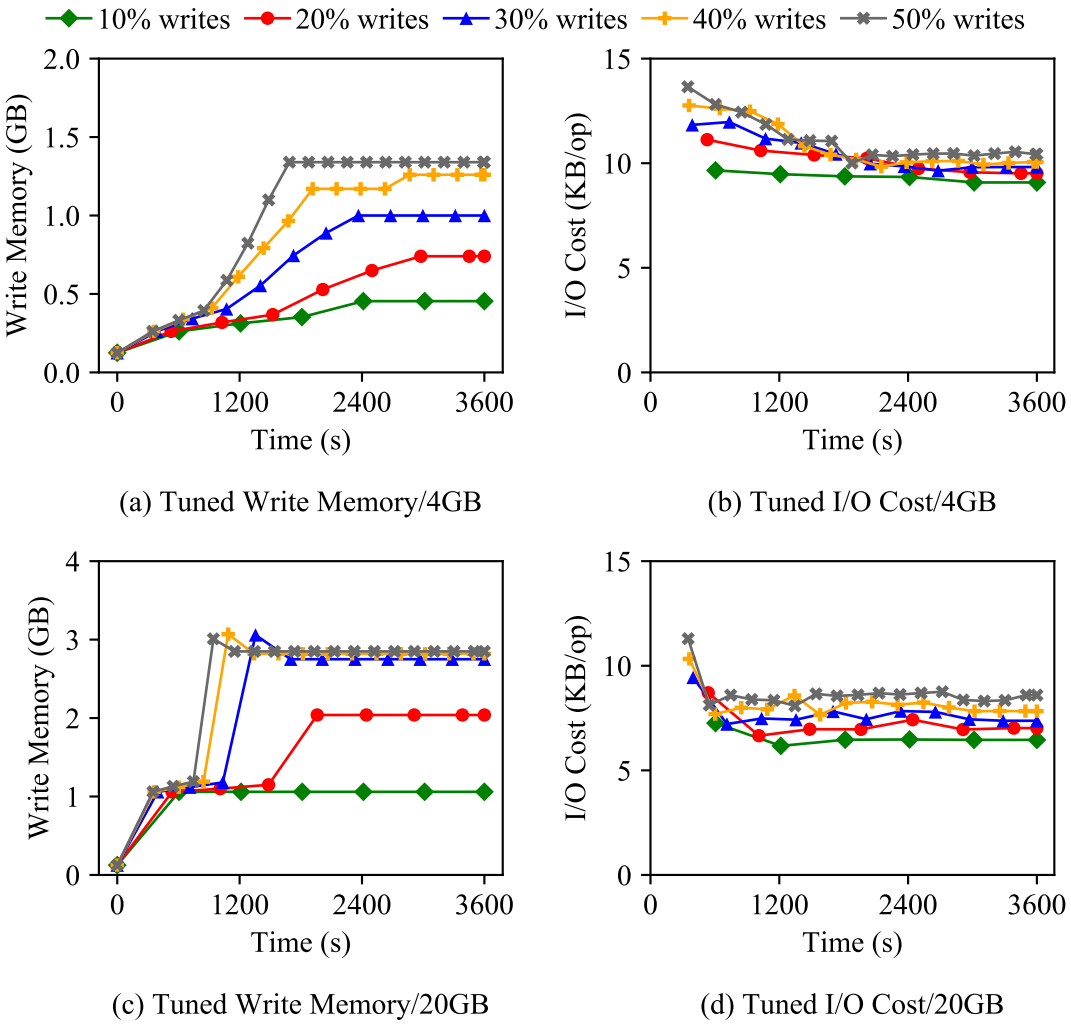
6.6.3 Evaluating the Memory Tuner

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 6.5.4.

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 [40] 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.



(a) Tuned Write Memory/4GB

(b) Tuned I/O Cost/4GB

(c) Tuned Write Memory/20GB

(d) Tuned I/O Cost/20GB

Figure 6.14: Evaluation of Memory Tuner on YCSB

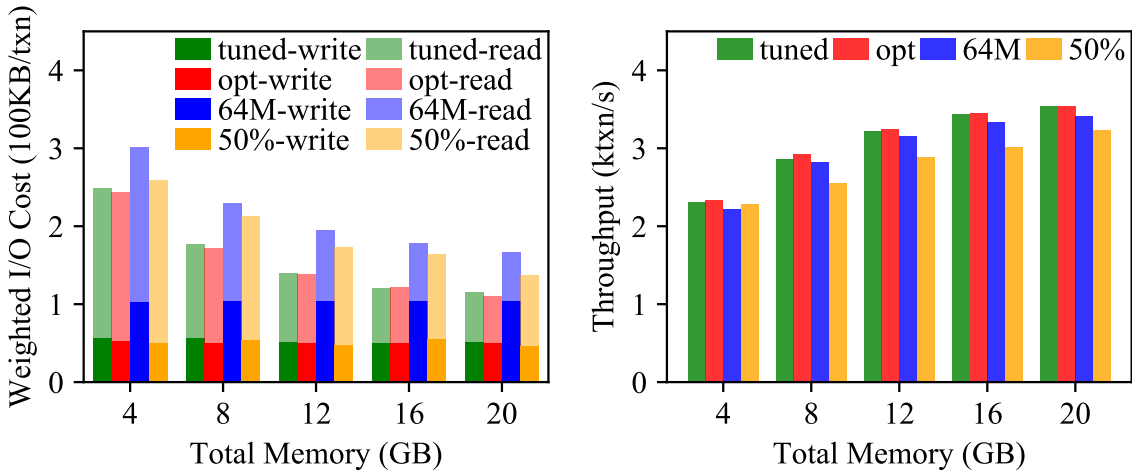
The tuned write memory size and the corresponding I/O costs over time are shown in Figure 6.14. Note that each point denotes one tuning step performed by the memory tuner. We see that the

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 6.14a and 6.14c, 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 6.14a and 6.14c, 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 6.14b and 6.14d, the overall I/O cost also decreases after the memory allocation is tuned over time.

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 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 6.15 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



(a) Weighted I/O Cost

(b) Transaction Throughput

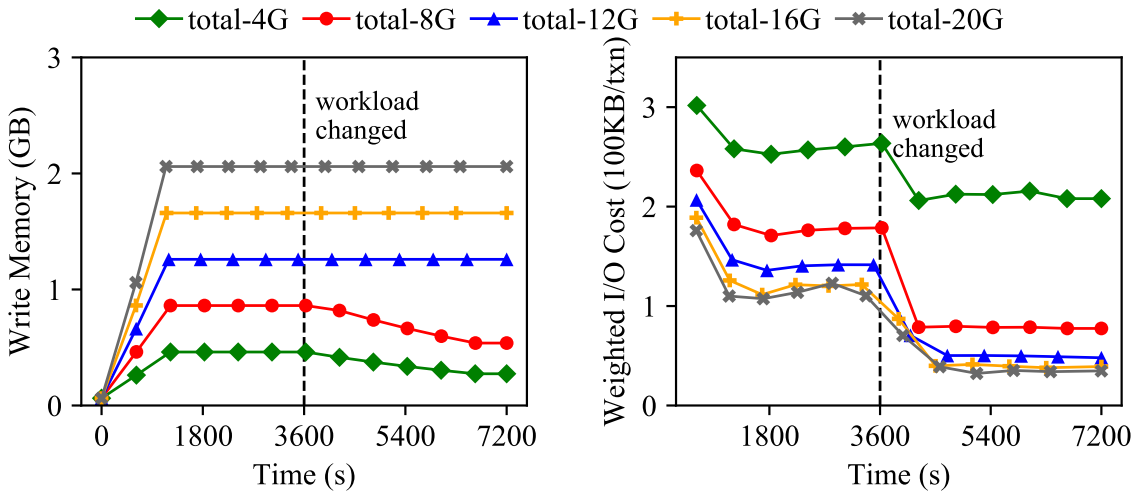
Figure 6.15: Memory Tuner's Accuracy on TPC-C

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.

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 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 6.16. 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.

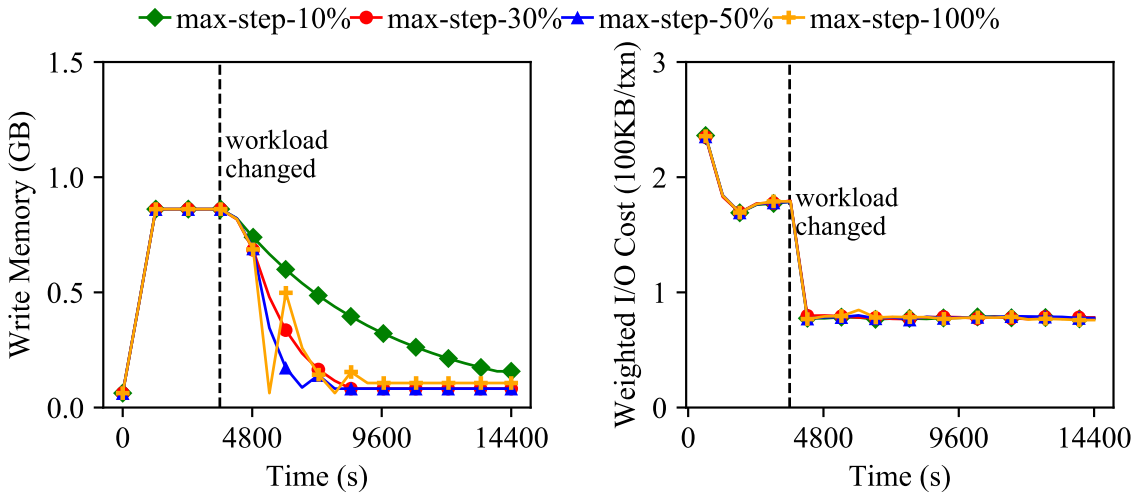


(a) Tuned Write Memory

(b) Tuned I/O Cost

Figure 6.16: Memory Tuner's Responsiveness on TPC-C

To study the impact of the maximum step size on the responsiveness and stability of the memory tuner, we further carried out an experiment that varies the maximum step size from 10% to 100%. The total memory was set at 12GB. Each experiment ran for four hours and the workload changed from the the default TPC-C mix into the read-heavy mix after the first hour. The tuned write memory and weighted I/O cost over time are shown in Figures 6.17a and 6.17b respectively. As the results show, increasing the maximum step size improves responsiveness by allowing the memory tuner to change the memory allocation more quickly. However, this also negatively impacts the memory tuner's stability and leads to some oscillation. Also note that decreasing the write memory more rapidly has a very small impact on the I/O cost since the buffer cache already occupies most of the memory. Thus, the memory tuner's default maximum step size is set at 10% to ensure stability while providing reasonable responsiveness.



(a) Tuned Write Memory

(b) Tuned I/O Cost

Figure 6.17: Impact of Maximum Step Size on Memory Tuner's Responsiveness

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.

6.7 Conclusions

In this chapter, 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.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this dissertation, we have presented techniques for optimizing LSM-based storage systems in Big Data Management Systems (BDMSs).

Chapters 2 and 3 laid the groundwork for these research contributions by first presenting an overview of LSM trees followed by a comprehensive survey of the recent LSM literature.

In Chapter 4, we presented a number of techniques for efficiently maintaining and exploiting of LSM-based secondary indexes and range filters. We first described and evaluated a series of optimizations for efficient point lookups, greatly improving the range of applicability for LSM-based secondary indexes. We further introduced two novel strategies for efficiently maintaining LSM-based auxiliary structures using a primary key index. The Validation strategy eliminates ingestion-time point lookups and defers secondary index maintenance to the background, significantly improving ingestion performance with a small overhead for secondary index queries. The Mutable-bitmap strategy attaches each disk component with a mutable bitmap to mark obsolete

records and only accesses the primary key index during data ingestions. This not only maximizes the the pruning capabilities of filters but also improves ingestion performance.

In Chapter 5, we have studied and evaluated the write stall problem for various LSM-tree designs. We first proposed a two-phase evaluation approach that combines the closed and open system models to experimentally evaluate the impact of write stalls on percentile write latencies. We then identified and explored the design choices for LSM merge schedulers. For full merges, we proposed a greedy scheduler to minimize write stalls. For partitioned merges, we found that a single-threaded scheduler is sufficient to provide a stable write throughput but that the maximum write throughput must be measured properly. Based on these findings, we argue that it is important to evaluate the performance variance of LSM-trees carefully to ensure the actual usability of the measured throughput.

In Chapter 6, 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 to enable adaptive memory management. We then introduced a partitioned memory component structure with new flush polices to better utilize the write memory in order to minimize the overall write cost. To perform adaptive memory management between write memory and the buffer cache, we further described a memory tuner that continuously tunes the memory allocation. We empirically demonstrated that the proposed techniques can successfully reduce the disk I/O cost and improve the overall performance of an LSM-based storage system.

7.2 Future Work

In Chapter 4, we described several new strategies to efficiently maintain LSM-based auxiliary structures. In general, these strategies are suitable for write-heavy workloads, but may not be a good choice for read-heavy workloads. For now, the user must decide in advance that which

maintenance strategy should be used based on the expected workload. In the future, it would be useful to unify these strategies so that the system can adaptively choose the best maintenance strategy based on the current workload.

In Chapter 5, we studied the performance stability problem of LSM-trees using a constant arrival workload. Even though the constant arrival workload revealed some interesting problems for LSM merge schedulers that cause unstable write throughput, arrival uniformity may not be a valid assumption for practical workloads that often contain a lot of inherent variances. The usability of LSM-trees could be further improved by considering the variances exhibited by practical workloads.

In Chapter 6, we proposed a number of adaptive memory management techniques for LSM-trees. Currently, this work has focused on minimizing the disk I/O cost of LSM-trees. Since the performance gap between the CPU and the disk becomes smaller and smaller due to the advancement of storage technologies, an important future direction is to extend this work to minimize the CPU cost and the disk I/O cost together. This will further improve the overall system performance of LSM-trees, especially on modern hardware.

Bibliography

[1] AsterixDB. https://asterixdb.apache.org/.

[2] Cassandra. http://cassandra.apache.org/.

[3] Compaction stalls: something to make better in RocksDB. http://smalldatum.blogspot.com/2017/01/compaction-stalls-something-to-make.html.

[4] Dragon: A distributed graph query engine. https://code.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/.

[5] HBase. https://hbase.apache.org/.

[6] LevelDB. http://leveldb.org/.

[7] MyRocks. https://http://myrocks.io/.

[8] Phoenix. https://phoenix.apache.org/.

[9] Read- and latency-optimized log structured merge tree. https://github.com/sears/bLSM/.

[10] RocksDB. http://rocksdb.org/.

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

[12] YCSB change log. https://github.com/brianfrankcooper/YCSB/blob/master/core/CHANGES.md.

[13] I. Absalyamov et al. Lightweight cardinality estimation in LSM-based systems. In *ACM SIGMOD*, pages 841–855, 2018.

[14] S. Agrawal et al. Database tuning advisor for Microsoft SQL Server 2005. In *ACM SIGMOD*, pages 930–932. ACM, 2005.

[15] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *PVLDB*, 8(8):850–861, 2015.

[16] W. Y. Alkowaileet et al. An LSM-based tuple compaction framework for Apache AsterixDB. *PVLDB*, 13(9):1388–1400, 2020.

[17] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[18] S. Alsubaiee et al. Storage management in AsterixDB. *PVLDB*, 7(10):841–852, 2014.

[19] S. Alsubaiee et al. LSM-based storage and indexing: An old idea with timely benefits. In *International ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich)*, pages 1–6, 2015.

[20] H. Amur et al. Design of a write-optimized data store. Technical report, Georgia Institute of Technology, 2013.

[21] M. Armbrust et al. PIQL: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.

[22] M. Armbrust et al. Generalized scale independence through incremental precomputation. In *ACM SIGMOD*, pages 625–636, 2013.

[23] M. Athanassoulis et al. MaSM: efficient online updates in data warehouses. In *ACM SIGMOD*, pages 865–876. ACM, 2011.

[24] M. Athanassoulis et al. Designing access methods: The RUM conjecture. In *EDBT*, pages 461–466, 2016.

[25] O. Balmau et al. FloDB: Unlocking memory in persistent key-value stores. In *European Conference on Computer Systems (EuroSys)*, pages 80–94, 2017.

[26] O. Balmau et al. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *USENIX Annual Technical Conference (ATC)*, pages 363–375, 2017.

[27] M. A. Bender et al. Don't thrash: how to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012.

[28] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82 – 87, 1976.

[29] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, July 1970.

[30] E. Bortnikov et al. Accordion: Better memory organization for LSM key-value stores. *PVLDB*, 11(12):1863–1875, 2018.

[31] Y. Bu et al. A bloat-aware design for big data applications. *SIGPLAN Not.*, 48(11):119–130, June 2013.

[32] G. Candea et al. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[33] W. Cao et al. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *PVLDB*, 11(12):1849–1862, 2018.

[34] Z. Cao et al. On the performance variation in modern storage stacks. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 329–343, 2017.

[35] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2):4:1–4:26, 2008.

[36] S. Chaudhuri et al. Variance aware optimization of parameterized queries. In *ACM SIGMOD*, pages 531–542. ACM, 2010.

[37] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1):133–162, Nov 1986.

[38] G. J. Chen et al. Realtime data processing at Facebook. In *ACM SIGMOD*, pages 1087–1098, 2016.

[39] H. T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1):311–336, Nov 1986.

[40] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *ACM SoCC*, pages 143–154, 2010.

[41] N. Dayan et al. Monkey: Optimal navigable key-value store. In *ACM SIGMOD*, pages 79–94, 2017.

[42] N. Dayan et al. Optimal Bloom filters and adaptive merging for LSM-trees. *ACM TODS*, 43(4):16:1–16:48, Dec. 2018.

[43] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *ACM SIGMOD*, pages 505–520, 2018.

[44] N. Dayan and S. Idreos. The log-structured merge-bush & the wacky continuum. In *ACM SIGMOD*, pages 449–466, 2019.

[45] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56:74–80, 2013.

[46] G. DeCandia et al. Dynamo: Amazon'y highly available key-value store. In *ACM SOSP*, pages 205–220, 2007.

[47] J. Do et al. Query processing on smart SSDs: Opportunities and challenges. In *ACM SIGMOD*, page 1221–1230, 2013.

[48] J. Do et al. Programming an SSD controller to support batched writes for variable-size pages. In *ICDE*, 2021.

[49] S. Dong et al. Optimizing space amplification in RocksDB. In *CIDR*, 2017.

[50] S. Duan et al. Tuning database configuration parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.

[51] B. Fan et al. Cuckoo filter: Practically better than Bloom. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 75–88, 2014.

[52] Y. Fang et al. Spatial indexing in Microsoft SQL Server 2008. In *ACM SIGMOD*, pages 1207–1216, 2008.

[53] G. Golan-Gueta et al. Scaling concurrent log-structured data stores. In *European Conference on Computer Systems (EuroSys)*, pages 32:1–32:14, 2015.

[54] G. Graefe. Modern B-tree techniques. *Found. Trends Databases*, 3(4):203–402, 2011.

[55] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, pages 47–57, 1984.

[56] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM CSUR*, 15(4):287–317, Dec. 1983.

[57] M. Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[58] D. Huang et al. TiDB: A raft-based HTAP database. *PVLDB*, 13(12):3072–3084, 2020.

[59] G. Huang et al. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *ACM SIGMOD*, pages 651–665, 2019.

[60] J. Huang et al. Statistical analysis of latency through semantic profiling. In *European Conference on Computer Systems (EuroSys)*, pages 64–79, 2017.

[61] J. Huang et al. A top-down approach to achieving performance predictability in database systems. In *ACM SIGMOD*, pages 745–758, 2017.

[62] J. Jia. *Supporting Interactive Analytics and Visualization on Large Data*. PhD thesis, Component Science Department, University of California, Irvine, 2017.

[63] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[64] S. Kannan et al. Redesigning LSMs for nonvolatile memory with NoveLSM. In *USENIX Annual Technical Conference (ATC)*, pages 993–1005, 2018.

[65] A. Khodaei et al. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *Database and Expert Systems Applications (DEXA)*, pages 450–466, 2010.

[66] T. Kim et al. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience*, 50(7):1114–1151, 2020.

[67] Y. Kim et al. A comparative study of log-structured merge-tree-based spatial indexes for big data. In *ICDE*, pages 147–150, 2017.

[68] J. Lawder. *The application of space-filling curves to the storage and retrieval of multi-dimensional data*. PhD thesis, PhD Thesis, University of London, UK, 2000.

[69] Y. Li et al. Tree indexing on solid state drives. *PVLDB*, 3(1-2):1195–1206, 2010.

[70] Y. Li et al. ElasticBF: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *2019 USENIX Annual Technical Conference (ATC)*, pages 739–752, 2019.

[71] Y. Li et al. Enabling efficient updates in KV storage via hashing: Design and performance evaluation. *ACM Transactions on Storage (TOS)*, 15(3):20, 2019.

[72] H. Lim et al. Towards accurate and fast evaluation of multi-stage log-structured designs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 149–166, 2016.

[73] D. Lomet and C. Luo. Efficiently reclaiming space in a log structured store. *CoRR*, abs/2005.00044, 2020.

[74] J. Lu et al. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *PVLDB*, 12(12):1970—1973, 2019.

[75] L. Lu et al. WiscKey: Separating keys from values in SSD-conscious storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 133–148, 2016.

[76] C. Luo. Breaking down memory walls in LSM-based storage systems. In *ACM SIGMOD*, pages 2817–2819, 2020.

[77] C. Luo and M. J. Carey. Efficient data ingestion and query processing for LSM-based storage systems. *PVLDB*, 12(5):531–543, 2019.

[78] C. Luo and M. J. Carey. On performance stability in LSM-based storage systems. *PVLDB*, 13(4):449–462, 2019.

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

[80] C. Luo and M. J. Carey. Breaking down memory walls: Adaptive memory management in LSM-based storage systems. *PVLDB*, 14(3):241–254, 2021.

[81] C. Luo et al. Umzi: Unified multi-zone indexing for large-scale HTAP. In *EDBT*, pages 1–12, 2019.

[82] S. Luo et al. Rosetta: A robust space-time optimized range filter for key-value stores. In *ACM SIGMOD*, page 2071–2086, 2020.

[83] Q. Mao et al. Experimental evaluation of bounded-depth LSM merge policies. In *IEEE International Conference on Big Data (Big Data)*, pages 523–532, 2019.

[84] C. Mathieu et al. Bigtable merge compaction. *CoRR*, abs/1407.3008, 2014.

[85] F. Mei et al. LSM-tree managed storage for large-scale key-value store. In *ACM SoCC*, pages 142–156, 2017.

[86] F. Mei et al. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In *ACM SoCC*, pages 477–489, 2018.

[87] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *ACM SIGMOD*, pages 361–370, 1992.

[88] P. Muth et al. The LHAM log-structured history data access method. *VLDBJ*, 8(3):199–221, 2000.

[89] E. J. O'Neil et al. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD Rec.*, 22(2):297–306, June 1993.

[90] P. O'Neil et al. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[91] A. Papagiannis et al. An efficient memory-mapped key-value store for flash storage. In *ACM SoCC*, pages 490–502, 2018.

[92] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *CACM*, 33(6):668–676, 1990.

[93] F. Putze et al. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics*, 14:4:4.4–4:4.18, 2010.

[94] M. A. Qader et al. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In *ACM SIGMOD*, pages 551–566, 2018.

[95] P. Raju et al. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *ACM SOSP*, pages 497–514, 2017.

[96] V. Raman et al. Constant-time query processing. In *ICDE*, pages 60–69, 2008.

[97] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[98] K. Ren et al. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *PVLDB*, 10(13):2037–2048, 2017.

[99] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1):26–52, 1992.

[100] G. M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *VLDB*, pages 257–262, 1982.

[101] S. Sarkar et al. Lethe: A tunable delete-aware LSM engine. In *ACM SIGMOD*, page 893–908, New York, NY, USA, 2020.

[102] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–44, 2006.

[103] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *ACM SIGMOD*, pages 217–228, 2012.

[104] M. Seidemann and B. Seeger. ChronicleDB: A high-performance event store. In *EDBT*, pages 144 – 155, 2017.

[105] S. Seshadri et al. Willow: A user-programmable SSD. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 67–80, 2014.

[106] P. J. Shetty et al. Building workload-independent storage with VT-trees. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 17–30, 2013.

[107] V. Srinivasan and M. J. Carey. Performance of on-line index construction algorithms. In *EDBT*, pages 293–309, 1992.

[108] A. J. Storm et al. Adaptive self-tuning memory in DB2. In *VLDB*, pages 1081–1092, 2006.

[109] R. Taft et al. CockroachDB: The resilient geo-distributed SQL database. In *ACM SIGMOD*, pages 1493–1509, 2020.

[110] J. Tan et al. iBTune: Individualized buffer tuning for large-scale cloud databases. *PVLDB*, 12(10):1221–1234, 2019.

[111] W. Tan et al. Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT*, pages 700–711, 2014.

[112] Y. Tang et al. Deferred lightweight indexing for log-structured key-value stores. In *International Symposium in Cluster, Cloud, and Grid Computing (CCGrid)*, pages 11–20, 2015.

[113] D. Teng et al. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 68–79, 2017.

[114] D. Teng et al. A low-cost disk solution enabling LSM-tree to achieve high performance for mixed read/write workloads. *ACM TOS*, 14(2):15:1–15:26, 2018.

[115] R. Thonangi et al. A practical concurrent index for solid-state drives. In *ACM CIKM*, pages 1332–1341, 2012.

[116] R. Thonangi and J. Yang. On log-structured merge for solid-state drives. In *ICDE*, pages 683–694, 2017.

[117] D. N. Tran et al. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage (TOS)*, 4(1):1–25, 2008.

[118] P. Unterbrunner et al. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[119] D. Van Aken et al. Automatic database management system tuning through large-scale machine learning. In *ACM SIGMOD*, pages 1009–1024, 2017.

[120] T. Vinçon et al. NoFTL-KV: Tackling write-amplification on KV-stores with native storage management. In *EDBT*, pages 457–460, 2018.

[121] P. Wang et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *European Conference on Computer Systems (EuroSys)*, pages 16:1–16:14, 2014.

[122] S. Wang et al. Lightweight indexing of observational data in log-structured storage. *PVLDB*, 7(7):529–540, 2014.

[123] S. Wang et al. Fast and adaptive indexing of multi-dimensional observational data. *PVLDB*, 9(14):1683–1694, 2016.

[124] X. Wu et al. LSM-trie: an LSM-tree-based ultra-large key-value store for small data. In *USENIX Annual Technical Conference (ATC)*, pages 71–82, 2015.

[125] J. Yang et al. F1 lightning: HTAP as a service. *PVLDB*, 13(12):3313–3325, 2020.

[126] A. C.-C. Yao. On random 2–3 trees. *Acta Informatica*, 9(2):159–170, 1978.

[127] T. Yao et al. Building efficient key-value stores via a lightweight compaction tree. *ACM TOS*, 13(4):29:1–29:28, 2017.

[128] T. Yao et al. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In *International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[129] H. Yoon et al. Mutant: Balancing storage cost and latency in LSM-tree data stores. In *ACM SoCC*, pages 162–173, 2018.

[130] Y. Yue et al. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(4):961–973, 2017.

[131] C. Yunpeng et al. LDC: a lower-level driven compaction method to optimize SSD-oriented key-value stores. In *ICDE*, pages 722–733, 2019.

[132] J. Zhang et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *ACM SIGMOD*, pages 415–432, 2019.

[133] Z. Zhang et al. Pipelined compaction for the LSM-tree. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 777–786, 2014.

[134] M. Ziauddin et al. Dimensions based data clustering and zone maps. *PVLDB*, 10(12):1622–1633, 2017.