UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Storage Design in Log-Structured Merge (LSM) Tree Databases

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Qizhong Mao

June 2022

Acknowledgments

First, I would like to thank my advisor, Professor Vagelis Hristidis, for guiding and supporting me with his wisdom and patience in the past five years. He has always been a great advisor, for my research, and for my life. I am grateful for his precious support through my PhD study, and all the help to my family and for my job finding during the pandemic. I would also like to thank other committee members, Professor Vassilis J. Tsotras, Professor Ahmed Eldawy and Professor K. K. Ramakrishnan, for their valuable comments on my research. I would not have been here without their help.

I would like to thank all my collaborators. Professor Vassilis J. Tsotras brought lots of novel ideas and constructive comments to most of my research projects. Professor Neal E. Young provided me with many helpful suggestions to the theories and proofs in my first research project. Professor Ahmed Eldawy and Tin Vu gave me useful comments and suggestions about spatial partitioning algorithms. Steven G. Jacobs and Mohiuddin Abdul Qader laid the foundation for my first two research projects. Professor Yihan Sun and Xin Zhang inspired me with the idea for the last research project. We also had many interesting discussions with my lab mates, Ildar Absalyamov, Nhat X.T. Le, and Luxun Xu. I really appreciate their help and support during my PhD journey.

Most importantly, I would like to thank my wife, Ying. We have been friends for over twenty years, and a family for four years. We lived and worked together, enjoyed all the happiness, and overcame all the hard times. My life would be incomplete without her support, companion and love.

To my wife and parents for all the support.

To little Charlotte, you are the most precious gift of my life.

ABSTRACT OF THE DISSERTATION

Efficient Storage Design in Log-Structured Merge (LSM) Tree Databases

by

Qizhong Mao

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2022
Dr. Vagelis Hristidis, Chairperson

In this cloud era, data is being generated rapidly from billions of network users, mobile devices, social networks, sensors, and many other devices and applications. Compared to traditional relational databases which were optimized for read-heavy workloads, many modern NoSQL database systems choose log-structured merge (LSM) architectures to support high write throughput, including AsterixDB, Bigtable, Cassandra, Dynamo, HBase, LevelDB, and RocksDB. My research interests focus on the architectural design and optimization of the storage engines of such LSM systems. Specifically, my thesis targets three aspects: merge policies, spatial data, and partitioning.

First, a merge policy, also known as compaction strategy, is a critical component of an LSM system. It defines how data is organized on disk and highly affects the system's read and write performance as well as space utilization. Five state-of-the-art merge policies from existing LSM systems, including Bigtable, Constant, Exploring, Tiered, and Leveled, with two recently proposed policies, Binomial and MinLatency, are selected for comparison and evaluation of write, read and transient space amplification. We build and experimentally compare all these policies on the same platform. The experimental results show these new policies outperform the other strategies, as they offer a better trade-off between write and read amplification.

Second, most of the existing LSM systems are optimized only for single dimensional data, that is, they lack support for spatial indexes for spatial queries. To support spatial indexes, an LSM system must either index spatial data by mapping the spatial keys into single dimensional keys or provide native support for a secondary LSM R-tree index. Using an OpenStreetMap dataset and a synthetic dataset, we experimentally compare LSM R-tree indexes with four different merge policies: Concurrent, Binomial, Tiered, and Leveled (with three partitioning algorithms). We discuss our observations and recommendations with respect to the merge policy, comparator, and partitioning in Leveled policy.

Third, the incremental merge style of the Leveled policy makes it possible to break a big merge into multiple small sub-merges via partitioning. For certain workloads, such as sequential insertions, Leveled policy supports trivial-moves, where a whole partition is moved to the next level without any processing. Such features are missing from stack-based merge policies, such as Tiered, which often have many time-consuming large merges, and have no effective support for trivial moves to minimize disk I/O. We propose a novel global-range partitioning algorithm for stack-based merge policies to 1) improve the parallelism of merges to improve the overall write throughput; 2) increase opportunities for trivial-moves; and 3) enable a hybrid of stack-based and leveled merge policies.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

In this cloud era, billions of network users, mobile devices, social networks, sensors, and many other devices and applications keep generating data rapidly. Many modern NoSQL database systems choose log-structured merge (LSM) [72], including AsterixDB [9], Bigtable [36], Cassandra [7], Dynamo [25], HBase [8], LevelDB [37], and RocksDB [29]. Compared to traditional relational databases which apply in-place update index structured that are optimized for read-heavy workloads, the out-of-place update mechanism in LSM architectures makes these systems capable of supporting high write throughput (see Section 2.1 for the details of LSM architectures).

My research focuses on the architectural design and optimization of the storage engines of such LSM systems. Specifically, this thesis targets three aspects: merge policies, spatial data, and partitioning.

Merge Policy A merge policy, also known as compaction strategy in some systems, is a critical component of an LSM that defines how data is organized on disk, and highly affects the system's write, read performance and space utilization. Most research has focused on optimizing LSM systems' core user operations like insert/update/delete, read (single get) and scan (iterator) based on existing merge policies. The study of different merge policies has been missing for a long time. We select five state-of-the-art merge policies from existing LSM systems, including BIGTABLE, CONSTANT, EXPLORING, TIERED, and LEVELED, with two recently proposed policies, BINOMIAL and MINLATENCY for comparison and evaluation of write, read and transient space amplification. These policies are implemented and experimentally compared on the same platform, AsterixDB. The results show these two new policies outperform the other strategies in the compared metrics, as they offer a better trade-off between write and read amplification.

Spatial Data Support Most of the existing LSM systems are designed as key-value stores, which are mainly optimized for single dimensional data. Almost all of them, except AsterixDB, lack the support for spatial indexes for spatial queries, as their internal indexes design and merge policies cannot handle multi-dimensional data efficiently. Some systems like ScyllaDB [78] choose to use other frameworks that are dedicated for spatial-temporal data (e.g., Spark [81, 96, 97]) as complements for spatial query. But this method often requires translation between multidimensional data and single dimensional data, which generally has high overhead. To natively support spatial queries, an LSM system must either index spatial data by mapping the spatial keys into single dimensional keys or provide native support for a secondary LSM R-tree index. We experimentally compare LSM R-tree indexes with four different merge policies: CONCURRENT, BINOMIAL, TIERED, and LEVELED (with three partitioning algorithms: Size, STR and R^*-Grove) using an OpenStreetMap dataset and a synthetic dataset. We discuss our observations and recommendations with respect to the choices of merge policy, comparator, and partitioning in Leveled policy.

Partitioning in LSM LevelDB and RocksDB deploy an incremental merge style of a partitioned policy (Leveled), which makes it possible to break a big merge into multiple small sub-merges to improve the overall merge performance. These twos systems also support a so-called *trivial-move* operation to move data without rewriting it in certain workloads such as sequential insertions. Such features are missing from stack-based LSM systems and when Tiered is used in RocksDB. When a stack-based merge policy is used, there often have many time-consuming large merges, and trivial moves are not always available to minimize disk I/O even for some special workloads. We propose a novel Global-Range partitioning algorithm for stack-based merge policies to 1) improve the parallelism of merges to improve the overall write throughput; 2) increase opportunities for trivial-moves; and 3) enable a hybrid of stack-based and leveled merge policies.

The rest of this thesis is organized as follows. Chapter 2 discusses the background information of LSM-trees including two LSM architectures and merge policies. Chapter 3 compares and evaluates seven LSM merge policies. Chapter 4 studies spatial index support in LSM systems. Chapter 5 presents novel partitioning algorithms for stack-based merge policies. Finally, Chapter 6 concludes this dissertation.

Chapter 2

Background

In Section 2.1 we discuss the fundamentals of an LSM tree and how data is maintained. In Section 2.2 we discuss two LSM architectures, stack-based and leveled, and several state-of-the-art merge policies we compared and evaluated for each architecture.

2.1 LSM tree

An LSM tree [72] generally consists of two layers, one layer in memory which contains one active *MemTable* (a.k.a memory component), and one layer on disk where data is organized into one or multiple *sorted run*s [2]. Every sorted run contains records sequentially ordered by the indexed key. Depending on the LSM tree architecture (discussed in Section 2.2), a sorted run can have one or multiple immutable *SSTable*s (a.k.a disk components). SSTables are typically implemented using a tree structure, such as B^+-tree. A tree structure usually partitions records into blocks or pages as nodes (some may group multiple blocks or pages into frames as nodes). This partition is usually referred as *local* *partition* within a physical file. On the other hand, a sorted run may be a virtual file that is partitioned into multiple physical files, which is referred as *global partition.* Local partition is often bound with the data structure used in physical files, where global partition is associated with the LSM tree architecture. Therefore, in this thesis, we will primarily focus on global partition only.

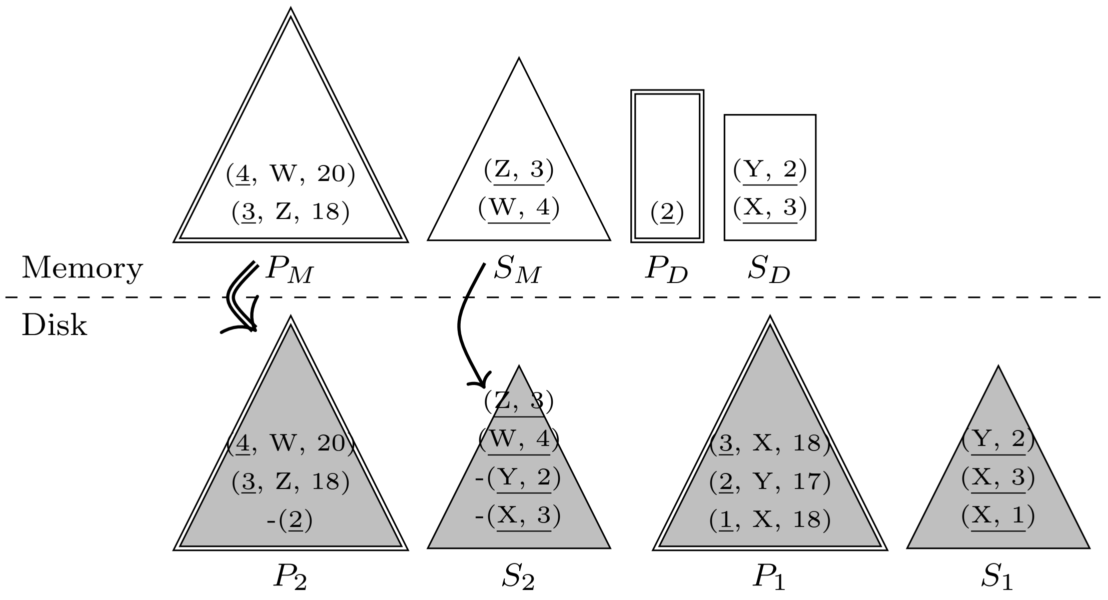


Figure 2.1: LSM flush operation. Primary (double line) and secondary (single line) MemTable are flushed independently to the top of SSTables of the corresponding index. Delete table is flushed together with the corresponding MemTable (P_M and P_D, S_M and S_D), creating anti-matter records (marked with -) in the flushed SSTable. Index keys are underlined. The primary index P's schema is (*CarID*, *OwnerID*, *ManufactureYear*). A secondary index S is built on *OwnerID*.

All records inserted or updated are batched into the MemTable. When the MemTable reaches its capacity, it is scheduled to be *flush*ed to disk, creating an SSTable, as shown in Figure 2.1. A flush operation sorts the records in the MemTable P_M, then bulk-writes the sorted records to a SSTable P_2. A *comparator*, which compares two keys, is used to sort records in the MemTable. A key is inserted to a separate in-memory delete table P_D when a record is deleted (e.g. 2). P_D is flushed together with the P_M, adding *anti-matter* (a.k.a tomestone) records (e.g. -(2)) to P_2.

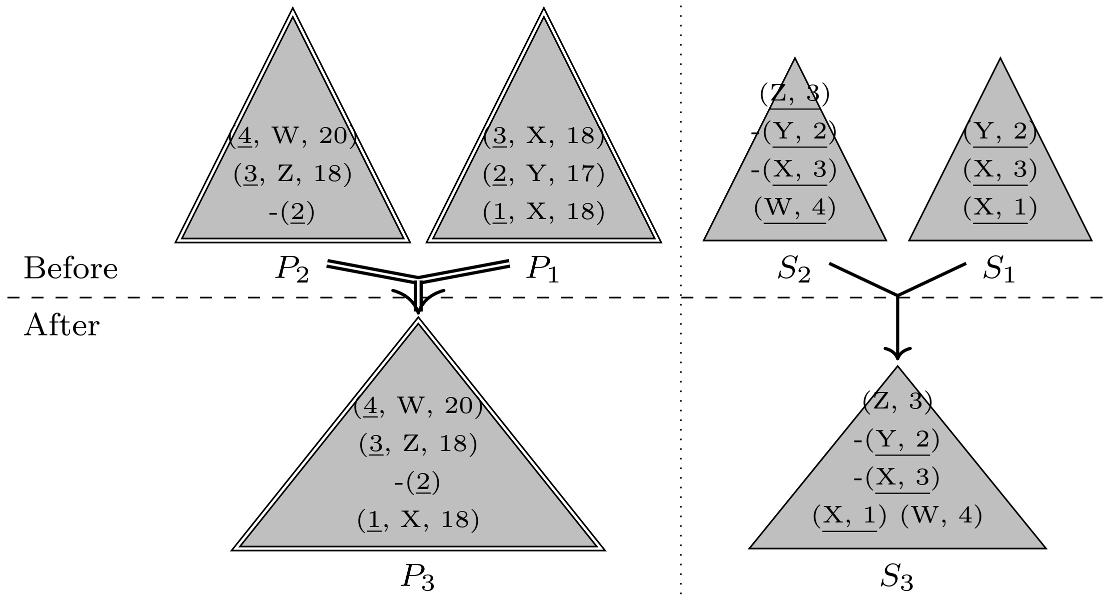


Figure 2.2: LSM merge operation. Primary index and secondary index are merged independently. Anti-matter records (marked with -) can be completely deleted if the oldest SSTable (e.g. P_1 or S_1) is involved in the merge.

Reads become slower as the number of SSTables increases. To improve the read performance, SSTables are merged based on a *merge policy* (a.k.a. compaction strategy). A merge operation scans all records from all merging tables and creates a sorted stream using a priority queue and the same comparator, then bulk-writes the unique records into new SSTable(s), as illustrated in Figure 2.2. Obsolete (old version) records are discarded during a merge, leaving only the newest version. For example, (3, X, 18) from P_1 gets removed because P_2 has a newer version (3, Z, 18). An anti-matter record will overwrite any old versions of the same record (e.g record with key 2), but will be overwritten by a new version of the same valid record (from a later insertion). Anti-matter records can be deleted when the oldest sorted run is involved in a merge. *Write amplification* is a common measurement of the write cost in an LSM system. A read query first checks the metadata of all tables, and adds tables whose key range contains the searched key to an ordered list of *operational tables*. All operational tables may contain records to answer the query, thus, the number of operational tables is usually used to compute the *read amplification*, which measures the read cost in the worst case[1].

2.2 LSM Architectures and Merge Policies

2.2.1 Stack-based LSM Tree

In a stack-based LSM tree, every single SSTable is a sorted run (thus SSTable and sorted run are interchangeable), where SSTables are ordered by the time created from flushes or merges. Stack-based merge policies generally merge only consecutive SSTables and create only one single SSTable per merge.

Most stack-based merge policies make merge decisions based on certain size ratio conditions, where every single merge involve similar sized SSTables. Such merge policies are often referred as *tiering style*. The term tiering came from the SizeTiered policy in Cassandra (described later in this section), while the term stack-based came from Bigtable [65, 68]. Tiering style merge policies are a subset of stack-based merge policies. A key difference is that SSTable sizes in tiering style merge policies are non-decreasing with respect to their time of creation, such that older SSTables are usually no smaller than any newer SSTables, while such restriction does not hold for the general stack-based merge policies, where there

[1]Some operational tables may be skipped by filters such as Bloom filter, reducing the actual read amplification.

is no relation between SSTable sizes and freshness. Certain stack-based policies choose to restrict the total number of SSTables to a constant number which limits the worst case read amplification. These policies are called *bounded-depth* policies [65].

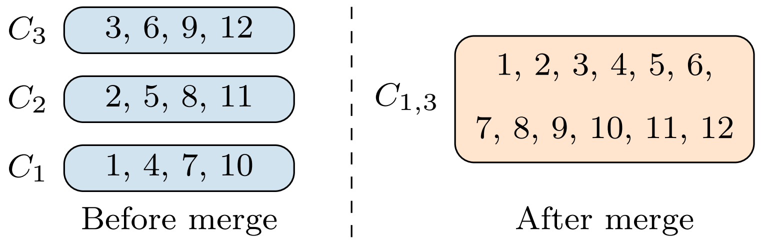
2.2.2 Leveled LSM Tree

In a Leveled LSM tree [26, 27] every level is a sorted run which is partitioned into multiple (typically) disjoint SSTables of the same size [62]. The number of SSTables in level $i \geq 2$ is B times more than the number in level $i-1$. There may also be a special level 0 which contains B_0 SSTables as a buffer, which holds flushed SSTables as multiple sorted runs. When a level reaches its capacity (B_0 or B^i), a SSTable is selected and merged with all overlapping SSTables in the next level, creating one or multiple new SSTables in the next level. While the oldest SSTable in level 0 must be selected, any SSTable in other levels can be selected. A point query only needs to check all SSTables in level 0, and at most 1 SSTable in every level $i \geq 1$. A range query may just need to check a few SSTables in each level, reducing the total size to be checked.

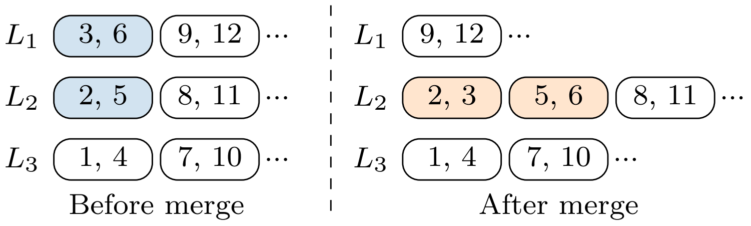
2.2.3 Stack-based vs. Leveled LSM Tree

The major differences between stack-based and leveled LSM trees in terms of merge operations are illustrated in Figure 2.3. All sorted runs (SSTables / levels) are ordered from newer to older in top-down direction, where blue and orange rounded rectangles represent input and output SSTables of a merge, respectively. Three consecutive SSTables C_1-C_3 are merged into a single SSTable $C_{1,3}$ in a stack-based LSM tree, where in a leveled LSM tree, one SSTable ($[3,6]$) from one level (L_1) is merged with the only overlapping SSTable ($[2,5]$)

in the next level (L_2), and the two output SSTables are placed in the next level ($L2$). At the same time, SSTables $[9, 12]$ and $[8, 11]$ or $[8, 11]$ and $[7, 10]$ can be merged as they do not overlapping with the current merging SSTables $[3, 6]$ and $[2, 5]$.



(a) Merge in stack-based LSM tree. Each rectangle is a SSTable storing a sorted run.



(b) Merge in leveled LSM tree. Each solid rectangle is a SSTable. Each row is a level / sorted run, containing one or multiple SSTables.

Figure 2.3: Examples of merges in stack-based and leveled LSM trees. Input and output SSTables in a merge are marked in blue and orange, respectively. Keys in a SSTable are represented by the numbers inside the rectangle.

Chapter 3

Comparison and Evaluation of State-of-the-Art LSM Merge Policies

3.1 Introduction

Many modern NoSQL systems [16, 17] use log-structured-merge (LSM) architectures [72] to achieve high write throughput. To insert a new record, a WRITE operation simply inserts the record into the memory-resident *MemTable* [17] (also called the in-memory component). UPDATE operations are implemented lazily, requiring only a single WRITE to the MemTable. DELETE operations are implemented similarly, by writing an *anti-matter* record for the key to the MemTable. Thus, each WRITE, UPDATE, or DELETE operation avoids any immediate disk access. When the MemTable reaches its allocated capacity (or for other reasons), it is flushed to disk, creating an immutable disk file called a component, or, usually, an *SSTable* (Sorted Strings Table [17]). This process continues, creating many SSTables over time.

Each READ operation searches the MemTable and SSTables to find the most recent value written for the given key. With a compact index stored in memory for each SSTable, checking whether a given SSTable contains a given key typically takes just one disk access [38, §2.5]. (For small SSTables, this access can sometimes be avoided by storing a Bloom filter for the SSTable in memory [24].) Hence, the time per READ grows with the number of SSTables. To control READ costs, the system periodically *merges* SSTables to reduce their number and to prune updated and anti-matter records. Each merge replaces some subset of the SSTables by a single new SSTable that holds their union. The merge batch-writes these items to the new SSTable on disk. The *write amplification* is the number of bytes written by all merges, divided by the number of bytes inserted by WRITE operations.

A *merge policy* (also known as a *compaction* policy) determines how merges are done. The policy must efficiently trade off total write amplification for total read cost (which increases with the average number of SSTables being checked per READ operation, known as *read amplification*). This chapter focuses on what we call *bounded depth* policies — those that guarantee a bounded number of disk accesses for each READ operation by ensuring that, at any given time, the SSTable count (the number of existing SSTables) never exceeds a given parameter k, typically 3–10, such that the read amplification is at most k. Maintaining bounded depth is important in applications that require low read latency, but bounded-depth policies are not yet well understood.

A recent theoretical work by Mathieu et al. [68] (including one of the current authors) formally defines a broad class of so-called *stack-based policies* (see Section 3.3 for the definition). This class includes policies of many popular NoSQL systems, including Bigtable [17], HBase [34, 48, 73], Accumulo [47, 73], Cassandra [52], Hypertable [45], and AsterixDB [4]. In contrast, *leveled* policies (used by LevelDB and its spin-offs [37]) split SSTables by key-space to avoid monolithic merges, so they do not fit the stack-based model. Note that all current leveled implementations yield unbounded depth, hence they are not considered here.

Mathieu et al. [68] also propose theoretical metrics for policy evaluation, and, as a proof of concept, propose new policies that, among stack-based policies, are optimal according to those metrics. Two such policies, MinLatency and Binomial (defined in Section 3.2) are bounded-depth policies which were designed to have minimum *worst-case write amplification* (subject to the depth constraint) among all stack-based policies. Mathieu et al. [68] observe that, according to the theoretical model, on some inputs *existing policies are far from optimal*, so, on some common workloads, compared to existing policies, MinLatency and Binomial can have lower write amplification.

Here we empirically compare MinLatency and Binomial to 3 representative bounded-depth merge policies from state-of-the-art NoSQL databases: a policy from AsterixDB [9], Exploring (the default policy for HBase [8]), and the default policy from Bigtable (as described by Mathieu et al. [68], which includes authors from Google); as well as the standard Tiered policy (the default policy for Cassandra [7]) and Leveled policy (the default policy for LevelDB [37]). Section 3.2 defines these policies. We implement the policies under consideration on a common platform — Apache AsterixDB [4, 9], — and evaluate them on inputs from the Yahoo! Cloud Serving Benchmark (YCSB) [20, 92]. This is the first implementation and evaluation of the policies proposed by Mathieu et al. [68] on a real NoSQL system. The empirical results validate the theoretical model. MinLatency and Binomial achieve write amplification close to the theoretical minimum, thereby outperforming the other policies by orders of magnitude on some realistic workloads. (See Section 3.4.)

Having a realistic theoretical model facilitates merge policy design both via theoretical analysis (as for MinLatency and Binomial), and because it enables rapid but faithful simulation of experiments. NoSQL systems are designed to run for months, incorporating hundreds of terabytes. Experiments can take weeks, even with appropriate adaptations. In contrast, the model allows some experiments to be faithfully simulated in minutes. (See Section 3.5.)

In summary, this chapter makes the following contributions:

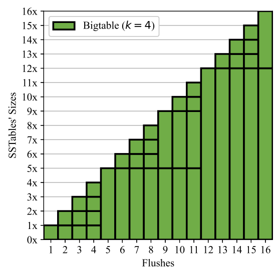
1. The implementation of several existing merge policies, including the popular Tiered and Leveled, from a variety of different NoSQL database systems, and two recently proposed merge policies, on a common, open-source platform, specifically Apache AsterixDB.

2. An experimental evaluation on write, read and transient space amplification using the Yahoo! Cloud Serving Benchmark (YCSB), confirming that the recently proposed policies can significantly outperform the state-of-the-art policies on some common workloads, such as append-only and update-heavy workloads.

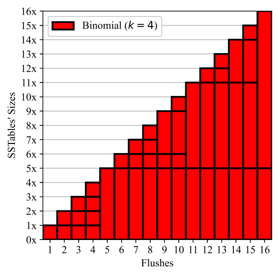
3. A study of how insertion order affects the write amplification of merge policies, especially for Leveled.

4. We have shown that Binomial and MinLatency outperform the popular Tiered and Leveled policies with a better trade-off between write amplification and average read amplification.

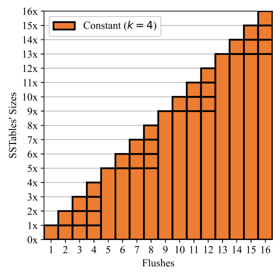
5. An empirical validation of a realistic cost model, which facilitates the design of merge policies via theoretical analysis and rapid simulation.



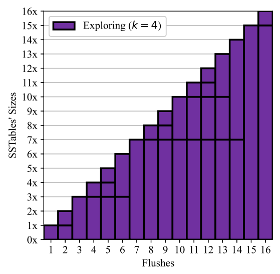
(a) Bigtable



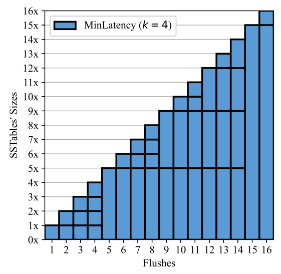
(b) Binomial



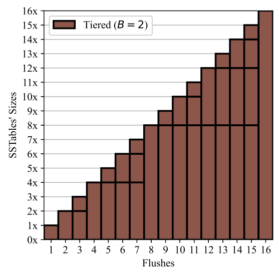
(c) Constant



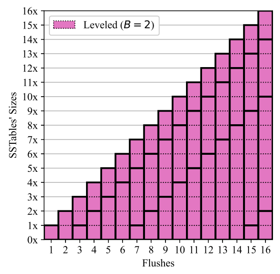
(d) Exploring



(e) MinLatency



(f) Tiered



(g) Leveled

Figure 3.1: Examples of SSTables states after 16 flushes for the 7 merge policies. SSTables are represented by rectangles (solid for the 6 stack-based policies, dotted for Leveled). Older SSTables are lower, newer SSTables are higher. Levels are represented by solid rectangles in the last plot. SSTables' sizes are with respect to the flush size.

3.2 Policies Studied

Bigtable (Google) The default for the Bigtable platform is as follows [68]. *When the MemTable is flushed, if there are fewer than k SSTables, add a single new SSTable holding the MemTable contents. Otherwise, merge the MemTable with the i most recently created SSTables, where i is the minimum such that, afterwards, the size of each SSTable exceeds the sum of the sizes of all newer SSTables.*[1] Roughly speaking, this tries to ensure that each SSTable is at most half the size of the next older SSTable. We denote this policy BIGTABLE.

Exploring (Apache HBase) EXPLORING is the default for HBase [8]. In addition to $\boldsymbol{k}$, it has configurable parameters $\boldsymbol{\lambda}$ (default 1.2), $\boldsymbol{C}$ (default 3), and $\boldsymbol{D}$ (default 10). When the MemTable (Memstore in HBase) is flushed, the policy orders the SSTables (HFiles in HBase) by time of creation, considers various contiguous subsequences of them, and merges one that is in some sense most cost-effective. Specifically: *Temporarily add the MemTable as its own (newest) SSTable, then consider every contiguous subsequence s such that*

- *s has at least C and at most D SSTables, and*
- *in s, the size of the largest SSTable is at most λ times the sum of the sizes of the other SSTables.*

In the case that there is at least one such subsequence s, merge either the longest (if there are at most k SSTables) or the one with minimum average SSTable size (otherwise). In the

[1]The implementation of this and other policies may temporarily create an SSTable holding the MemTable contents, and then merge that SSTable with the other SSTables.

remaining case, and only if there are more than k SSTables, merge a contiguous subsequence of C SSTables having minimum total size.

Constant (AsterixDB before version 0.9.4) CONSTANT is as follows. *When the MemTable is flushed, if there are fewer than k SSTables, add a single new SSTable holding the MemTable contents. Otherwise, merge the MemTable and* all *k SSTables into one.*

Tiered and Leveled TIERED policy is the default for Cassandra (termed as SizeTiered Compaction Strategy). LEVELED is the default for LevelDB [37] and RocksDB [29] (termed as Leveled Compaction). In theory, both policies have one core configurable parameter, the size ratio $\boldsymbol{B}$. In practice, TIERED may need multiple parameters (3 in Cassandra) to determine SSTables of similar sizes, LEVELED also has an extra paramter that control the number of SSTables in level 0 as an on disk buffer. The total SSTable size in one tier or level is $\boldsymbol{B}$ times larger than the previous tier or level. The differences are:

- In TIERED, every tier must have at most $\boldsymbol{B}$ SSTables, each SSTable is $\boldsymbol{B}$ larger than the SSTable size in the previous tier. In LEVELED, all SSTables are of the same size, the number of SSTables in one level is $\boldsymbol{B}$ more than the previous level.

- Any two SSTables can have overlapping key space in TIERED, while all SSTables must not have overlapping key space in the same level in LEVELED.

- TIERED only allows merging consecutive SSTables. While in LEVELED, one SSTable is picked to be merged with all SSTables in the next level that have overlapping key ranges with the picked SSTable (if any). These SSTables do not have to be consecutive.

- In Tiered, every merge involves at least two SSTables. In Leveled, only one SSTable can be merged if there is no overlapping SSTable in the next level.

- Only one SSTable is created in a merge in Tiered. In Leveled, the number of SSTables created in a merge is typically the same as the number of SSTables being merged.

- In Tiered, the new SSTable size is typically the same as the total size of the SSTables being merged. In Leveled, all input and output SSTables have the same size.

Next are the definitions of the MinLatency and Binomial policies which were proposed by Mathieu et al [68]. First, define a utility function B, as follows. Consider any binary search tree T with some nodes $\{1, 2, \ldots, n\}$ in search-tree order (each node is larger than those in its left subtree, and smaller than those in its right subtree). Given a node t in T, define its *stack (merge) depth* to be the number of ancestors smaller (larger) than t. (Hence, the depth of t in T equals its stack depth plus its merge depth.)

Fix any two positive integers k and m, and let $n = \binom{m+k}{k} - 1$. Let $\tau^*(m, k)$ be the unique n-node binary search tree on nodes $\{1, 2, \ldots, n\}$ that has maximum stack depth $k - 1$ and maximum write depth $m - 1$. For $t \in \{1, 2, \ldots, n\}$, define $B(m, k, t)$ to be the stack depth of node t in T.

Compute the function $B(m, k, t)$ via the following recurrence. Define $B(m, k, 0)$ to be zero, and for $t > 0$ use

$$B(m,k,t) = \begin{cases} B(m-1,k,t) & \text{if } t < \binom{m+k-1}{k}, \\ 1 + B\Big(m, k-1, t - \binom{m+k-1}{k}\Big) & \text{if } t \geq \binom{m+k-1}{k}. \end{cases}$$

The policies are defined as follows.

MinLatency For each $t = 1, 2, \ldots, n$, in response to the t-th flush, the action of the policy is determined by t, as follows:

Let $m' = \min\{m : \binom{m+k}{m} > t\}$ and $i = B(m', k, t)$. Order the SSTables by time of creation, and merge the i-th oldest SSTable with all newer SSTables and the flushed MemTable (leaving i SSTables).

Binomial For each $t = 1, 2, \ldots, n$, in response to the t-th flush, the action of the policy is determined by t, as follows:

Let $T_k(m) = \sum_{i=1}^{m} \binom{i+\min(i,k)-1}{i}$ and $m' = \min\{m : T_k(m) \geq t\}$.

Let $i = 1 + B(m', \min(m', k) - 1, t - T_k(m'-1) - 1)$. Order the SSTables by time of creation, and merge the i-th oldest SSTable with all newer SSTables and the flushed MemTable (leaving i SSTables).

As described in Section 3.3, these policies are designed carefully to have the minimum possible *worst-case write amplification* among all policies in the aforementioned class of stack-based policies.

Bigtable, Constant and (although it is not obvious from its specification) MinLatency are *lazy* — whenever the MemTable is flushed, if there are fewer than k SSTables, the policy leaves those SSTables unchanged, and creates a new SSTable that holds just the flushed MemTable's contents. For this reason, these policies tend to keep the number of SSTables close to k. In contrast, for moderate-length runs (4^k or fewer flushes, as discussed later), Exploring and Binomial often merge multiple SSTables even when fewer than k SSTables are already present, so may keep the average number of SSTables well below k, potentially allowing faster read operations.

Examples of all these seven merge policies for the first 16 flushes are shown in Figure 3.1. For the six stack-based policies (Figure 3.1a - 3.1f), a new SSTable is added to the top of the stack in every flush. Several SSTables are merged into one SSTable. For example in Figure 3.1a, before the 12th flush, there are 2 SSTables of size 2x and 2 SSTabls of size 1x. After the 12th flush, all the 5 SSTables are merged into one big SSTable of size 12x. The number of SSTables of Bigtable, Binomial, Constant, Exploring and MinLatency never exceeds $k = 4$. Binomial and MinLatency choose different SSTables to merge starting from the 9th flush, based on their own computations (Figure 3.1b and 3.1e). For Tiered, a merge is triggered every $B = 2$ flushes and multiple merges are triggered at the 4th, 8th, 12th and 16th flush (Figure 3.1f). For Leveled, multiple merges may be triggered at every flush starting from the 3rd flush, while only one merge is triggered at the 2nd flush (Figure 3.1g). For example, before the 12th flush, there are 2 SSTable in level 1 (top rectangle), 4 rectangles in level 2, and 5 SSTable in level 3 (bottom rectangle). After the 12th flush, a new SSTable is added to level 1, triggering a merge which selects an SSTable in level 1 and merges it to level 2. Then level 2 has 5 SSTables, and another merge is triggered which selects an SSTable in level 2 and merges it to level 3. Eventually, there are still 2 SSTables in level 1 and 4 SSTables in level 2, but 6 SSTables in level 3.

3.3 Design of MinLatency and Binomial

This section reviews definition of the class of so-called *stack-based* merge policies in [68], the *worst-case write amplification* metric, and how MinLatency and Binomial are designed to minimize that metric among all policies in that class.

3.3.1 Bounded-depth stack-based merge policies

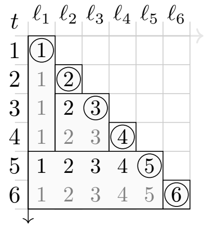
Informally, a *stack-based policy* must maintain a set of SSTables over time. The set is initially empty. At each time $t = 1, 2, \ldots, n$, the MemTable is flushed, having current size in bytes equal to a given integer $\ell_t \geq 0$. In response, the merge policy must choose some of its current SSTables, then replace those chosen SSTables by a single SSTable holding their contents and the MemTable contents. As a special case, the policy may create a new SSTable from the MemTable contents alone.

Each newly created SSTable is written to the disk, batch-writing a number of bytes equal to its size, which *by assumption* is the sum of the sizes of the SSTables it replaces, plus ℓ_t if the merge includes the flushed MemTable. (This ignores updates and deletes, but see the discussion below.)

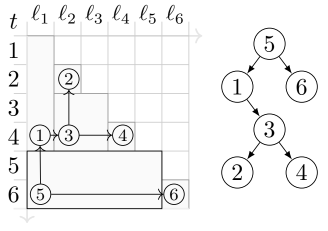
A *bounded-depth* policy (in the context of a parameter k) must keep the SSTable count at k or below. Subject to that constraint, its goal is to minimize the *write amplification*, which is defined to be the total number of bytes written in creating SSTables, divided by $\sum_{t=1}^{n} \ell_t$, the sum of the sizes of the n MemTable flushes. (Write amplification is a standard measure in LSM systems [27, 56, 57, 79].) Tiered is stack-based but not bounded-depth, while Leveled is neither stack-based nor bounded-depth.

t	SSTables at time t	bytes written	SSTable count
1	$\sigma_1 = \{\mathbf{1}\}$	ℓ_1	1
2	$\sigma_2 = \{1\}, \{\mathbf{2}\}$	ℓ_2	2
3	$\sigma_3 = \{1\}, \{\mathbf{2}, \mathbf{3}\}$	$\ell_2 + \ell_3$	2
4	$\sigma_4 = \{1\}, \{2, 3\}, \{\mathbf{4}\}$	ℓ_4	3
5	$\sigma_5 = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}$	$\ell_1 + \ell_2 + \ell_3 + \ell_4 + \ell_5$	1
6	$\sigma_6 = \{1, 2, 3, 4, 5\}, \{\mathbf{6}\}$	ℓ_6	2

(a)



(b)



(c)

Figure 3.2: **(a)** An eager, stable schedule σ ($n = 6$, $k = 3$). **(b)** A graphical representation of σ. Each shaded rectangle is an SSTable (over time). Row t is the stack at time t. **(c)** The binary-search-tree representation of σ.

For intuition, consider the example $k = 2$ and $\ell_t = 1$ uniformly for $t \in \{1, 2, \ldots, n\}$. The optimal write amplification is $\Theta(\sqrt{n})$.

Next is the precise formal definition, as illustrated in Figure 3.2(a):

Problem 1 (k-Stack-Based LSM Merge) *A problem instance is an $\ell \in \mathbb{R}_+^n$. For each $t \in \{1, \ldots, n\}$, say* flush t *has* (flush) size *ℓ_t. A solution is a sequence $\sigma = \{\sigma_1, \ldots, \sigma_n\}$, called a* schedule*, where each σ_t is a partition of $\{1, 2, \ldots, t\}$ into at most k parts, each called an* SSTable*, such that σ_t is refined by*[2] *$\sigma_{t-1} \cup \{\{t\}\}$ (if $t \geq 2$). The* size *of any SSTable F is defined to be $\ell(F) = \sum_{t \in F} \ell_t$ — the sum of the sizes of the flushes that comprise F. The goal is to minimize σ's* write amplification*, defined as $W(\sigma) = \sum_{t=1}^{n} \delta(\sigma_t, \sigma_{t-1}) / \sum_{t=1}^{n} \ell_t$, where $\delta(\sigma_t, \sigma_{t-1}) = \sum_{F \in \sigma_t \setminus \sigma_{t-1}} \ell(F)$ is the sum of the sizes of the new SSTables created during the merge at time t.*

Formally, a *(bounded-depth) stack-based merge policy* is a function P mapping each problem instance $\ell \in \mathbb{R}_+^n$ to a solution σ. In practice, the policy must be *online*, meaning that its choice of merge at time t depends only on the flush sizes $\ell_1, \ell_2, \ldots, \ell_t$ seen so far.

[2]Each part in σ_t is the union of some parts in $\sigma_{t-1} \cup \{\{t\}\}$.

Because future flush sizes are unknown, no online policy P can achieve minimum possible write amplification for *every* input ℓ. Among possible metrics for analyzing such a policy P, the focus here is on *worst-case* write amplification: the maximum, over all inputs $\ell \in \mathbb{R}_+^n$ of size n, of the write amplification that P yields on the input. Formally, this is the function $n \mapsto \max\{W(P(\ell)) : \ell \in \mathbb{R}_+^n\}$.

Updates and Deletes The formal definitions above ignore the effects of key UPDATEs and DELETEs. While it would not be hard to extend the definition to model them, for designing policies that minimize worst-case write amplification, this is unnecessary: these operations only *decrease* the write amplification for a given input and schedule, so any online policy in the restricted model above can easily be modified to achieve the same worst-case write amplification, even in the presence of UPDATEs and DELETEs.

Additional terminology Recall that a policy is *stable* if, for every input, it maintains the following invariant at all times among the current SSTables: *the* WRITE *times of all items in any given SSTable precede those of all items in every newer SSTable.* (Formally, every SSTable created is of the form $\{i, i+1, \ldots, j\}$ for some i, j.) As discussed previously, this can speed up READs. We note without proof that any unstable solution can be made stable while at most doubling the write amplification. Likewise, each uniform input has an optimal stable solution. All policies tested here are stable.

A policy is *eager* if, for every input ℓ, for every time t, the policy creates just one new SSTable (necessarily including the MemTable flushed). Every input has an optimal eager solution, and all bounded-depth policies tested here except for EXPLORING are eager.

An online policy is *static* if each σ_t is determined *solely by k and t*. In a static policy, the merge at each time t is predetermined — for example, for $t = 1$, merge just the flushed MemTable; for $t = 2$, merge the MemTable with the top SSTable, and so on — independent of the flush sizes $\ell_1, \ell_2, \ldots$ The MinLatency and Binomial policies are static. Static policies ignore the flush sizes, so it may seem counter-intuitive that static policies can achieve optimum worst-case write amplification.

3.3.2 MinLatency and Binomial

Among bounded-depth stack-based policies, MinLatency and Binomial, by design, have the minimum possible worst-case write amplification. Their design is based on the following relationship between schedules and binary search trees.

Fix any k-Stack-based LSM Merge instance $\ell = (\ell_1, \ldots, \ell_n)$. Consider any eager, stable schedule σ for ℓ. (So σ creates just one new SSTable at each time t.) Define the (rooted) *merge forest* $\mathcal{F}$ for σ as follows: for $t = 1, 2, \ldots, n$, represent the new SSTable F_t that σ creates at time t by a new node t in $\mathcal{F}$, and, for each SSTable F_s (if any) that is merged in creating F_t, make node t the parent of the node s that represents F_s.

Next, create the *binary search tree* T for σ from $\mathcal{F}$ as follows. Order the roots of $\mathcal{F}$ in decreasing order (decreasing creation-time t). For each node in $\mathcal{F}$, order its children likewise. Then let $T = T(\sigma)$ be the standard left-child, right-sibling binary tree representation of $\mathcal{F}$. That is, T and $\mathcal{F}$ have the same vertex set $\{1, 2, \ldots, n\}$, and, for each node t in T, the left child of t in T is the first (oldest) child of t in $\mathcal{F}$ (if any), while the right child of t in T is the right (next oldest) sibling of t in $\mathcal{F}$ (if any; here we consider the roots to be siblings). It turns out that (because σ is stable) the nodes of T must be in search-tree order (each node is larger than those in its left subtree and smaller than those in its right subtree). Figures 3.2(a) and 3.2(c) give an example.

What about the depth constraint on σ, and its write amplification? Recall that the *stack (merge) depth* of a node t is the number of ancestors that are smaller (larger) than t. While the details are out of scope here, the following holds:

For any eager, stable schedule σ:

1. *σ obeys the depth constraint if and only if every node in $T(\sigma)$ has stack depth at most $k-1$,*

2. *the write amplification incurred by σ on ℓ equals*

$$\frac{\sum_{t=1}^{n}(\mathsf{mergedepth}(t, T(\sigma)) + 1)\ell_t}{\sum_{t=1}^{n}\ell_t} \leq 1 + \max_{t=1}^{n} \mathsf{mergedepth}(t, T(\sigma)).$$

The mapping $\sigma \to T(\sigma)$ is invertible. Hence, *any binary search tree t with nodes $\{1, 2, \ldots, n\}$, maximum stack depth $k-1$, and maximum merge depth $m-1$ yields a bounded-depth schedule σ (such that $T(\sigma) = t$), having write amplification at most m on any input $\ell \in \mathbb{R}_+^n$.*

Rationale for MinLatency MinLatency uses this observation to produce its schedule [68]. First consider the case that $n = \binom{m+k}{k} - 1$ for some integer m. Among the binary search trees on nodes $\{1, 2, \ldots, n\}$, there is a unique tree with maximum stack depth $k-1$ and maximum merge depth $m-1$. Let $\tau^*(m, k)$ denote this tree, and let $\sigma^*(m, k)$ denote the corresponding schedule.

MinLatency is designed to output $\sigma^*(m,k)$ for any input of size n. Since $\tau^*(m,k)$ has maximum merge depth $m-1$, as discussed above, $\sigma^*(m,k)$ has write amplification at most m, which by calculation is

$$(1+O(1/k))\, k\, n^{1/k}/c_k, \tag{3.1}$$

where $c_k = (k+1)/(k!)^{1/k} \in [2,e]$. This bound extends to arbitrary n, so MinLatency's worst-case write amplification is at most (3.1).

This is optimal, in the following sense: for every $\epsilon > 0$ and large n, no stack-based policy achieves worst-case write-amplification less than $(1-\epsilon)k\, n^{1/k}/c_k$. This is shown by using the bijection described above to bound the minimum possible write amplification for *uniform* inputs.

Binomial and the small-n and large-n regimes As mentioned previously, due to the fact that MinLatency and Bigtable are lazy, they produce schedules whose average SSTable count is close to k. When n is large, any policy with near-optimal write amplification must do this. Specifically, in what we call the *large-n regime* — after the number of flushes exceeds 4^k or so — any schedule with near-optimal write amplification (e.g., for uniform ℓ) *must* have average SSTable count near k. In this regime, Binomial behaves similarly to MinLatency. Consequently, in this regime, Binomial still has minimum worst-case write amplification.

However, in what we call the *small-n regime* — until the number of flushes n reaches 4^k — it is possible to achieve near-optimal write-amplification while keeping the average SSTable count somewhat smaller. Binomial is designed to do this [68]. In the small-n

regime, it produces the schedule σ for the tree $\tau^*(m,m)$, for which the maximum stack depth and maximum merge depth are both $m \approx \log_2(n)/2$, so BINOMIAL's average SSTable count and write amplification are about $\log_2(n)/2$, which is at most k (in this regime) and can be less. Consequently, in the small-n regime, BINOMIAL can opportunistically achieve average SSTable count well below k. In this way it compares well to EXPLORING, and it behaves well even with unbounded depth ($k = \infty$).

3.4 Experimental Evaluation

3.4.1 Test Platform: AsterixDB

Apache AsterixDB [4, 9] is a full-function, open-source Big Data Management System (BDMS), which has a shared-nothing architecture, with each node in an AsterixDB cluster managing one or more storage and index partitions for its datasets based on LSM storage. Each node uses its memory for a mix of storing MemTables of active datasets, buffering of file pages as they are accessed, and other memory-intensive operations. AsterixDB represents each SSTable as a B^+-tree, where the number of keys at each internal node is roughly the configured page size divided by the key size. (Internal nodes store keys but not values.) Secondary indexing is also available using B^+-trees, R-trees, and/or inverted indexes [5]. As secondary indexing is out of the scope of this chapter, our experiments involve only primary indexes.

AsterixDB provides *data feed*s for rapid ingestion of data [39]. A *feed adapter* handles establishing the connection with a data source, as well as receiving, parsing and translating data from the data source into ADM objects [4] to be stored in AsterixDB.

Several built-in feed adapters available for retrieving data from network sockets, local file system, or from applications like Twitter and RSS.

3.4.2 Experimental Setup

The experiments were performed on a machine with an Intel i3–4330 CPU running CentOS 7 with 8 GB of RAM and two mirrored (RAID 1) 1 TB hard drives. AsterixDB was configured to use 1 node controller, so all records are stored on the same disk location. The relatively small RAM size of 8 GB limits caching, to better simulate large workloads. The MemTable capacity was configured at 4 MB. The small MemTable capacity increases the flush rate to better simulate longer runs.

The workload was generated using the Yahoo! Cloud Serving Benchmark [20, 92], with default parameters used in the load phase. The full workload consists of 80,000,000 WRITEs, each writing one record with a primary key of 5 to 23 bytes plus 10 attributes of 1000 bytes, giving a total size of about 1 kB. Each primary key is a string with a 4-byte prefix and a long integer (as a string). Insert order was set to the default *hashed*.

To achieve high ingestion rate, we implemented a YCSB database-interface layer for AsterixDB using the "socket_adapter" data feed (which retrieves data from a network socket) with an upsert data model, so that records are written without a duplicate key check to achieve a much higher throughput. Upsert in AsterixDB and Cassandra is the equivalent of standard insert in other NoSQL systems, where, if an inserted record conflicts in the primary key with an existing record, it overwrites it.

The MemTable flushes were triggered by AsterixDB when the MemTable was near capacity, so the input ℓ generated by the workload was nearly uniform, with each flush size

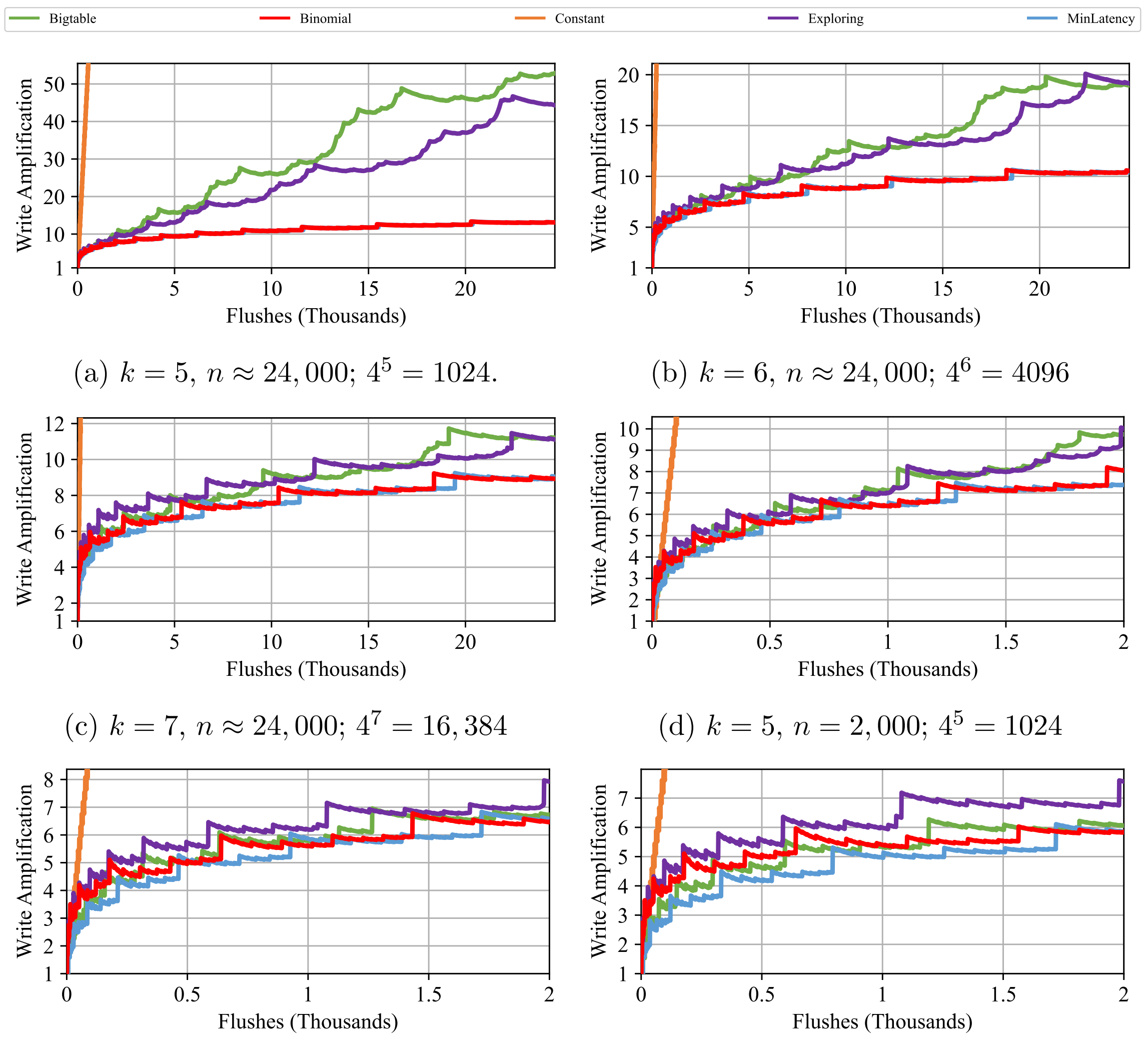
ℓ_t about 4 MB. This represents about 3,300 records per flush, so the input size n — the total number of flushes in the run — was just over $24,000$.

For each of the five bounded-depth stack-based policies tested, and for each $k \in \{3, 4, 5, 6, 7, 8, 10\}$, we executed a single run testing that policy, configured with that depth (SSTable count) limit k. For TIERED and LEVELED policies, we executed the same runs with size ratio $B \in \{4, 8, 16, 32\}$, the number of SSTables in level 0 was set to 2 in LEVELED policy. LEVELED also used a strategy that picks the SSTable which overlaps with the minimum number of SSTables in the next level for merges in order to reduce the write amplification. All other policy parameters were set to their default values (see Section 3.2). Each of the 43 runs started from an empty instance, then inserted all records of the workload into the database, generating just over 24,000 flushes for the merge policy.

For some smaller k values, some of the bounded-depth policies had significantly large write amplification and so did not finish the run. BINOMIAL and MINLATENCY finished in about 16 hours, but BIGTABLE and EXPLORING ingested less than 40% of the records after two days, so were terminated early. Similarly, CONSTANT was terminated early in *all* of its runs.

As our focus is on write amplification, which is not affected by READs, the workload contains no READs (but see Section 3.4.3).

The data for all 43 runs is tabulated in Appendix A



(a) $k = 5$, $n \approx 24,000$; $4^5 = 1024$.

(b) $k = 6$, $n \approx 24,000$; $4^6 = 4096$

(c) $k = 7$, $n \approx 24,000$; $4^7 = 16,384$

(d) $k = 5$, $n = 2,000$; $4^5 = 1024$

(e) $k = 6$, $n = 2,000$; $n \ll 4^6$

(f) $k = 7$, $n = 2,000$; $n \ll 4^7$

Figure 3.3: Write amplification vs. number of flushes over time for runs with $k \in \{5, 6, 7\}$. The top row shows $n \approx 24,000$; the bottom shows $n = 2,000$. The transition from the small-n regime to the large-n regime (if present) occurs at 4^k flushes. Complete data is in Appendix A.

3.4.3 Policy Comparison

Write Amplification

At any given time t during a run, define the *write amplification* (so far) to be the total number of bytes written to create SSTables so far divided by the number of bytes flushed so far ($\sum_{s=1}^{t} \ell_s$). This section illustrates how write amplification grows over time during the runs for the various policies. The 5 bounded-depth stack-based policies all share a common parameter k, which is the maximum number of SSTables. On the other hand, TIERED and LEVELED both share a different parameter B, which is the size ratio between tiers or levels. Because these two parameters carry different meanings, it is not meaningful to compare the write amplification of these 7 policies directly with the same value of k and B. Thus in this subsection, we compare and evaluate them into 2 groups: one group containing the 5 bounded-depth stack-based policies are compared for the same value of k, while the other group of TIERED and LEVELED are compared for the same value of B.

Bounded-depth Stack-based Policies

We focus on the runs with $k \in \{5, 6, 7\}$, which are particularly informative. The runs for each k are shown in Figures 3.3a–3.3c, each showing how the write amplification grows over the course of all $n \approx 24,000$ flushes. Because workloads with at most a few thousand flushes are likely to be important in practice, Figures 3.3d–3.3f repeat the plots, zooming in to focus on just the first $2,000$ flushes ($n = 2,000$).

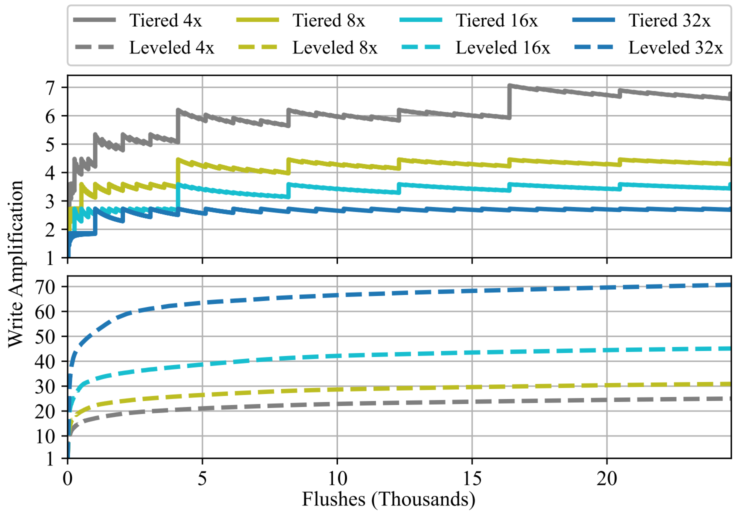
In interpreting the plots, note that the caption of each sub-figure shows the threshold 4^k. The small-n regime lasts until the number of flushes passes this threshold, whence the large-n regime begins. Note that (depending on n and k), some runs lie entirely within the small-n regime ($n \leq 4^k$), some show the transition, and in the rest (with $n \gg 4^k$) the small-n regime is too small to be seen clearly. In all cases, the results depend on the regime as follows. During the small-n regime, MINLATENCY has smallest write amplification, with BINOMIAL, BIGTABLE, and then EXPLORING close behind. As the large-n regime begins, MinLatency and Binomial become indistinguishable. Their write amplification at time t grows sub-linearly (proportionally to $t^{1/k}$), while those of Bigtable and Exploring grow linearly (proportionally to t). Although we do not have enough data for Constant, its write amplification is $O(t/k)$ as it merges all SSTables in every k flushes. These results are consistent with the analytical predictions from the theoretical model [68].

Tiered Policy and Leveled Policy

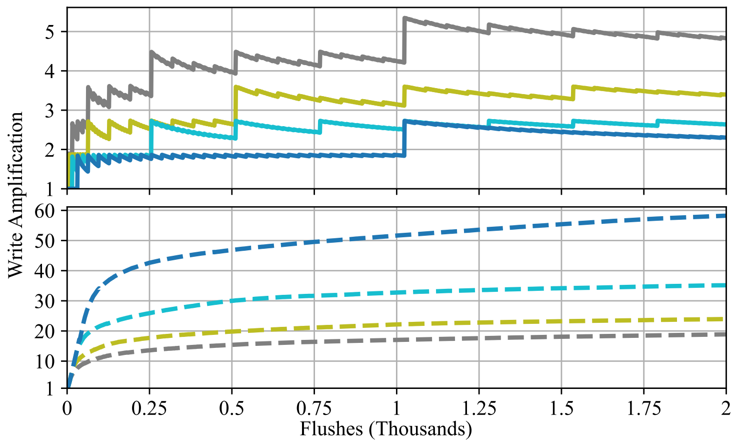
The runs for Tiered and Leveled are shown in Figure 3.4a for $n \approx 24,000$ flushes with $B \in \{4, 8, 16, 32\}$. Tiered achieved lowest write amplification than any other policy tested, while Leveled has significantly higher write amplification than all the other policies except for Constant. The write amplification is $O(\log t)$ and $O(B \log t)$ for Tiered and Leveled, respectively. From the figure, it is observable that smaller size ratio leads to higher write amplification in Tiered but lower write amplification in Leveled, which verifies the theoretical numbers. Runs with $2,000$ flushes are shown in Figure 3.4b which shows the same results. Unlike the bounded-depth policies, the small-n regime does not apply to Tiered and Leveled.

Read Amplification

Read amplification is the number of disk I/Os per read operation. In this chapter, we focus on point query only. In practice, accessing one SSTable only costs one disk I/O, assuming all metadata and all internal nodes of B^+-trees are cached. Therefore, the *worst-case* read amplification can be computed as the SSTable count for all stack-based policies,



(a) $n \approx 24,000$



(b) $n = 2,000$

Figure 3.4: With $B \in \{4, 8, 16, 32\}$, TIERED are shown as solid lines in the top sub-figures, LEVELED are shown as dashed lines in the bottom sub-figures.

or approximately the number of levels for LEVELED policy (number of SSTables in level 0 is 2 in our experiments), although techniques such as Bloom filter can skip checking most of the SSTables, making the actual read amplification be only 1.

As noted previously, MinLatency and Bigtable, being lazy, tend to keep the read amplification near its limit k. In the large-n regime, any policy that minimizes worst-case write amplification must do this. But, in the small-n regime, Binomial opportunistically achieves smaller average read amplification, as does Exploring to some extent.

Figure 3.5 has a line for each policy except Constant. The line for Constant was generated from simulation. It can be clearly seen from the figure that Constant is far from optimal, thus below we concentrate on the other policies. The curve shows the trade-off between final write amplification and average read amplification achieved by its policy: it has a point (x, y) for every run of the bounded-depth stack-based policy with $k \in \{4, 5, 6, 7, 8, 10\}$, or Tiered and Leveled with $B \in \{4, 8, 16, 32\}$ (and $n \approx 24,000$), where x is the final write amplification for the run and y is the average read amplification over the course of the run. Both the x-axis and y-axis are log-scaled.

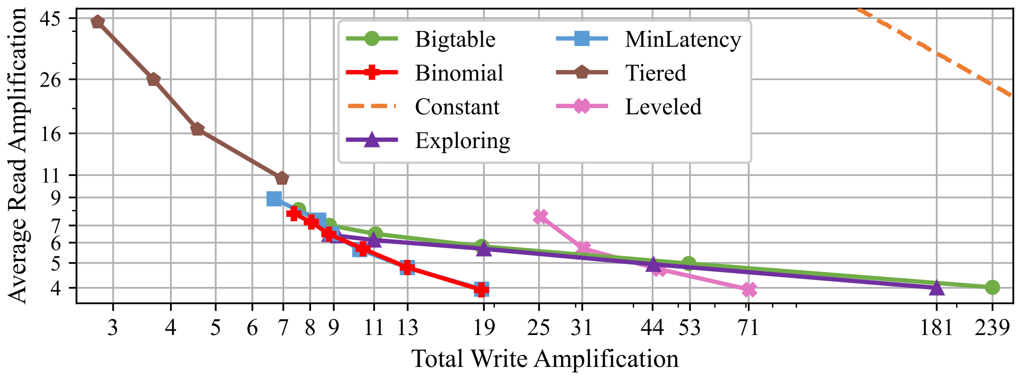


Figure 3.5: Average read amplification vs. total write amplification (x and y log-scaled).

First consider the runs with $k \in \{7, 6, 5, 4\}$. Within each curve, these correspond to the *four rightmost / lowest points* (with $k = 4$ being rightmost / lowest). These runs are dominated by the large-n regime, and each policies has average SSTable count (y coordinate) close to k. In this regime the Binomial and MinLatency policies achieve the same (near-optimal) trade-off, while the Exploring and Bigtable policies are far from the optimal frontier due to their larger write amplification.

Next consider the remaining runs, for $k \in \{10, 8\}$. On each curve, these are the two leftmost / highest points, with $k = 10$ leftmost. In the curve for Exploring, its two points are indistinguishable. These runs stay within the small-n regime. In this regime, Binomial achieves a slightly better tradeoff than the other policies. MinLatency and Bigtable give comparable tradeoffs. For Exploring, its two runs lie close to the optimal frontier, but low: increasing the SSTable limit (k) from 8 to 10 makes little difference.

The read amplification of Tiered and Leveled is inverse of their write amplification, that is $O(B \log t)$ for Tiered and $O(\log t)$ for Leveled. Tiered, – the 4 points from left to right correspond to $B \in \{32, 16, 8, 4\}$ respectively – has higher read amplification than any other policy tested but has significantly lower write amplification. Leveled, – the 4 points from left to right correspond to $B \in \{4, 8, 16, 32\}$ respectively – has comparable read amplification to the other policies except Tiered, but has much higher write amplification. Usually, a merge policy cannot achieve low write and read amplification at the same time. Most researches tried to improve the trade-off curve of Tiered and Leveled such that it can get closer to the optimal frontier [23, 24]. As shown in the figure, Binomial and MinLatency are both closer to the optimal frontier, which has better trade-off between write and read. Bigtable and Exploring are closer to the optimal frontier with a few large k values, but they have to pay very high write cost to reduce the read cost.

Transient Space Amplification

Recently various works [10, 27, 51] have discussed the importance of *space amplification*. In an LSM-tree based database system, space amplification is mostly determined by the amount of obsolete data from updates and deletions in a stable state which are yet to be garbage-collected in merges. Because our primary focus is an append-only workload without any updates or deletions, there would be no obsolete data, so the space amplification of all policies would be almost the same. Therefore, comparing space amplification among these policies is not interesting here.

On the other hand, what is more interesting is the *transient space amplification*, which measures the temporary disk space required for creating new components [30, 62] during merges. We compute transient space amplification as the maximum total size of all SSTables divided by the total data size flushed (inserted) so far. For example, a flush in Tiered or Leveled can trigger several merges in sequence, where only the largest merge will be counted. A maximum of transient space amplification of 2 can happen when a major merge involves all existing SSTables. A policy with higher transient space amplification needs larger disk space to load the same amount of data, causing lower disk space utilization. The highest transient space amplification observed in our experiments for each policy are shown in Figure 3.6, where Binomial, MinLatency, Bigtable and Exploring use $k = 4$, Tiered uses $B = 4$ and Leveled uses $B = 32$. All stack-based policies tested (including Tiered) could eventually reach a transient space amplification of 2, while Leveled has very low transient space amplification that is close to 1, and hence utilizes disk space much better. Among the five stack-based policies, Tiered offered lowest transient space amplification but highest read amplification; the transient space amplification is lower than the others for BIGTABLE but its write amplification is high. But in generally, any policy which tries to reduce the total number of SSTables to a minimum can have a high transient space amplification close to 2 due to major merges.

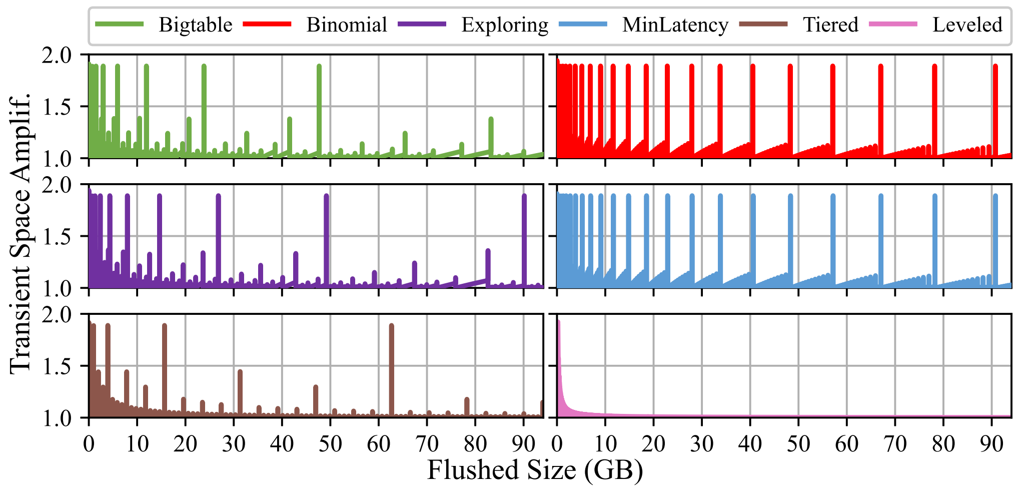
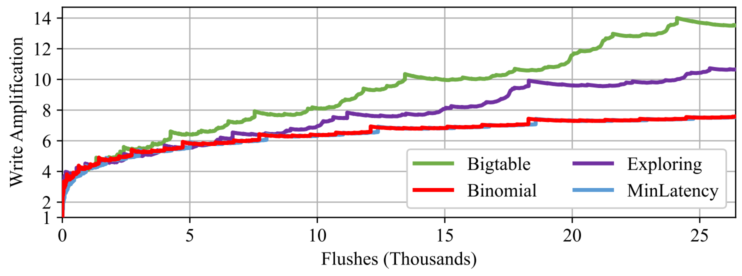


Figure 3.6: Transient space amplification of stack-based policies with $k = 4$ or $B = 4$ and LEVELED with $B = 32$.

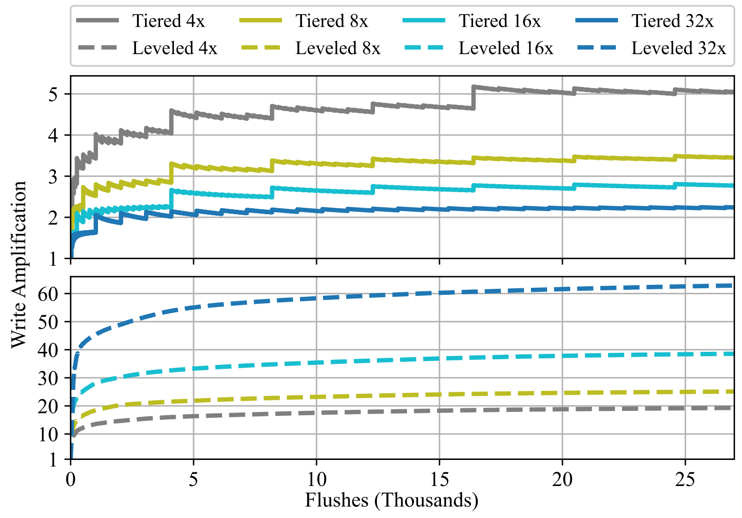
3.4.4 Updates and Deletions

UPDATE and DELETE operations insert records whose keys already occur in some SSTable. As a merge combines SSTables, if it finds multiple records with the same key, it can remove all but the latest one. Hence, for workloads with UPDATE and DELETE operations, the write amplification can be reduced. But the experimental runs described above have no UPDATE or DELETE operations. As a step towards understanding their effects, we did additional runs with $k = 6$ and $B \in \{4, 8, 16, 32\}$, with 70% of the WRITE operations replaced by UPDATEs, each to a key selected randomly from the existing keys according to a Zipf distribution with exponent $E = 0.99$, concentrated on the recently inserted items, similar to the "Latest" distribution in YCSB. The flush rate is reduced, as UPDATEs to keys currently in the MemTable are common but do not increase the number of records in the MemTable. To compensate, we increased the total number of operations by 50%, resulting in about $n \approx 26,400$ flushes.

Figure 3.7a and Figure 3.7b plot the write amplification versus flushes for the 4 runs of the bounded-depth policies and for the 8 runs of TIERED and LEVELED, respectively.



(a) Four bounded-depth policies with $k = 6$



(b) TIERED and LEVELED with $B \in \{4, 8, 16, 32\}$.

Figure 3.7: Runs with random UPDATEs ($n \approx 26,400$).

The primary effect (not seen in the plots) is a reduction in the total number of flushes, but the write amplification (even as a function of the number of flushes) is also somewhat reduced, compared to the original runs (Figure 3.3b). The relative performance of the various policies is unchanged. Experiments with other key distributions (uniform over existing keys, or Zipf concentrated on the oldest) yielded similar results that we don't report here.

Although the theoretical model mostly focuses on the append-only workload, via the experiments shown in Figure 3.7, the write amplifications are still aligned with the model's prediction even with a update-heavy workload. In practice, with more updates or deletions, the error between the theoretical and the actual write cost of Binomial and MinLatency can become more and more significant. One way to solve this problem is to periodically re-evalute the current status of SSTables, and recompute the number of flushes based on the total SSTable size. Hence Binomial and MinLatency are still good candidates even for update-heavy workloads in real-world applications.

3.4.5 Insertion Order

Unlike stack-based policies where merges are independent of SSTable contents, Leveled is highly sensitive to the insertion order, which affects the number of overlapping SSTables in every merge. For workloads with sequentially inserted keys, SSTables do not overlap with each other, and hence they are simply moved to the next level by updating only the metadata with no data copy [75]. For an append-only workload with sequential insertion order, Leveled can achieve a minimum write amplification that is close to 1, as all merges are just movements of SSTables. However, if updates or deletions were added to such workload, we found that LEVELED could have very similar write amplification as a workload with non-sequential insertion order. We re-ran the same experiments for LEVELED with updates, except that we changed the insert order from *hashed* to *ordered*, such that new keys inserted are in sequential order, while some of the inserted keys can be updated later. Results of these runs are shown in Figure 3.8, which is almost identical to Figure 3.7b. The insertion order does not impact the write amplification of the other stack-based policies by much, thus we do not report their write amplifications here.

A minor observation from these runs is, compared to the runs with *hashed* insertion order, the total number of flushes is slightly reduced, leading to a slightly lower write amplification. This is because AsterixDB implements MemTable as B^+-tree. The MemTables are usually 1/2 to 2/3 full with *hashed*, so flushes are triggered more often. As SSTables are created using bulk loading method, their B^+-tree fill factors are very high, making the flushed SSTable size smaller than the MemTable size. For a workload with sequential insertion order, both MemTables and SSTables have very high fill factors, so flushes are triggered less frequently.

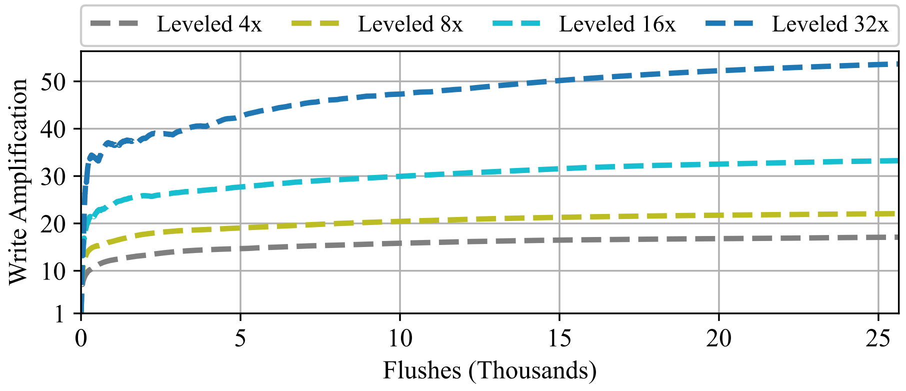


Figure 3.8: Runs of sequential insertion with random UPDATEs for LEVELED ($n \approx 26,400$)

3.5 Model Validation and Simulation

In each run, the *total time* spent in each merge operation is well-predicted by the bytes written. This is demonstrated by the plot in Figure 3.9, which has a point for *every individual merge* in every run, showing the elapsed time for that merge versus the number of bytes written by that merge.

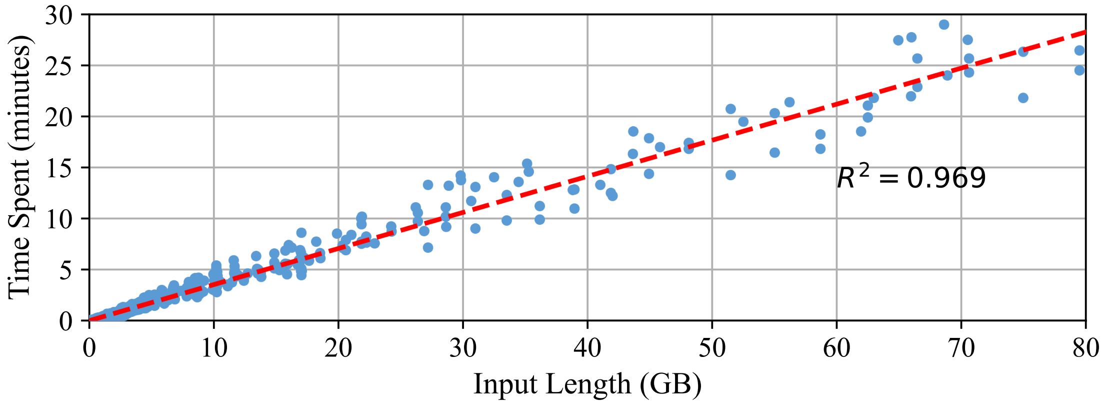
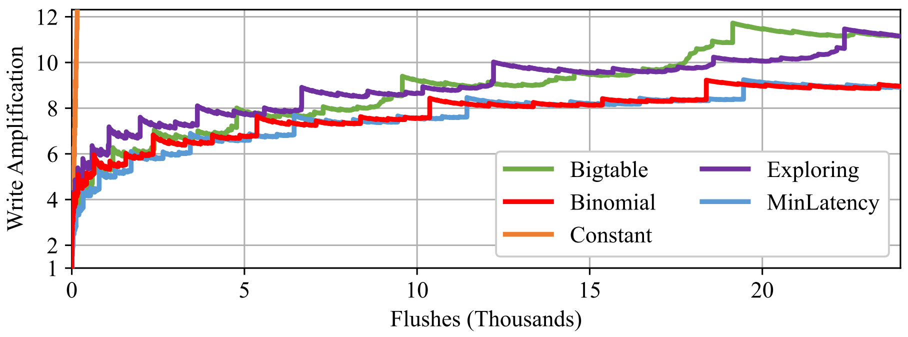


Figure 3.9: Time for each merge vs. bytes written

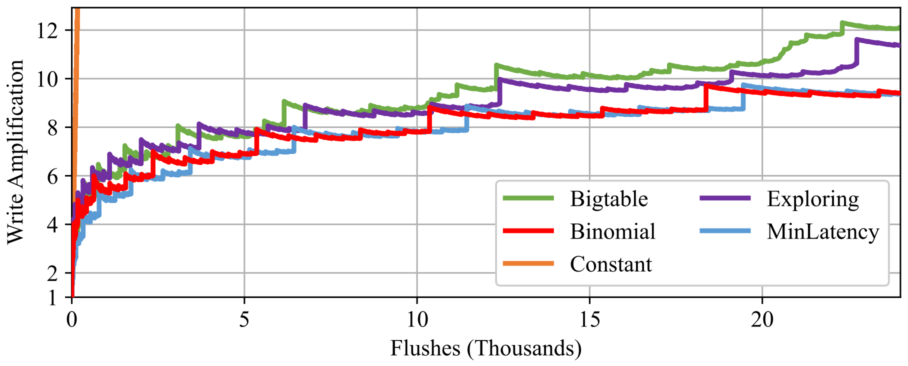
Also, observed write amplification is in turn well predicted by the theoretical model. More specifically, using the assumptions of the theoretical model, we implemented a simulator [85] that can simulate a given policy on a given input for any stack-based policies. For each of the 35 runs of the bounded-depth policies from the experiment, we simulated the run (for the given policy, k, and $n \approx 24,000$ uniform flushes).

Figure 3.10a illustrates the five runs with $k = 7$, over time, by showing the write amplifications over time as observed in the *actual* runs. Figure 3.10b shows the same for the *simulated* runs. For the static policies MinLatency and Binomial, the observed write amplification tracks the predicted write amplification closely. For Exploring and

Bigtable, the observed write amplification tracks the predicted write amplification, but not as closely. (For these policies, small perturbations in the flush sizes can affect total write amplification.)



(a) Observed write amplification over time. ($k = 7, n = 24,000$)



(b) Simulated write amplification over time. ($k = 7, n = 24,000$)

Figure 3.10: Observed and simulated write amplification over time.

Figure 3.11 shows that the simulated write amplification is a reasonable predictor of the write amplification observed empirically. That figure has two plots. The first (top) plot in that figure has a curve for each policy (except Constant), with a point for each of the six or seven runs that the policy completed, showing the observed final write amplification versus the simulated final write amplification. The two extreme points in the upper right are for Bigtable and Exploring with $k = 4$, with very high write amplification. To better show the remaining data, the second (bottom) plot expands the first, zooming in to the lower left corner (the region with $x \in [7, 39]$). For each curve, the R^2 value of the best-fit linear trendline is shown in the upper left of the first plot. (The trendlines are not shown.) The R^2 values are very close to 1, demonstrating that the simulated write amplification is a good predictor of the experimentally observed write amplification.

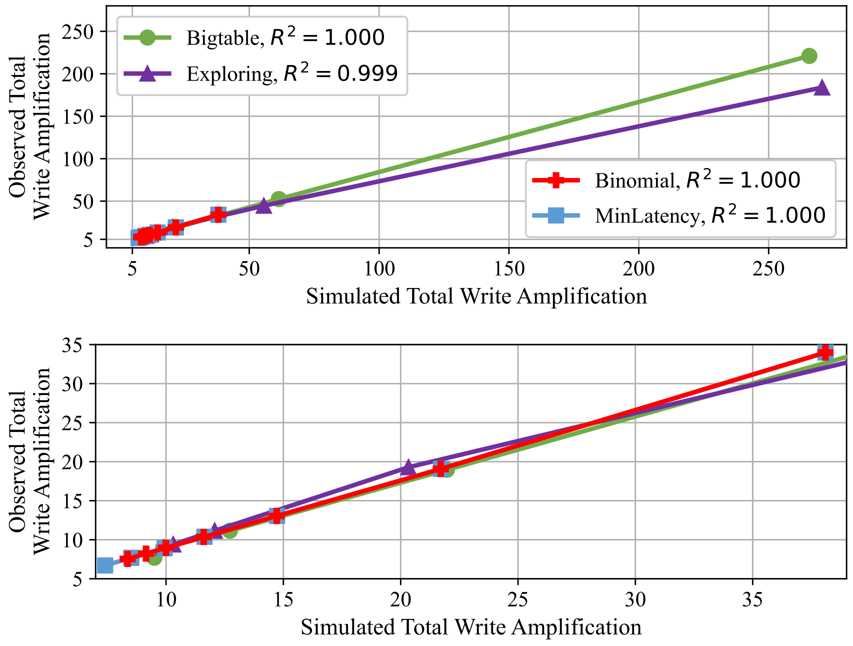
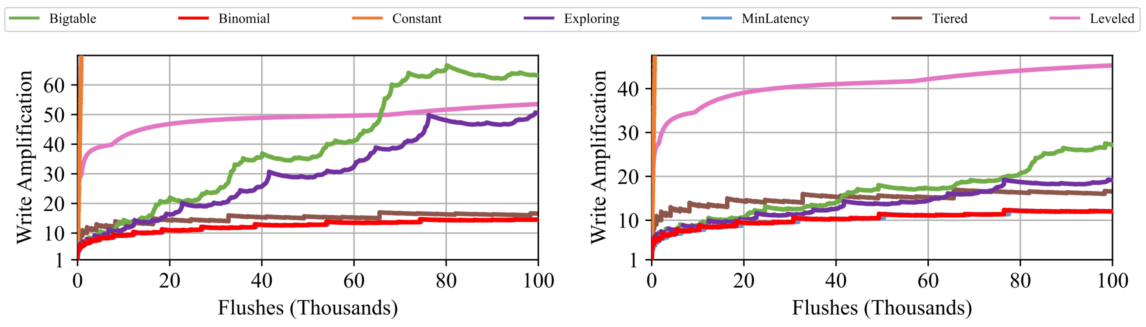
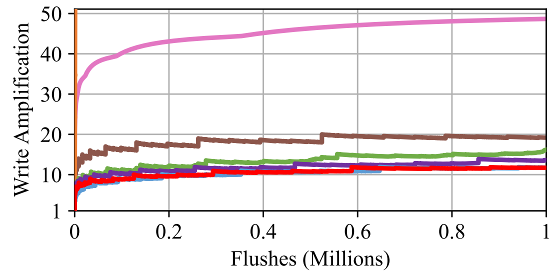


Figure 3.11: Observed versus simulated write amplification. The bottom plot zooms in to the portion with $x \in [7, 39]$.



(a) $k = 6$ (Tiered: $B = 2$, Leveled: $B = 9$), $n = 100,000$; $4^k = 4,096$.

(b) $k = 7$ (Tiered: $B = 2$, Leveled: $B = 6$), $n = 100,000$; $4^k = 16,384$.



(c) $k = 10$ (Tiered: $B = 2$, Leveled: $B = 4$), $n = 1,000,000$; $4^k \approx 10^6$.

Figure 3.12: Simulated total write amplification for 10^5 flushes (and 10^6 for $k = 10$).

Policy design via analysis and simulation A realistic theoretical model facilitates design in at least two ways. As described earlier, the model allows a precise theoretical analysis of the underlying combinatorics, as illustrated by the design of MinLatency and Binomial. It also allows accurate simulation. As noted in the introduction, LSM systems are designed to run for months, incorporating terrabytes of data. Even with appropriate adaptations, real-world experiments can take days or weeks. Replacing experiments by (much faster) simulations can moderate this bottleneck. As a proof of concept, Figure 3.12 shows *simulated* write amplification over time for Bigtable, Binomial, Constant, Exploring, and MinLatency for $k \in \{6, 7, 10\}$. As these policies' average read amplification are all less than k ($\frac{k}{2}$ for Constant), the following settings were used for Tiered and Leveled to achieve similar average read amplification (assuming one SSTable overlaps with B SSTables in the next level in every merge for Leveled):

- Figure 3.12a: $k = 6$, $n = 100,000$,$B = 2$ for Tiered and $B = 9$ for Leveled, their average read amplification are 8.15 and 6.25, respectively;
- Figure 3.12b: $k = 7$, $n = 100,000$, $B = 2$ for Tiered and $B = 6$ for Leveled, their average read amplification are 8.15 and 7.33, respectively;
- Figure 3.12c: $k = 10$, $n = 1,000,000$, $B = 2$ for Tiered and $B = 9$ for Leveled, their average read amplification are 9.88 and 10.53, respectively.

The smallest size ratio allowed for Tiered is 2, which also provides the lowest average read amplification it can achieve. In all settings, the write amplification of Leveled are 3 to 4 times larger than Binomial and MinLatency, they are only comparable with Exploring and slightly better than Bigtable in the first plot. Tiered, on the other hand, had lower write amplification than Bigtable and Exploring, but always higher than Binomial and MinLatency, and its average read amplification are higher too, except for the last plot, which is slightly lower than 10.

These simulations took only minutes to complete.

3.6 Discussion

As predicted by the theoretical model, policy behavior fell into two regimes: the *small-n* regime (until the number of flushes reached about 4^k) and the *large-n* regime (after that). MinLatency achieved the lowest write amplification, with Binomial a close second to it. Bigtable and Exploring were not far behind in the small-n regime, but in the large-n regime their write amplification was an order of magnitude higher. In short, the two newly proposed policies achieve near-optimal worst-case write amplification among all stack-based policies, outperforming policies in use in industrial systems, especially for runs with many flushes.

The trade-offs between write amplification and average read amplification were also studied in this chapter. In the large-n regime, all bounded-depth policies except (sometimes) Exploring had average read amplification near k. MinLatency and Binomial, but not Exploring or Bigtable, were near the optimal frontier. In the small-n regime, all policies were close to the optimal frontier, with Binomial and Exploring having average read amplification below k. On the other hand, although popular in the literature, the trade-offs of Tiered and Leveled are much worse than Binomial and MinLatency. These two policies might be overrated if we focus more on the cost of writes and reads.

Limitations and Future Work

Non-uniform flush sizes, updates, dynamic policies. The experiments here are limited to near-uniform inputs, where most flush sizes are about the same. Most LSM database systems, including AsterixDB, Cassandra, HBase, LevelDB and RocksDB, use uniform flush size. Some of them support checkpointing, which flushes the MemTable at some timeout interval, or when the commit log is full, potentially creating smaller SSTable before the MemTable is full. Although uniform or near-uniform flush is more common in the literature, workloads with variable flush sizes are of interest. Variable flush size may be used to coordinate multiple datasets sharing the same memory budget, or balance the write buffer and the read buffer cache for dynamic workloads. For example, a recent work [61] described an architecture which provides adaptive memory management to minimize overall write costs. For moderately variable flush sizes, we expect the write amplification of most policies (and certainly of MinLatency and Binomial) to be similar to that for uniform inputs. Regardless of variation, the write amplification incurred by MinLatency and Binomial is guaranteed to be no worse than it is for uniform inputs.

Most of the experiments here are limited to append-only workloads. A few preliminary results here suggest that a moderate to high rate of updates and deletes mainly reduce flush rate, and slightly reduce write amplification. At a minimum, updates and deletes are guaranteed not to increase the write amplification incurred by MinLatency and Binomial. But inputs with updates, deletes, and non-uniform flush sizes can have optimal write amplification substantially below the worst case of $\Theta(n^{1/k})$. In this case, dynamic policies such as Bigtable, Exploring, and new policies which are designed using the theoretical framework of *competitive-analysis* (as in [68]), may, in principle, outperform static policies such as Binomial and MinLatency. Future work will explore how significant this effect may be in practice. On the other hand, all the six evaluated stack-based policies are not very sensitive to workloads with updates or deletes, their relative ranking of write and read cost almost remain the same. The exception is Leveled, which is very sensitive to updates or deletes if the insertion order is nearly sequential. With a very low rate of updates or deletes, write amplification of Leveled increased significantly.

Compression Many databases support data compression to reduce the data size on disk, at a cost of higher CPU usage to retrieve data with decompression. In general, a system with stronger compression has lower write amplification and space amplification because of smaller data size [27]. Moreover, compression makes flushed SSTable size smaller than the MemTable size which can potentially affect the performance of BINOMIAL and MINLATENCY.

Read costs The experimental design here focuses on minimizing write amplification, relying on the bounded-depth constraint to control read costs such as *read amplification* — the average number of disk accesses required per read for point queries. Most LSM systems (other than Bigtable) offer merge policies that are not depth-bounded, instead allowing the SSTable count to grow, say, logarithmically with the number of flushes. A natural objective would be to minimize a linear combination of the read and write amplification — this could control the stack depth without manual configuration. (This is similar to BINOMIAL's behaviour in the small-n regime, where it minimizes the worst-case maximum of the read and write amplification, achieving a reasonable balance.) For read costs, a more nuanced accounting is desirable: It would be useful to take into account the effects of Bloom filters, and dynamic policies that respond to varying *rates* of READs are also of interest. Moreover, read amplification of point queries or range queries, query response time and throughput, sequential versus random access to SSTables can be of interest as well.

Secondary indexes The existence of secondary indexes impacts merging. For example, AsterixDB (with default settings) maintains a one-to-one correspondence between the SSTables for the primary indexes and the SSTables for the secondary indexes. Ideally, merge policies should take secondary indexes into account.

3.7 Related Work

Historically, the main data structure used for on-disk key-value storage is the B^+-tree. Nonetheless, LSM architectures are becoming common in industrial settings. This is partly because they offer substantially better performance for write-heavy workloads [44]. Further, for many workloads, reads are highly cacheable, making the *effective* workload write-heavy. In these cases, LSM architectures substantially outperform B^+-trees.

In 2006 Google released Bigtable [17, 36], now the primary data store for many Google applications. Its default merge policy is a bounded-depth stack-based policy. We study it here. Spanner [21], Google's Bigtable replacement, likely uses a stack-based policy, though details are not public.

Apache HBase [8, 34, 48] was introduced around 2006, modeled on Bigtable, and used by Facebook 2010–2018. Its default merge policy is Exploring, the precursor of which was a variant of Bigtable called RatioBased. Both policies are configurable as bounded-depth policies. Here we report results only for Exploring, as it consistently outperformed RatioBased.

Apache Cassandra [7, 52] was released by Facebook in 2008. Its first main merge policy, SizeTiered, is a stack-based policy that orders the SSTables by size, groups similar-sized SSTables, and then merges a group that has sufficiently many SSTables. SizeTiered is not *stable* — that is, it does not maintain the following property at all times: *the* write *times of all items in any given SSTable precede those of all items in every newer SSTable.* With a stable policy, a READ can scan the recently created SSTables first, stopping with the first SSTable that contains the key. Unstable policies lack this advantage: a READ operation must check *every* SSTable. Apache Accumulo [47] which was created in 2008 by the NSA, uses a similar stack-based policy. We don't test these policies here, as our test platform supports only stable policies, and we believe they behave similarly to BIGTABLE or EXPLORING.

Previous to this work, our test platform — Apache AsterixDB — provided just one bounded-depth policy (CONSTANT), which suffered from high write amplification [5]. AsterixDB has removed support for CONSTANT, and, based on the preliminary results provided here, added support for BINOMIAL. Our recent work [66] shows that BINOMIAL can provide superior write and read performance for LSM secondary spatial indexes, too.

Leveled policies LevelDB [26, 37] was released in 2011 by Google. Its merge policy, unlike the policies mentioned above, does not fit the stack-based model. For our purposes, the policy can be viewed as a modified stack-based policy where each SSTable is split (by partitioning the key space into disjoint intervals) into multiple smaller SSTables that are collectively called a *level* (or *sorted run*). Each READ operation needs to check only one SSTable per level — the one whose key interval contains the given key. Using many smaller tables allows smaller, "rolling" merges, avoiding the occasional monolithic merges required by stack-based policies.

In 2011, Apache Cassandra added support for a leveled policy adapted from LevelDB. (Cassandra also offers merge policies specifically designed for time-series workloads.)

In 2012, Facebook released a LevelDB fork called RocksDB [27, 29]. RocksDB offers several policies: the standard TIERED and LEVELED, LEVELED-N which allows multiple sorted runs per level, a hybrid of TIERED+LEVELED, and FIFO which aims for cache-like data [30].

Mixed of tiered and leveled policies In the literature, TIERED provides very low write amplification but very high read amplification. On the other hand, LEVELED provides good read amplification at the cost of high write amplification. It is natural to combine these 2 policies together to achieve a more balanced trade-off between write and read amplification. In RocksDB [27, 29], there are 2 policies of such mix of TIERED and LEVELED policies. The Leveled-N policy allows N sorted run in a single level instead of 1 sorted run per level. Similar idea was also described in [23, 24]. The other policy is called tiered+leveled, which uses TIERED for the smaller levels and LEVELED for the larger levels. This policy allows transition from TIERED to LEVELED at a certain level. SlimDB [76] is one example of this policy. It is an interesting research direction to evaluate and compare their trade-offs between write and average read amplification with BINOMIAL and MINLATENCY.

None of the leveled or mixed policies are stack-based or bounded-depth policies.

Other merge-policy models and optimizations Independently of Mathieu et al. [68], Lim et al. [56] propose a similar theoretical model for write amplification and point out its utility for simulation. The model includes a statistical estimate of the effects of for UPDATEs and DELETEs. For leveled policies, Lim et al. use their model to propose tuning various policy parameters — such as the size of each level — to optimize performance. Dayan et al. [23, 24] propose further optimizations of SizeTiered and leveled policies by tuning aspects such as the Bloom filters' false positive rate (vs. size) according to SSTable size, the per-level merge frequency, and the memory allocation between buffers and Bloom filters.

Multi-threaded merges (exploiting SSD parallelism) are studied in [18, 27, 57, 88]. Cache optimization in leveled merges is studied in [84]. Offloading merges to another server is studied in [1].

Some of the methods above optimize read performance; those complement the optimization of write amplification considered here. None of the above works consider bounded-depth policies.

This chapter focuses primarily on write amplification (and to some extent read amplification). Other aspects of LSM performance, such as I/O throughput, can also be affected by merge policies but are not discussed here. For a more detailed discussion of LSM architectures, including compaction policies, see [62].

3.8 Conclusions

This chapter compares several bounded-depth LSM merge policies, including representative policies from industrial NoSQL databases and two new ones based on recent theoretical modeling, as well as the standard Tiered policy and Leveled policy, on a common platform (AsterixDB) using Yahoo! cloud serving benchmark. The results have validated the proposed theoretical model and show that, compared to existing policies, the newly proposed policies can have substantially lower write amplification. Tiered and Leveled, while popular in the literature, generally underperform because of their worse trade-off between writes and reads. The theoretical model is realistic, and can be used, via both analysis and simulation, for the effective design and analysis of merge policies. For example, we shared our experimental findings with the developers of Apache AsterixDB [9], and BINOMIAL, designed via the theoretical model, has now been added as an LSM merging policy option to AsterixDB.

Chapter 4

Comparison of LSM Indexing Techniques for Storing Spatial Data

4.1 Introduction

Due to the increasing need of (mobile) applications such as navigation systems, location-based review systems, and geo-tagged social media, the volume and ingestion rate of spatial data are increasing rapidly. Database systems have been moving to log-structured merge (LSM) tree [72] storage architectures to facilitate high write throughput. Such systems include Apache AsterixDB [4], Cassandra [52], and HBase [34], Google Bigtable [17], and LevelDB [26], Facebook RocksDB [27], and ScyllaDB [78]. LSM systems provide superior write performance than most relational databases. However, most of these systems primarily focus on key-value store, and do not have native support of spatial queries, which often rely on spatial indexes.

In most applications, a spatial index cannot live alone and must be created as a secondary index that is dependent on a primary index to query any non-spatial attributes. Most LSM systems do not have the direct support of the general secondary index, not to mention the support of spatial index. In AsterixDB, LSM-fication is a generic framework to convert a class of indexes to LSM secondary indexes [5]. Using this framework, we have two options to index spatial data. The first option is a B^+-tree-based solution that indexes single-dimensional data projected from multidimensional spatial data through linearization. Note that this option also applies to systems (e.g, LevelDB and RocksDB) that use Sorted-String-Table (SSTable) and binary search methods to support range queries. The second option is a native spatial index, for example, R-tree, as a local index. To the best of our knowledge, AsterixDB is the only LSM storage engine with native support of LSM R-tree index; all other LSM-based systems only support B^+-tree index at most. Based on the results from [49, 50], the R-tree-based solution is the general preferable option for LSM spatial index in most scenarios, hence in this chapter, we focus on LSM R-tree indexes only.

In addition to the organization of the local index discussed above, which determines how data is organized in a single LSM component (file), another key design choice for spatial LSM indexes is the merge policy, which determines when and how components are merged. The two main merge paradigms we consider are stack-based and leveled. In stack-based policies, components are organized as a stack, where the most recent components are higher in the stack. LEVELED policies use (almost) fixed-size components, with newer components on higher levels; lower levels have more components per level. Stack-based LSM tree usually has better write performance and good read performance. Leveled LSM tree is the most popular paradigm in the industry with very good read performance, but higher write amplification in general [67].

The typical query for spatial indexing is a region query, where the region is typically expressed as a Minimum Bounding Rectangle (*MBR*). For each component, we maintain its MBR, so it is easy to filter components based on the query MBR. This filtering is generally not effective in stack-based policies, as most components have very large MBRs, comparable to the whole space in many applications. On the other hand, this filtering can be more effective for LEVELED policy, because the components on the same level are mostly disjoint in key ranges. In the case of R-tree indexing, this means that the components at the same level have non-overlapping MBRs, or possibly limited overall, depending on the partitioning algorithm employed.

To achieve minimal spatial overlap in LEVELED policies, spatial partitioning algorithms, specifically Sort-Tile-Recursive (STR) [54] and R^*-Grove [86], are employed. There are several subtle implementation decisions that significantly affect the merge performance. We found that a critical one is the choice of *comparator*, which compares two spatial records, because different comparator performs differently in high and low selectivity queries; certain combinations of comparator and partitioning algorithm in LEVELED policy can effectively create disk components of disjoint MBRs, which significantly improves filtering efficiency.

A key contribution of the chapter is that we implemented several LSM spatial indexing algorithms on a common database system, AsterixDB, and compared them for write and read performance using two spatial workloads. A key conclusion is that stack-based policies generally perform better with low write and read cost. Although LEVELED policy had very high write amplification, certain configurations could achieve comparable write throughput to stack-based policies. Its read performance was also very competitive in low selectivity queries.

In summary, this chapter makes the following contributions:

1. We study how an LSM architecture can be extended to support secondary spatial indexes (Section 4.3.3). We consider several design decisions and architectures.

2. We examine a number of optimized partitioning algorithms for Leveled LSM R-tree index, which minimize the overlap among MBRs while also minimizing the I/O cost (Section 4.3.4).

3. We implement all compared LSM spatial indexing policies on AsterixDB. Source code is publicly available at [64].

4. We experimentally compared all LSM spatial indexing algorithms using a real-world dataset and a synthetic dataset (Section 4.4).

5. We discuss our observations and recommendations, which challenge the current popularity of LEVELED policies (Section 4.5).

4.2 Background

4.2.1 Policies Studied

In this chapter, the following three stack-based merge policies are selected for evaluation:

- Binomial policy was originally proposed by Mathieu et al. [68], then formally defined and evaluated in [65, 67]. The name Binomial came from the fact that it uses a Binary Search Tree to make merge decisions. It is a bounded-depth policy that maintains an optimal write cost with an upper bound of worst case read amplification by only one parameter k, which restricts the maximum number of disk components. Compared to the other online merge policies, whose merge schedules are based on heuristic information such as component sizes, Binomial policy is an offline policy whose merge schedule is pre-determined only on the number of flushes. It was originally designed for append-only workload, but can be adjusted for workloads with updates or deletions as well.

- Tiered (a.k.a SizeTiered) policy is the default policy in Cassandra [15], and had been adopted as Universal Compaction [31] in RocksDB. It groups disk components into tiers. Every tier has B disk components. Whenever a tier has B disk components, all the B disk components are merged into a new component of B times larger into the top of the next tier. B is also called size ratio or fanout factor. The implementation of Tiered policy varies in different systems. For example, besides selecting similar sized components to merge, the Universal compaction in RocksDB can also select components that have more overlapping keys to reduce space amplification, or simply merge several components to enforce the total number of components to the number specified by *level0_file_num_compaction_trigger* [31], if the other two options cannot be performed. In this chapter, we implemented the TIERED policy similar to Cassandra, which only selects components based on size ratio, thus ignoring components' contents.

- CONCURRENT policy was recently added to AsterixDB to replace Prefix policy as its new default policy [58]. Unlike Prefix policy, which tends to merge similar sized components and excludes components whose sizes are larger than a user defined threshold, CONCURRENT policy is bounded-depth by a parameter k. The disk components to be merged are determined by a minimum length C, a maximum length D and a size ratio λ. Starting from the newest disk component, the policy considers any longest sequence with disk components $\{D_i, D_{i-1}, \dots, D_1\}$ where $C + 1 \leq i \leq D$, and merges them into a single disk component if $|D_i| \leq \lambda \sum_{j=1}^{i-1} |D_j|$, where $|D_j|$ is the size of disk component D_j.

4.2.2 Comparing Different Merge Policies

To better illustrate the difference among the four compared merge policies, we list their sorted runs sizes after some number of flushes in Table 4.1. Component sizes in TIERED and CONCURRENT are always non-decreasing. For BINOMIAL, it is possible that some newer sorted runs are larger. For example, after 40 flushes, the third sorted run has size 20 while the fourth sorted run has size 15. Also the number of sorted runs in BINOMIAL never exceeds $k = 4$.

Merge Policy	BINOMIAL ($k = 4$)	TIERED ($B = 4$)	CONCURRENT (default)	LEVELED ($B_0 = 2, B = 4$)
20 Flushes	1, 4, 15	4, 16	3, 17	1, 1, 4, 14
40 Flushes	2, 3, 20, 15	4, 4, 16, 16	1, 1, 3, 35	1, 1, 4, 16, 18
60 Flushes	10, 50	4, 4, 4, 16, 16, 16	1, 59	1, 1, 4, 16, 38
80 Flushes	10, 20, 50	16, 64	3, 77	1, 1, 4, 16, 58
100 Flushes	15, 35, 50	4, 16, 16, 64	1, 1, 3, 95	1, 1, 4, 16, 64, 14
120 Flushes	3, 10, 106	4, 4, 16, 16, 16, 64	1, 119	1, 1, 4, 16, 64, 34

Table 4.1: Sorted run sizes of the four compared merge policies, where newer sorted runs are on the left. Each number is the size of a sorted run with respect to the MemTable size. TIERED, CONCURRENT and LEVELED always have sorted runs in non-decreasing order. The first two sorted runs in LEVELED policy are two disk components in level 0, the other numbers are the number of disk components of size 1 in the corresponding levels. Default parameters for CONCURRENT: $k = 30, C = 3, D = 10, \lambda = 1.2$.

4.3 LSM Secondary Spatial Index

In this section, we first cover how LSM secondary indexes are maintained (Section 4.3.1), which affects the trade-off between write and read performance. We then discuss two approaches to index the spatial data, which are special type of secondary data, on LSM systems: a B^+-tree-based solution is discussed in Section 4.3.2, an R-tree-based solution is discussed in Section 4.3.3. We discuss how different merge policies affect the spatial index performance and present a partitioning algorithm for the LEVELED policy.

4.3.1 LSM Secondary Index

Before talking about the spatial index, we first explain how LSM secondary indexes are constructed and maintained. An LSM secondary index has almost identical architecture to the primary index, except it is sorted by a composite key $\langle SK, PK \rangle$, where SK is the secondary key, and PK is the primary key. Records are first ordered by SK then by PK in disk components. When a record gets updated or deleted in the primary index, the current composite key in a secondary index may become invalid as SK is no longer valid for PK. During record insertion, an *eager* strategy uses the PK ($\underline{2}$ and $\underline{3}$ in Figure 2.1) to find the old value of SK (Y for $\underline{2}$ and X for $\underline{3}$), and then inserts an anti-matter record with the old SK and PK (two entries in S_D) to all secondary indexes, so any read on a secondary index will only return valid records thus the primary index does not need to be checked. A *lazy* strategy does not update secondary indexes during records insertion but needs an extra step, querying the primary index, to verify the returned records. Writes are usually slower in the eager strategy due to the checking on all secondary indexes but reads can be faster. On the other hand, the lazy strategy provides faster writes, but reads are slower due to the extra validation in the query time. Detailed discussion about these secondary indexing strategies can be found in [5, 59, 74].

An LSM secondary index can have its own memory component budget, and flushes and merges are triggered independently of the primary index. Or it can share a global budget with the primary index. In this design, the primary index and all secondary indexes are always flushed together, but they may use different merge policies. Merges may not be triggered at the same time, although certain merge policies (CorrelatedPrefix policy in AsterixDB) can enforce the merges for all indexes at the same time. Read queries on an LSM secondary index are very similar to the merge operation.

4.3.2 Spatial LSM Index based on B^+-tree

Most of the works on LSM tree are optimized for single-dimensional data. Unlike single-dimensional data, there is usually no clear definition of how to order multidimensional spatial data. The most common approach is to project multidimensional data to single-dimensional data to be indexed by a B^+-tree. The projection is made through *linearization*. One of the most common linearization methods is space-filling curve. The two most well-known space-filling curves are Z-order curve and Hilbert curve. The BUILDINDEXES function of Algorithm 4.1 presents the pseudocode of building a spatial index on B^+-tree via space-filling curve. Both GeoMesa [43] and DataStax Cassandra [14] support this type of spatial index using GeoHash [43, 71], which is based on Z-order curve. More details are discussed in Section 4.6.

A space-filling curve partitions space into cells of the same size and uses fixed-length bit strings (usually 32/64-bit numbers) to represent each cell. A toy example of a Hilbert curve with 4 bits is shown in Figure 4.1. For point-type data, the value of the cell in which a point resides will be saved as the secondary key SK, and a B^+-tree is built on these cell values. Spatial queries may be handled in two ways to obtain the cells to be scanned. The first method is to find the cell values for all corners of the query MBR (the light purple region) and scans all cells between the smallest cell (2) and the largest cell (13). This method utilizes sequential disk I/O but may waste lots of resources checking records not in the contained cells (e.g., $3 - 6$ and $9 - 12$). It is generally preferred when the difference between the two values is small. Another method is to identify the exact cells in

Algorithm 4.1 Spatial index based on B^{+}-tree - Part 1

```
1: function SFCPoint($\vec{a}$)    ▷ The space-filling curve (SFC) value of point $\vec{a}$
2: function BuildIndexes
3:     while not end do
4:         $pk$ = NextRecord.GetPrimaryKey()    ▷ Primary key
5:         $\vec{v}$ = NextRecord.GetSpatialKey()    ▷ Secondary key
6:         $o$ = NextRecord.GetOthers()    ▷ Any other attributes
7:         WritePrimaryIndex($pk, \vec{v}, o$)
8:         WriteBTreeIndex(SFCPoint($\vec{v}$), $pk$)
9:     end while
10: end function
11: function SFCRect($\vec{a}, \vec{b}$)    ▷ All SFC values of a rectangle as $\vec{a}$ to $\vec{b}$
12: function PrimaryKeys($v$)    ▷ All primary keys whose spatial key's SFC value is $v$
13: function SpatialKey($pk$)    ▷ The spatial key (point) of a primary key $pk$
```

which the query MBR covers (2, 7, 8, and 13), then scans every covered cell. This method minimizes the disk I/O, but more random I/Os are involved. It can be used if the minimum and maximum cell values are far apart or very few cells are covered. Both methods get the records whose SK fall into the query cells, but every record must be further checked using its spatial attribute. As shown in the function SpatialSearch of Algorithm 4.1, the secondary spatial index is first searched to get all the primary keys whose spatial keys match any of the space-filling curve values from the searching MBR. This process may be implemented as a range query or multiple point queries. Next, the spatial attribute of each

Algorithm 4.2 Spatial index based on B^+-tree - Part 2

```
14: $\vec{a}, \vec{b}$    ▷ Lower (left) and upper (right) points of the searching rectangle
15: function SpatialSearch($\vec{a}, \vec{b}$)
16:     $S = \{\ \}$    ▷ (Hash) set for unique primary keys
17:     for $v_c \in$ SFCRect($\vec{a}, \vec{b}$) do
18:         for $pk \in$ PrimaryKeys($v_c$) do    ▷ $B^+$-tree search
19:             $S$.Add($pk$)
20:         end for
21:     end for
22:     $R = [\ ]$    ▷ List of matching records
23:     for $pk \in S$ do
24:         $\vec{v}$ = SpatialKey($pk$)    ▷ Primary index search
25:         if $\vec{v}$.WithinRect($\vec{a}, \vec{b}$) then
26:             $R$.Add(Record($pk$))
27:         end if
28:     end for
29:     return $R$
30: end function
```

unique primary key must be obtained from the primary index. Then, the spatial attribute will be verified with the searching MBR to determine if the record shall be returned.

Spatial index with linearized data on B^+-tree can be very efficient due to the superior random and sequential read performance of B^+-tree. It is also relatively easy

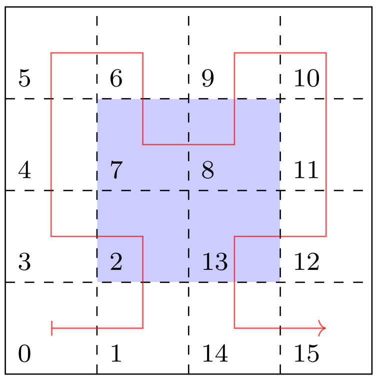


Figure 4.1: Example of Hilbert curve and spatial intersection query.

for an existing database system to support spatial index with some extended framework (GeoMesa) Despite these advantages, these methods have some common drawbacks. The major issue is that this type of index requires some prior knowledge about the space, such as the minimum and maximum values of each dimension, and object distribution, to decide the number of cells to use. Storing cell values costs extra disk space and I/O during index writes and reads. Spatial objects in some cells may be very dense, making scans in these cells relatively slow.

4.3.3 Spatial LSM Index based on R-tree

Spatially close objects may not have close cell values, as shown in Figure 4.1. A natural way is to place nearby records into the same groups. R-tree [40] and R^*-tree [11] are widely used as local indexes for spatial data, which partition records into disk blocks based on their spatial locations (in this chapter, we use R-tree and R^*-tree interchangeably). The R-tree has similar implementation to B^+-tree, except it partitions leaf nodes and creates internal nodes by MBRs. Spatial queries may need to traverse multiple paths to leaf nodes to find records. To bulk-write an R-tree, records are sorted by a comparator, then packed into multiple partitions as leaf nodes and create internal nodes accordingly in a bottom-up fashion. Common comparators used in R-tree include space-filling curve comparators (**Hilbert curve** or Z-order curve), and **simple** bitwise comparator (Algorithm 4.3). Because only the relative order of two records is needed, space filling curves values are only computed during run-time, and do not need to be stored together with the records, which saves disk space and reduces disk I/O. The simple comparator compares two points by each dimension, which is essentially the Nearest-X algorithm [54, 77]. Note that it is a generalized version of the comparator used for single-dimensional data.

Algorithm 4.3 Comparator implementations - Part 1

```
1: $\vec{a}$                                   ▷ A point type spatial object represented by a vector
2: $|\vec{a}|$                                 ▷ The number of dimensions of $\vec{a}$
3: function SFCCompare($\vec{a}$, $\vec{b}$)   ▷ $|\vec{a}| = |\vec{b}|$
4:     $v_a$ = SFCPoint($\vec{a}$)             ▷ Algorithm 4.1
5:     $v_b$ = SFCPoint($\vec{b}$)             ▷ Algorithm 4.1
6:     if $v_a < v_b$ then
7:         return -1                           ▷ $\vec{a}$ is smaller
8:     else if $v_a > v_b$ then
9:         return 1                            ▷ $\vec{b}$ is smaller
10:    else
11:        return 0                            ▷ $\vec{a}$ and $\vec{b}$ are equal
12:    end if
13: end function
```

Algorithm 4.4 Comparator implementations - Part 2

```
14: function SimpleCompare($\vec{a}$, $\vec{b}$)                         ▷ $|\vec{a}| = |\vec{b}|$
15:     for $i = 1 \rightarrow |\vec{a}|$ do                             ▷ Compare each dimension
16:         if $\vec{a}[i] < \vec{b}[i]$ then
17:             return -1                                                  ▷ $\vec{a}$ is smaller
18:         else if $\vec{a}[i] > \vec{b}[i]$ then
19:             return 1                                                   ▷ $\vec{b}$ is smaller
20:         else
21:             continue                                                   ▷ Check the next dimension
22:         end if
23:     end for
24:     return 0                                                           ▷ $\vec{a}$ and $\vec{b}$ are equal
25: end function
```

In an LSM R-tree index, SK is the spatial location of every record, typically as an array of numbers. The same records are compared multiple times during flushes, merges, and queries. With a space-filling curve comparator, linearized values of records must be re-computed every time, potentially adding delays to those operations. In most cases, R-tree (or R^*-tree) is the preferred option for spatial index [49, 50]; hence in this chapter, we only focus on the LSM R-tree designs.

A spatial query first determines the list of operational components by checking each component's MBR, represented by the minimum key (bottom left point) $\vec{K}_{min}$ and the maximum key (top right point) $\vec{K}_{max}$, where $\vec{K}$ represents an array. Given two components

$C : \langle \vec{K}_{min}, \vec{K}_{max} \rangle$ and $C' : \langle \vec{K}'_{min}, \vec{K}'_{max} \rangle$ and the number of dimensions $D \geq 1$, the two components are overlapping if and only if (4.1) is satisfied, or disjoint otherwise.

$$\forall d \in [1, D] : \vec{K}_{min}[d] \leq \vec{K}'_{max}[d] \wedge \vec{K}'_{min}[d] \leq \vec{K}_{max}[d] \tag{4.1}$$

Then, a spatial search can scan all operational components and return the results directly, as shown in Algorithm 4.5. Depending on the actual query, the primary index may not be involved in the spatial search.

As described in Section 2.2.1, stack-based merge policies are often unaware of disk components' contents like key boundaries, which merges are scheduled in the same way regardless of the type or dimensions of the data. Disk components have a high chance to have intersected MBRs with each other, making MBR based filtering at component level less important for stack-based policies. Also, R-tree employs MBR-based filtering on the disk block level internally; only a small portion of disk components is read even if the component size is large. Despite a spatial query usually needs to scan all disk components, the read amplification is not high, due to the low average number of disk blocks scanned per component. To the best of our knowledge, AsterixDB is the only system that uses stack-based LSM R-tree indexes.

Stack-based LSM R-tree indexes mostly rely on the local index of disk components for spatial queries, which has little room to improve in the policies themselves. However, it is very different for LEVELED LSM R-tree index. A Leveled LSM R-tree index may have thousands of disk components. A spatial query can potentially check all disk components in the worst case, which leads to very high read amplification and low locality. Two key

Algorithm 4.5 Spatial index based on R-tree

```
1: function BuildIndexes
2:     while not end do
3:         $pk$ = NextRecord.GetPrimaryKey()                ▷ Primary key
4:         $\vec{v}$ = NextRecord.GetSpatialKey()              ▷ Secondary key
5:         $o$ = NextRecord.GetOtherAttributes()         ▷ Any other attributes
6:         WritePrimaryIndex($pk, \vec{v}, o$)
7:         WriteRTreeIndex($\vec{v}, pk$)
8:     end while
9: end function
10: function SpatialSearch($\vec{a}$, $\vec{b}$)
11:     $R = [\,]$                                          ▷ List of matching records
12:     for $\langle \vec{v}, pk \rangle \in$ Rect($\vec{a}, \vec{b}$) do       ▷ $R$-tree search
13:         $R$.Add(Record($pk$))
14:     end for
15:     return $R$
16: end function
```

design decisions are (a) how to partition records into components during merges and (b) what comparator to use to order records inside a component to allow faster merges. We will discuss them in the next section in detail. To the best of our knowledge, no current system is using leveled LSM R-tree indexes, which is surprising given the popularity of Leveled merge policies. We have implemented the discussed policies on AsterixDB for our experiments.

4.3.4 Partitioning in Leveled LSM ***R***-tree

A partitioning algorithm is necessary to split records into different disk components, which affects the performance of write and read operations of a leveled LSM tree. It must be capable of distributing records into a fixed number of partitions such that the number of records in all partitions are roughly the same. In this section we discuss three partitioning algorithms, size, STR and R^*-Grove, along with two comparators, the Hilbert curve comparator, and the simple comparator.

Size Partitioning Size partitioning is the default partitioning algorithm used in leveled LSM-trees. It simply distributes sorted records into multiple disk components such that all disk components have roughly the same size. A priority queue takes streams of sorted records from each merging component as inputs, and outputs a stream of sorted record from all merging components, similar to the sort-merge join algorithm. Because records are already sorted in each component, storing them in memory for sorting is not needed. Size partitioning only fetches one disk block from each merging disk component at a time, so the memory requirement is minimal. The order of the records depends on the comparator being used. By default, AsterixDB sorts spatial records by a Hilbert curve comparator for 2-D point data and Z-order curve comparator for the other types of spatial data. The two space filling curve based comparators cannot guarantee spatially disjoint disk components, as shown in Figure 4.2b as the partitioning result from Figure 4.2a. On the other hand, if size partitioning is coupled with the simple comparator, this combination can achieve a similar result as STR partitioning, which will be discussed in the next paragraph.

STR Partitioning Sort-Tile-Recursive (STR) [54] was originally proposed to pack blocks for R-tree for point data. We adopt this partitioning algorithm in leveled LSM R-tree index. When disk components are merged, we apply STR to partition all merging records to multiple spatially disjoint groups and create a separate disk component for each group. That way, all disk components in one sorted run are disjoint, regardless of the comparator. For non-point data, we apply STR to the center points of spatial objects, but MBRs are computed from their actual MBRs. The comparator only affects the order of the records inside each component, but the components' MBRs remain the same. There are two major drawbacks of STR partitioning. The first is that STR requires storing all merging records in memory for sorting, leading to much higher CPU and memory usage, otherwise, exernal sorting is needed which incurs much higher disk I/O cost. Thus, it is generally slower than size partitioning. The second is that because STR gives higher weights on more significant dimensions, it tends to create narrow but tall rectangles (as shown in Figure 4.2c from the same input), which may make read queries less efficient as a read query may need to check more disk components although only a small portion of each disk component is actually needed. This may be even more severe for higher dimensional data [86, 87].

***R^**-Grove Partitioning** We also ported R^*-Grove [86, 87] partitioning, which aims to create square-like and balanced partitions for analytic frameworks like Apache Hadoop and Spark, into AsterixDB for our experiments. R^*-Grove partitions spatial records in three phases: a sampling phase which draws a random sample of the input records, a boundary computation phase which generates partition boundaries with desired level of load balance, and a final partitioning phase which puts every record into the corresponding partition. Like

STR, comparator does not affect the partitioning but only affects the internal organization of disk components. As shown in Figure 4.2d (from the same input), R^*-Grove tends to create more square-like MBRs so fewer disk components may be checked. However, it makes multiple passes to scan all records, and is computationally more expensive than STR, for which merges are usually slower.

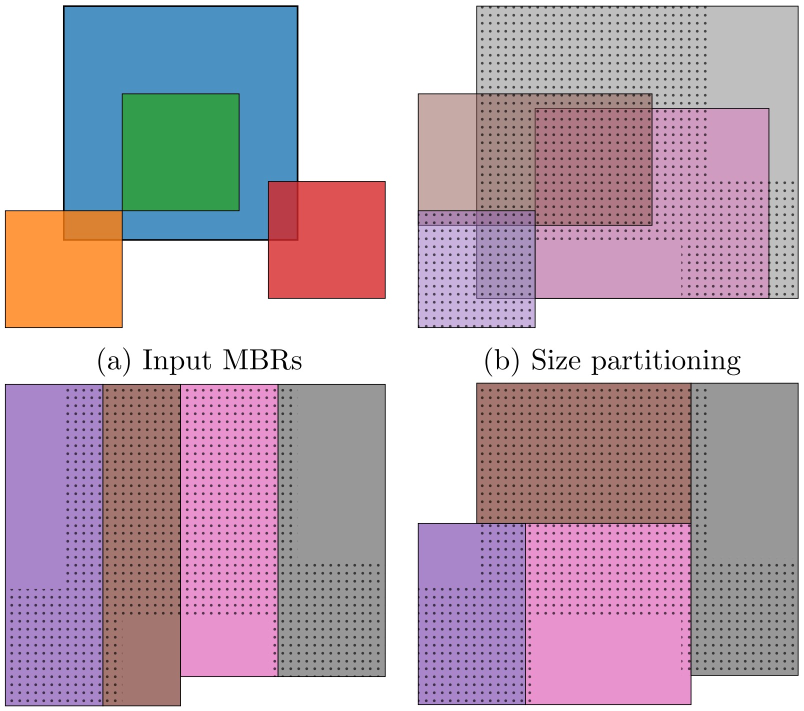

(a) Input MBRs
(b) Size partitioning
(c) STR partitioning
(d) R^*-Grove partitioning

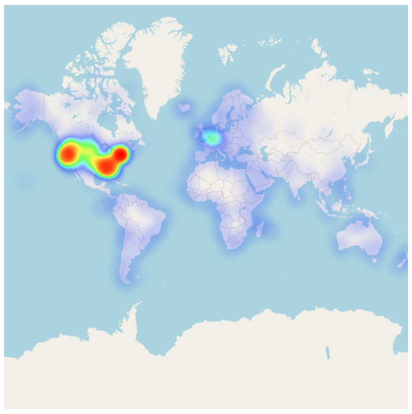
Figure 4.2: Examples of three partitioning algorithms from the same input. Points are uniformly distributed in each of the four input MBRs and are marked with dots in the three partitioned sub-figures.

Both STR and R^*-Grove face an issue of high memory usage, which limits the total size of components to be merged. A possible solution is to make two passes on all merging disk components. The first pass samples a small number of records from all merging disk components, then STR or R^*-Grove can be applied on the sampled records to obtain partition boundaries. The second pass scans all records and put them into a corresponding partition whose MBR contains the record (or the center if it is not point type). This method only uses a small amount of memory, but significantly increases the number of disk I/Os during merge operations.

4.4 Experimental Evaluation

4.4.1 Datasets and Workloads

Two geo-location datasets of exactly 100,000,000 2-D points were used in all experiments. One is a real-world dataset randomly sampled from OpenStreetMap (*OSM* for short) [35, 41]; the other is a synthetic dataset which longitude and latitude values were uniform randomly generated. Points in the OSM dataset are highly clustered in urban areas all over the world, especially in the United States and western Europe (Figure 4.3a). Points in the random dataset are uniformly distributed around the globe (Figure 4.3b).



(a) OpenStreetMap

(b) Random

Figure 4.3: Heatmap of the two datasets, coordinates range from $[-180°, -90°]$ to $[180°, 90°]$.

For each dataset, we generated a workload with interleaved reads and writes as follows:

1. A *Load* phase of 50,000,000 records. Each record is associated with a unique ID in long type and a random string of 1,000 bytes as a synthetic attribute (e.g., geo-location description). Points are stored as two double type numbers. Every record is exactly one kilobyte long.

2. An *Insert* phase containing 500,000 records.

3. A *Read* phase containing 10,000 spatial intersection queries. The query rectangle center is a point randomly picked from all previously inserted points. The rectangle size is determined by a random selectivity $10^{-\sigma}, \sigma \in \{3, 4, 5\}$, that the width and height are $360 \times 10^{-\sigma}$ and $180 \times 10^{-\sigma}$, respectively.

The *Load* phase was executed once in the beginning, then the *Insert* phase and *Read* phase were interleaved for 100 times that 100,000,000 total records were inserted (leading to 100 GB primary index and 2.4 GB LSM *R*-tree index), and 1,000,000 queries were executed. This interleaved workload guarantees the same data size in the corresponding insert phase and read phase in all experiments for fair comparisons.

Read queries were generated in a way that every query can return at least one record. We also tested other selectivity values with $\sigma \in \{1, 2\}$ and $\sigma \in [6, 10]$. We observed the same results for $\sigma \in \{1, 2\}$ with $\sigma = 3$, and $\sigma \in [6, 10]$ with $\sigma = 5$, hence we only reported results for $\sigma \in \{3, 5\}$ ($\sigma = 4$ and $\sigma = 5$ are very similar). To avoid access to the primary index, we used COUNT(*) function so only the LSM *R*-tree index would be scanned. AsterixDB provides several built-in spatial functions, only "*spatial_intersect*" operates on the LSM R-tree index. Many other types of spatial query are usually based on pruning using MBR intersections, such as circle range, kNN and distance join, it is reasonable to focus on this type of rectangular intersection queries.

4.4.2 Experimental Setup

Apache AsterixDB [9] is a full-function, open-source Big Data Management System (BDMS) on LSM storage. The primary index of a dataset is stored as LSM B^+-tree, the spatial index is stored as LSM R-tree. All secondary indexes and the primary index share a global memory budget; thus, they are always flushed together. AsterixDB uses the eager strategy to maintain secondary indexes. Spatial records are ordered by a Hilbert curve comparator or a Simple comparator. MBR of a disk component is computed from all records when it is created from a flush or a merge.

All experiments were performed on 5 AWS *m5.large* instances. Each instance has 2 vCPUs running on Intel Xeon Platinum 8175M, 8 GB of memory, and 200 GB general purpose SSD (gp2). All 5 instances are located in the same zone "*us-west-2b*", connections within instances only used private IP to minimize network latency. AsterixDB was configured to use a single node in each server. Other configurations were set to the defaults. The average size of the flushed disk components in LSM R-tree index was around 2 MB.

4.4.3 Merge Policy Configurations

The following merge policy configurations were applied to the LSM R-tree index:

- Binomial: $k \in \{4, 10\}$.
- Tiered: $B \in \{4, 10\}$.
- Concurrent: Default ($k = 30$, $C = 3$, $D = 10$, $\lambda = 1.2$).
- Leveled: $B_0 = 2$, $B = 10$, size, STR and R^*-Grove partitioning.

Both *Hilbert* curve comparator and *Simple* comparator were paired with each configuration. To avoid interference from the primary index, we set Binomial policy with $k = 8$ for the primary index in all runs. For runs of Leveled policy, we used a selection algorithm to pick a disk component that overlaps with the fewest disk components in the next level, aiming at minimizing their write amplification.

4.4.4 Write Performance

Write Amplification A merge policy with higher write amplification writes more data, which may reduce the write throughput, potentially slow down other operations as well. We present the write amplification of policies with different configurations for the two datasets in Figure 4.4. Write amplification of all stack-based policies are not affected by the dataset because the policies are all content-unaware. Comparators only affect the order of records within disk components, but not component sizes. The write amplification of a stack-based policy are the same for all its configurations, so they are combined in the figure. Binomial with $k = 10$, Tiered with $B = 10$, and Concurrent had the lowest write amplification as they merge infrequently. Binomial with $k = 4$ and Tiered with $B = 4$ had slightly higher write amplification as they merged more eagerly, and Binomial must bound the number of disk components.

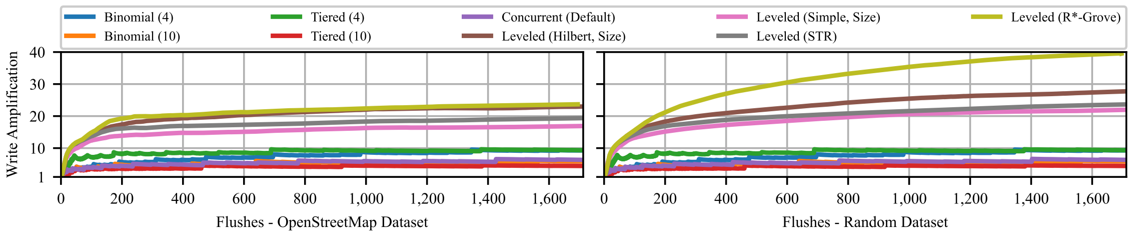
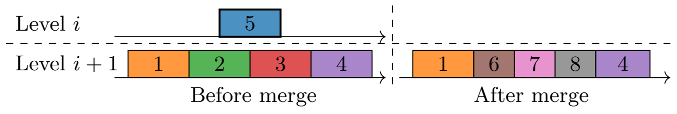


Figure 4.4: Write amplification of compared policies with different configurations.

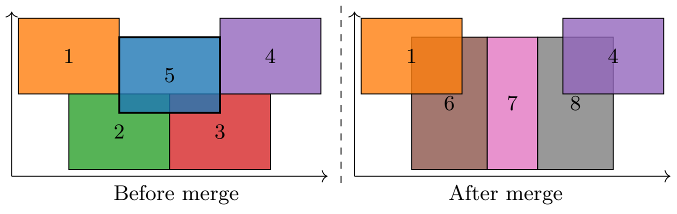
All Leveled policy runs had much higher write amplification than any stack-based policy. Write amplification for the random dataset is higher than the OSM dataset. For the random dataset, it has a higher chance of having more overlapping disk components involved in every merge.

Runs using R^*-Grove partitioning had the highest write amplification among all and are even more significant in the random dataset. Runs using size partitioning with Hilbert curve comparator had the second highest write amplification, as this setting failed to generate disjoint disk components. Runs using STR partitioning with either comparator had slightly lower write amplification because STR partitioning guarantees disjoint disk components, so merge sizes were smaller on average. Runs using size partitioning with Simple comparator achieved the lowest among them because merging records were ordered by the longitude values; thus, they were partitioned into disjoint groups, creating almost disjoint disk components.

The write amplification of runs using R^*-Grove is much higher than the other runs of Leveled policy, especially in the random dataset. A key reason is that a partitioning algorithm that generates disjoint key ranges can only guarantee that the merged components do not overlap with any other component in the level for single dimensional data, as shown in Figure 4.5a, where component 1 from level i has overlapping key range with component 3 and 4 from level $i+1$ (each rectangle represents the component's key range in the whole key space). However, as shown in Figure 4.5b, creating disjoint merged components may fail to guarantee disjoint components in the level. Having overlapping components in a level does not only increase the read amplification, but also increases the write amplification as the probability of merging with more components becomes higher. STR partitioning also has this issue, but it is not so obvious. MBRs created from STR partitioning tend to be tall and thin, which will look like a vertically stretched version of Figure 4.5a, that there will be only a few of overlapping components in every level. But for R^*-Grove, overlapping components can frequently occur, leading to much higher write amplification.



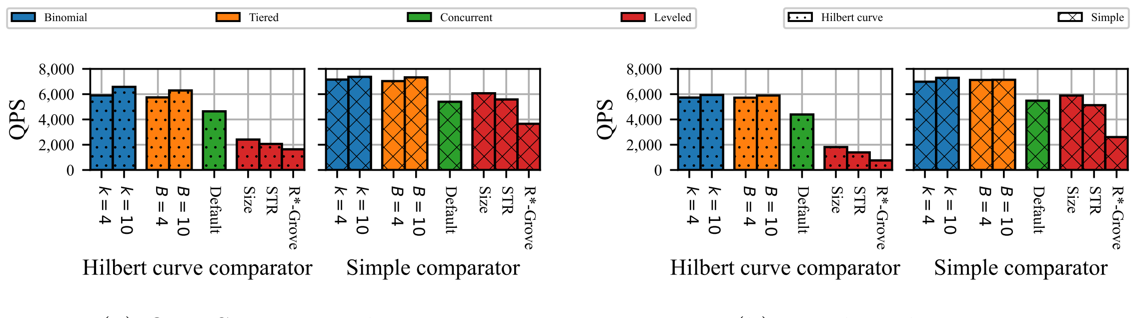
(a) Single-dimension



(b) Two-dimension

Figure 4.5: Components' key boundaries before and after a merge, where components 2, 3 and 5 are merged and replaced by 6, 7 and 8 (illustration purpose only, not from real data).

Write Throughput We have further measured the write throughput and listed the numbers in Figure 4.6. All stack-based policies showed a very high write throughput. BINOMIAL and TIERED runs had the highest write throughput. CONCURRENT had much lower write throughput though its write amplification are low. For runs of LEVELED policy, write throughput of runs with Simple comparator was very close to BINOMIAL and TIERED despite they had high write amplification, runs with Hilbert curve comparator still got the lowest throughput as expected. The write throughput of runs using R^*-Grove partitioning were much lower than the others.



(a) OpenStreetMap dataset (b) Random dataset

Figure 4.6: Average write throughput (requests per second) for all policies with different configurations.

All indexes shared a global memory budget in AsterixDB; any secondary index was always flushed together with the primary index. The write throughput of an LSM secondary index can be dominated by the throughput of the primary index, as the primary index is much larger (2.4 GB v.s 100 GB). For this reason, write throughput of BINOMIAL and TIERED runs were slowed down. We did not observe write stalls or spikes in write throughput in the R-tree index either, which should be common in stack-based policies [60, 95].

Hilbert curve comparator is generally slower in computation than Simple comparator as it needs multiple internal iterations to compare two values, significant overheads could be added to write throughput. To verify this hypothesis, we ran a set of small experiments using the same source codes of both comparators plus a Z-order curve comparator from AsterixDB to sort arrays of 1,000,000 random points of 2, 3 and 4 dimensions, respectively. Results in Figure 4.7 showed that Simple comparator is about six times faster than the other two.

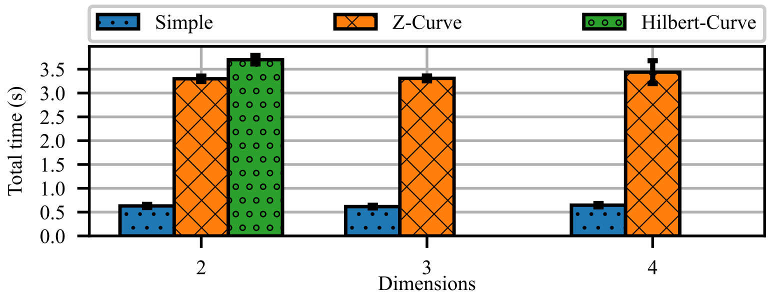


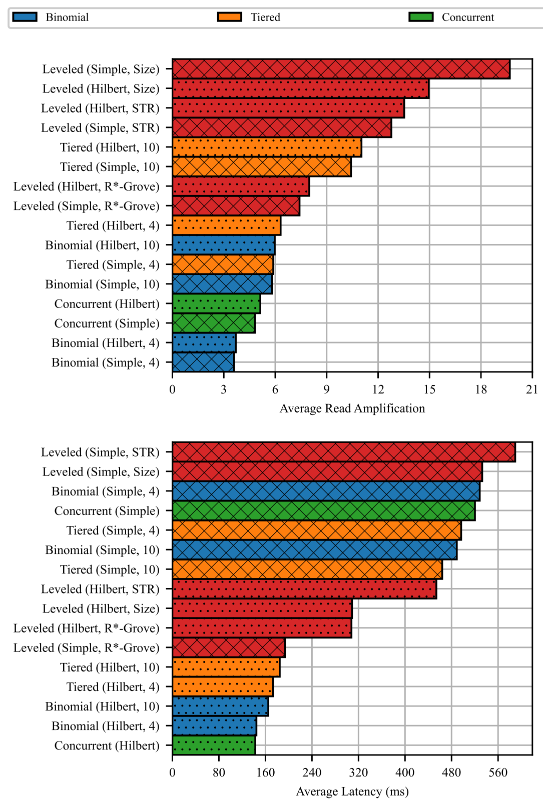
Figure 4.7: Total time to sort arrays of 1,000,000 random points. AsterixDB's Hilbert curve comparator only supports two dimensional points due to its slow computation for higher dimensions.

4.4.5 Read Performance

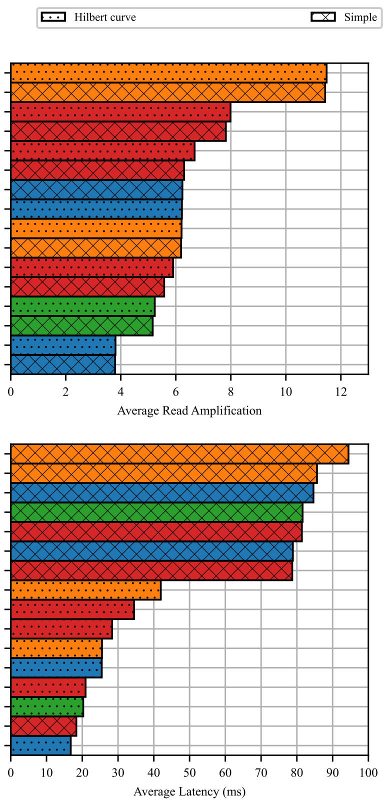
We measured the read performance by the following two metrics: (a) average (mean) *read amplification*, i.e., the number of operational disk components of each spatial query, and (b) average (mean) *read latency*, i.e., the total time spent to scan all operational disk components. Latency here is different from query response time that it measures the time accessing every operational disk component and excludes the time of query compilation and network latency.

High Selectivity (10^{-3}) A spatial query with higher selectivity covers a more substantial area, which returns more results on average. We measured the average number of returned records of about 28,000 for the OSM dataset and 75 for the random dataset. The difference between these two numbers signified from the clustering of the OSM dataset, where a large selectivity query hit more points in highly clustered areas than unclustered areas in the random dataset.

The average read amplification and latency for the OSM dataset are shown in Figure 4.8a. In general, the read amplification of a stack-based LSM index is the same as the total number of disk components, because all disk component must be scanned. For leveled LSM index, only 10 to 20 disk components were scanned, even though over 1,000 disk components were available; MBR based filtering was very efficient. Two runs using size partitioning had the highest two read amplification. Looking into latency, all policies with Hilbert curve comparator had lower latency than those with Simple comparator, except for the runs using R^*-Grove partitioning. With Hilbert curve comparator, stack-based policies still had the lowest latency numbers, the latency of Leveled policy using size partitioning and R^*-Grove partitioning were not bad. Surprisingly, Leveled policy using R^*-Grove partitioning with Simple comparator achieved very competitive results to the faster five runs of the stack-based policies with Hilbert curve comparator, while most of the other Leveled policy runs were slower. Overall, Hilbert curve comparator would be preferred for large selectivity queries, read latency is almost linearly correlated to the read amplification; thus stack-based policies might be better, but Leveled policy using R^*-Grove partitioning with Simple comparator is also a good option.



(a) OpenStreetMap dataset



(b) Random dataset

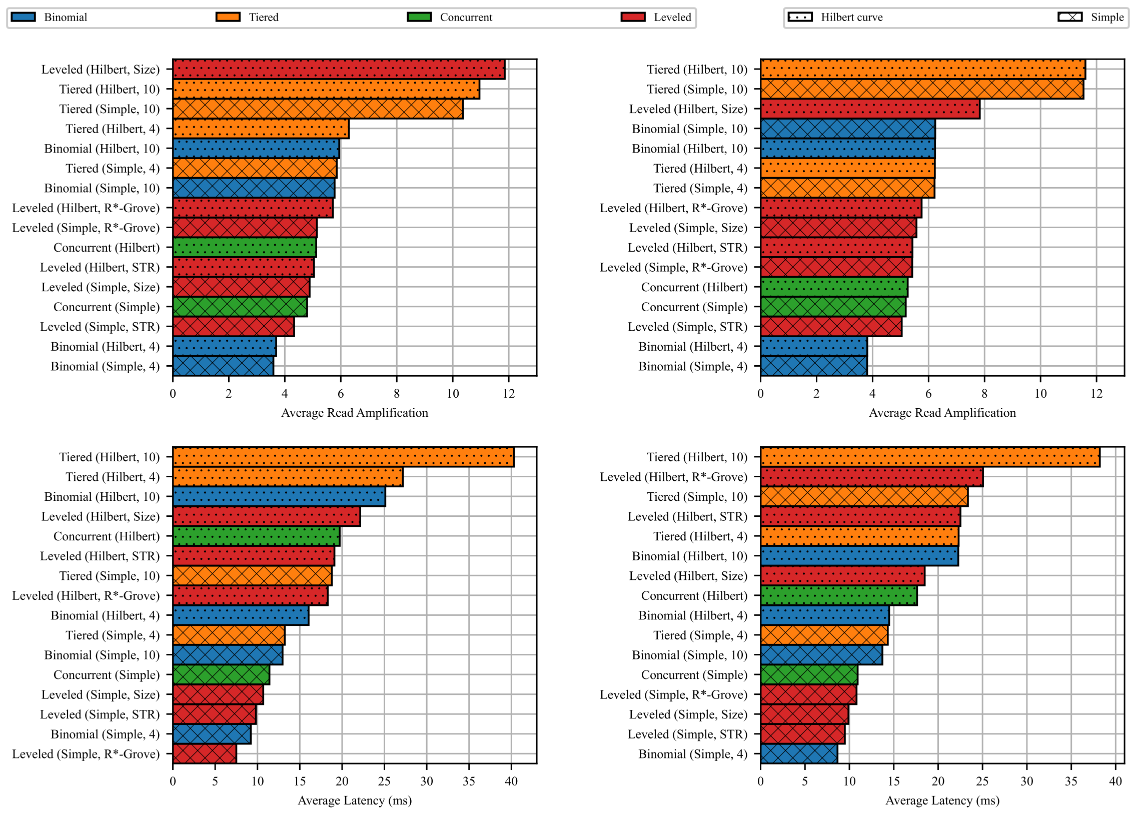
Figure 4.8: Average read amplification and latency for selectivity 10^{-3}.

For the random dataset, read amplification were about the same to the OSM dataset for all stack-based policies, as shown in Figure 4.8b. However, read amplification of LEVELED policy runs dropped to below 8 with significant improvements, meaning that MBR based filtering had been even more efficient for this type of dataset. Runs with Hilbert curve comparator still outperformed runs with Simple comparator for read latency, except for the runs using R^*-Grove partitioning. Still, runs of LEVELED policy ranked very well among them, especially the run using R^*-Grove partitioning with Simple comparator which ranked second, thanks to their lower read amplification. Latency numbers in the random dataset were 6 to 8 times shorter than the numbers in the OSM dataset. Here, reads tend to be slower in the highly clustered dataset for high selectivity queries.

Low Selectivity (10^{-5}) Common spatial queries usually return less than 100 results, which cover a relatively small area. In our experiments, we measured an average of 12 results from the OSM dataset and 1 from the random dataset. However, the number could be 0 most of the time for the random dataset if query rectangles were not generated from existing points.

Similar to high selectivity queries, write amplification of all the stack-based policies remained the same for both datasets, as almost all disk components were scanned, as shown in Figure 4.9a and Figure 4.9b. Except for one run of Leveled policy using size partitioning with Hilbert curve comparator, the other five runs of Leveled policy became very competitive in both datasets, which even had lower read amplification than some stack-based policies. The two runs using STR partitioning, and the runs using size partitioning and R^{*}-Grove partitioning with Simple comparator, had much better MBR based filtering for low selectivity queries.

Because of the lower read amplification and fewer returned records, read latency numbers were all smaller than those in the high selectivity queries. Runs with the Hilbert curve comparator were slower than those with Simple comparator. The slower computation of the Hilbert curve comparator became a significant bottleneck for low selectivity queries, while it showed superior efficiency for high selectivity queries. The three Leveled runs with Simple comparator were all in the top four among all. Queries could finally take



(a) OpenStreetMap dataset (b) Random dataset

Figure 4.9: Average read amplification and latency for selectivity 10^{-5}.

advantage of their better MBR based filtering capabilities to provide much faster index access time. From comparing the two figures of read latency for the two datasets, we can see the numbers are very close to the same policy with the same configuration. The impact of data clustering was not evident on read performance for small selectivity queries.

4.5 Discussion

Among all compared policies, Binomial was the winner in almost all settings, showing the best read amplification and latency numbers, while maintaining the highest write throughput and near-top write amplification. CONCURRENT was only second to BINOMIAL in terms of read performance in most settings, but it had relatively low write throughput, although its write amplification was low. Its multi-threaded merge was a bottleneck in write throughput. There could be some optimizations to multi-threaded merges, but most need hardware or operating system support [27, 88]. TIERED had the lowest write amplification and very high write throughput, but the read performance was sacrificed. The Leveled policy had the highest write amplification, but writes could still be fast with proper configurations. In our experiments, the Leveled policy showed good read performance mostly in low selectivity queries, although the combination of R^*-Grove partitioning and Simple comparator achieved outstanding read performance at the cost of the lowest write performance. Therefore it may not be a good option for high selectivity queries in general. There could be cases where it may be better suited. Leveled architecture is a perfect fit for object stores (Amazon S3, Microsoft Azure, etc.) which tend to have many relatively small files (or so-called blobs). Comparing to stack-based policies, it can manage records more efficiently via file (disk component) based filtering, rather than relying on local indexes.

In terms of policy configuration, Hilbert curve comparator performed better than Simple comparator in high selectivity queries but was worse in low selectivity queries due to its slow computation. If Leveled policy must be chosen, size partitioning is generally a good option for high selectivity queries, while STR partitioning and R^*-Grove partitioning are still very competitive, especially in low selectivity queries. With a larger index size, STR and R^*-Grove might be better options because they guarantee to create disjoint disk components to have better MBR based filtering capability. However, the higher CPU and memory requirement during merges as well as the low write performance of R^*-Grove must be considered.

How LSM secondary index is maintained may have a major impact on the write and read performance of a secondary index. With the eager strategy, write throughput may be determined by the primary index, while with the lazy strategy, read latency may be dominated by the time to verify returned records against the primary index.

Limitations and Future Work In this chapter we focused on the write and read performance of R-tree based LSM spatial indexes. Based on the results from [50], we did not include comparisons against indexes based on B^+-tree, which may be a more common approach on existing LSM database systems. It may be worthwhile to revisit these designs on different LSM architectures, since B^+-tree usually has better write and read performance than R-tree for certain types of non-intersection spatial queries. The lack of optimizations on hardware and operating system limited the MBR based filtering efficiency for Leveled policy. We would expect some better results for Leveled policy if some optimizations could be done, such as hardware support for MBR based filtering (e.g. FPGA based filtering) to utilize STR or R^*-Grove partitioning.

4.6 Related Work

Most LSM systems only support single-dimension indexes such as B^+-tree. To support spatial index, they must rely on some linearization method to project multidimensional data into a single dimension to be loaded in B^+-tree. GeoMesa [43] is a spatial-temporal index that supports so-called Bigtable-style databases including Google Bigtable [17], Apache Accumulo [47], Apache HBase [34]. It uses a customized GeoHash [43, 71] implementation based on the Z-order curve to encode spatial and temporal data into bit strings. STEHIX [19] and Brahimet al [14] took a similar approach but only limited to HBase and DataStax Cassandra, respectively. Kim et al [49, 50] studied five LSM spatial indexes, four of them fall into this category: DHB-tree, DHVB-tree, and SHB-tree all map point data with space-filling curves (Hilbert curve); SIF builds an inverted index (based on B^+-tree) but the main idea is similar to SHB-tree. Like SIF, a posting list based LSM inverted index design described in [74] can also be extended to support spatial index.

Another common approach is to build LSM spatial indexes on R-tree. R-HBase [42] and BGRP-tree [83] partition the data space into grid cells or regions and use an in-memory R-tree to index the partitions, although the local indexes are still built on B^+-trees. Nanjappan implemented a separate R^*-tree index outside of Cassandra [70]. LevelGIS [91] uses a three-layer hierarchical structure of R-tree index on LevelDB to support spatial queries.

LSM RUM-Tree [80] utilizes Update Memo on AsterixDB's R-tree index to support update-intensive spatial workloads, which is orthogonal to our work.

Galvizo et al. [33] described how multi-valued fields (multidimensional data) are indexed in AsterixDB, which could also be extended to improve the R-tree index performance as a future work. While many open-source projects add spatial index support to LevelDB, RocksDB, and some other LSM systems, most still use the first approach which is B^+-tree with linearized spatial data, only a few of them adopt the LSM R-tree approach. To the best of our knowledge, none of these projects have been deployed in practice. RocksDB used to provide a utility called SpatialDB, but it got abandoned and removed from GitHub since January 2019.

Some systems choose to use an LSM database only for storage and use some other structures for spatial queries. For example, DataStax stores geospatial data in Cassandra, but builds geospatial index and handles geospatial queries via Solr [22]. Compared to native LSM secondary spatial index, the key drawback of such systems is that insertions are slower as they need to be written to two or more systems.

Partitioning algorithms can highly affect the write and read performance of a leveled LSM R-tree index. Some R-tree packing algorithms can be directly used for partitioning in merges. The combinations of size partitioning with Hilbert curve comparator and Simple comparator have the same effect as Hilbert Sort [46] and Nearest-X [77], respectively, which are both outperformed by STR partitioning [54]. OMT [53] is a top-down R-tree bulk-loading algorithm which might be portable as well. Other partitioning algorithms include sampling-based methods like SpatialHadoop [28] and R^*-Grove [86, 87], and quad-tree-based method like [89] for Hadoop. Most of these algorithms designed for R-tree bulk-loading or Hadoop can efficiently handle static data, where the data is written only once. However, they are usually not designed for dynamic data, where the data frequently changes in some write-heavy workloads. A better partitioning algorithm that takes both heavy writes and reads is much desired, that LSM spatial index can benefit a lot from it.

4.7 Conclusions

In this chapter, we compared and evaluated secondary spatial index performance of both stack-based and leveled LSM architectures with four representative merge policies, on a common platform (AsterixDB). The results from both the OpenStreetMap dataset and the synthetic random datset have shown that Binomial policy is probably the best candidate for LSM R-tree-based spatial index, although it is not specifically optimized for multidimensional spatial data. While having higher write amplification and generally lower write throughput, with proper configuration, Leveled policy can achieve close or even better read performance to some of the better stack-based policies. Although most stack-based policies do not benefit from MBR based filtering at the disk component level, MBR based leveled partitioning can provide much better filtering efficiency to improve spatial query performance in proper settings. Compared to analytic frameworks, a key challenge of MBR based leveled partitioning in LSM tree is to maintain more disjoint square-liked MBRs while keeping the write cost low. We also showed that the choice of different comparators and partitioning algorithms for a Leveled policy depends on spatial queries' selectivity.

Chapter 5

Effective Partitioning for Stack-based Merge Policies

5.1 Introduction

Most of the existing LSM systems can be put into two categories, stack-based and leveled. The former category includes AsterixDB [9], Bigtable [36], Cassandra [7] and HBase [8]. Stack-based policies are generally more efficient for applications that need to support high ingestion rate. Merges in these systems may be large in size, which often leads to fewer but larger files. This design makes resource management easier as merges must be executed single-threaded, and there is no need to maintain many file handlers in memory. On the other hand, large merges take a long time to complete, leading to a higher chance of stopping data ingestion (a.k.a *write stalls*). Also, canceling a merge may be expensive, especially when the work is almost completed, and most merged data has to be discarded.

Leveled policies mainly includes LevelDB [37] and RocksDB [29]. They use a partitioned merge style to keep files relatively small. Data is merged incrementally between sorted runs (levels). Instead of having few large merges, they choose to perform many small merges. This design has several benefits: 1) merges are small, so they can be finished faster; 2) more threads can be used to parallelize multiple merges at the same time; 3) aborting a merge is not too expensive compared to aborting a stack-based merge; and 4) files may not need to be rewritten in merges in certain special workloads. Generally speaking, stack-based LSM systems have better write performance and leveled LSM systems have better read performance.

In Chapters 3 and 4, we discussed how some stack-based merge policies, including the two proposed Binomial and MinLatency, can achieve optimal write amplification while maintaining a very low read amplification. To further tackle their major problems of having insufficient merge throughput due to single-threaded large merges, we propose two partitioning algorithms for stack-based merge policies, such that large merges can be parallelized, and disk I/O can be saved via the support of trivial-moves for certain workloads.

In summary, this chapter makes the following contributions:

1. We motivate and present the problem of facilitating parallelism during stack-based merges. A baseline partitioning method is proposed, size-based, where a new partition is created when the previous partition is above a maximum size. We show what the key challenges are to increase parallelism. Specifically , We identify the *chaining problem* that makes size based partitioning ineffective to stack-based merge policies (Section 5.3).

2. We propose two partitioning methods: Local-Range and Global-Range partitioning that can mitigate the problems of size-based partitioning. These partitioning methods facilitate multiple parallel sub-merges and trivial-moves. (Section 5.4).

3. We implement the two proposed partitioning algorithms on RocksDB and evaluate their write and read performance against the no-partitioning and size partitioning on RocksDB's UniversalCompaction.

5.2 Background

5.2.1 Stack-based Merge

In Chapters 2, 3 and 4, we discuss the differences between stack-based merge policies and leveled merge policies. Stack-based merge policies always merge whole sorted runs into a single output sorted run (Figure 2.3). In the standard design, a single sorted run is equivalent to a single SSTable. An SSTable's size may become very large after a few merges. In general, stack-based merge policies tend to create a few large SSTables. When these SSTables are merged, only a single thread can be used to scan the input SSTables and write to a single output SSTable. This often leads to relatively low disk and CPU utilization, given the fact that database servers have more CPU cores and faster SSDs to support highly parallelized computation and disk I/O.

In the previous chapters, most stack-based merge policies make merge decisions based on file sizes, such as BIGTABLE, CONCURRENT, EXPLORING and TIERED. BINOMIAL and MINLATENCY make merge decisions offline assuming flushed SSTables are of the same size. They all assume SSTables to be merged are all overlapping with each other. This is typically true for many workloads where keys inserted are almost random. However, there are some special workloads in which keys are (almost) sorted. Such examples include workloads where keys are time-correlated, or in a bulk-loading stage. Usually, for such workloads, there is no need to perform any merges, as SSTables created will be strictly non-overlapping with each other. Any single `Get` query needs to check at most one SSTable. Traditional stack-based merges do not take this important information into account, and still make unnecessary merges, wasting disk I/O to merge already sorted data.

RocksDB's UniversalCompaction RocksDB is a fork of LevelDB, which has no support of stack-based merge policies in the beginning. The SizeTiered merge policy (CompactionStrategy) in Cassandra was ported to RocksDB and renamed as UniversalCompaction. Although it originated from the SizeTiered policy, it now has quite different settings and behaviors. Instead of always picking similar sized SSTables to merge, UniversalCompaction also supports picking SSTables of different sizes based on their total sizes via an option *stop_style*. UniversalCompaction in RocksDB is not a strict stack-based merge policy though. For example, sorted runs are allowed to be placed into levels other than 0 to have partitioned SSTables, but it still keeps the backward compatibility to have large sorted runs of single SSTable in level 0. Thus, it can still be viewed as a stack-based merge policy.

5.2.2 Partitioned Leveled Merge

In contrast to stack-based merge, leveled merge in LevelDB [37] and RocksDB [29] chooses to break large sorted runs into smaller non-overlapping SSTables. One sorted run is represented as a list of SSTables, whose key ranges are sorted and non-overlapping. For example, let $[l_i, u_i]$ denote the key range (smallest key and largest key) of the $i-th$ SSTable in a sorted run, there must be $l_i \leq u_i < l_{i+1} \leq u_{i+1}$. Note here, although not common, an SSTable can contain a single key, making its $l = u$. But an SSTable's smallest key must be strictly larger than the largest key in the previous SSTable in the same sorted run. Searching in a sorted run is divided into two or three binary searches for single `Get` or `Iterator` (scan) operation respectively. In a `Get` operation, a binary search is performed to find the first candidate SSTable whose largest key u is no less than the searched key. Then this SSTable's smallest key is checked. If the smallest key is larger than the searched key, this whole sorted run can be skipped as no SSTable in this sorted run contains the searched key. If the smallest key is smaller than the searched key, the key will be binary searched again using the SSTable's index. In an `Iterator` operation, similar to `Get`, at most binary searches are needed to find the first and the last SSTables that have overlapping ranges with the seached range. Then a binary search is performed in the first SSTable to return the first result, followed by a linear scan to return all matching results in all the candidate SSTables.

In a merge, only a portion of the SSTables in sorted runs are selected to keep every merge size small. A typical leveled merge usually sets a maximum SSTable size limit θ such that most SSTables will be this size. In a partitioned leveled merge, output key-value pairs are partitioned on-the-fly based on the current writing SSTable size. Algorithm 5.1 shows the pseudocode of how a merge is executed with size partitioning. If the total size to output is S, there will be $\lceil \frac{S}{\theta} \rceil$ SSTables created. All SSTables created will be the same size θ except for the last one, which may be smaller. It is worth noting that, a triditional stack-based merge can be viewed as the same algorithm with $\theta = \infty$, so that only one SSTable is created from a merge.

Algorithm 5.1 Merge via size based partitioning

```
1: Input: $T$                                    ▷ List of input SSTables to be merged
2: Input: $\theta$                                ▷ Size constraint of SSTables
3: Input: $r_{out}$                               ▷ The output sorted run
4: $iter = \textsc{Iterator}(T)$                  ▷ Merging iterator on all SSTables in $T$
5: $t = \textsc{NewTable}()$                      ▷ Create one SSTable
6: while $iter.\textsc{HasNext}()$ do
7:     $\langle k, v\rangle = iter.\textsc{GetNext}()$   ▷ Get key and value, then advance the iterator
8:     if $t.\textsc{Size}() + k.\textsc{Size}() + v.\textsc{Size}() > \theta$ then   ▷ SSTable size overflow
9:         $r_{out}.\textsc{Add}(t)$              ▷ Complete $t$ and add to $r_{out}$
10:        $t = \textsc{NewTable}()$              ▷ Create another SSTable
11:    end if
12:    $t.\textsc{Write}(k, v)$
13: end while
14: $r_{out}.\textsc{Add}(t)$                     ▷ Complete the last $t$ and add to $r_{out}$
```

Both LevelDB and RocksDB support a so-called *trivial-move* operation. When an SSTable does not overlap with any SSTable in the next sorted run (level), this single SSTable can be selected to "merge" into the next sorted run. Instead of rewriting this SSTable, LevelDB and RocksDB can directly move the file into the next sorted run and just update some minimum size metadata. The I/O cost of a trivial-move is almost none, thus merges in sequential workloads or bulk-loads of sorted data are extremely fast.

5.3 Problem Definition

5.3.1 Motivation: Expensive Stack-based Merges

Although most stack-based merge policies have much better write performance than the partitioned leveled merge policy, they suffer from the following problems:

- During large merges where many sorted runs or some big sorted runs are involved, one merge can occupy lots of CPU, memory and disk resources for a long time. This often leads to write stalls, which slows down or completely stops new writes into the LSM tree.

- Only a single thread can be used for one merge, mainly because there is only one SSTable to write for the output sorted run. For modern storage devices like SSD, this design may not fully utilize the powerful hardware to achieve higher write throughput. The chaining problem described below is another constraining factor that does not allow multiple threads to work on the merge in parallel.

- For certain special workloads, for example, a sequential workload where all ingested keys are in a monotonically increasing order, a partitioned leveled policy can use a *trivial-move* [32] technique to move the files without rewriting them. This design can save lots of disk I/O for this type of workload. However, existing stack-based merge policies do not benefit from trivial-moves, as they always rewrite existing SSTables.

A naive solution to the above problems is to apply partitioning to sorted runs in stack-based LSM trees. Instead of having a single SSTable for each sorted run, every sorted run can be partitioned into one or multiple disjoint SSTables, like the partitioned leveled LSM tree. With a good partitioning algorithm, it is possible to split a big merge into multiple smaller sub-merges, where each sub-merge only processes a subset of SSTables from the input sorted runs. If multiple sub-merges can be scheduled in one merge, they can be executed in multi-threads, which will improve the overall write throughput, and reduce the total time to complete the merge. On the other hand, SSTables that are non-overlapping with any other SSTables in a merge can be trivial-moved to the output sorted run, this can save disk I/O and further improve the write performance. Note here we do not require the whole sorted run be non-overlapping with other merging sorted runs.

5.3.2 Chaining Problem: Key Obstacle to Parallelizing Merges

It is natural to adopt the size based partitioning algorithm to stack-based LSM trees. However, this algorithm will frequently cause the chaining problem, where most or even all the input SSTables must be grouped in one sub-merge. As shown in Figure 5.1, to merge sorted run A with keys $\{1, 3, 5, 7, 9, 11, 13\}$ and sorted run B with keys $\{4, 6, 8, 10, 12, 14\}$ with the maximum SSTable size limit (as the number of keys) $\theta = 3$, only one sub-merge can be scheduled using size partitioning, because there is no way to split the to-be-merged components into disjoint groups. On the other hand, the proposed Global-Range partitioning (discussed in Section 5.4) can effectively schedule two sub-merges and one trivial-move, making the merge of these two sorted runs more efficient.

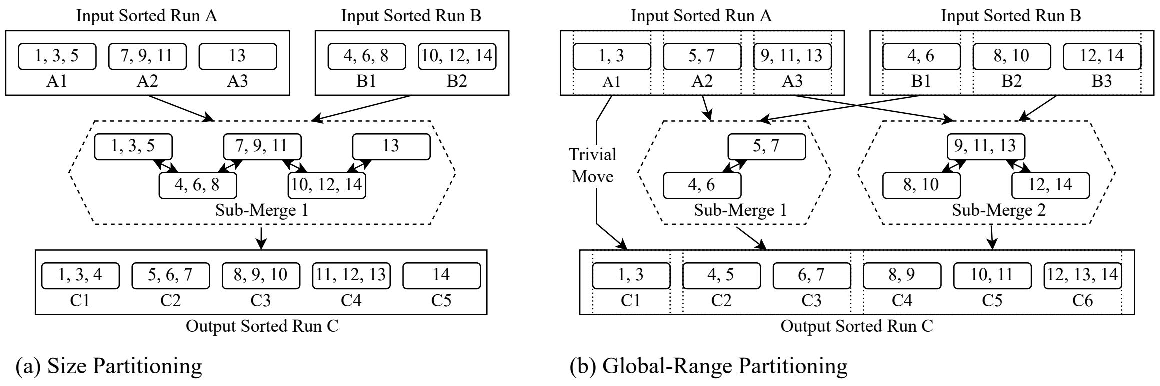


Figure 5.1: Size partitioning vs. Global-Range partitioning.

5.3.3 Trivial-Move: I/O Saving during Merge

RocksDB's UniversalCompaction supports trivial-moves with a strict requirement — a trivial-move can only be performed when only one single SSTable is selected in a merge, and this SSTable is non-overlapping with any other SSTable in the output level. In practice, this usually requires that all SSTables in level 0 are non-overlapping with each other. Because a stack-based merge policy only merges consecutive SSTables, an SSTable will always be merged if it is between two overlapping SSTables, even if this SSTable is not overlapping with any other SSTable in the database. For example, to merge three consecutive SSTables A, B and C, whose key ranges are $[1, 10]$, $[21, 30]$ and $[6, 15]$ respectively, B $[21, 30]$ must be merged together with A and B though it is not overlapping with the other two.

5.4 Proposed Local-Range and Global-Range Partitioning

In this section, we first show how a stack-based merge is executed with partitioning in Section 5.4.1. Then we present an overview of Local-Range and Global-Range partitioning in Section 5.4.2. We show the details of Local-Range and Global-Range partitioning in

Section 5.4.3. Lastly, we show the challenges and our solutions in implementing these two partitioning algorithms Section 5.4.4

5.4.1 Partitioned Stack-based Merge

In a leveled merge, when an SSTable is selected to merge and it is not overlapping with any SSTables in the next level, this SSTable will be trivial-moved. This means, any leveled merge of at least two SSTables will never contain any SSTables that can be trivial-moved. However, when a stack-based merge is executed with partitioning, it is possible that some SSTables do not overlap with other SSTables, since all SSTables in the merging sorted runs must be selected. For this reason, the partitioned stack-based merge should be handled differently to leveled merge. Algorithm 5.2 describes how sub-merges can be generated and executed in one stack-based merge of multiple partitioned sorted runs, where some SSTables may be trivial-moved and the others may be sub-merged.

By checking the metadata of all the input SSTables, we are able to place the SSTables into one or multiple non-overlapping groups. Then we can execute either a trivial-move or a sub-merge for each group in parallel. Parallel execution via multi-threaded sub-merges provides better write throughput with shorter total merge time, at the cost of higher CPU, memory, and disk usages. The performance of multi-threaded sub-merges is heavily hardware dependent, which may be limited by the number of CPU cores, storage media access speed, etc.

If one sub-merge contains only one input SSTable, meaning this SSTable has no overlapping SSTables in the whole merge, this SSTable can be safely *trivial-moved* to save

Algorithm 5.2 Partitioned stack-based merge - Part 1

```
1: Input: $R_{in}$                         ▷ List of input sorted runs ($|R_{in}| \geq 2$)
2: Input: $r_{out}$                        ▷ The output sorted run
3: $M = [\,]$                              ▷ List of all SSTables to be merged
4: for $r$ in $R_{in}$ do                  ▷ $r$: sorted run
5:     for $t$ in $r$ do                   ▷ $t$: SSTable
6:         $M$.Append($t$)
7:     end for
8: end for
9: $M$.SortByLowerBound()
10: $C = [\,]$                             ▷ List of non-overlapping SSTables groups
11: $G = [\,]$                             ▷ Group of overlapping SSTables
12: $u_{max} = 0$                          ▷ Current upper bound of $G$
```

the disk I/O for rewriting it. The execution of a trivial-move is simple, all files (if more than one) associated with this SSTable can be moved or hard-linked (filesystem dependent) into the output sorted run. Although moving or hard-linking files incurs no disk I/O, minimum disk I/O may be required to update the SSTable or database's metadata such as the manifest file.

If one sub-merge has more than one input SSTables, we must partitioned-sub-merge (Algorithm 5.3, line 30) them. The size based partition method in Algorithm 5.1 can be used as one implementation of the PartitionedSubMerge function aiming at creating similar sized SSTables with a maximum SSTable size limit θ. As discussed in

Algorithm 5.3 Partitioned stack-based merge - Part 2

```
13: for $t$ in $M$ do                                          ▷ $t$: SSTable
14:     $u_t = t$.GetUpperBound()
15:     if $|G| = 0$ then                           ▷ The first SSTable, $G$ is empty
16:         $u_{max} = u_t$
17:     else if $t$.GetLowerBound() $> u$ then              ▷ $t$ is disjoint with $G$
18:         $C$.Append($G$)                       ▷ Complete $G$ and append it to $C$
19:         $G, u_{max} = [\,], u_t$                      ▷ Reset the current group $G$
20:     else                         ▷ $t$ overlaps with $G$, update $u_{max}$ if needed
21:         $u_{max} =$ Max($u_{max}, u_t$)
22:     end if
23:     $G$.Append($t$)
24: end for
25: $C$.Append($G$)                      ▷ Complete the last $G$ and append it to $C$
26: for $T$ in $C$ do                           ▷ $T$: list of overlapping SSTables
27:     if $|T| = 1$ then
28:         TrivialMove($r_{out}, T[0]$)            ▷ Move the only SSTable to $r_{out}$
29:     else
30:         PartitionedSubMerge($r_{out}, T$)   ▷ Execute a sub-merge with partitioning
31:     end if
32: end for
```

Section 5.3.2, the chaining problem may prevent Size partitioning from generating sub-merges effectively. To solve this problem, we propose two algorithms, *Local-Range* and *Global-Range* partitioning, which partition SSTables based on predetermined key ranges.

5.4.2 Overview of Local-Range and Global-Range Partitioning

Similar to k-d tree [13], both Local-Range and Global-Range partitioning apply binary space partitioning on keys rather than SSTable sizes. The key difference between these two algorithms is, Local-Range partitioning uses the actual key range from the SSTables to be merged (actual smallest and largest keys) as the key space to binary-split, while Global-Range partitioning uses the key range from the whole key space (smallest and largest possible keys). They recursively partition the keys until all partitions are no larger than θ. Figure 5.2 presents an overview of the three partitioning algorithms (Size partitioning, Local-Range and Global-Range partitioning) and No-partitioning on a sorted run of 40 keys.

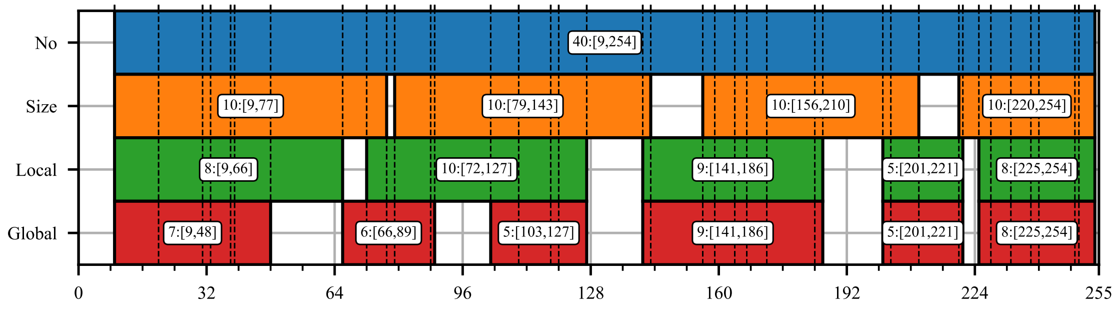


Figure 5.2: Overview of No-partitioning, Size, Local-Range partitioning and Global-Range partitioning on a sorted run of 40 8-bit unsigned integer keys. Maximum SSTable size limit $\theta = 10$. SSTable size as the number of keys, the smallest key and the largest key are annotated in each rectangle representing an SSTable. Actual keys are shown in dashed vertical lines.

Local-Range partitioning first splits the key range [9, 254] at the mean $\lfloor\frac{9+254}{2}\rfloor = 131$, then the left and right partitions are further split at their means $\lfloor\frac{9+131}{2}\rfloor = 70$ and $\lfloor\frac{132+254}{2}\rfloor = 193$, respectively. On the other hand, Global-Range partitioning makes the first split at the mean of the whole key space [0, 255], thus the first split is always at $\lfloor\frac{0+255}{2}\rfloor = 127$. Then the left and right partitions are split at $\lfloor\frac{0+127}{2}\rfloor = 63$ and $\lfloor\frac{128+255}{2}\rfloor = 191$, respectively. In both algorithms, splits will stop when all partitions are no larger than 10.

Comparing the three partitioning algorithms and no-partitioning, no-partitioning creates a single partition that covers almost the entire key space. Size partitioning creates similar sized partitions (all have 10 keys) except the last one. Local-Range partitioning creates partitions where adjacent partitions' key range sizes are similar. For example, the first three partitions have key range sizes 58, 56 and 46 respectively. The last two partitions have key range sizes 21 and 20 respectively, and their total key range size is 54, which is close to the first three partitions. Global-Range partitioning creates partitions with fixed bounds $[0, 63]$, $[64, 95]$, $[96, 127]$, $[128, 191]$, $[192, 223]$ and $[224, 255]$ respectively, where the maximum key range size of each partition is always 256×2^{-i}, where $0 \leq i \leq 8$. It is also worth pointing out that these partitioning algorithms may create the same partitions depending on the distribution of keys, like the last three partitions in Local-Range and Global-Range partitioning.

When multiple partitioned sorted runs are merged, Global-Range partitioning has a higher chance to generate more sub-merges. An example of partitioned stack-based merges on three sorted runs is shown in Figure 5.3. Size and Local-Range partitioning can generate only one sub-merge, while Global-Range partitioning can generate five sub-merges.

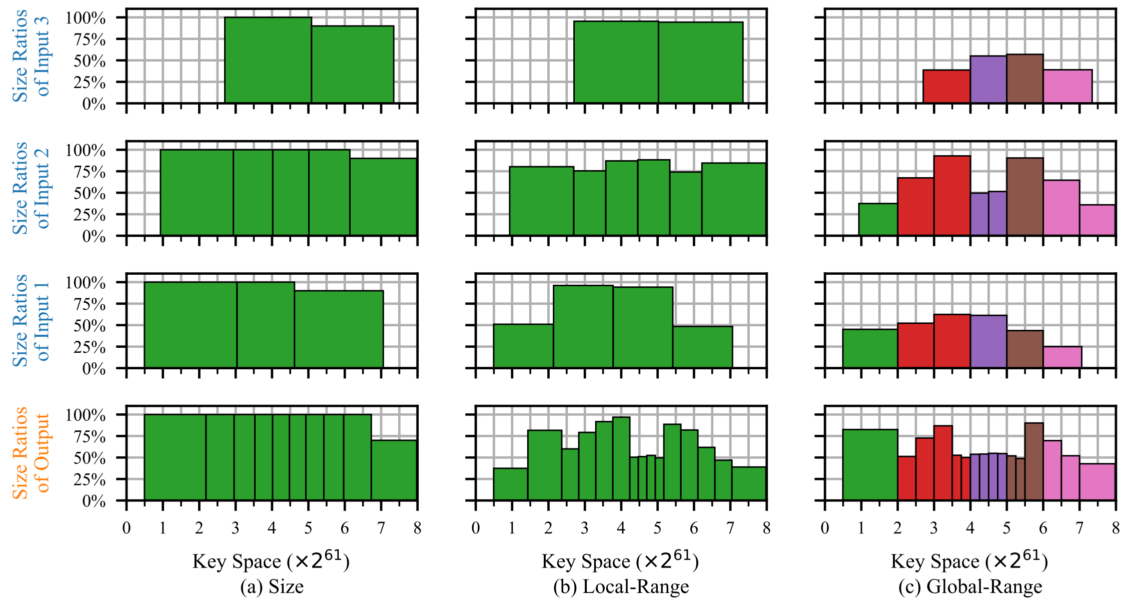


Figure 5.3: Examples of Size, Local-Range and Global-Range partitioning. The top 3 rows are the SSTable size ratios with respect to θ in 3 input sorted runs. The last row is the size ratios of the output sorted run from merging the 3 input sorted runs. Each column represents the results of a different partition algorithm. SSTables of the same color are grouped in one sub-merge. Keys are the same in the sorted runs in every row. Sorted runs 1, 2 and 3 are created from previous flushes and merges.

Although Global-Range partitioning can generally provide higher parallelism via multiple sub-merges, it tends to create more SSTables. In the given example, 10 SSTables are created after merge using Size partitioning, while this number is 15 and 16 for Local-Range and Global-Range partitioning respectively. In practice, both Local-Range and Global-Range will create 1.5 to 2 times more SSTables than Size partitioning with the same θ, although Local-Range partitioning usually creates fewer SSTables than Global-Range partitioning. Having too many SSTables may increase the overhead of maintaining SSTables in the system, and may reduce the `Iterator` performance due to the fact that more files need to be scanned.

There are also two major challenges implementing Local-Range and Global-Range partitioning, estimating data size within a range, and mapping variable-length byte string keys to unsigned integers. We will address these two challenges in Section 5.4.4.

5.4.3 Local-Range and Global-Range Partitioning Implementation

The section is organized as follows. We first show the basic implementation of range-based PartitionedSubMerge (Algorithm 5.3 line 30) with a list of predetermined key bounds, which is used by both Local-Range and Global-Range partitioning. Then we show the implementations of Local-Range and Global-Range partitioning to compute the key bounds. Lastly, we show the challenges in the implementations and our solutions.

Range-based PartitionedSubMerge

In a sub-merge with Size partitioning, the key range of every output SSTable is computed on the fly during the SSTable construction. The execution of a sub-merge is different when the key ranges are provided beforehand. Generally, a key range is defined as a pair of a lower bound l and an upper bound u, identified as $[l, u]$[1]. During the execution of a sub-merge, a key value pair will be written to a new SSTable when the key is larger than the current range's upper bound. The pseudocode of this range-based is shown in Algorithm 5.4.

[1]Some implementation may use an exclusive upper bound as $[l, u)$

Algorithm 5.4 Merge via range-based partitioning

```
1: Input: $T$                                   ▷ List of input SSTables to be merged
2: Input: $r_{out}$                             ▷ The output sorted run
3: $B$ = ComputeBounds($T$)                     ▷ Sorted list of pairs of lower and upper bounds
4: $iter$ = Iterator($T$)                       ▷ Merging iterator on all SSTables in $T$
5: $t$ = NewTable()                             ▷ Create one SSTable
6: $i = 1$
7: while $iter$.HasNext() do
8:     $\langle k, v \rangle$ = $iter$.GetNext()      ▷ Get key and value, then advance the iterator
9:     if $k > B[i]$.GetUpperBound() then       ▷ This key belongs to the next SSTable
10:        $r_{out}$.Add($t$)                    ▷ Complete $t$ and add to $r_{out}$
11:        $t$ = NewTable()                      ▷ Create another SSTable and reset $t$
12:        $i = i + 1$                           ▷ Process the next upper bound
13:    end if
14:    $t$.Write($k, v$)
15: end while
16: $r_{out}$.Add($t$)                           ▷ Complete the last $t$ and add to $r_{out}$
```

Key Bounds Computation

To compute the key bounds B (Algorithm 5.4 line 3), Local-Range and Global-Range partitioning both use the same BinarySplit function, as described in Algorithm 5.5.

The basic idea of this function is to keep splitting at the mean of a range and stop when the approximate data size of a sub-range is no greater than θ. Unlike Size

Algorithm 5.5 Function BinarySplit

```
1: function BinarySplit($(l, u)$)                          ▷ $l \leq u$
2:     $s = $ SizeInRange$(l, u)$        ▷ Approximate data size between $l$ and $u$
3:     if $s = 0$ then
4:         return [ ]
5:     else if $s \leq \theta$ then
6:         return $[(l, u),]$
7:     else
8:         $m = \lfloor \frac{l+u}{2} \rfloor$                          ▷ Mean of $l$ and $u$
9:         return BinarySplit$(l, m)$ + BinarySplit$(m + 1, u)$    ▷ List concatenation
10:    end if
11: end function
```

partitioning which aims at creating partitions of the same data size, this function aims at creating partitions of the same key range size. Next, we discuss the Local-Range and Global-Range partitioning with the help of BinarySplit in the paragraphs below.

Local-Range Partitioning Intuitively, we can partition the local key range of SSTables being merged with BinarySplit. By doing so, we can ensure the two sub-ranges' sizes are equal after a split. If keys are uniformly distributed in those SSTables, the mean and median of the key range will likely be close to each other, making almost perfect partitioning in both data size and key range size. The pseudocode is shown in Algorithm 5.6.

As we can see in Figure 5.3 (b), SSTables are almost symmetrical in both data size and key range size. It still has the chaining problem just like Size partitioning. The major

Algorithm 5.6 Function ComputeBoundsLocalRange

```
1: function ComputeBoundsLocalRange($T$)    ▷ $T$: List of SSTables in a sub-merge
2:     $k_{min}$ = GetMinKey($T$)
3:     $k_{max}$ = GetMaxKey($T$)
4:     return BinarySplit($k_{min}, k_{max}$)
5: end function
```

issue is that this method does not guarantee high quality bounds to reduce the possibility of overlapping SSTables.

Global-Range Partitioning In Local-Range partitioning, we use the actual key range of a sub-merge as the reference to create the partition upper bounds. However, it fails to create fixed bounds which still leads to many overlapping SSTables. To overcome this problem, we can use the maximum key space as the reference to create the partition upper bounds. By doing a binary split from the maximum key space, we can ensure the mean values are always in a fixed set.

An example of Global-Range partitioning is illustrated in Figure 5.4, which shows the execution steps of Sub-Merge 2 in Figure 5.1 with a key space of 8 bits (e.g, data type `uint8_t`). As we can observe from the figure, the key range size of a node in the binary tree at level i ($i \geq 0$, root is at level 0) is always 2^{-i} of the total key space size (256 here). For any two SSTables t_1 and t_2 ($t_1 \neq t_2$), let n_1 and n_2 denote the lowest nodes in a *full* binary tree that fully contain t_1 and t_2 respectively. If n_1 is an ancestor or a descendent of n_2 (n_1 and n_2 can be the same node), this indicates t_1 and t_2 are overlapping, so that they must be merged. Otherwise, n_1 and n_2 have a lowest common ancestor that is different from them, this indicates t_1 and t_2 are non-overlapping, so that they have a higher chance to be trivial-moved.

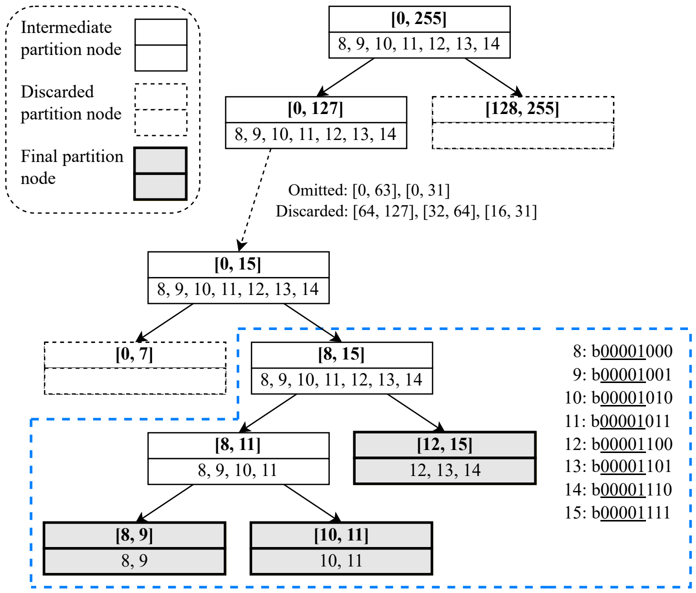


Figure 5.4: Binary tree for Global-Range partitioning.

Although this method can generate better quality partitions, it is computationally expensive. The size of the binary tree can be large with many levels. We observe that many recursions are spent on discarding nodes in the beginning and the end of the whole key space, such as node $[0, 7]$ and $[128, 255]$. When the key space is large, this becomes a waste of resources. To make the algorithm run faster, we must make the binary tree smaller. We notice that node $[8, 15]$ is the first node that splits to two non-empty nodes. We can directly create a binary tree rooted at node $[8, 15]$ for Global-Range partitioning, as shown in the area in blue dashed lines. By looking at the binary representations of all possible values of this node, we can easily see they all share a common binary prefix b$\underline{00001}*$. To obtain the root node in the smaller tree, we can compute the longest binary prefix (b$\underline{00001}*$) of the smallest key (8: b<u>00001</u>000) and the largest key (14: b<u>00001</u>110) to be partitioned, the root node must have the same binary prefix, with $*$ replaced by all 0s (8: b<u>00001</u>000) and 1s (15: b<u>00001</u>111) as the key range. This completes the final Global-Range partitioning in Algorithm 5.7.

Algorithm 5.7 Function ComputeBoundsGlobalRange

```
1:  function ComputeBoundsGlobalRange($T$)    ▷ $T$: List of SSTables in a sub-merge
2:      $b_{min}$ = ToBinary(GetMinKey($T$))
3:      $b_{max}$ = ToBinary(GetMaxKey($T$))
4:      for $i = 1 \to |b_{min}|$ do    ▷ $|b_{min}|$: Total number of bits
5:          if $b_{min}[i] \neq b_{max}[i]$ then
6:              $f = i$    ▷ $f$: The position of the first distinct bit
7:              break
8:          end if
9:      end for
10:     for $i = f \to |b_{min}|$ do
11:         $b_{min}[i] = 0$    ▷ Unset the bit
12:         $b_{max}[i] = 1$    ▷ Set the bit
13:     end for
14:     $l$ = ToUInt($b_{min}$)    ▷ Convert binary string to unsigned integer
15:     $u$ = ToUInt($b_{max}$)    ▷ Convert binary string to unsigned integer
16:     return BinarySplit($l, u$)
17: end function
```

As indicated in Figure 5.3 (c), Global-Range partitioning has a higher chance to create non-overlapping SSTables, leading to more trivial-moves and sub-merges, thus the merge throughput can be improved.

5.4.4 Challenges in Implementation

In the implementations of Local-Range and Global-Range partitioning, we face two major challenges. The first is that we need to find an efficient and effective way to compute the data size within a range without having too much disk I/O. The second is that, Global-Range partitioning assumes unsigned integer keys such as `uint64_t`, we must use a hash function to convert a key (mainly variable-length byte string) to an unsigned integer to make binary space partition. To support range queries, this hash function must be order-preserving.

Data Size Estimation via StreamHist

To make the most accurate partitions that respect the maximum SSTable size limit θ, we can build a binary tree like the one in Figure 5.4 and place all the key value pairs in the nodes. However, this requires excessive disk I/O and space, which contradicts the purpose of improving the write performance. To obtain the approximate data size within a range, we can use a sample-based method to maintain some statistics of keys for every SSTable. In this work, we use the C++ `StreamHist` implementation [82] of Ben-Haim's Streaming Parallel Decision Trees [12] to maintain a histogram for each SSTable during its construction. The histograms will be then used in the SizeInRange function in Algorithm 5.5 line 2, which gives very good accuracy with minimum overhead.

Algorithm 5.8 shows the implementation of SizeInRange function using `StreamHist`. Note the Sum function of `StreamHist` computes the estimated size from $-\infty$ to the function argument value. In our implementation, we use a hash table to cache the Sum results as they may be reused many times in the recursions of BinarySplit.

Algorithm 5.8 Data size estimation in range

```
1: $h_m$ = NewStreamHist()                         ▷ An empty StreamHist
2: for $t$ in $T$ do                                ▷ $t$: SSTable
3:     $h_t = t$.GetStreamHist()
4:     $h_m$.Merge($h_t$)                           ▷ Merge $h_t$ into $h_m$
5: end for
6: function SizeInRange($l, u$)
7:     return $h_m$.Sum($u$) − $h$.Sum($l$)
8: end function
```

We modify the implementation of `StreamHist` by removing unnecessary attributes and methods with some optimizations for the speed. The space overhead of `StreamHist` is minimum. In our implementation, a `StreamHist` of 64-bit double type with 128 bins occupies about 2 kB in memory and less space on disk (some attributes can be shared from the SSTable's metadata, compression can further reduce the size). In a database of 1000 SSTables, the total space overhead is about 2 MB only.

Because a `StreamHist` must be maintained for every SSTable, every single key written to an SSTable during flushes and merges must be written to the `StreamHist` as well, which incurs some overhead in time. In our testing (Figure 5.5), an unweighted

StreamHist with 128 bins has an average of 300 nanoseconds or less, and the weighted version has an average of 4 microseconds or less, for a single insertion regardless of the data type (`double` or `long double`) used.

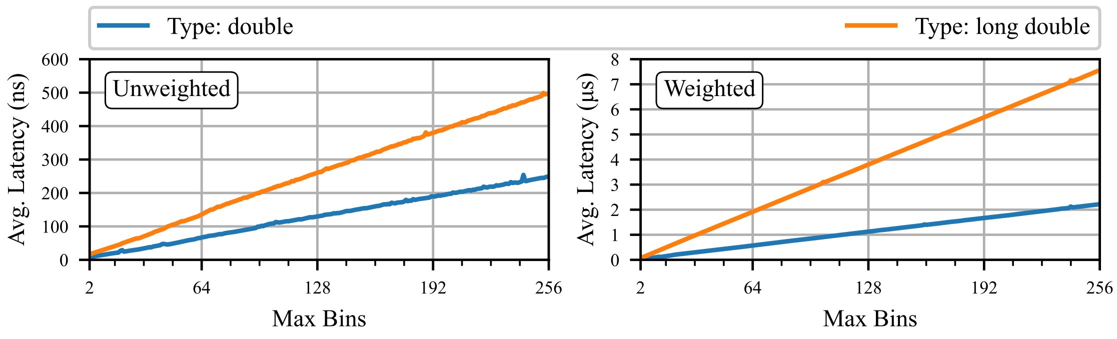


Figure 5.5: Average insertion latency in `StreamHist`.

Order-preserving Hash Function

While Local-Range partitioning does not require unsigned integer type of keys, it still needs the `StreamHist` which requires numerical data for the estimation of data sizes. Moreover, Global-Range partitioning also requires unsigned integer type of keys. For non-numerical type of keys such as variable-length byte strings[2], we apply a hash function that converts a key to `uint64_t` and stores this value in the `StreamHist`. If range query support is not required, we can use any standard hashing function for this purpose. Also, the SSTables will likely use hash indexes instead of binary-search-based indexes (e.g., B^+-tree). To support range queries, we propose an order-preserving hash function for the conversion from variable-length byte string to `uint64_t`.

[2]Numerical types like singed integers and floating numbers can be easily one-to-one mapped to unsigned integers while preserving the order.

In general, due to the Pigeonhole principle [3], it is impossible to convert an arbitrary string of variable-length into a unique unsigned integer, because the original key space is almost infinite, while the key space size of `uint64_t` is only 2^{64}. For an arbitrary length byte string s using the whole extended ASCII characters (0x00—0xFF), its mapped `uint64_t` value via $f(s) = \lambda(s)\left(2^{64}-1\right)$, where $\lambda(s)$ is a real number scalar in $[0,1)$. Mathematically, we want $\lambda(s_1) < \lambda(s_2)$ if and only if $s_1 < s_2$. Define a function g such that $\lambda(s) = \sum_{i=1}^{|s|} \mathrm{g}(s[i])$, where $s[i]$ is the i-th byte value between 0 and 255. g must satisfy the following five constraints.

1. $0 < \mathrm{g}(s[i]) < 1$, otherwise $\lambda(s)$ may be greater than 1.

2. Let $s[1\to i]$ be a sub-string of s from byte 1 to byte i inclusively. $\mathrm{g}(s[1\to i]) < \mathrm{g}(s[1\to i+1])$ for all $1 \leq i \leq |s|-1$. This gives more weights on the more significant bytes.

3. $\mathrm{g}(s[i]) > 0$ for all $0 \leq s[i] \leq 255$. This ensures Constraint 2. and distinguishes the case with no byte at position i $(i > |s|)$.

4. $\mathrm{g}(s_1[i]) < \mathrm{g}(s_2[i])$ for any two bytes $0 \leq s_1[i] < s_2[i] \leq 255$ to maintain the order of strings.

5. For strings $s_1 < s_2$, let k be the first position of where they differ, that is $s_1[1\to k-1] = s_2[1\to k-1]$ and $s_1[k] < s_2[k]$. We must guarantee $\mathrm{g}(s_1[k]) + \sum_{i=k+1}^{|s_1|} \mathrm{g}(s_1[i]) < \mathrm{g}(s_2[k])$. The remaining part of s_2 is truncated because $s_2[1\to k]$ is the smallest possible string starting with this prefix.

Let us assume the function $\mathrm{g}(s[i]) = \frac{s[i]}{x^i}$, where x is a variable to be determined. Doing so ensures Constraint 3. since $0 \leq s[i] \leq 255$. To ensure 1. and 4., we just need to make $x > 1$. Plug this formula into the inequation in Constraint 5., we just need to find a value of x that satisfies Inequation 5.1:

$$\frac{s_1[k]+1}{x^k} + \sum_{i=k+1}^{|s_1|} \frac{s_1[i]+1}{x^i} < \frac{s_2[k]+1}{x^k}, \forall s_1[k] < s_2[k] \tag{5.1}$$

Transforming 5.1 we get Inequation 5.2

$$\sum_{i=k+1}^{|s_1|-1} \frac{s_1[i]+1}{x^i} < \frac{s_2[k]-s_1[k]}{x^k} \leq \frac{1}{x^k}, \forall s_1[k] < s_2[k] \tag{5.2}$$

The maximum value of $\sum_{i=k+1}^{|s_1|} \frac{s_1[i]+1}{x^i}$ happens when $|s_1| = \infty$ and all $s_1[i] = 255$, then we have

$$\begin{aligned} \max\left(\sum_{i=k+1}^{|s_1|} \frac{s_1[i]+1}{x^i}\right) &\leq \sum_{i=k+1}^{|s_1|} \frac{255+1}{x^i} \\ &= \frac{256}{x^{k+1}} \times \left(\frac{1-x^{-(|s_1|-k)}}{1-x^{-1}}\right) \\ &< \frac{256}{x^{k+1}} \times \left(\frac{1-x^{-\infty}}{1-x^{-1}}\right) \\ &= \frac{256}{x^{k+1}} \times \left(\frac{1}{1-x^{-1}}\right), \forall x > 1 \end{aligned} \tag{5.3}$$

Now we have $\frac{256}{x^{k+1}} \times \left(\frac{1}{1-x^{-1}}\right) \leq \frac{1}{x^k}$, which is $x\left(1-x^{-1}\right) \geq 256$. Solving this inequation we get $x \geq 257$. The final formula for λ_s is shown in Equation 5.4.

$$\lambda(s) = \sum_{i=1}^{|s|} \frac{s[i]+1}{257^i} \tag{5.4}$$

For an empty string, its $\lambda(s) = 0$. For any non-empty string, $\text{MIN}\left(\lambda(s)\right) = \frac{0+1}{257^1} = \frac{1}{257}$, where the smallest string contains a single byte '0x00'. $\text{MAX}\left(\lambda(s)\right) = \sum_{i=1}^{\infty} \frac{255+1}{257^i} = \frac{256}{257} \times \left(\frac{1}{1-257^{-1}}\right) = 1$, where the largest string is an infinity length string with all '0xFF's. Therefore, $\lambda(s) < 1$ is true for any *finite* length string.

Although this hash is a one-to-one mapping from any string to a unique real number in [0, 1), storing the real number in a floating type in the computer will lose its precision, and incurs rounding errors when converting this floating number to an unsigned integer. There can be collisions (two different strings hashed to the same integer) when two different strings are mapped to the same unsigned integer. We cannot maintain the strict relation $f(s_1) < f(s_2)$ for any strings $s_1 < s_2$, but a weak relation $f(s_1) \leq f(s_2)$ is still guaranteed. In practice, this hash function has an exceptionally low collision rate. In our testing, we find no collision in 100,000,000 random variable-length byte strings. However, the collision rate starts increasing for strings with certain common prefixes. Table 5.1 summarizes the collision rates of strings with common prefixes. In practice, we can apply the hash function on the suffixes to reduce the collision rate.

$\boldsymbol{l} =$ \ $\boldsymbol{p} =$	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$\boldsymbol{p+1}$	0	0	0	0	0	0.99+	0.99+
$\boldsymbol{p+2}$	0	0	0	0	0.69	0.99+	0.99+
$\boldsymbol{p+3}$	0	0	0	0.13	0.96	0.99+	0.99+
$\boldsymbol{p+4}$	0	0	5.1e-4	0.17	0.96	0.99+	0.99+
$\boldsymbol{p+5}$	0	2e-6	5.9e-4	0.18	0.97	0.99+	0.99+
$\boldsymbol{p+6}$	0	2.6e-6	6.6e-4	0.17	0.97	0.99+	0.99+
$\boldsymbol{p+7}$	0	1.8e-6	6.9e-4	0.16	0.98	0.99+	0.99+

(a) Type: `double` (64-bit)

$\boldsymbol{l} =$ \ $\boldsymbol{p} =$	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$\boldsymbol{p+1}$	0	0	0	0	0	0.93	0.99+
$\boldsymbol{p+2}$	0	0	0	0	0	0.93	0.99+
$\boldsymbol{p+3}$	0	0	0	0	1.8e-3	0.93	0.99+
$\boldsymbol{p+4}$	0	0	0	6.9e-6	0.03	0.93	0.99+
$\boldsymbol{p+5}$	0	0	0	1.2e-4	0.03	0.93	0.99+
$\boldsymbol{p+6}$	0	0	4.0e-7	1.2e-4	0.03	0.93	0.99+
$\boldsymbol{p+7}$	0	0	4.0e-7	1.2e-4	0.03	0.93	0.99+

(b) Type: `long double` (80/128-bit)

Table 5.1: Average collision rates of byte strings with common prefixes. Columns are the prefix lengths (p), rows are the string lengths (l). Collision rates remain unchanged for larger p or l.

We also test the performance of this hash function. As shown in Figure 5.6, the overhead of hashing a byte string is also minimum (less than 90 nanoseconds for 128-byte long random strings). Also we notice that the performance is about the same for both `double` and `long double` types. To achieve better precision with lower collision rate, we use `long double` in the implementation of this hash function.

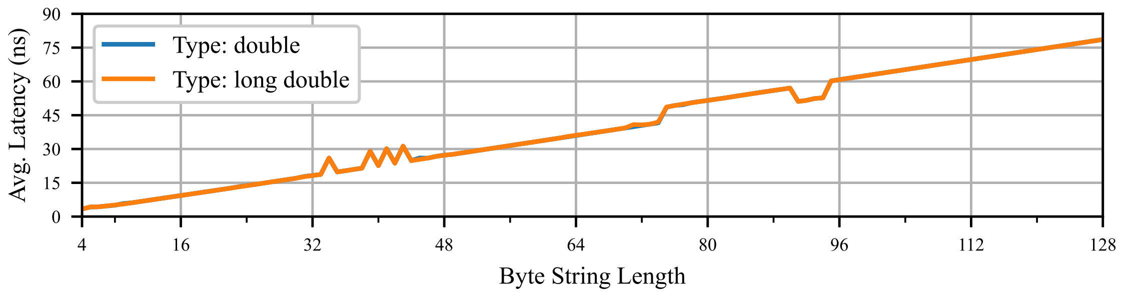


Figure 5.6: Average latency of the order-preserving hash function for variable-length strings. Average latency per byte is 0.63 nanoseconds for both `double` and `long double` types.

5.5 Experiments

5.5.1 Experimental Setup

We implement the proposed partitioning algorithms for UniversalCompaction in RocksDB, and compare the four partitioning algorithms (No-partition, Size, Local-Range and Global-Range) with the default Leveled merge policy. We also make the following optimizations to RocksDB:

1. Allow trivial-moves within a merge, such that some SSTables can be trivial-moved and the others can be sub-merged.

2. Sub-merges are executed in a thread-pool, instead of having a single thread for every sub-merge. This is because the proposed partitioning algorithms are able to create many (over 60) sub-merges in certain workloads, which may overload the system with too many threads.

3. Allow multiple SSTables to form a single sorted run in level 0. Binary search is also made available in level 0 on SSTables in the same sorted run to find candidate SSTables for a given `Get` or `Iterator` operations. This is more efficient a linear search on all SSTables in level 0.

Basic Experimental Settings

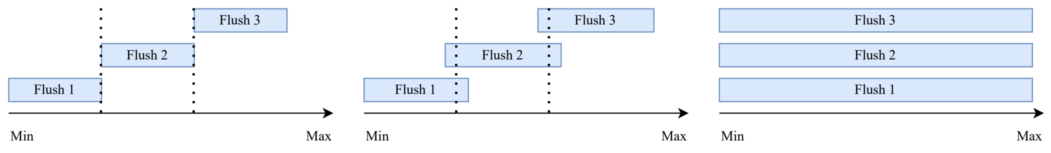
To compare and evaluate the write and read performance of these four partitioning algorithms, we conduct two sets of experiments as shown in Table 5.2. The UInt experiment set has three different distributions to insert keys. The distributions have a significant impact on the performance of the two proposed partitioning algorithms. All runs in the UInt and Str experiments have a loading phase of 41,943,040 key-value pairs (about 40 GB). They also have 100,000 reads and 10,000 range scans of size 1000 after the loading phase. 10,000 keys are used to warm the table cache before doing any reads and scans. UInt runs do not need a hash function since the keys are in the binary format of `uint64_t`. The hash function described in Section 5.4.4 is used in the Str runs.

We did not include experiments that use a sequential key distribution. This is because in a sequential distribution, all flushed SSTables are non-overlapping, making all SSTables in the database strictly non-overlapping with each other. No actual merge will

Experiment	**Key Data Type**	**Key Length**	**Value Length**	**Total KV Length**	**Key Distribution**
UInt	Byte string of 64-bit unsigned integer	8	1016	1024	Unique random
					Semi-sequential
					Zipf ($q = 0.5$)
Str	Byte string	[4, 32]	[992, 1020]	1024	Random length, random bytes

Table 5.2: Summary of experiments.

be performed but only trivial moves, thus all partitioning algorithms take no effect in this case. The semi-sequential distribution is similar to a pure sequential distribution, except two adjacent flushed SSTable may have a small overlapping range. This can happen when keys are generated using an incremental counter, and they are written to the database asynchronously. Figure 5.7 show an example of three key distributions for the first three flushed SSTables. The key range sizes in (c) extend to the whole key space.



(a) Pure Sequential (b) Semi-sequential (c) Random

Figure 5.7: UInt key distributions.

RocksDB Options

Most options in RocksDB are kept default. Non-default options are listed in Table 5.3. *num_levels* is set to 1 in UniversalCompaction to force placing SSTables in level 0 and create unpartitioned sorted runs with No-partitioning, in order to simulate the original SizeTiered merge policy. *level0_slowdown_writes_trigger* and *level0_stop_writes_trigger* are adjusted to limit the maximum number of sorted run in the database. These two options

Option	**UniversalCompaction**	**LevelCompaction**
max_subcompactions	16	
max_compaction_bytes	Unlimited	
target_file_size_base	64 MB	
num_levels	1	4
level0_slowdown_writes_trigger	8	5
level0_stop_writes_trigger	9	6
compaction_options_universal.allow_trivial_move	`true`	N/A
compaction_options_universal.min_merge_width	4	N/A
compaction_options_universal.stop_style	`kCompactionStopStyleSimilarSize` `kCompactionStopStyleTotalSize`	N/A

Table 5.3: RocksDB options.

and *num_levels* are also adjusted in LevelCompaction to have the same maximum number of sorted runs in the database. LevelCompaction does not have a partitioning option set, thus it is default to Size partitioning. Maximum SSTable size constraint θ is set to 64 MB using *target_file_size_base* except for the No-partitioning runs. In all the runs with No-partitioning and Size partitioning, `StreamHist` is disabled so they do not have the overhead of converting a key to `uint64_t` and writing to `StreamHist`. Runs with Local-Range and Global-Range partitioning are configured to use unweighted `StreamHist`s of 128 bins[3].

Test Platform

All experiments are done on a AWS EC2 *c5.2xlarge* instance with 8 cores and 16 GB memory. The instance uses EBS *gp3* storage. As shown in Figure 5.8, the peak write

[3]The weighted version improves the data size estimation marginally with a larger time overhead.

throughput is around 725 MB/s using 6 to 8 threads. Having more threads does not improve the write throughput and will further decrease the write throughput per each thread.

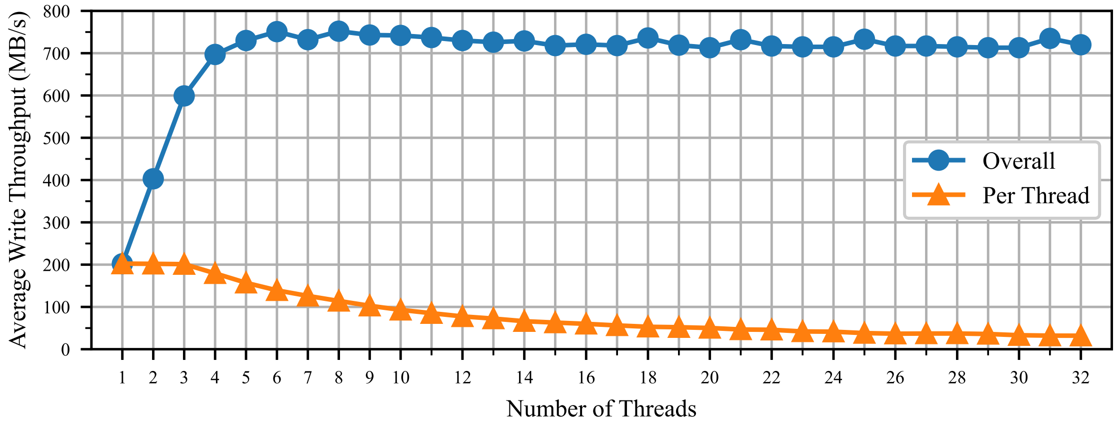


Figure 5.8: Multi-threaded write throughput on EBS *gp3* storage

5.5.2 Experimental Results with Unsigned Integer Keys

Semi-sequential Key Distribution

In a semi-sequential key distribution, any two adjacent sorted runs will overlap with each other. UniversalCompaction with No-partitioning cannot schedule any trivial-moves since all sorted runs contain a single SSTable. On the other hand, any partitioned runs, including LevelCompaction, have similar write performance (Figure 5.9) as most SSTables can be trivial-moved.

Merges with Size partitioning are slightly faster than Local-Range and Global-Range partitioning (Figure 5.10). This is mainly because it does not have the overhead of writing to `StreamHist`, and it has fewer SSTables to merge and trivial-move (Figure 5.11).

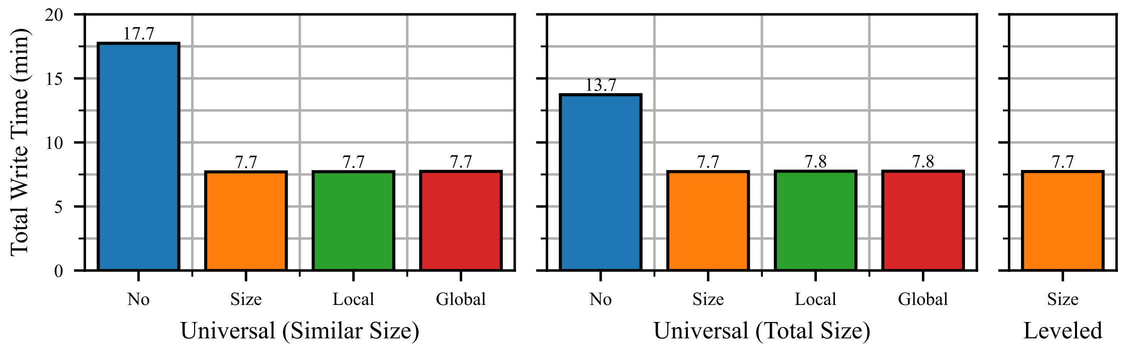


Figure 5.9: Semi-sequential UInt: Total write time.

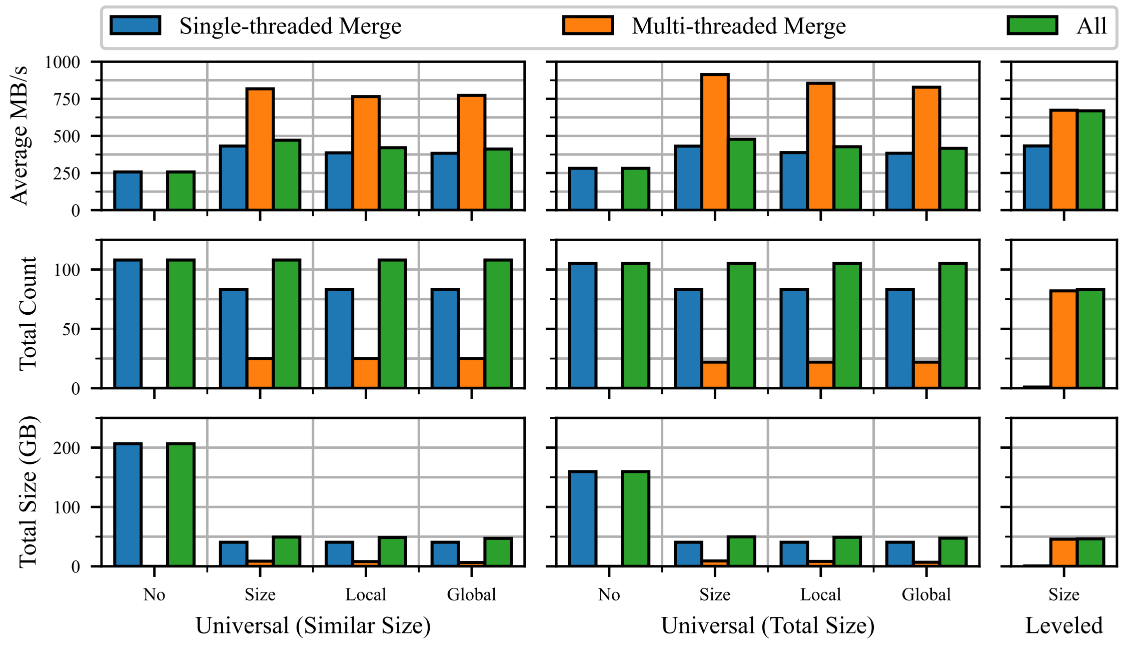


Figure 5.10: Semi-sequential UInt: Merge statistics.

When it comes to read performance, Size partitioning is the winner (Figure 5.12). The tree partitioning algorithms all achieve the lowest `Get` latencies because of smaller file size. In terms of `Iterator` performance, Size partitioning is only second to No-partitioning, which has much fewers files to scan.

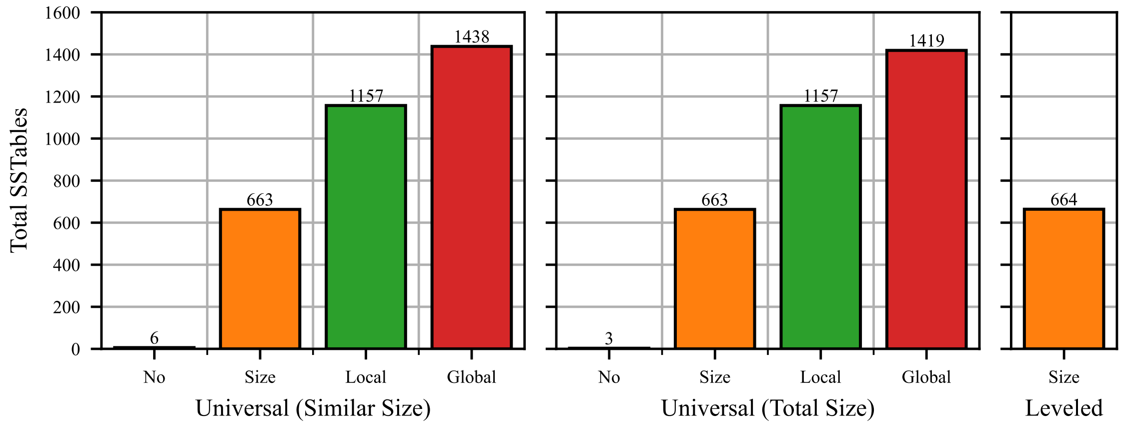


Figure 5.11: Semi-sequential UInt: SSTable statistics.

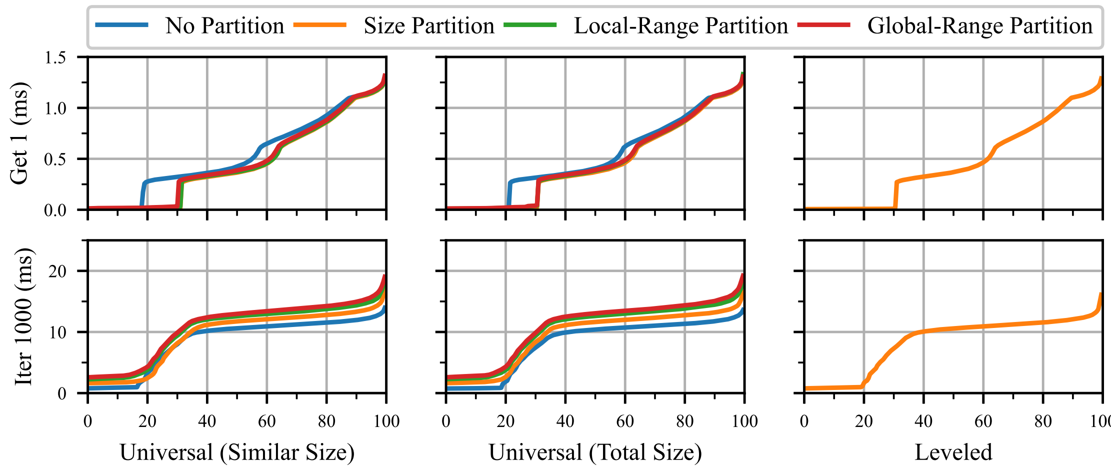


Figure 5.12: Semi-sequential UInt: Read performance.

Random Key Distribution

In a random key distribution, all sorted runs have a similar key span over the whole key space $[0, 2^{64}-1]$. Global-Range partitioning achieves 18% and 7% improvement over the No-partitioning and Size partitioning using similar size stop style and total size stop style respectively (Figure 5.13). Size partitioning is even slower than No-partitioning because no sub-merges can be scheduled, and it has extra overhead to scan multiple SSTables.

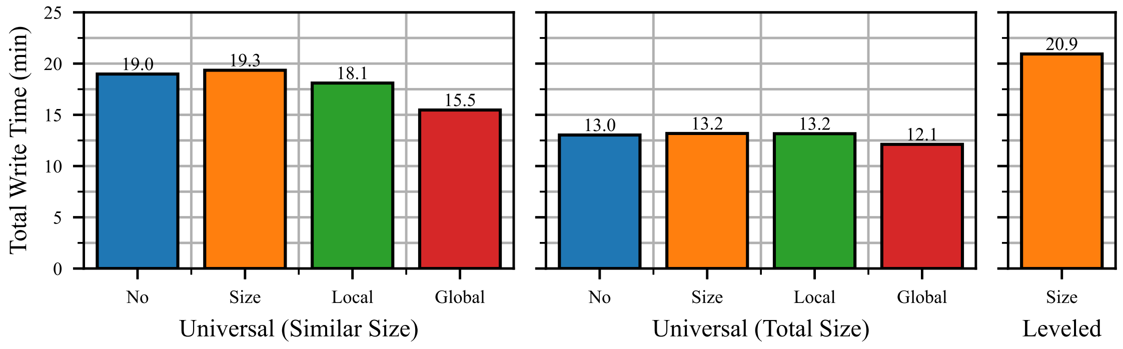


Figure 5.13: Random UInt: Total write time.

Global-Range partitioning has the highest merge throughput, and Local-Range partitioning is only second to it (Figure 5.14). These two partitioning algorithms can effectively process large merges with multi-threading. On the other hand, Size partitioning cannot schedule any sub-merge, thus behaves similarly to No-partitioning.

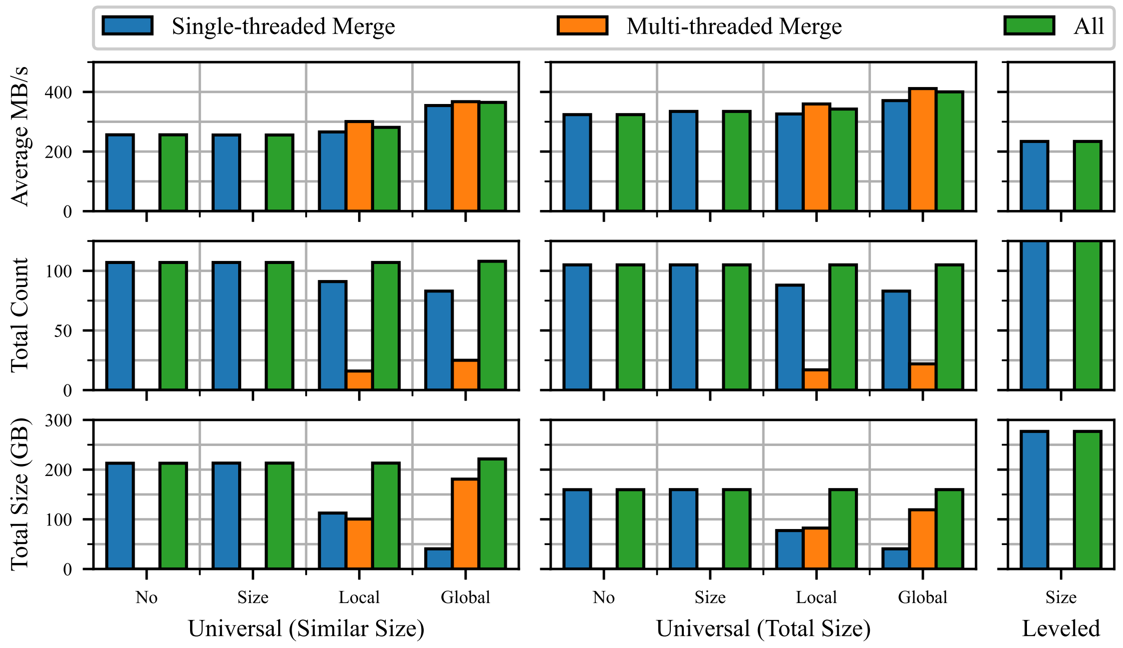


Figure 5.14: Random UInt: Merge statistics. Leveled has over 600 merges (truncated).

Even though Local-Range and Global-Range partitioning tend to create more SSTables (Figure 5.15), read performance is not affected much. As shown in Figure 5.16, all the four partitioning algorithms have about the same latency in both stop styles. No-partitioning is only a little bit faster still due to fewer SSTables to scan.

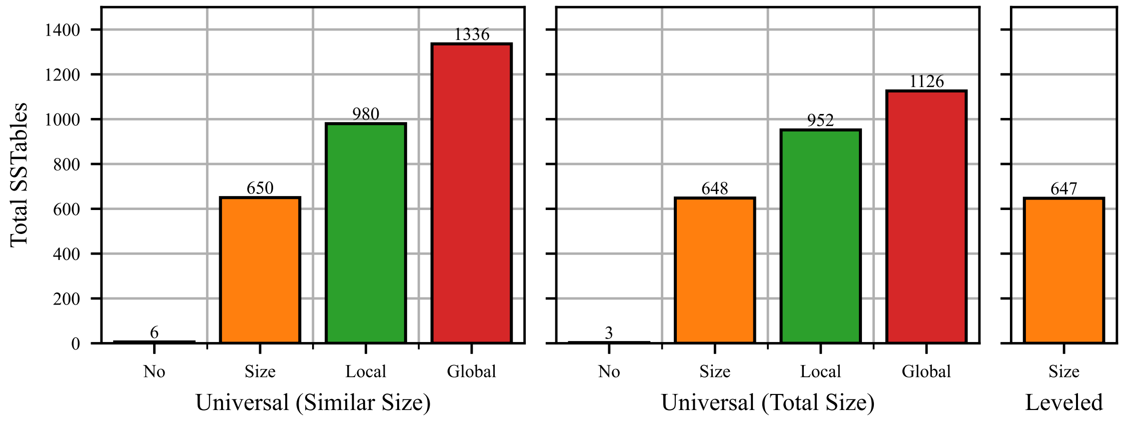


Figure 5.15: Random UInt: SSTable statistics.

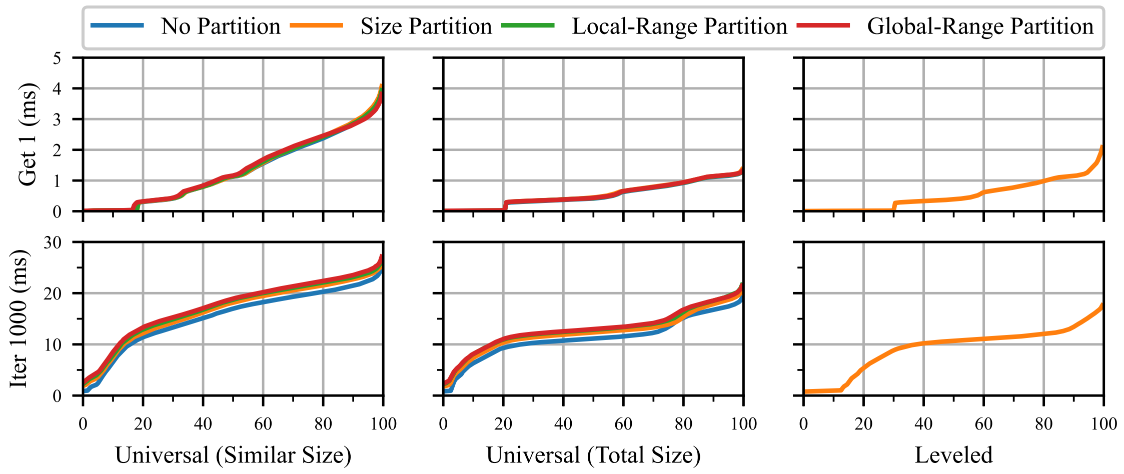


Figure 5.16: Random UInt: Read performance.

Skewed Key Distribution

We use a zipf distribution with $q = 0.5$ to generate a skewed distribution with unique keys. We observe almost the same write (Figure 5.17), merge (Figure 5.18) and read (Figure 5.19) performance as the random key distribution. The only difference is, fewer SSTables are created in this distribution using Local-Range and Global-Range partitioning. In the runs of random key distributions, compared to Size partitioning, Local-Range partitioning creates 50% more SSTables and Global-Range partitioning creates 100% more SSTables. While in the skewed key distribution, they both create 50% more SSTables, which is not a big overhead to maintain.

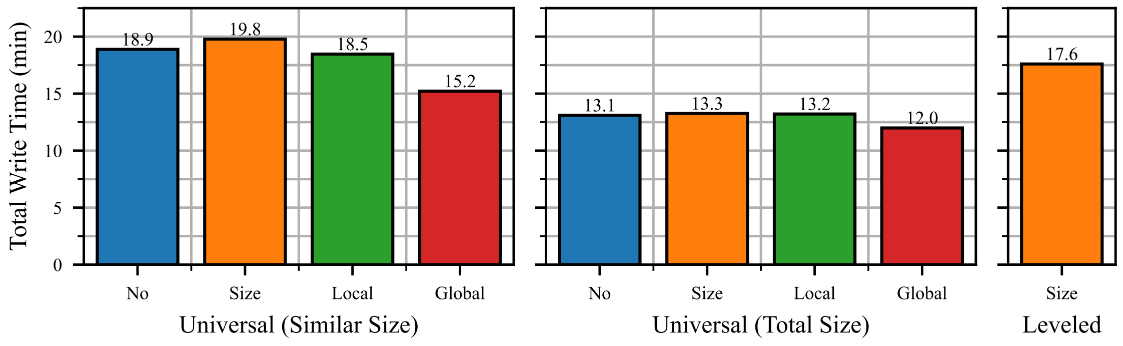


Figure 5.17: Skewed UInt: Total write time.

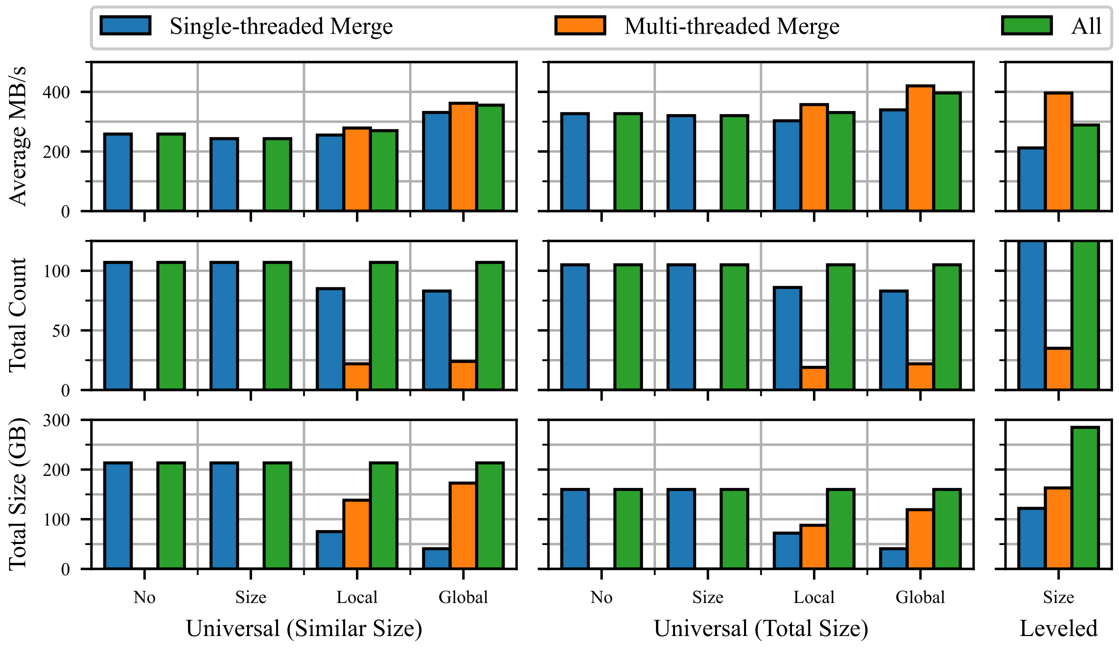


Figure 5.18: Skewed UInt: Merge statistics. Leveled has over 600 merges (truncated).

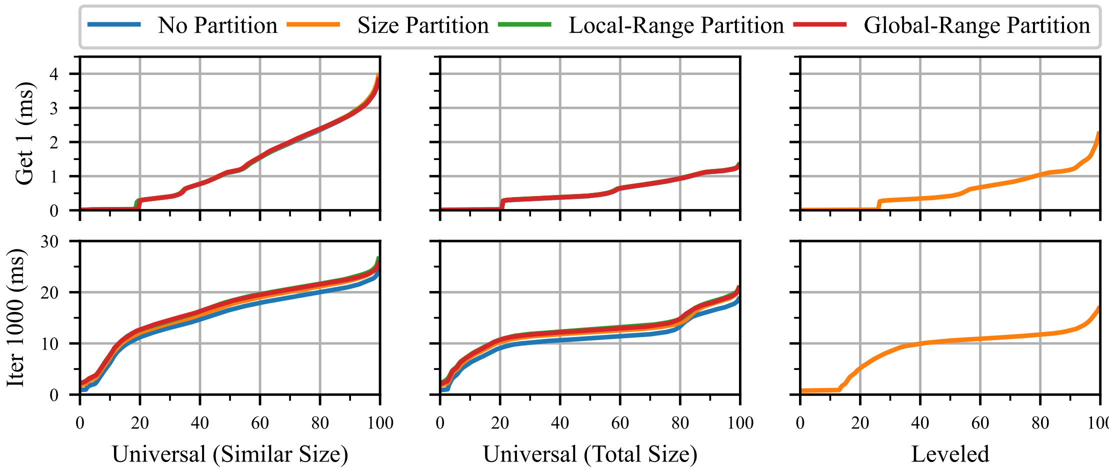


Figure 5.19: Skewed UInt: Read performance.

5.5.3 Experimental Results with Variable-length Byte String Keys

While random byte strings almost follow an uniform distribution similar to the random integers, the actual performance is different. Local-Range partitioning is only a few

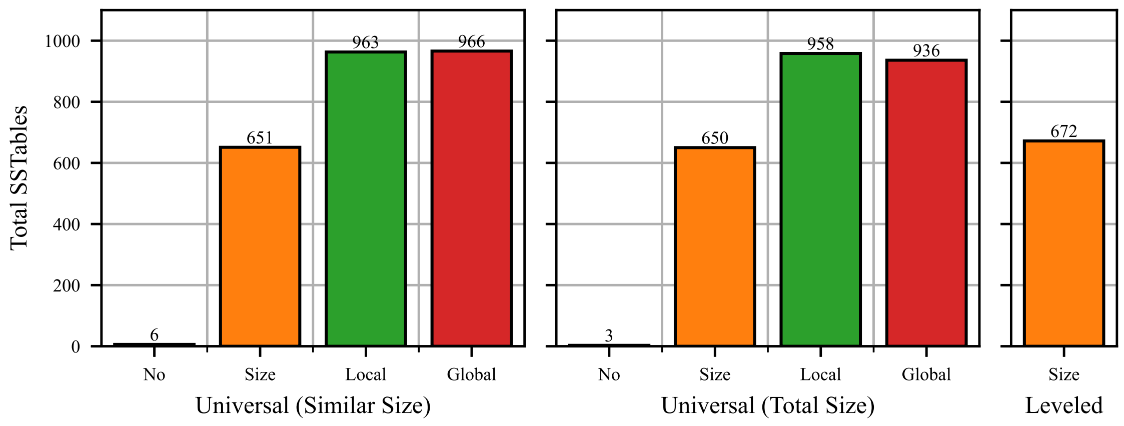


Figure 5.20: Skewed UInt: SSTable statistics.

seconds slower than the Global-Range partitioning in total write time (Figure 5.21). Their read performance is also very similar, as shown in Figure 5.22. Global-Range partitioning has slightly higher merge throughput (Figure 5.23), likely because it creates 10% to 20% more SSTables than Local-Range partitioning (Figure 5.24), leading to more sub-merges.

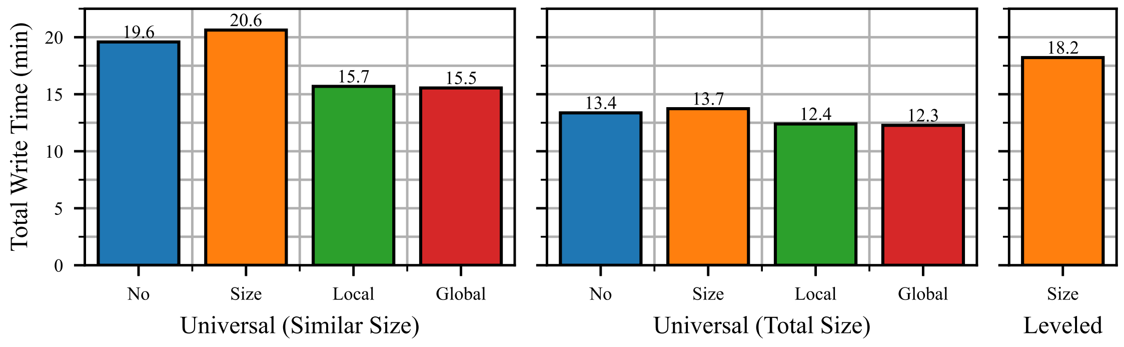


Figure 5.21: Byte String: Total write time.

Overall, both Local-Range and Global-Range partitioning outperform No-partitioning and Size partitioning in writes. No-partitioning has slightly better iteration performance

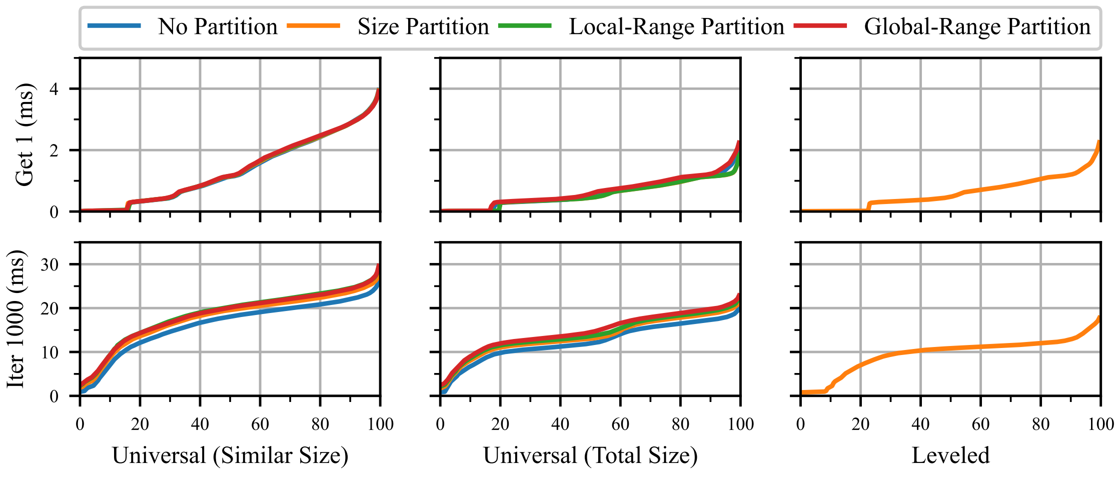


Figure 5.22: Byte String: Read performance.

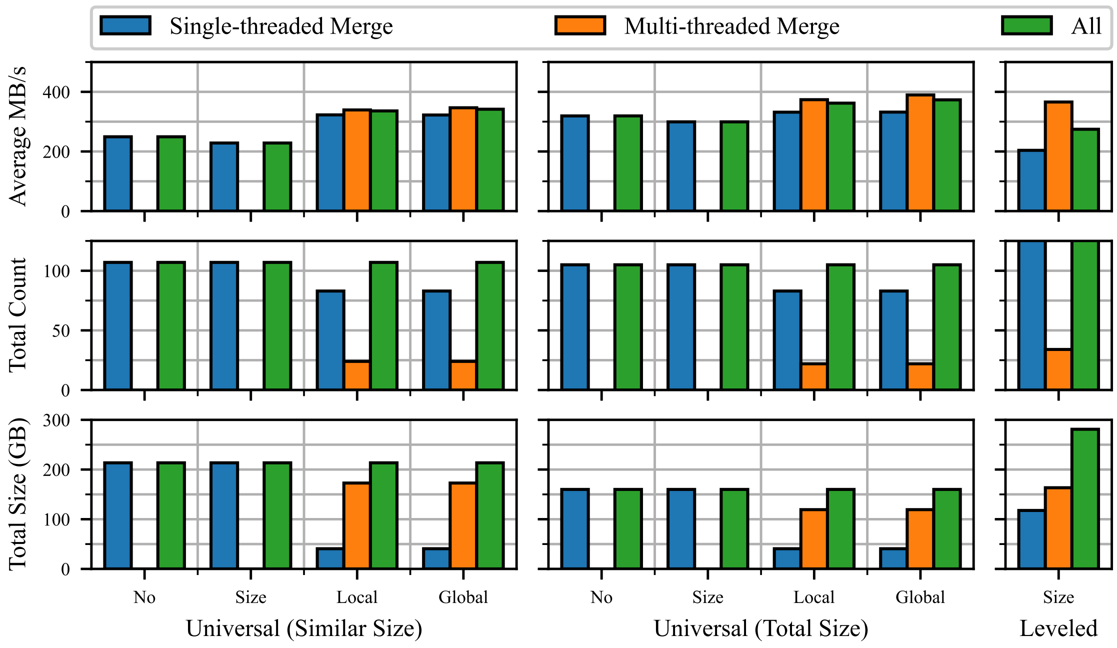


Figure 5.23: Byte String: Merge statistics.

due to fewer SSTables to scan. The other three partitioning algorithms have similar read performance in both get and iteration operations.

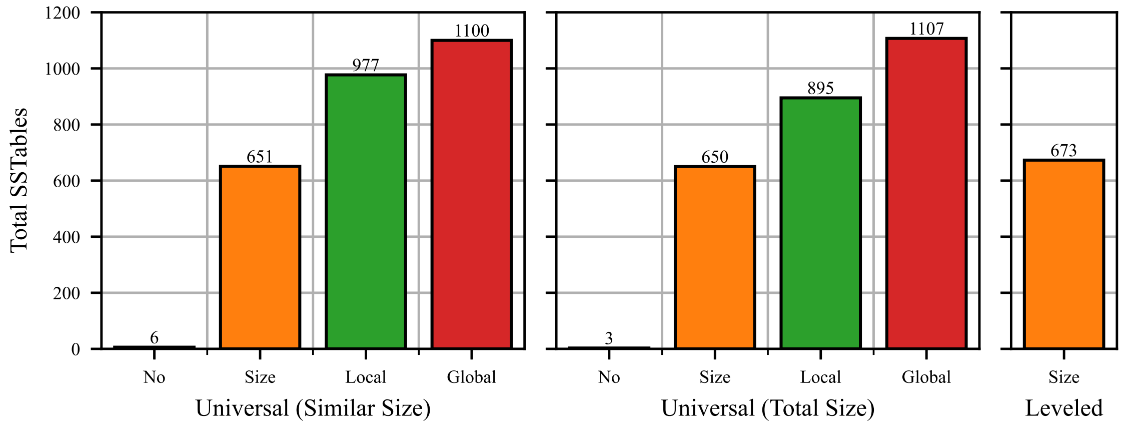


Figure 5.24: Byte String: SSTable statistics. Leveled has over 600 merges (truncated).

5.6 Discussion

The two proposed local-range and global-range partitioning algorithms can both effectively generate better quality partitions to increase the parallelism of LSM merges in most of the tested workloads. Global-range partitioning generally has the best merge throughput and achieves the highest write throughput, even with the overhead of maintaining `StreamHists`. The major problem with Global range partitioning is that it tends to create more SSTables, and some very small SSTables may exist, leading to higher cost to maintain the metadata of SSTables for sorted runs. Although this drawback does not affect the `Get` operation performance, `Iterator` performance will degrade with more files to scan. Local-range partitioning can also achieve high parallelism of LSM merges, and it creates fewer SSTables than global-range partitioning. When a system has many range queries, local-range partitioning may be a better option. There are two ways to reduce the number of SSTables created with local-range and global-range partitioning. Small partitions can be grouped together to create a larger partition. These partitions can be treated as virtual SSTables that share the same physical file (data, index and filters). A physical SSTable may keep a list of metadata of these virtual SSTables. Another way is to increase the maximum SSTable size limit θ to compensate. θ can be set to 1.5 or 2 times of the limit used in size partitioning, given the fact that the average SSTable size of local-range and global-range partitioning is $\frac{1}{2}$ and $\frac{2}{3}$ of that of size partitioning for the same θ. For certain special workloads where keys are pure sequential or semi-sequential, all partitioning algorithms have similar performance. Size partitioning may be a better choice for these workloads, as it creates the least number of SSTables, having the smallest memory and disk overhead.

Limitations and Future Work

Dynamic resource allocation As shown in Figure 5.8, having too many threads for merges may not effectively improve the overall write throughput. Threads are also used by other operations like flushes and user queries (`Get`, `Iterator`, etc.), there may not be enough CPU resources to execute many sub-merges. In this work, we limit the maximum number of sub-merges via an option as the hard limit. It would be better if an algorithm can make dynamic decisions for sub-merges that take the number of available CPU cores into consideration.

Data Size Estimation Currently, `StreamHist` is unable to handle updates. The precision of estimating data size will degrade significantly with lots of updates. Also, it is only used for the two proposed partitioning algorithms only. Given its capability of providing more statistics for SSTable contents, it is possible to use this information for a better merge policy. The requirement of converting a key to `uint64_t` may be removed if a better data structure can be used, which can reduce space overhead and provide better estimation of data sizes.

Hybrid stack-based and leveled merge policy With the partitioning for stack-based merge policies, the boundaries between stack-based and leveled merge policies become blurred. Transitions between these two types of merge policies are made easier, leading to the potential of a new hybrid policy. This new policy can have the ability to achieve high write throughput like the stack-based policies and keep the read latencies low like the leveled policies. It will be more tunable to adapt to dynamically changing workloads.

5.7 Related Work

Partitioning Algorithms Besides the default Size partitioning used in LevelDB and RocksDB, many partitioning algorithms were proposed mainly for the Leveled policy only. Some of them can also be used on stack-based policies like UniversalCompaction in RocksDB. The partitioning algorithm proposed in PE File [44] is application dependent. It generally applies a binary split at the median of keys, which can be viewed as a special version of Size partitioning. Zhang et al. [99] and SifrDB [69] use a similar partitioned stack-based design as the Size partitioning in LevelDB and RocksDB. The lightweight-compaction-tree [93, 94] uses vertical grouping for partitioning, similar to the one used in Leveled merge policy. Its partitioning also benefits load balancing across database nodes. PebblesDB [75] has a similar idea of range-based partitioning. It randomly picks an inserted key by some least-significant-bits from the hashed value as a guard for every range in a level. During merges, keys are written to the SSTables protected by the corresponding guard value. The Write-Buffer tree [6], LSM-trie [90] and HashKV [55] apply hash-based partitioning to distribute keys evenly into SSTables for workloads that do not need range query support. Most of the aforementioned work does not apply to stack-based merge policies, have limited support of range query, and may have the same chaining problem.

Optimizations for Range Query While partitioning a large sorted runs into set of smaller SSTables can generally improve the system's overall write performance, and has almost no impact on the single `Get` query performance, it makes medium to long range queries slower due to the fact that more files must be scanned. There is various existing research that target at improving range query performance in a partitioned LSM tree. REMIX [100] proposes a so-called Split Compaction, which splits data into multiple physical partitions by file sizes, to improve range query performance of LSM trees. Split compaction does not partition within a sorted run, but creates a collection of SSTables as a separate LSM tree. Surf [98] and Rosetta [63] both propose effective range query filters to reduce the number of SSTables to be checked to answer a query. Rosetta filter is constructed with multiple Bloom filters for every binary prefix of a range query. Since every single SSTable in Global-Range partitioning has a common binary prefix for all the keys, it can be combined with Rosetta and reduce the number of Bloom filters needed for every SSTable. It may also be possible to combine Rosetta filter with `StreamHist` to provide more accurate estimation of data size and faster computation speed to make Global-Range partitioning more efficient.

5.8 Conclusions

In this work, we propose two partitioning algorithms, named Local-Range partitioning and Global-Range partitioning, for stack-based merge policies. We compare these two partitioning algorithms with the existing Size partitioning, and No-partitioning on RocksDB. The experimental results show the proposed algorithms can improve the merge throughput for up to 30%, the overall write throughput for up to 20% over No-partitioning or Size partitioning using RocksDB's UniversalCompaction, while having the same or even better `Get` latency and less than 10% of higher `Iterator` latency.

Chapter 6

Conclusions

In this thesis, we have presented a various efficient storage design in LSM-tree databases.

In Chapter 2, we studied the fundamental I/O components of LSM systems, and two architectures with merge policies.

In Chapter 3, we studied five existing merge policies, BIGTABLE, CONSTANT, EXPLORING, TIERED and LEVELED, and comapred them with two newly proposed policies BINOMIAL and MINLATENCY using YCSB on a common platform AsterixDB. The results validated the the theoretical model and showed that the two proposed policies have substantially lower write amplification while maintaining a low read amplification.

In Chapter 4, we compared and evaluated secondary spatial index performance of both stack-based and leveled LSM architectures with four representative merge policies, on AsterixDB. The results from both the OpenStreetMap dataset and the synthetic random dataset have shown that BINOMIAL policy is probably the best candidate for LSM R-tree-based spatial index, although it is not specifically optimized for multidimensional spatial data. Based on our experimental results, we showed our recommendation for the choice of merge policy, comparator and partitioning algorithms depending on the workload and queries' selectivity.

Finally in Chapter 5, we proposed two novel range-based partitioning algorithms for stack-based merge policies. These two partitioning algorithms can effectively improve the parallelism during merges in a variety of different workloads, increasing the overall write throughput, while having minimum effect on the single get and iterator performance.

Bibliography

[1] Ahmad MY, Kemme B (2015) Compaction management in distributed key-value datastores. Proc VLDB Endow 8(8):850–861

[2] Ahn JS, Qader MA, Kang WH, Nguyen H, Zhang G, Ben-Romdhane S (2019) Jungle: towards dynamically adjustable key-value store by combining LSM-tree and copy-on-write B^+-tree. In: 11th USENIX Workshop on Hot Topics in Storage and File Systems

[3] Ajtai M (1994) The complexity of the pigeonhole principle. Combinatorica 14(4):417–433

[4] Alsubaiee S, Altowim Y, Altwaijry H, Behm A, Borkar V, Bu Y, Carey M, Cetindil I, Cheelangi M, Faraaz K, Gabrielova E, Grover R, Heilbron Z, Kim YS, Li C, Li G, Ok JM, Onose N, Pirzadeh P, Tsotras V, Vernica R, Wen J, Westmann T (2014) AsterixDB: A scalable, open source BDMS. Proc VLDB Endow 7(14):1905–1916

[5] Alsubaiee S, Behm A, Borkar V, Heilbron Z, Kim YS, Carey MJ, Dreseler M, Li C (2014) Storage management in AsterixDB. Proc VLDB Endow 7(10):841–852

[6] Amur H, Andersen DG, Kaminsky M, Schwan K (2013) Design of a write-optimized data store. Tech. rep., Georgia Institute of Technology

[7] Apache Software Foundation (2019) Apache Cassandra. URL `http://cassandra.apache.org`

[8] Apache Software Foundation (2019) Apache HBase. URL `https://hbase.apache.org`

[9] Apache Software Foundation (2020) Apache AsterixDB. URL `https://asterixdb.apache.org`

[10] Balmau O, Dinu F, Zwaenepoel W, Gupta K, Chandhiramoorthi R, Didona D (2019) SILK: Preventing latency spikes in log-structured merge key-value stores. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp 753–766

[11] Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R^*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD international conference on Management of data, pp 322–331

[12] Ben-Haim Y, Tom-Tov E (2010) A streaming parallel decision tree algorithm. Journal of Machine Learning Research 11(2)

[13] Bentley JL (1975) Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9):509–517

[14] Brahim MB, Drira W, Filali F, Hamdi N (2016) Spatial data extension for Cassandra NoSQL database. Journal of Big Data 3(1):11

[15] Cassandra (2019) How is data maintained? URL `https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlHowDataMaintain.html`

[16] Cattell R (2011) Scalable SQL and NoSQL data stores. SIGMOD Rec 39(4):12–27

[17] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: A distributed storage system for structured data. ACM Trans Comput Syst 26(2):4:1–4:26

[18] Chen F, Lee R, Zhang X (2011) Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture, IEEE, IEEE, pp 266–277

[19] Chen X, Zhang C, Ge B, Xiao W (2015) Spatio-temporal queries in HBase. In: 2015 IEEE International Conference on Big Data (Big Data), IEEE, pp 1929–1937

[20] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, SoCC '10, pp 143–154

[21] Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W, Kanthak S, Kogan E, Li H, Lloyd A, Melnik S, Mwaura D, Nagle D, Quinlan S, Rao R, Rolig L, Saito Y, Szymaniak M, Taylor C, Wang R, Woodford D (2013) Spanner: Google's globally distributed database. ACM Trans Comput Syst 31(3):8:1–8:22

[22] DataStax (2021) Geospatial queries for Point and LineString. URL `https://docs.datastax.com/en/dse/6.0/cql/cql/cql_using/search_index/queriesGeoSpatial.html`

[23] Dayan N, Idreos S (2018) Dostoevsky: Better space-time trade-offs for LSM-Tree based key-value stores via adaptive removal of superfluous merging. In: Proceedings of the 2018 International Conference on Management of Data, ACM, SIGMOD '18, pp 505–520

[24] Dayan N, Athanassoulis M, Idreos S (2017) Monkey: Optimal navigable key-value store. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD, pp 79–94

[25] DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W (2007) Dynamo: Amazon's highly available key-value store. ACM SIGOPS operating systems review 41(6):205–220

[26] Dent A (2013) Getting started with LevelDB. Packt Publishing Ltd

[27] Dong S, Callaghan M, Galanis L, Borthakur D, Savor T, Strum M (2017) Optimizing space amplification in RocksDB. In: CIDR, CIDR, vol 3, p 3

[28] Eldawy A, Mokbel MF (2015) SpatialHadoop: A MapReduce framework for spatial data. In: 2015 IEEE 31st international conference on Data Engineering, IEEE, pp 1352–1363

[29] Facebook, Inc (2020) RocksDB. URL `https://rocksdb.org`

[30] Facebook, Inc (2020) RocksDB Wiki: Compaction. URL `https://github.com/facebook/rocksdb/wiki/Compaction`

[31] Facebook, Inc (2020) RocksDB Wiki: Universal Compaction. URL `https://github.com/facebook/rocksdb/wiki/Universal-Compaction`

[32] Facebook, Inc (2022) RocksDB Wiki: Compaction trivial move. URL `https://github.com/facebook/rocksdb/wiki/Compaction-Trivial-Move`

[33] Galvizo G (2021) On indexing multi-valued fields in AsterixDB. Master's thesis, University of California, Irvine

[34] George L (2011) HBase: the definitive guide: random access to your planet-size data. O'Reilly Media, Inc.

[35] Ghosh S, Vu T, Eskandari MA, Eldawy A (2019) UCR-STAR: The UCR spatio-temporal active repository. SIGSPATIAL Special 11(2):34–40, DOI 10.1145/3377000.3377005

[36] Google LLC (2019) Bigtable. URL `https://cloud.google.com/bigtable`

[37] Google LLC (2019) LevelDB. URL `https://github.com/google/leveldb`

[38] Graefe G, et al (2011) Modern B-tree techniques. Foundations and Trends® in Databases 3(4):203–402

[39] Grover R, Carey MJ (2015) Data ingestion in AsterixDB. In: EDBT, OpenProceedings.org, pp 605–616

[40] Guttman A (1984) *r*-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on Management of data, pp 47–57

[41] Haklay M, Weber P (2008) OpenStreetMap: User-generated street maps. IEEE Pervasive Computing 7(4):12–18

[42] Huang S, Wang B, Zhu J, Wang G, Yu G (2014) R-HBase: A multi-dimensional indexing framework for cloud computing environment. In: 2014 IEEE International Conference on Data Mining Workshop, IEEE, pp 569–574

[43] Hughes JN, Annex A, Eichelberger CN, Fox A, Hulbert A, Ronquest M (2015) GeoMesa: a distributed architecture for spatio-temporal fusion. In: Geospatial Informatics, Fusion, and Motion Video Analytics V, International Society for Optics and Photonics, vol 9473, p 94730F

[44] Jermaine C, Omiecinski E, Yee WG (2007) The partitioned exponential file for database storage management. The VLDB Journal 16(4):417–437

[45] Judd D (2008) Scale out with HyperTable. Linux magazine, August 7th 1, URL `http://www.linux-mag.com/id/6645`

[46] Kamel I, Faloutsos C (1993) On packing *R*-trees. In: Proceedings of the second international conference on Information and knowledge management, pp 490–499

[47] Kepner J, Arcand W, Bestor D, Bergeron B, Byun C, Gadepally V, Hubbell M, Michaleas P, Mullen J, Prout A, et al (2014) Achieving 100,000,000 database inserts per second using Accumulo and D4M. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, IEEE, pp 1–6

[48] Khetrapal A, Ganesh V (2006) HBase and Hypertable for large scale distributed storage systems. Dept of Computer Science, Purdue University 10(1376616.1376726)

[49] Kim Y (2016) Transactional and spatial query processing in the big data era. PhD thesis, University of California, Irvine

[50] Kim Y, Kim T, Carey MJ, Li C (2017) A comparative study of log-structured merge-tree-based spatial indexes for big data. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), IEEE, pp 147–150

[51] Kuszmaul BC (2014) A comparison of fractal trees to log-structured merge (LSM) trees. Tokutek White Paper

[52] Lakshman A, Malik P (2010) Cassandra: A decentralized structured storage system. SIGOPS Oper Syst Rev 44(2):35–40

[53] Lee T, Lee S (2003) OMT: Overlap minimizing top-down bulk loading algorithm for *R*-tree. In: CAISE Short paper proceedings, vol 74, pp 69–72

[54] Leutenegger ST, Lopez MA, Edgington J (1997) STR: A simple and efficient algorithm for *R*-tree packing. In: Proceedings 13th International Conference on Data Engineering, IEEE, pp 497–506

[55] Li Y, Chan HH, Lee PP, Xu Y (2019) Enabling efficient updates in KV storage via hashing: Design and performance evaluation. ACM Transactions on Storage (TOS) 15(3):1–29

[56] Lim H, Andersen DG, Kaminsky M (2016) Towards accurate and fast evaluation of multi-stage log-structured designs. In: 14th USENIX Conference on File and Storage Technologies (FAST 16), USENIX Association, pp 149–166

[57] Lu L, Pillai TS, Gopalakrishnan H, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2017) WiscKey: Separating keys from values in SSD-conscious storage. ACM Trans Storage 13(1):5:1–5:28

[58] Luo C (2019) Merge policies and schedulers in AsterixDB. URL `https://cwiki.apache.org/confluence/x/iQ3jBw`

[59] Luo C, Carey MJ (2018) Efficient data ingestion and query processing for LSM-based storage systems. arXiv preprint arXiv:180808896

[60] Luo C, Carey MJ (2019) On performance stability in LSM-based storage systems. Proc VLDB Endow 13(4):449–462, DOI 10.14778/3372716.3372719

[61] Luo C, Carey MJ (2020) Breaking down memory walls: Adaptive memory management in LSM-based storage systems (extended version). arXiv preprint arXiv:200410360

[62] Luo C, Carey MJ (2020) LSM-based storage techniques: a survey. The VLDB Journal 29(1):393–418

[63] Luo S, Chatterjee S, Ketsetsidis R, Dayan N, Qin W, Idreos S (2020) Rosetta: A robust space-time optimized range filter for key-value stores. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp 2071–2086

[64] Mao Q (2020) Spatial index on AsterixDB. URL `https://github.com/autopear/asterixdb/tree/SpatialIndex`

[65] Mao Q, Jacobs S, Amjad W, Hristidis V, Tsotras VJ, Young NE (2019) Experimental evaluation of bounded-depth LSM merge policies. In: 2019 IEEE International Conference on Big Data (Big Data), IEEE, pp 523–532

[66] Mao Q, Qader MA, Hristidis V (2020) Comprehensive comparison of LSM architectures for spatial data. In: 2020 IEEE International Conference on Big Data (Big Data), IEEE, pp 455–460

[67] Mao Q, Jacobs S, Amjad W, Hristidis V, Tsotras VJ, Young NE (2021) Comparison and evaluation of state-of-the-art LSM merge policies. The VLDB Journal 30(3):361–378, DOI 10.1007/s00778-020-00638-1, URL `https://doi.org/10.1007/s00778-020-00638-1`

[68] Mathieu C, Staelin C, Young NE, Yousefi A (2014) Bigtable merge compaction. arXiv preprint arXiv:14073008 abs/1407.3008

[69] Mei F, Cao Q, Jiang H, Li J (2018) SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In: Proceedings of the ACM Symposium on Cloud Computing, pp 477–489

[70] Nanjappan A (2019) R*-Tree index in Cassandra for geospatial processing

[71] Niemeyer G (2008) GeoHash. URL `http://geohash.org`

[72] O'Neil P, Cheng E, Gawlick D, O'Neil E (1996) The log-structured merge-tree (LSM-tree). Acta Inf 33(4):351–385

[73] Patil S, Polte M, Ren K, Tantisiriroj W, Xiao L, López J, Gibson G, Fuchs A, Rinaldi B (2011) YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In: Proceedings of the 2Nd ACM Symposium on Cloud Computing, ACM, SOCC '11, pp 9:1–9:14

[74] Qader MA, Cheng S, Hristidis V (2018) A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In: Proceedings of the 2018 International Conference on Management of Data, pp 551–566

[75] Raju P, Kadekodi R, Chidambaram V, Abraham I (2017) PebblesDB: Building key-value stores using fragmented log-structured merge trees. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp 497–514

[76] Ren K, Zheng Q, Arulraj J, Gibson G (2017) SlimDB: a space-efficient key-value storage engine for semi-sorted data. Proceedings of the VLDB Endowment 10(13):2037–2048

[77] Roussopoulos N, Leifker D (1985) Direct spatial search on pictorial databases using packed *R*-trees. In: Proceedings of the 1985 ACM SIGMOD international conference on Management of data, pp 17–31

[78] ScyllaDB Inc (2021) ScyllaDB. URL `https://www.scylladb.com/`

[79] Sears R, Ramakrishnan R (2012) bLSM: A general purpose log structured merge tree. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, SIGMOD '12, pp 217–228

[80] Shin J, Wang J, Aref WG (2021) The LSM RUM-Tree: A log structured merge *R*-Tree for update-intensive spatial workloads. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), IEEE, pp 2285–2290

[81] Spark A (2018) Apache Spark. Retrieved January 17(1):2018

[82] Spöcker G (2021) streamhist-cpp. URL `https://github.com/guntersp/streamhist-cpp`

[83] Takasu A, et al (2015) An efficient distributed index for geospatial databases. In: Database and Expert Systems Applications, Springer, pp 28–42

[84] Teng D, Guo L, Lee R, Chen F, Ma S, Zhang Y, Zhang X (2017) LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In: Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on, IEEE, IEEE, pp 68–79

[85] UCR DBLab (2019) Merge Simulator. URL `https://github.com/UC-Riverside-DatabaseLab/MergeSimulator`

[86] Vu T, Eldawy A (2018) *R*-Grove: growing a family of *R*-trees in the big-data forest. In: Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp 532–535

[87] Vu T, Eldawy A (2020) R^*-Grove: Balanced spatial partitioning for large-scale datasets. arXiv preprint arXiv:200711651

[88] Wang P, Sun G, Jiang S, Ouyang J, Lin S, Zhang C, Cong J (2014) An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In: Proceedings of the Ninth European Conference on Computer Systems, ACM, EuroSys '14, pp 16:1–16:14

[89] Whitman RT, Park MB, Ambrose SM, Hoel EG (2014) Spatial indexing and analytics on Hadoop. In: Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems, pp 73–82

[90] Wu X, Xu Y, Shao Z, Jiang S (2015) LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp 71–82

[91] Xu R, Liu Z, Hu H, Qian W, Zhou A (2020) An efficient secondary index for spatial data based on LevelDB. In: International Conference on Database Systems for Advanced Applications, Springer, pp 750–754

[92] Yahoo! (2019) Yahoo! cloud serving benchmark. URL `https://github.com/brianfrankcooper/YCSB`

[93] Yao T, Wan J, Huang P, He X, Gui Q, Wu F, Xie C (2017) A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In: Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST), pp 1–13

[94] Yao T, Wan J, Huang P, He X, Wu F, Xie C (2017) Building efficient key-value stores via a lightweight compaction tree. ACM Transactions on Storage (TOS) 13(4):1–28

[95] Yao T, Zhang Y, Wan J, Cui Q, Tang L, Jiang H, Xie C, He X (2020) MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association, Online, pp 17–31

[96] Yu J, Wu J, Sarwat M (2015) Geospark: A cluster computing framework for processing large-scale spatial data. In: Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems, pp 1–4

[97] Yu J, Zhang Z, Sarwat M (2019) Spatial data management in apache Spark: the Geospark perspective and beyond. GeoInformatica 23(1):37–78

[98] Zhang H, Lim H, Leis V, Andersen DG, Kaminsky M, Keeton K, Pavlo A (2018) Surf: Practical range query filtering with fast succinct tries. In: Proceedings of the 2018 International Conference on Management of Data, pp 323–336

[99] Zhang W, Xu Y, Li Y, Li D (2016) Improving write performance of LSMT-based key-value store. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), IEEE, pp 553–560

[100] Zhong W, Chen C, Wu X, Jiang S (2021) REMIX: Efficient range query for LSM-trees. In: 19th USENIX Conference on File and Storage Technologies (FAST 21), pp 51–64

Appendix A

Merge Policy Data

For each of the 43 runs, Tables A.1 and A.2 show the total write amplification and the average read amplification at five points during the run: after $1,000$, $3,000$, $5,000$, $10,000$, and $20,000$ flushes. If it happens that the MemTable is flushed while a merge is ongoing, the SSTable count may briefly exceed k. For this reason, the average read amplification slightly exceeded k in a few runs (with $k \in \{3,4,5\}$ — see the highlighted cells in the tables).

Policy	k/B	$n = 1,000$		$3,000$		$5,000$		$10,000$		$20,000$	
		Write Amplif.	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**
Bigtable	3	37.57	3.48	87.37	3.71	165.89	3.79	N/A			
	4	11.64	3.97	31.89	4.35	46.79	4.46	86.33	4.61	175.06	4.74
	5	7.13	4.53	11.19	4.79	15.80	4.96	26.16	5.17	46.17	5.36
	6	5.78	5.03	7.78	5.36	9.08	5.55	12.52	5.76	18.71	5.96
	7	5.34	5.60	6.69	5.92	7.85	6.10	9.22	6.31	11.46	6.52
	8	5.05	6.00	6.52	6.34	6.52	6.57	7.32	6.80	8.31	7.04
	10	5.31	6.97	5.79	7.39	6.63	7.51	7.26	7.73	7.87	7.98
Binomial	3	12.05	2.95	17.26	3.01	20.61	3.03	25.72	3.08	32.07	3.13
	4	8.61	3.71	10.67	3.82	12.72	3.85	14.76	3.91	17.56	3.95
	5	6.38	4.49	8.84	4.65	9.34	4.70	10.86	4.77	12.49	4.83
	6	5.61	5.21	7.33	5.48	8.16	5.57	8.84	5.64	10.34	5.69
	7	5.40	5.39	6.44	6.05	6.77	6.24	7.57	6.40	8.96	6.49
	8	5.40	5.41	6.30	6.21	6.44	6.64	7.37	7.00	7.64	7.23
	10	5.40	5.38	6.30	6.19	6.44	6.62	7.30	7.08	7.19	7.68
Exploring	3	30.67	3.31	94.57	3.63	164.31	3.76	N/A			
	4	12.52	3.80	22.88	4.07	34.83	4.23	68.71	4.43	153.02	4.61
	5	7.00	4.31	10.55	4.66	13.08	4.79	21.72	5.06	37.00	5.26
	6	6.20	4.51	7.68	5.06	8.75	5.23	11.22	5.50	16.93	5.77
	7	5.99	4.58	7.41	5.19	7.73	5.42	8.66	5.81	10.07	6.12
	8	5.97	4.56	7.12	5.20	7.45	5.47	8.19	5.91	8.71	6.33
	10	5.90	4.55	6.99	5.19	7.33	5.48	7.98	5.97	8.37	6.40

Table A.1: Total observed write and average read amplification for all runs, for various n.

Policy	k/B	$n = 1,000$		$3,000$		$5,000$		$10,000$		$20,000$	
		Write Amplif.	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**	**Write Amplif.**	**Read Amplif.**
MinLatency	3	12.10	3.00	17.26	3.03	20.62	3.04	25.72	3.08	32.07	3.12
	4	7.89	3.73	10.68	3.84	12.74	3.88	14.79	3.94	17.58	3.98
	5	6.38	4.51	8.11	4.66	9.37	4.70	10.90	4.76	12.48	4.82
	6	5.86	5.24	6.75	5.46	7.52	5.52	8.78	5.58	10.41	5.64
	7	4.97	6.09	5.96	6.30	6.59	6.37	7.55	6.44	9.13	6.51
	8	4.34	6.77	5.30	7.10	6.03	7.13	6.89	7.24	7.64	7.33
	10	3.69	8.24	4.52	8.56	5.03	8.63	5.87	8.78	6.93	8.91
Tiered	4	4.25	8.36	5.01	9.39	5.87	9.84	5.98	10.53	6.73	11.34
	8	3.15	11.75	3.46	13.81	4.24	14.54	4.28	15.31	4.30	16.86
	16	2.52	17.94	2.67	22.05	3.41	23.35	3.43	24.37	3.43	26.19
	32	1.86	31.54	2.45	33.24	2.57	34.36	2.66	36.95	2.70	41.98
Leveled	4	17.05	5.36	19.87	6.13	21.07	6.48	22.87	6.95	24.48	7.48
	8	22.17	4.05	25.03	4.68	26.47	4.81	28.66	5.23	30.37	5.62
	16	32.75	3.59	36.60	3.86	38.68	3.92	42.15	4.33	44.47	4.67
	32	51.65	2.95	60.98	3.48	63.50	3.69	66.54	3.84	69.62	3.92

Table A.2: Total observed write and average read amplification for all runs, for various n (continued).