Spooky: Granulating LSM-Tree Compactions Correctly

Niv Dayan
University of Toronto
nivdayan@cs.toronto.edu

Tamar Weiss, Shmuel Dashevsky, Michael Pan,
Edward Bortnikov, Moshe Twitto
Pliops
{tamarw,shmueld,michaelp,ebortnik,moshet}@pliops.com

ABSTRACT

Modern storage engines and key-value stores have come to rely on the log-structured merge-tree (LSM-tree) as their core data structure. LSM-tree operates by gradually merge-sorting data across levels of exponentially increasing capacities in storage. A crucial design dimension of LSM-tree is its compaction granularity. Some designs perform *Full Merge*, whereby entire levels get compacted at once. Others perform *Partial Merge*, whereby smaller groups of files with overlapping key ranges are compacted independently. This paper shows that both strategies exhibit serious flaws. With Full Merge, space-amplification is exorbitant. The reason is that while compacting the LSM-tree's largest level, there must be at least twice as much storage space as data to store both the original and new files until the compaction is finished. On the other hand, Partial Merge exhibits excessive write-amplification. The reason is twofold. (1) The files getting compacted typically do not have perfectly overlapping key ranges, and so some non-overlapping data is superfluously rewritten in each compaction. (2) Files with different lifetimes become interspersed within the SSD leading to high SSD garbage-collection overheads. As the data size grows, these problems grow in magnitude.

We introduce Spooky, a novel compaction granulation method to address these problems. Spooky partitions data at the largest level into equally sized files, and it partitions data at smaller levels based on the file boundaries at the largest level. This allows merging one group of perfectly overlapping files at a time to limit space-amplification and compaction overheads. At the same time, Spooky writes larger though fewer files simultaneously so that files with different lifetimes do not become as interspersed within the SSD. This cheapens garbage-collection. We show empirically that Spooky achieves >2x lower space-amplification than Full Merge and >2x lower write-amplification than Partial Merge at the same time.

PVLDB Reference Format:
Niv Dayan and Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, Moshe Twitto. Spooky: Granulating LSM-Tree Compactions Correctly . PVLDB, 15(11): 3071 - 3084, 2022.
doi:10.14778/3551793.3551853

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551853

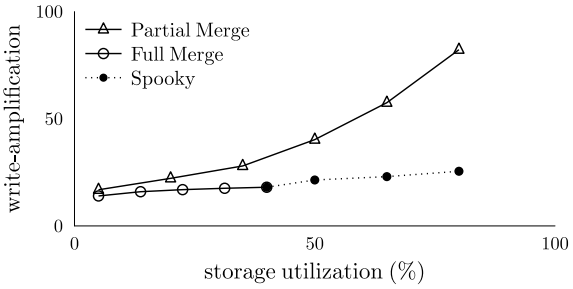


Figure 1: Existing LSM-trees cannot reach high storage utilization while maintaining moderate write-amplification.

1 INTRODUCTION

LSM-Tree. LSM-tree has become the backbone of modern key-value stores and storage engines. It ingests key-value entries from the application by buffering them in memory. When the buffer fills up, it flushes these entries to storage (typically a flash-based SSD) as a sorted array called a *run*. LSM-tree then compacts smaller runs into larger ones to (1) restrict the number of runs that a query has to search, and to (2) discard obsolete entries, for which newer versions with the same keys have been inserted. It organizes these runs based on their ages across levels of exponentially increasing capacities. LSM-tree is used widely including in OLTP [37], HTAP [61], social graphs [65], FTL design [21], data series [49–51], blockchain [30], stream-processing [16], etc.

Compaction Granularity. The compaction policy of an LSM-tree dictates which data to merge under which circumstances. Existing work has rigorously studied how to tune the eagerness of a compaction policy as a means of striking different trade-offs between the costs of reads, writes, and space [22, 23, 45, 67, 70, 74, 80]. This paper focuses on an orthogonal yet crucial design dimension: compaction granularity. Existing compaction designs can broadly be lumped into two categories with respect to how they granulate compactions: *Full Merge* and *Partial Merge* [72]. Each entails a particular flaw.

Partial Merge. With Partial Merge, each run is partitioned into multiple small files of equal sizes. When a level reaches capacity, one file from within that level is selected and merged into files with overlapping key ranges at the next larger level. Partial merge is used by default in LevelDB [35] and RocksDB [33]. Its core problem is high write-amplification. The reason is twofold. First, the files chosen to be merged typically do not have perfectly overlapping key ranges. Each compaction therefore superfluously rewrites some non-overlapping data [57]. Second, simultaneous compactions at different levels cause files with different lifespans to become physically interspersed within the SSD [12, 28]. This makes it hard for the SSD to perform efficient internal garbage-collection, especially as the data size grows. We illustrate this problem in Figure 1 with

the curve labeled Partial Merge. Write-amplification increases at an accelerating rate as storage utilization (i.e., user data size divided by storage capacity) increases. Figure 1 is based on an experiment described in Sections 3.

Full Merge. With Full Merge, entire levels are merged all at once. Full merge is used by default in Cassandra [3], HBase [4]. The core problem is that until a merge operation is finished, the files being merged cannot be disposed of. This means that compacting the LSM-tree's largest level, which is exponentially larger than the rest, requires having twice as much storage capacity as user data until the operation is finished. As a result, Full Merge cannot reach a storage utilization of over 50%. It therefore wastes most of the available storage capacity, as shown in Figure 1.

Research Problem. Figure 1 shows that neither Partial Merge nor Full Merge can achieve a high storage utilization and moderate write-amplification at the same time. Is it possible to bridge this gap and attain the best of both worlds?

Spooky. We introduce **S**pooky: **P**artiti**o**ned C**o**mpaction for **K**e**y**-Value Stores. Spooky partitions the LSM-tree's largest level into equally-sized files, and it partitions a few of the subsequent largest levels based on the file boundaries at the largest level. This allows merging one group of perfectly overlapping files at a time to restrict both space-amplification and compaction overheads. At smaller levels, Spooky performs Full Merge to limit write-amplification yet without inflating space requirements as these levels are exponentially smaller. In addition, Spooky writes and deletes files sequentially within each level and across fewer levels at a time. As a result, fewer files become physically interspersed within the SSD to cheapen garbage-collection.

Spooky is a meta-policy: it is orthogonal to other compaction aspects such as eagerness [22, 23, 45, 67, 69, 80] , key-value separation [15, 55, 58, 81], hardware acceleration/customization [11, 26, 77, 85, 91], performance stabilization [8, 56, 59, 74], etc. As such, it both complements and enhances existing LSM-tree variants and the variety of applications that they each optimize for.

Contributions. Our contributions are as follow.

(1) We posit Full and Partial Merge as the two core compaction granulation strategies used by modern LSM-trees. We provide cost models and experiments to show that Full Merge suffers from exorbitant space-amplification while Partial Merge suffers from excessive write-amplification.
(2) We introduce Spooky, a new compaction granulation approach that (I) partitions data at larger levels into perfectly overlapping groups of files that can be merged using little extra space or superfluous work, and that (II) issues SSD-friendly I/O patterns that cheapen garbage-collection.
(3) We show experimentally that Spooky achieves >2x lower space-amplification than Full Merge and >2x better write-amplification than Partial Merge at the same time.
(4) We show that Spooky's lower write-amplification improves both read and write throughput.

2 LSM-TREE FUNDAMENTALS

LSM-tree organizes data across L levels of exponentially increasing capacities. Level 0 is an in-memory buffer (aka memtable). All other levels are in storage. The capacity at Level i is T times larger than at Level $i-1$. When the largest level reaches capacity, a new larger level is added. The number of levels L is $\approx \log_T(N/B)$, where N is the data size and B is the buffer size. Figure 2 lists terms used to describe LSM-tree throughout the paper.

For each insert, update, or delete request issued by the application, a data entry comprising a key and a value is put in the buffer (in case of a delete, the value is a tombstone [71]). When the buffer fills up, it gets flushed to Level 1 as a file sorted based on its entries' keys. Whenever one of the levels in storage reaches capacity, some file from within it is merged into one of the next larger levels. Whenever two entries with the same key are encountered during compaction, the older one is considered obsolete and discarded.

Each file can be thought of as a static B-tree whose internal nodes are cached in memory. There is an in-memory Bloom filter [13] for each file to allow point reads to skip accessing files that do not contain a given key. A point read searches the levels from smallest to largest to find the most recent version of an entry with a given key. A range read sort-merges entries within a specified key range across all levels to return the most recent version of each entry in the range to the user.

Concurrency Control. In the original LSM-tree paper, each level is a mutable B-tree, and locks are held to transfer data from one B-tree to another when a level reaches capacity [67]. To obviate locking bottlenecks, however, most modern LSM-trees employ multi-version concurrency control [32]. In RocksDB, for example, a new version object is created after each compaction/flush operation. This object contains a list of all files active at the instant in time that the compaction/flush finished. Point and range reads operate over files within the version object that was newest when they commenced to return consistent output to the user.

Space-Amplification. LSM-tree occupies more storage space than the size of the user data. The reason is twofold. First, obsolete entries take up space until compaction discards them. We refer to this as *durable* space-amplification.

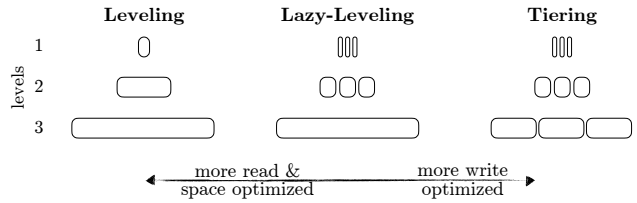
Second, multi-version concurrency control prevents the system from reclaiming space occupied by a file until the compaction operating on this file has finished [3, 4, 33, 35]. We refer to the temporary extra space needed during compaction to store the original and merged data at the same time as *transient* space-amplification.

We distinguish between *logical* and *physical* data size as the LSM-tree's size before and after space-amplification (space-amp) is considered. We define total space-amp in Equation 1 as the factor by which the physical data size is greater than the logical data size. Durable and transient space-amp are each defined here as a fraction of the logical data size. The inverse of total space-amp is storage-utilization, the fraction of the storage device that can store user data. It is generally desirable to operate at a high storage utilization to take advantage of the available hardware.

$$\text{total space-amp} = 1 + \text{transient space-amp} + \text{durable space-amp} \quad (1)$$

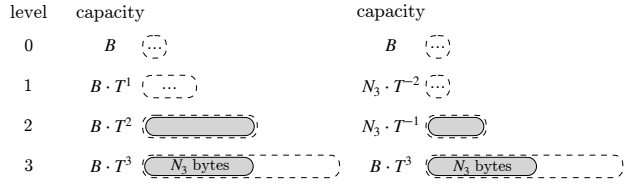
Write-Amplification. The LSM-tree's compactions cause each data entry to be rewritten to storage multiple times. The average number of times an entry is physically rewritten is known as write-amplification (write-amp). It is generally desirable to keep write-amp low as it consumes storage bandwidth and lifetime.

In addition to compactions, there is another important source of write-amp for LSM-tree: SSD garbage-collection (GC) [28]. Modern



Term	Definition
N	overall data size (bytes)
B	buffer size (bytes)
T	LSM-tree size ratio
L	number of LSM-tree levels
Ni	data size at Level i (bytes)
Ci	capacity at Level i (bytes)
X	level to which Spooky performs dividing merge

Figure 2: LSM-tree spans various merge policies and parameters that allow optimizing for different applications/workloads. Each rectangle in the figure represents a run, which is a unit of sorted data that comprises one or more files.



(A) Without DCA **(B) With DCA**

Figure 3: DCA restricts durable space-amplification by shrinking capacities at smaller levels based on the data size at the largest level. This example assumes leveling.

flash-based SSDs lay out data sequentially across *erase units* [1, 75]. As the SSD fills up, GC kicks in to reclaim space. It does this by (1) picking an erase unit with ideally as little remaining live data as possible, (2) migrating any live data to other erase units, and (3) erasing the target unit [1, 24, 36]. Hence, GC can cause data to be rewritten multiple times.

Since GC occurs opaquely within the SSD, it has historically been difficult to measure. Hence, most papers on LSM-tree to date only focus on optimizing compaction overheads [22, 23, 66, 69, 70, 80]. In this paper, we emphasize that total write-amp is the product of both sources of write-amp, as expressed in Equation 2. Hence, these sources must be co-optimized to curb total write-amp.

$$\text{total write-amp} = \text{compaction write-amp} \times \text{GC write-amp} \tag{2}$$

Merge Policy. The compaction policy of an LSM-tree determines which data to merge when. The two mainstream policies are known as *leveling* and *tiering*, as illustrated in Figure 2.

With leveling, new data arriving at a level is immediately merged with whichever overlapping data already exists at that level. As a result, each level contains at most one sorted unit of data, also referred to as a *run*. Each run consists of one or more files.

With tiering [45], each level contains multiple runs. When a level reaches capacity, all runs within it are merged into a single run at the next larger level. Tiering is more write-optimized than leveling as each compaction spans fewer data. However, it is less read-efficient as each query has to search more runs per level. It is also less space-efficient as it takes longer to identify and discard obsolete entries. With both leveling and tiering, the size ratio T can be fine-tuned to control the trade-off between compaction vs. query and space overheads [19].

Figure 2 also illustrates a hybrid policy named *lazy leveling*, which applies leveling at the largest level and tiering at smaller levels. Its write cost is similar to that of tiering while still having similar space and point read overheads to those of leveling [22]. It therefore offers good trade-offs in-between. We apply Spooky to all three policies in this work to demonstrate its broad applicability.

Dynamic Capacity Adaptation (DCA). Durable space-amp exhibits a pathological worst-case. When a new level is added to the LSM-tree to accommodate more data, its capacity is set to be larger by a factor of T than the capacity at the previously largest level. When this happens, the data size at the new largest level is far smaller than its capacity. As the now second largest level fills up with new data, it can come to contain as much data as the current data size at the largest level. In this case, durable space-amp may be two or greater, as illustrated in Figure 3 Part (A). To remedy this, RocksDB introduced a technique coined Dynamic Capacity Adaptation (DCA) [31]. As shown in Figure 3 Part (B), DCA restricts the capacities at Levels 1 to $L-1$ based on the data size rather than the capacity at Level L. With leveling or lazy leveling, DCA bounds the worst-case durable space-amp to the expression in Equation 3. We leverage DCA in conjunction with Spooky throughout the paper.

$$\text{durable space-amp} \leq \frac{1}{T-1} \tag{3}$$

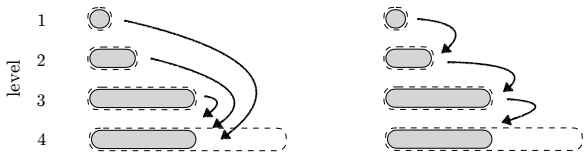
Compaction Granularity. The granularity of compaction controls how much data to merge in one go when a level has reached capacity. Compaction granularity is an orthogonal design dimension to the merge policy [72]. There are two mainstream granulation approaches: Full vs. Partial. They profoundly impact the balance between write-amp and space-amp.

Full Merge. With Full Merge, compaction is performed at the granularity of whole levels. Full Merge is used in Cassandra, Hbase, and Universal Compaction in RocksDB.

Full Merge lends itself to *preemption*, a technique used to reduce write-amp by predicting across how many levels the next merge operation will recurse and merging them all at once [22]. Figure 4 Part (A) illustrates an example where Levels 1 to 4, which are all nearly full, are merged all at once. In contrast, Part (B) shows how without preemption, a merge operation from Level 1 recurses to Level 4, causing data originally at Levels 1 and 2 to be rewritten three and two times, respectively. We leverage preemption in conjunction with Full Merge throughout the paper to optimize write-amp.

The core problem with Full Merge is that while compacting the largest level, which contains most of the data, transient space-amp skyrockets as the original files cannot be disposed of until the compaction is finished [63, 66].

Partial Merge. With *Partial Merge*, used by default in LevelDB and RocksDB, each run is partitioned into multiple files (a.k.a. Sorted String Tables or SSTs). When Level i fills up, some file from Level i is picked and merged into the files with overlapping key ranges at Level $i+1$. Different methods have been proposed for how to pick this file (e.g., randomly or round-robin). The best-known technique, coined *ChooseBest* [76], picks the file that overlaps with the least amount of data at the next larger level to minimize write-amp.



(A) Full Preemptive Merge (B) Full Recursive Merge

Figure 4: Preemptive merge allows some of the data is skip merging merged across some of the levels, thus reducing write-amp.

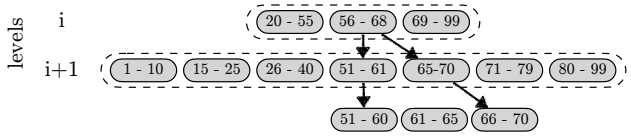


Figure 5: With Partial Merge, each run is partitioned into multiple files, and smaller independent groups of intersecting files are merged at a time.

For example, in Figure 5, the leftmost SST at Level $L-1$ overlaps with four SSTs at Level L, the middle with two, and the rightmost with three. Hence, the middle file is picked. Partial Merge exhibits lower transient space-amp than Full Merge as compaction is more granular. However, it exhibits a higher write-amp than Full Merge as we show in Section 3.

3 PROBLEM ANALYSIS

This section analyzes write-amp and space-amp for Full vs. Partial Merge to formalize the problem. We assume the leveling merge policy for both baselines. We also assume uniformly random insertions to model the worst-case write-amp.

Modeling Compaction Write-Amp. With Full Merge, the i^{th} run arriving at a level after the level was last empty entails a write-amp of i to merge with the existing run at that level. After $T-1$ runs arrive, a preemptive merge spanning this level takes place and empties it again. Hence, each level contributes $\frac{1}{T} \cdot \sum_{i=1}^{T-1} i = \frac{T-1}{2}$ to write-amp, resulting in Equation 4.

$$\text{compaction write-amp with Full Merge} = \frac{T-1}{2} \cdot L \tag{4}$$

With Partial Merge, a file picked using ChooseBest from a full Level i intersects with $\approx T/2$ files' worth of data on average at Level $i+1$ [57, 76]. The overlap among these files typically isn't perfect, however, leading to additional overhead. For example, in Figure 5, the file picked from Level i with key range 56-68 does not perfectly overlap with the two intersecting files at Level $i+1$, which have a wider combined key range of 51-70. This means that entries at both edges of the compaction, in the ranges of 51-56 and 68-70, are superfluously rewritten. We coin this problem *superfluous edge merging*. On average, one additional file's worth of data is superfluously included in each compaction, leading to the write-amp expression in Equation 5.

$$\text{compaction write-amp with Partial Merge} = \frac{T+1}{2} \cdot L \tag{5}$$

Compaction Write-Amp Experiments. We verify Equations 4 and 5 using RocksDB to provide a principled comparison between the baselines that they each represent. We use the default RocksDB compaction policy to represent Partial Merge, and we created a Full Preemptive Merge implementation within RocksDB[1]. The size ratio T is set to 5 and the buffer size B is set to 64MB. The rest of the implementation and setup details are in Sections 5 and 6.

[1]We did not use RocksDB's Universal Compaction to represent Full Merge because it is not a leveled merge policy but rather a bounded-depth policy (more in Section 7). Therefore, it does not lend itself to an apples-to-apples comparison with the default RocksDB compaction policy for our purposes.

Figure 6 Part (A) issues unique random insertions to an initially empty database. The x-axis reports the number of times the buffer has flushed (i.e., N/B). The y-axis measures write-amp for each baseline against its model predictions. We observe that Full Merge exhibits a lower write-amp than Partial Merge. The reasons are that (1) it uses preemption to skip merging entries across nearly full levels, and (2) it avoids the problem of superfluous edge merging by compacting whole levels at a time.

Figure 6 Part (B) repeats the experiment with different size ratios on the x-axis while fixing N/B, the number of times the buffer has flushed, to 3500. With Full Merge, the size ratio has an approximately linear relationship to write-amp. It is therefore possible to reduce write-amp by tuning the size ratio to smaller values. With Partial Merge, however, write-amp cannot be reduced beyond the global minimum shown in the figure. The reason is that with Partial Merge, smaller size ratios cause the relative amount of superfluous work (i.e., rewriting one file's worth of non-overlapping data) to increase relative to the amount of useful work performed in each compaction (i.e., merging $\approx T/2$ files' worth of overlapping data). In summary, Full Merge exhibits lower write-amp than Partial Merge across the board, and it is also more amenable to tuning.

Garbage-Collection Write-Amp. We now measure SSD garbage-collection (GC) with Full vs. Partial Merge. We run a large-scale experiment that fills up an initially empty LSM-tree with unique random insertions followed by random updates for one day on a 960GB SSD. We use the Linux nvme command to report the data volume that the operating system writes to the SSD vs. the data volume that the SSD physically writes to flash. This allows calculating write-amp due to GC throughout the experiment. For both baselines, the size ratio is set to 5, the buffer size to 64MB, and DCA is turned on. The rest of the setup is given in Section 6.

Since our goal is to elicit SSD garbage-collection, it is important to fill up the SSD so that it is stressed for free space. The Full Merge baseline, however, is unable to utilize most of the available storage capacity due to its high transient space-amp. We therefore employ different logical data sizes in this experiment: 369GB and 644GB with Full and Partial Merge, respectively. As we will see, the physical data size comes out to $\approx$ 800GB with both baselines thus providing us with a degree of experimental control.

Figure 7 Part (A) plots three curves for Partial Merge: (1) compaction write-amp, (2) GC write-amp, and (3) their product, total write-amp. During the experiment, GC write-amp steadily increases (up to $\approx$ 3). The reason is that multiple compactions across different levels of the LSM-tree occur simultaneously. This causes files to become physically interspersed within the same SSD erase units.

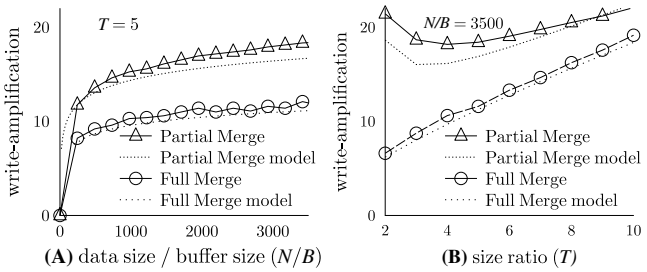


Figure 6: Full Merge exhibits lower write-amp than Partial Merge as the data grows (A) and across different tunings (B). Our cost models predict write-amp reasonably well.

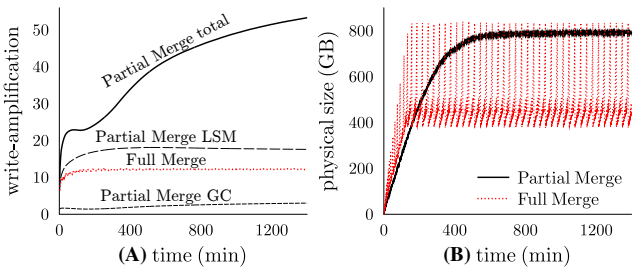


Figure 7: Partial Merge exhibits skyrocketing write-amp as we approach device capacity (A), while Full Merge wastes most of the available storage capacity (B).

Since files from different levels have exponentially varying lifespans, some data in each erase unit is disposed of quickly while other data lingers for longer. Hence, the SSD must internally migrate older data to reclaim space. This is a manifestation of a well-known problem of "hot" and "cold" data mixing within an SSD to inflate GC overheads [27, 75]. We observe that while GC write-amp is far smaller than compaction write-amp, it multiplies with it to cause total write-amp to exceed fifty by the experiment's end.

It is tempting to think that using larger files with Partial Merge would eliminate these GC overheads by causing larger units of data to be written and erased all at once across SSD erase units. We falsify this notion later in Section 6 by showing that even with large files, GC overheads remain high because simultaneous partial compactions still cause files from different levels to mix physically.

For Full Merge, the figure only depicts compaction write-amp as no GC write-amp was detected. The reason is that Full Merge writes one large file at a time. The SSD erase units that store this large file also get cleared all at once when this file is deleted. This allows the SSD to reclaim space without migrating data internally.

Measuring and Modeling Space-Amp. Figure 7 Part (B) reports the physical data size of each baseline for the same experiment as in Figure 7 Part (A). With Full Merge, the curve is sawtooth-shaped since compactions into the LSM-tree's largest level occasionally cause transient space-amp to skyrocket. In contrast, the curve for Partial Merge is smooth because compactions are more granular. The physical data size for both baselines is approximately equal even though the Partial Merge baseline accommodates nearly twice as much logical data.

We provide space-amp models for Partial and Full merge in Equations 6 and 7 to enable an analytical comparison. The term $\frac{1}{T-1}$ in both equations accounts for the worst-case durable space-amp from Equation 3. Otherwise, space-amp with Full Merge is higher by an additive factor of one to account for the fact that its transient space-amp occasionally requires as much extra space as the logical data size. By contrast, transient space-amp with Partial Merge is assumed here to be negligible due to the small file sizes used (i.e., typically 2-64MB).

$$\text{maximum space-amp with Partial Merge} = 1 + \frac{1}{T-1} \tag{6}$$

$$\text{maximum space-amp with Full Merge} = 2 + \frac{1}{T-1} \tag{7}$$

Scalability with Data Size. Figure 1 in Section 1 repeats the experiment in Figure 7 while varying storage utilization. The size ratio is set to ten this time, and we run each trial for at least one day for write-amp to converge. For each trial, we report storage-utilization on the x-axis against total write-amp on the y-axis. For Partial Merge, total write-amp increases at an accelerating rate as storage utilization increases. This is a well-known phenomenon with SSDs: as they fill up, erase blocks with little remaining live data become increasingly hard to find [75]. GC migration overheads therefore increase rapidly. The outcome is that each additional byte of user data costs disproportionately more to store in terms of forgone performance. In contrast, Full Merge maintains a low total write-amp but is unable to reach a storage utilization of over 50%.

Problem Summary. Full Merge exhibits exorbitant space-amp. On the other hand, Partial Merge exhibits massive write-amp as storage utilization increases. Can we devise a new compaction granulation approach that enables high storage utilization and moderate write-amp at the same time?

4 SPOOKY

We introduce Spooky, a new method of granulating LSM-tree merge operations that eliminates the contention between write-amp and space-amp. As shown in Figure 8, Spooky comprises six design decisions. (1) It partitions the largest level into equally-sized files, and (2) it partitions a few of the subsequent largest levels based on the file boundaries at the largest level. (3) This allows Spooky to perform *partitioned merge*, namely compacting one group of perfectly overlapping files at a time across the largest levels to restrict both write-amp and space-amp. (4) At smaller levels, Spooky performs full preemptive merge. This improves write-amp without harming space-amp as these levels are exponentially smaller. (5) Spooky restricts the number of files being simultaneously written to the SSD to limit the mixing of hot and cold data within the same SSD erase blocks. (6) Within each level, Spooky writes data sequentially and later disposes of it sequentially as well. Hence, large swaths of data written sequentially to the SSD are also deleted sequentially. This allows the SSD to reclaim space more cheaply.

For ease of exposition, Section 4.1 describes a limited form of Spooky that performs partitioned merge only at the largest two levels. Section 4.2 generalizes Spooky to perform partitioned merge across more levels to further reduce space overheads. Section 4.3 focuses on Spooky's ability to accommodate skewed workloads. Sections 4.1 to 4.3 assume the leveled merge policy, and Section 4.4 extends Spooky to tiered and hybrid merge policies.

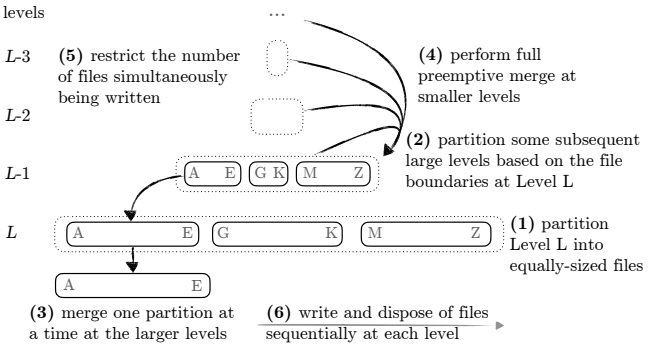


Figure 8: Spooky restricts space-amp and write-amp at the same time by introducing and adhering to six principles of compaction granulation.

4.1 Two-Level Spooky

Two-Level Spooky (2L-Spooky) performs partitioned merge across the largest two levels of an LSM-tree as shown in Figure 8. Level L (the largest level) is partitioned into files, each of which comprises at most N_L/T bytes, where N_L is the data size at Level L and T is the LSM-tree's size ratio. This divides Level L into at least T files of approximately equal sizes. Level $L-1$ (the second largest level) is also partitioned into files such that the key range at each file overlaps with at most one file at Level L. This allows merging one pair of overlapping files at a time across the largest two levels. At Levels 0 to $L-1$, 2L-Spooky performs full preemptive merge. Algorithm 1 describes 2L-Spooky's workflow, which is invoked whenever the buffer fills up and decides which files to merge in response. Let us go through its steps in detail.

Full Preemptive Merge at Smaller Levels. Algorithm 1 first picks some full preemptive merge operation to perform along Levels 0 to $L-1$. Specifically, It chooses the smallest level q in the range $1 \leq q \leq L-1$ that wouldn't reach capacity if we merged all data at smaller levels into it (Line 2). It then compacts all data at Levels 0 to q and places the resulting run at Level q (Lines 3-7).

Dividing Merge. A compaction into Level $L-1$ is coined a *dividing merge*. A run written by a dividing merge is partitioned such that each output file perfectly overlaps with at most one file at Level L (Lines 6-7). On the other hand, any run written at Levels 1 to $L-2$ is stored as one file (Line 4).

Partitioned Merge. When Level $L-1$ reaches capacity (Line 8), 2L-Spooky triggers a partitioned merge. As shown in Figure 8, this involves merging one pair of overlapping files from Levels $L-1$ and L at a time into Level L (Lines 9-11). If the projected size of an output file is greater than N_L/T, the output is split into two equally-sized files (Line 10). The pairs of files are merged in the order of the keys they contain. This ensures that data is written and disposed of in the same order.

Seamless Adaptation. Under skewed deletes, one file may abruptly shrink causing the others to become relatively larger. In this case, Spooky splits any file that is now greater than the maximum file size (N_L/T bytes) into two during the next partitioned merge. If the cumulative size of two or more adjacent pairs of overlapping files at Levels $L-1$ and L is lower than the maximum file size, we merge them all into one file at Level L during the next partitioned merge.

```
1  Function Merge_Workflow( ):
2      int target_lvl = smallest level q such that 1 ≤ q ≤ L − 1 and
         C_q < Σ_{i=0}^{q} S_i
3      if target_lvl < L-1 then
4          merge_into_one_file(0, target_lvl)
5      else if target_lvl == L-1 then
6          array<key> boundaries = get_largest_level_file_boundaries()
7          merge_and_partition_output(0, L-1, boundaries)
8      if N_{L−1} ≥ C_{L−1} then
9          for pair p of overlapping files at Levels L − 1 and L do
10             int file_max = C_{L−1}
11             merge(p, file_max)
12         if N_L ≥ C_L then
13             add a new level with capacity C_{L+1} = C_L · T
14             increment L
15         else if N_L < C_L/T then
16             set capacity of Level L − 1 to C_L/T and remove Level L
17             decrement L
18         for i = L − 1; L ≥ 1; i − − do
19             C_i = N_L/T^{L−i}
```

Algorithm 1: Compaction workflow for 2L-Spooky.

Hence, all files at Level L have similar sizes between $N_L/(T \cdot 2)$ and N_L/T bytes. We omit these details from Algorithm 1 for readability. These file adjustments occur seamlessly during partitioned merge operations and therefore involve no additional overhead.

Evolving the Tree. After a partitioned merge, Algorithm 1 checks if Level L is now at capacity. If so, we add a new level (Lines 12-14). On the other hand, if many deletes took place and the largest level significantly shrank, we remove one level (Lines 15-17). If the number of levels changed, the run at the previously largest level is placed at the new largest level. We then perform dynamic capacity adaptation to restrict durable space-amp (Lines 18-19).

Write-Amp Analysis. The full preemptive merge operations at smaller levels achieve the modest write-amp of Full Merge across Levels 1 to $L-2$. At Level $L-1$, each entry is rewritten one extra time relative to pure Full Merge. The reason is that Level $L-1$ has to reach capacity before a partitioned merge is triggered (i.e., there is no preemption at Level $L-1$). At Level L, the absence of overlap across different pairs of files prevents superfluous rewriting of non-overlapping data and thus keeps write-amp on par with Full Merge. Hence, 2L-Spooky increases write-amp by an additive factor of one relative to Full Merge, as stated in Equation 8.

$$\text{write-amp with 2L-Spooky} = L \cdot \frac{T-1}{2} + 1 \tag{8}$$

Design for Low SSD Garbage-Collection Overheads. With our design so far, at most three files can be simultaneously written to storage: one due to the buffer flushing, one due to a full preemptive merge, and one due to a partitioned merge. Hence, at most three files can become physically interspersed within the underlying SSD. In addition, Spooky writes data sequentially at every level and later disposes of it in the same order. As long as there are no concurrent writes from other running applications, this design cheapens SSD garbage-collection as large contiguous storage areas that are written at the same time tend to also be cleared at the same time. GC overheads are harder to reason about analytically as they depend on the particular SSD firmware, which is opaque. We therefore measure Spooky's impact on GC empirically in Section 6.

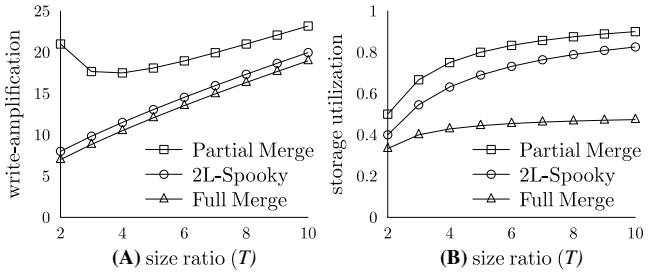


Figure 9: 2L-Spooky significantly improves space-amp relative to Full Merge and write-amp relative to Partial Merge.

Transient Space-Amp Analysis. A dividing merge and a partitioned merge never occur at the same time, yet they are both the bottlenecks in terms of transient space-amp. Hence, transient space-amp for the system is lower bounded by the expression $\max(file_{max}, C_{L-1})/N_L$. The term $file_{max}$ denotes the maximum file size at Level L and controls transient space-amp for a partitioned merge. The term C_{L-1} denotes the capacity at Level $L-1$ and controls transient space-amp for a dividing merge. Note that while it is possible to decrease $file_{max}$ to lower transient space-amp for a partitioned merge, the overall transient space-amp would still be lower bounded by C_{L-1}, and so setting $file_{max}$ to be lower than C_{L-1} is inconsequential. This explains our motivation for setting $file_{max}$ to $C_{L-1} = N_L/T$ (Line 10). The overall transient space-amp for 2L-Spooky is therefore $1/T$.

Total Space-Amp Analysis. Due to dynamic capacity adaptation, Spooky's durable space-amp is upper-bounded by Equation 3. We plug it along with Spooky's transient space-amp of $1/T$ into Equation 1 to obtain Spooky's total space-amp in Equation 9.

$$\text{space-amp with 2L-Spooky} = 1 + \frac{1}{T} + \frac{1}{T-1} \tag{9}$$

Summary. Figure 9 Part (A) plots 2L-Spooky's write-amp against Full and Partial Merge (Eqs. 5, 4, and 8) as we vary the size ratio. The data size N is assumed to be 1TB while the buffer size B is 64MB. 2L-Spooky significantly reduces write-amp relative to Partial Merge while almost matching Full Merge. Note that the figure ignores the impact of SSD garbage-collection and therefore understates the total write-amp differences between these baselines.

Figure 9 Part (B) plots storage utilization, the inverse of space-amp, for all three baselines (Eqs. 7, 6 and 9). 2L-Spooky improves space-utilization compared with Full Merge by 10% to 30% depending on the tuning of T. Compared with partial merge, however, space-utilization with 2L-Spooky is $\approx$ 10% worse across the board.

In summary, while 2L-Spooky enables new attractive write/space cost balances, its write-amp is still higher than with Full Merge by one and its space-utilization is higher than with Partial Merge by $\approx$ 10%. It therefore leaves something to be desired. We improve it further in Section 4.2.

4.2 Generalizing Spooky for Better Trade-Offs

In Section 4.1, we saw that dividing merge operations into Level $L-1$ create a lower bound of $1/T$ on transient space-amp. To recap, the reason is that Level $L-1$ comprises a fraction of $1/T$ of the data size. This level is rewritten from scratch during each dividing merge operation, and the input files cannot be deleted until the merge is complete. This section generalizes Spooky to support dividing merge operations into any level to overcome this bound.

```
1  Function Merge_Workflow:
2      array<key> boundaries = get_largest_level_file_boundaries()
3      int target_lvl = smallest level q such that 1 ≤ q ≤ X and
         C_q < Σ_{i=0}^{q} N_i
4      if target_lvl < X then
5          merge_into_one_file(0, target_lvl)
6      else if target_lvl == X then
7          merge_and_partition_output(0, X, boundaries)
8      if N_X ≥ C_X then
9          target_lvl = smallest level z such that X + 1 ≤ z ≤ L and
             C_z < Σ_{i=X}^{z} N_i. If no such level exists, however, pick Level L.
10         for partition p of intersecting files at Levels X to z do
11             if X + 1 ≤ target_lvl ≤ L − 1 then
12                 merge(p, boundaries)
13             else if target_lvl == L then
14                 int file_max = C_X
15                 merge(p, file_max)
16         if N_L ≥ C_L then
17             add a new level with capacity C_{L+1} = C_L · T
18             increment L
19         else if N_L < C_L/T then
20             set capacity of Level L − 1 to C_L/T and remove Level L
21             decrement L
22         for i = L − 1; L ≥ 1; i − − do
23             C_i = N_L/T^{L−i}
```

Algorithm 2: Spooky's generalized merge workflow.

Algorithm 2 describes Spooky's generalized compaction workflow. The workflow takes a parameter X, which determines the level into which we perform dividing merge operations. Level X is the smallest level at which we start partitioning runs based on the file boundaries at Level L (the largest level). For example, X, is set to $L-1$ in Figure 8 and to $L-2$ in Figure 10. In addition, we now partition Level L into files whose sizes are dictated by the capacity at Level X (i.e., at most $C_X = N_L/T^{L-X}$ bytes each).

Merging at Smaller Levels. Algorithm 2 is different from Algorithm 1 in that full preemptive merge operations only take place along levels 0 to $X-1$ while dividing merge operations now take place into Level X (rather than into Level $L-1$ as before). All else is the same as in Algorithm 1.

Partitioned Preemptive Merge. When Level X fills up, Spooky performs a partitioned merge operation along the largest $L-X$ levels, one group of at most $L-X$ perfectly overlapping files at a time. An important design decision in the generalized workflow is to combine the idea preemption with partitioned merge to limit the write-amp emanating from larger levels. Specifically, when Level X is full, Algorithm 2 picks the smallest level z in the range $X+1 \le z \le L$ that would not reach capacity if we merged all data within this range of levels into it (Line 9). Then, one group of overlapping files across Levels X to z is merged at a time into Level z. If the target Level z is not the largest level, the resulting run is partitioned based on the file boundaries at the largest level (Lines 11-12) to facilitate future partitioned merge operations.

Example. In Figure 10, X is set to $L-2$. In Part (A), the cumulative data size at Levels $L-2$ and $L-1$ does not exceed the capacity at Level $L-1$, and so we merge one pair of files from Levels $L-2$ and $L-1$ at a time into Level $L-1$. In Part (B), however, the data size at Levels $L-2$ and $L-1$ does exceed the capacity at Level $L-1$, and so Level L is chosen as the target. We therefore merge

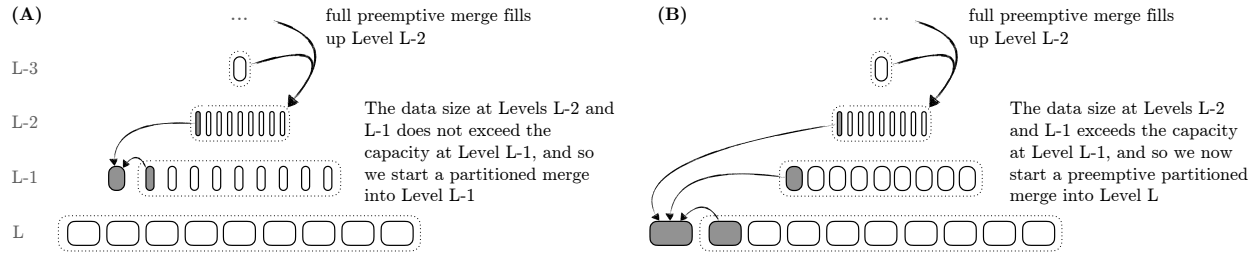


Figure 10: Spooky performs partitioned preemptive merge along larger levels to lower write-amp.

three overlapping files from Levels $L-2$, $L-1$ and L at a time into Level L. In this case, preemption allows us to merge the data from Level $L-2$ once rather than twice on the way to Level L.

Write-Amp Analysis. At Levels 1 to $X-1$, full preemptive merge operations keep write-amp on par with our Full Merge baseline. At Level X, each entry is rewritten one extra time relative to Full Merge as there is no preemption at this level. At Levels $X+1$ to L, write-amp is the same as with Full Merge as we effectively perform full preemptive merge across groups of perfectly overlapping files. Hence, write-amp is the same as for 2L-Spooky in Equation 8. By setting X to Level L, Spooky becomes identical to Full Merge and can therefore be seen as a generalization of this baseline.

$$\text{write-amp with Spooky} = L \cdot \frac{T-1}{2} + \begin{cases} 0 & X = L \\ 1 & \text{otherwise} \end{cases} \tag{10}$$

Total Space-Amp Analysis. A partitioned merge entails a transient space-amp of at most $\frac{1}{T^{L-X}}$ as Level L is partitioned into at least T^{L-X} files of approximately equal sizes. A dividing merge operation also entails a transient space-amp of $\frac{1}{T^{L-X}}$ as the capacity at this level is a fraction of $\frac{1}{T^{L-X}}$ of the data size. The overall transient space-amp, which is the maximum of these two expressions, is therefore also $\frac{1}{T^{L-X}}$. By plugging this expression along with Equation 3 for durable space-amp into Equation 1, we obtain Spooky's total space-amp in Equation 11.

$$\text{space-amp with Spooky} = 1 + \frac{1}{T^{L-X}} + \frac{1}{T-1} \tag{11}$$

Controlling the Number of Open Files. While decreasing the parameter X reduces transient space-amplification, it also increases the number of files in the system. Many operating systems limit the number of files that can be simultaneously open to safeguard against resource over-utilization. It is important not to exceed this amount to prevent the system from crashing. We provide Equation 12 to model the maximum number of files with respect to X. The intuition for the equation is that Spooky performs partitioned merge across the largest $(L-X)$ levels, each of which contains T^{L-X} files. In addition, there is one file across each of the smaller $(X-1)$ levels. In section 6, we show that tuning X to $L-2$ provides a good practical balance between transient space-amp and the number of open files. Such a tuning also ensures that metadata volume remains modest and that files remain large enough to be written to storage using large sequential writes.

$$\text{maximum number of files} = (L-X) \cdot T^{L-X} + (X-1) \tag{12}$$

New Space/Write Trade-Offs. Overall, Spooky significantly improves write-amplification relative to Partial Merge while slightly increasing transient space-amplification. At the same time, Spooky significantly improves space-amplification relative to Full Merge while slightly increasing write-amplification. Overall, Spooky provides new and improved trade-offs in-between.

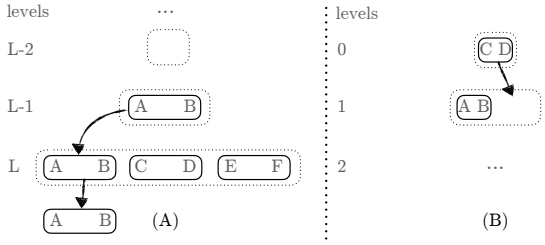


Figure 11: Spooky optimizes for skewed writes by avoiding compactions of non-overlapping files.

4.3 Optimizing for Skew

So far, we have been analyzing Spooky under the assumption of uniformly random insertions to reason about the worst-care write-amp and quality of service guarantees. With skewed updates, however, Spooky provides additional advantages. Specifically, Spooky avoids having to compact files at larger levels ($X+1$ to L) that do not overlap with newer updates during a partitioned merge. Figure 11 Part (A) illustrates an example with 2L-Spooky. As shown, all newer data at Level $L-1$ overlaps with just one of the files at Level L. Spooky therefore only compacts the two overlapping files while leaving all other files untouched during the next partitioned merge. In contrast, Full Merge would have to rewrite all of Level L as it stores every level as one file. Hence, Spooky not only matches Full Merge in the worst-case but also improves on it in the skewed case.

Spooky also allows dividing data at smaller levels into multiple files to optimize for skew. In Figure 11 Part (B), for instance, the buffer is flushed into Level 1, though its contents do not intersect with the existing file at Level 1. They are therefore flushed as a new small file rather than being compacted into the existing file at Level 1. This feature is particularly useful for handling sequential writes in the key space efficiently.

4.4 Supporting Tiered & Hybrid Merge Policies

While we have focused so far on how to apply Spooky to the leveled merge policy, our overarching vision is towards navigable systems that can learn and adapt across a wide design space to optimize for different workloads [6, 38–43]. Hence, it is important to show that Spooky generalizes to other merge policies as well. Figure 12 Part (A) shows how to do so with lazy leveling, which consists of one run at the largest level and multiple runs at each of the smaller levels.

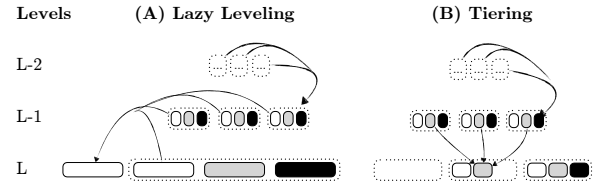


Figure 12: Spooky is compatible with any merge policy, including tiered and hybrid ones.

This example assumes 2L-Spooky to visualize the core idea clearly. Each shading corresponds to a disjoint part of the key space. As shown, each dividing merge operation compacts all data at Levels 0 to $L-2$ into a new run at Level $L-1$, and it partitions this run based on the file boundaries at Level L. During a partitioned merge, we draw one overlapping group of files from each of the runs at Level $L-1$ and compact them with the corresponding file at Level L. This design allows benefiting from even lower write-amp than before while still providing the same space-amp guarantees.

Figure 12 Part (B) applies Spooky to the tiered merge policy. Level L consists of multiple runs, and the data at both Levels $L-1$ and L is partitioned based on the file boundaries at the oldest run at Level L. A dividing merge operation compacts all data at Levels 0 to $L-2$ into a new run at Level $L-1$. During a partitioned merge, we add one run to Level L by merging one group of overlapping files from level $L-1$ at a time into Level L. This design trades durable space-amp for even better write-amp guarantees. This instance of Spooky represents an improved version of LWC-tree [83, 84], which carefully orchestrates merge operations across larger levels to restrict transient space-amp while executing them at a large enough granularity that garbage-collection overheads within the SSD are mitigated.

5 IMPLEMENTATION

This section discusses Spooky's implementation within RocksDB.

Encapsulation. There is an abstract compaction_picker.h class within RocksDB. Its role is to implement the logic of which files to compact under which conditions and how to partition the output into new files. We implemented Spooky by inheriting from this class and implementing the logic of Algorithm 2. Our implementation is therefore encapsulated in one file. This highlights an advantage of Spooky from an engineering perspective as it leaves all other system aspects (e.g., recovery, concurrency control, etc.) unchanged.

rLevels. We refer to levels in the RocksDB implementation as rLevels to prevent ambiguity with levels in our LSM-tree formalization introduced in Section 2.

rLevel 0. In RocksDB, rLevel 0 is the first rLevel in storage, and it is special: it is the only rLevel whose constituent files may overlap in terms of the keys they contain. When rLevel 0 has accrued α files flushed from the buffer, the compaction picker, and hence our Algorithm 2, is invoked. Once there are β files at rLevel 0, write throttling is turned on. When there are γ files at rLevel 0, the system stalls to allow ongoing compactions to finish. We tune these parameters to $\alpha = 4$, $\beta = 4$ and $\gamma = 9$ throughout our experiments[2]. Note that in effect, rLevel 0 can be seen as an extension of the buffer, and so it loosely corresponds to Level 0 in our LSM-tree formalization from Section 2. Flushing the buffer to rLevel 0 contributes an additive factor of one to write-amp, and so our implementation has a higher write-amp by one than the earlier write-models models (in Eqs. 8 and 10).

Level to rLevel Mappings. In RocksDB, all rLevels except rLevel 0 can only store one run (i.e., a non-overlapping collection of files). To support tiered and hybrid merge policies, whereby each level can contain multiple runs, we had to overcome this constraint. We did so by mapping each level in our LSM-tree formalization to one or more consecutive RocksDB rLevels. For example, in a tiered merge policy, Level 1 in our formalization corresponds to rLevels 1 to T, Level 2 to rLevels $T+1$ to $2 \cdot T$, etc.

Assuming Tiered/Hybrid Merge Policies. Our implementation has a parameter G for the number of greedy levels from largest to smallest that employ the leveling merge policy. Hence, when $G \geq L$, we have pure leveling, when $G = 0$ we have pure tiering, and when $G = 1$ we have lazy leveling. Thus, our implementation can assume different merge policies with different trade-offs for various application scenarios. The size ratio T can further be varied to fine-tune these trade-offs.

Full Merge. For our full merge baseline, we use our Spooky implementation yet with partitioned merge turned off. Hence, full preemptive merge is performed across all levels.

Avoiding Stalling. RocksDB's default compaction policy can perform internal rLevel 0 compactions, whereby multiple files at rLevel 0 are compacted into a single file that gets placed back at Level 0. The goal is to prevent the system from stalling when rLevel 0 is full (i.e., has γ files) yet there is an ongoing compaction into rLevel 1 that must finish before we trigger a new compaction from rLevel 0 to rLevel 1. We also enable rLevel 0 compactions within our Spooky implementation to prevent stalling. Specifically, whenever a full preemptive merge is taking place and we already have α or more files of approximately equal sizes, created consecutively, and not currently being merged, we compact these files into one file, which gets placed back at rLevel 0.

Concurrency. Our implementation follows RocksDB in that each compaction runs on background thread/s. We use the sub-compaction feature of RocksDB to partition large compactions across multiple threads. Our design allows for partitioned compactions, full preemptive compactions, and Level 0 compactions to run concurrently. Hence, there can be at most three compactions running simultaneously, though each of these compactions may be further parallelized using sub-compactions.

6 EVALUATION

We now show experimentally that Spooky is the only compaction granulation strategy that achieves high storage utilization, moderate write-amp, and high performance at the same time.

Platform. Our machine has an 11th Gen Intel i7-11700 CPU with sixteen 2.50GHz cores. The memory hierarchy consists of 48 KB of L1 cache, 512 KB of L2 cache, 16 MB of L3 cache, and 64GB of DDR memory. An Ubuntu 18.04.4 LTS operating system is installed on a 240GB KIOXIA EXCERIA SATA SSD. The experiments are running on a 960GB Samsung NVME SSD model MZ1L2960HCJR-00A07 with the ext4 file system.

[2]In the RocksDB code, α is referred to as level0_file_num_compaction_trigger, β as level0_slowdown_writes_trigger, and γ as level0_stop_writes_trigger.

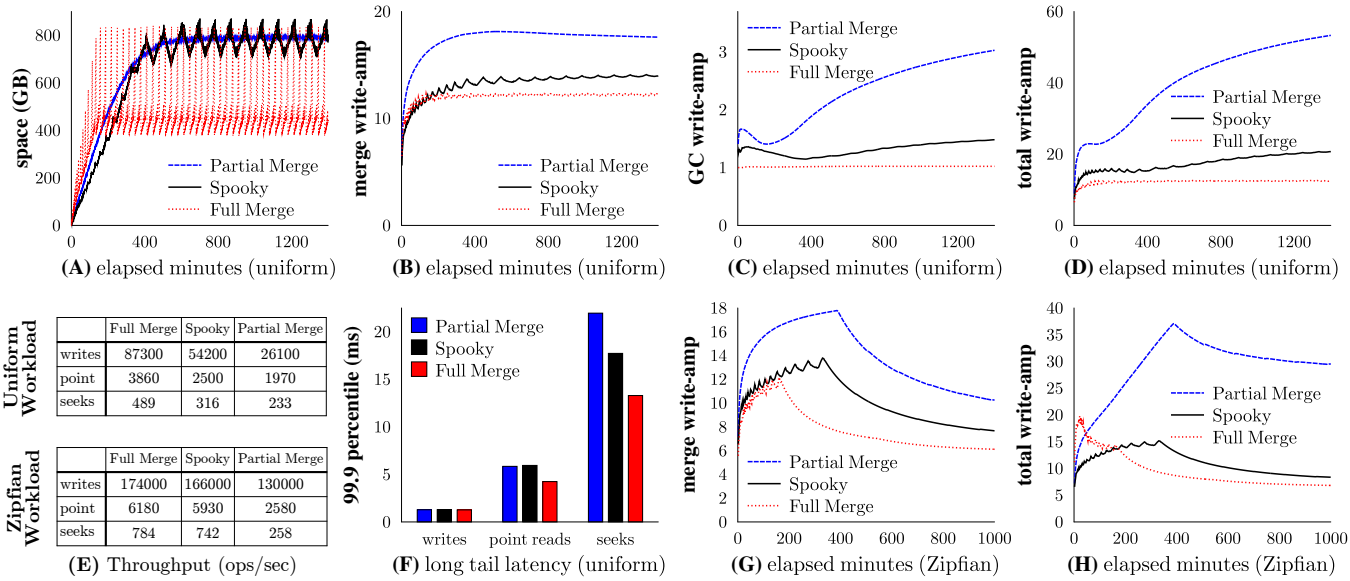


Figure 13: Full Merge exhibits low write-amp but cannot reach high storage utilization. Partial Merge can reach high storage utilization at the expense of high write-amp. Spooky is the only compaction granulation scheme that achieves high storage utilization, moderate write-amp, and good performance at the same time.

Setup. We use db_bench to run all experiments as it is the standard tool used to benchmark RocksDB. Every entry consists of a randomly generated 16B key and a 512B value. Unless otherwise mentioned, all baselines use the leveling merge policy, a size ratio of 5, and a memtable size of 64MB. Bloom filters are enabled and assigned 10 bits per entry. Dynamic capacity adaptation is applied for all baselines. The data block size is 4KB. We use one application thread to issue inserts/updates, and we employ sixteen background threads to parallelize compactions.

We use the implementation from Section 5 to represent Spooky and Full Merge. For Spooky, we set the parameter X, the level into which Spooky performs dividing merge operations, to $L-2$, the third largest level, though we also later experiment with other tunings of X. For Partial Merge, we use the standard compaction policy of RocksDB with the default file size of 64MB.

Experimental Control. Before each experimental trial, we delete the database from the previous trial and reset the SSD using the fstrim command. This allows the SSD to reclaim space internally. We then fill up the drive from scratch for the next trial. This ensures that subsequent experimental trials do not impact each other.

To elicit and measure the influence of SSD garbage-collection, it is crucial to run all experiments at a high storage utilization so that the SSD is stressed for free space. However, our Full Merge baseline is unable to reach a storage utilization of over 50%, which is also its core flaw. For this reason, the Full Merge baseline employs a smaller logical data size in some experiments.

Performance Monitoring. We run the du Linux command to monitor the physical database size every five seconds. We also run the nvme command every two minutes to report the SSD's internal garbage-collection write-amp. We use RocksDB's internal statistics to report the number of bytes flushed and compacted every two minutes to allow computing write-amp due to compactions. We use db_bench to report throughput for queries and updates.

Spooky Enables High Storage Utilization. Figure 13 Parts (A) to (D) measure space and write-amplification over time for the different baseline as we issue unique uniformly random insertions followed by uniformly random updates for one day.

Part (A) shows that Partial Merge exhibits negligible transient space-amp, so it can store more logical data (644GB in this case). In contrast, Full Merge exhibits a sawtooth-shaped curve due to its massive transient space-amp. It can therefore support a smaller logical data size (369GB in this experiment). For Spooky, the curve is also sawtooth-shaped due to its partitioned merge operations. The teeth are significantly smaller than with Full Merge, though, due to the finer merge granularity. This allows Spooky to match Partial Merge in terms of logical data size (also 644GB in this experiment).

Spooky Reduces Compaction Overheads. Figure 13 Part (B) measures write-amp due to compaction operations over time. Write-amp with Spooky is slightly higher than with Full Merge since Spooky stores ≈ 2x more logical data. At the same time, Spooky improves on Partial Merge by ≈ 30% while matching it in terms of logical data size. The reason is that Spooky eliminates superfluous edge merging while leveraging preemption across most levels.

Spooky Reduces Garbage-Collection. Figure 13 Part (C) measures the SSD's garbage-collection write-amp over time. Full Merge exhibits no overheads since all writes and deletes are large and sequential, and because the data size is smaller so the SSD is not stressed for free space. Partial Merge exhibits the highest overheads because its many simultaneous small compactions cause many files with different lifespans to become interspersed within the same SSD erase units. With Spooky, garbage-collection is significantly cheaper. The reason is that it writes fewer though larger files simultaneously. This leads to less interspersing of files with disparate lifespans within the SSD and hence > 2x cheaper garbage-collection.

Figure 13 Part (D) reports total write-up, the product of the write-amps due to compaction and garbage-collection from Parts (B) and

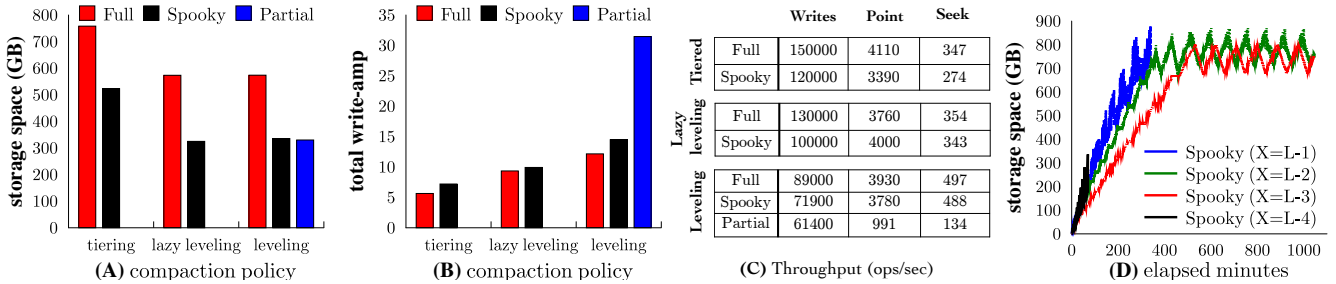


Figure 14: Spooky is compatible across a wide spectrum of compaction policies, including tiering and lazy leveling (Parts A-C). Spooky generally works best with $X = L - 2$ to balance transient space-amp and the number of open files (Part D).

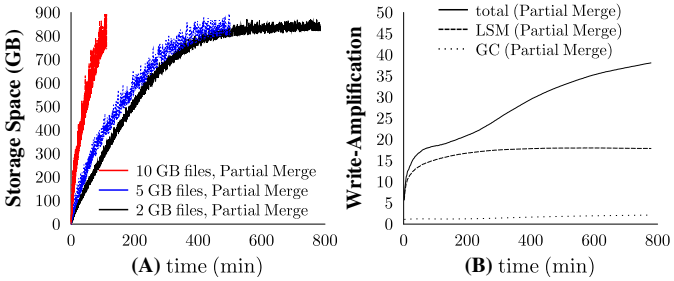


Figure 15: Large files with Partial Merge cause the system to crash due to transient space-amp (A). With the largest file size we tried that did not crash (2GB), SSD garbage-collection is still a severe performance issue (B).

(C). Spooky reduces total write-amp by $\approx x2.5$ relative to Partial Merge while matching it in terms of logical data size.

Spooky Improves Read/Write Performance. The top table in Figure 13 Part (E) compares throughput for the three baselines with a uniform workload (to three significant figures). The point reads and seek measurements were gathered by adding one thread to issue queries in parallel to the thread issuing writes.

Spooky improves update throughput relative to partial merge by approximately the same factor by which it improves on its total write-amp. This shows that reductions in total write-amp translate to direct improvements in write throughput.

Furthermore, Spooky significantly improves query performance relative to Partial Merge because there is less background write-amp to impede queries. Full Merge exhibits the best throughput results overall, but this is because its dataset is smaller. The 99.9 percentile latency in Figure 13 Part (F) shows that Spooky's throughput improvements do not involve a sacrifice in performance stability.

Spooky Improves Skewed Workloads. We extended db_bench to support Zipfian workloads and ran an experiment that first fills up a database with uniformly random unique insertions followed by ten hours of Zipfian point updates. The Zipf distribution parameter is set to one. The hot keys are randomly distributed across the key space, and they change every two hours to reflect changing hotspots. As in previous experiments, Spooky and Partial Merge employ a logical data size of 644GB while Full Merge uses a smaller logical data size of 369GB.

Figures 13 Parts (G) shows write-amplification arising from compaction operations of the LSM-tree, while Part (H) shows total write-amplification, which also accounts for SSD garbage-collection. Spooky significantly improves on Partial Merge while being able to store the same data size. The bottom part of Figure 13 Part (E) gives throughput measurements for Zipfian writes, point reads, and seeks. Again, Spooky significantly improves on Partial Merge and almost matches Full Merge while being able to store nearly 2x more data.

Spooky Applies Across Numerous Merge Policies. Figures 15 Parts (A) to (C) evaluate Spooky with the Tiered and Lazy Leveled merge policies, described in Section 4.4, along with the previous leveled strategies. All baselines have 250GB of logical data, and the workload is uniformly random. Relative to Full Merge, Spooky significantly reduces transient space-amplification without significantly impacting write-amplification for any merge policy.

Figures 15 Part (C) provides corresponding throughput figures for writes as well as for point reads and seeks issued from a parallel thread. From the theory, we would expect the leveling baselines to exhibit the best read performance as there are fewer runs to access. Interestingly, we observe that leveled partial merge (RocksDB's default compaction policy) exhibits the worse read performance. The reason is that its background write-amplification is far higher, and this interferes with read performance. In contrast, all Spooky baselines significantly reduce write-amplification and thereby improve both read and write throughput at the same time.

Tuning X. Figure 14 Part (D) explores different tunings of X for Spooky under the leveling merge policy and a logical data size of 644GB. With X tuned to $L - 1$, the second largest level, the system runs out of space and crashes since large files are merged at a time. As we decrease X to $L - 2$ and $L - 3$, space requirements decrease at a diminishing rate. With X set to $L - 4$, the system crashes due to having too many open files (the limit in our OS is set to 1024). Thus, setting X to $L - 2$ is sufficient for most practical purposes.

Larger Files Only Slightly Improve Partial Merge. It is tempting to think that using larger files with Partial Merge would eliminate GC overheads by causing larger units of data to be written and erased all at once across SSD erase units. In Figure 15 Part (A), we try three large file sizes. With 10GB and 5GB files, the database crashed after exceeding the SSD storage capacity, and so the respective curves are incomplete. The reason is that larger files entail a higher transient space-amp that causes the physical data size to vary widely, as evidenced by the noisier curves. Since 2GB files were the largest we succeeded in finishing the experiment with, we measure write-amp for Partial Merge with 2GB files in Figure 15 Part (B). GC overheads are slightly lower than in Figure 13 Part (C), which uses 64MB files. However, these overheads are still considerable and result in higher total write-amp than with Spooky. The

reason is that even with 2GB files, concurrent compactions cause these files to become physically interspersed. These findings are in line with contemporaneous work [11]. Overall, GC overheads with Partial Merge are not a tuning issue but rather an intrinsic problem, to which Spooky offers a novel solution.

Key Takeaway. Figure 1 measures total write-amp as we increase storage utilization. The size ratio T is set to 10. Full Merge cannot exceed a storage utilization of over 50%. Partial Merge reaches 80% storage utilization but its write-amp is exorbitant. In contrast, Spooky matches Partial Merge in terms of storage utilization while achieving, in this case, 3x lower total write-amp. Spooky is therefore the best choice enabling excellent storage utilization and performance at the same time.

7 RELATED WORK

Other Merge Policies. In addition to tiered/leveled merge policies that offer logarithmic read/write scalability (see Sections 2 and 3), other merge policies with different scalability trade-offs have been proposed. Bounded depth merge policies fix the number of runs to a constant to limit query costs, yet this causes write-amp to increase more rapidly [63, 64]. In contrast, LSM-bush scales write-amp at a slower rate of $O(\log \log N)$ yet pays more in terms of query costs. Both of these strains of merge policies have been described as employing Full Merge, and so such policies can be readily combined with Spooky to restrict transient space-amp.

Compaction Granulation Techniques. Since the tiered merge policy was first deployed by Cassandra in conjunction with Full Merge, several recent works propose finer granulation techniques to limit transient space-amp specifically with tiering [73]. They propose to partition the multiple runs at each level based on (1) the hashes of keys [80], (2) randomly [69], (3) into equally-sized files (or in other words using Partial Merge) [66, 92], or (4) based on the file boundaries at the oldest run in the system [83, 84]. In the context of bounded-depth policies, striped compaction, Striped Compaction in Hbase performs full key-space partitioning

LSM-trie partitions the multiple runs at each level into perfectly overlapping files based on the hashes of their keys, but it forgoes the ability to perform range reads [80]. PebblesDB partitions the key space randomly, which can occasionally result in excessively large partitions [69]. Some designs partition runs into equally sized files [66, 92], yet this leads to superfluous edge merging and thus higher write-amp. LWC-tree [83, 84] partitions runs at smaller levels based on file boundaries at larger levels while dividing files at larger levels based on size. Still, LWC-tree does not prevent the system from merging multiple partitions at once into the largest level, and so the system may crash. In contrast, Spooky ensures only one large merge operation happens at a time, while at the same time forcing all writes within each level to be sequential to also restrict GC garbage-collection. Furthermore, Spooky is generalized across the whole space of merge policies rather than just tiering.

The partitioned exponential file is a collection of non-overlapping leveled LSM-trees, each of which is allocated contiguously in the LBA space [46]. As the sizes of partitions change, fragmentation is created leading to either poorer storage utilization or higher write-amp for de-fragmentation. Furthermore, no transient space-amp guarantees are provided as multiple partitions may merge into their largest level at the same time. Spooky avoids the first problem by allocating data within each level through a layer of indirection, and the second by merging into the largest level one partition at a time. In addition, Spooky is applicable across all merge policies, and it addresses the problem of SSD garbage-collection.

Richer Hardware. Zoned Namespaces (ZNS) [11, 17, 85], no-FTL [77] and computational storage [68] have recently been proposed for controlling the placement of LSM-tree data across erase blocks within storage to cheapen garbage-collection. Spooky is also designed to address the problem of SSD garbage-collection, yet it does so through the standard block device interface. It is therefore applicable across a wider range of storage devices. At the same time, Spooky lends itself to ZNS and no-FTL by writing and disposing of large swaths of data sequentially. Thus, Spooky can be integrated with such schemes to further reduce garbage-collection overheads.

In addition, specialized hardware such as FPGAs [37, 91], heterogeneous storage [87], data processors [26], open-channel SSDs [78], GPUs [5], and non-volatile memory [48, 52, 86] have all been proposed as means of speeding up LSM-trees. Other hardware techniques leverage multi-core CPUs [89], lock-free synchronization [34] and distributed processing [2, 53]. Spooky is fully compatible with these approaches in that its only task is to select which files to merge. The execution of compaction is orthogonal and can easily be performed on different types of hardware and storage.

Key-Value Separation. Recent work proposes to store the values of data entries in an external log [15, 58, 81] or in a separate tiered LSM-tree [55]. This improves write-amp while sacrificing scan performance. Spooky can be combined with the LSM-tree component/s in such designs for better write-amp vs. space-amp balances.

In-Memory Optimizations. Various optimizations have been proposed for LSM-tree's in-memory data structures including adaptive or learned caching [60, 79, 82], leaned fence pointers [18], tiered buffering [7, 14], selective flushing [7], smarter Bloom filters [19, 20, 88, 93, 95] or replacements thereof [25, 29, 44, 62, 70, 90, 91], and materialized indexes for scans [94]. Spooky is fully complementary with such works as it only impacts the decision of which files to merge and how to partition the output.

Performance Stability. Recent work focuses on maintaining stable performance during compaction operations [59]. Various prioritization [8, 9, 47], synchronization [74], deamortization [10, 54], and throttling techniques [56] have been proposed. Our Spooky implementation on RocksDB performs compaction on concurrent threads to avoid blocking the application, yet it could benefit from these more advanced techniques to more evenly schedule the overheads of large compaction operations in time.

8 CONCLUSION

Existing methods for granulating LSM-tree compactions either waste most of the available storage capacity or involve exorbitant write-amplification. We introduce Spooky, the first approach to achieve high storage utilization and moderate write-amplification at the same time.

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. *ATC* (2008).

[2] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *PVLDB* (2015).
[3] Apache. 2022. Cassandra. *http://cassandra.apache.org* (2022).
[4] Apache. 2022. HBase. *http://hbase.apache.org/* (2022).
[5] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D Owens. 2018. GPU LSM: A dynamic dictionary data structure for the GPU. In *IEEE IPDPS*. IEEE.
[6] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. *EDBT* (2016).
[7] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. *USENIX ATC* (2017).
[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *USENIX ATC*.
[9] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2020. Silk+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads. *TOCS* (2020).
[10] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-Oblivious Streaming B-trees. *SPAA* (2007).
[11] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs.
[12] Matias Bjørling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. 2013. The Necessary Death of the Block Device Interface. *CIDR* (2013).
[13] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM* 13, 7 (1970), 422–426.
[14] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *PVLDB* 11, 12 (2018), 1863–1875.
[15] Helen H W Chan, Yongkun Li, Patrick P C Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. *ATC* (2018).
[16] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime data processing at facebook. In *SIGMOD*.
[17] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. 2020. A New LSM-style Garbage Collection Scheme for ZNS SSDs. In *USENIX HotStorage*.
[18] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From wisckey to bourbon: A learned index for log-structured merge trees. In *USENIX OSDI*.
[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. *SIGMOD* (2017).
[20] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *TODS* (2018).
[21] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *SIGMOD* (2016).
[22] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. *SIGMOD* (2018).
[23] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *SIGMOD*.
[24] Niv Dayan, Martin Kjær Svendsen, Matias Bjorling, Philippe Bonnet, and Luc Bouganim. 2013. EagleTree: exploring the design space of SSD-based algorithms. *VLDB* (2013).
[25] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD*.
[26] Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, et al. 2021. The end of Moore's law and the rise of the data processor. *PVLDB* (2021).
[27] Peter Desnoyers. 2014. Analytic models of SSD write performance. *ACM TOS* (2014).
[28] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2021. Toward a better understanding and evaluation of tree structures on flash ssds. *VLDB* (2021).
[29] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *CoRR* (2021).
[30] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *SIGMOD*.
[31] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. *CIDR* (2017).
[32] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *USENIX FAST*.
[33] Facebook. 2022. RocksDB. *https://github.com/facebook/rocksdb* (2022).
[34] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling Concurrent Log-Structured Data Stores. *EuroSys* (2015).
[35] Google. 2022. LevelDB. *https://github.com/google/leveldb/* (2022).
[36] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. *ASPLOS* (2009).
[37] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *SIGMOD*.
[38] Stratos Idreos, Manos Athanassoulis, Niv Dayan, Demi Guo, Mike S Kester, Lukas Maas, and Kostas Zoumpatianos. 2015. Past and future steps for adaptive storage data systems: From shallow to deep adaptivity. In *Real-Time Business Intelligence and Analytics*. Springer.
[39] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Learning Key-Value Store Design. *arXiv preprint arXiv:1907.05443* (2019).
[40] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *CIDR*.
[41] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE DEBULL* 41, 3 (2018), 64–75.
[42] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike Kester, Niv Dayan, Demi Guo, Minseo Kang, et al. 2019. Learning data structure alchemy. *IEEE DEBULL* (2019).
[43] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. *SIGMOD* (2018).
[44] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *USENIX ATC*.
[45] H. V. Jagadish, P. P. S. Narayan, Sridhar Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. *VLDB* (1997).
[46] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. 2007. The Partitioned Exponential File for Database Storage Management. *VLDBJ* (2007).
[47] Peiquan Jin, Jianchuan Li, and Hai Long. 2021. DLC: A New Compaction Scheme for LSM-tree with High Stability and Low Latency. In *EDBT*.
[48] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *USENIX ATC*.
[49] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A scalable bottom-up approach for building data series indexes. *PVLDB* (2018).
[50] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut Palm: Static and Streaming Data Series Exploration Now in your Palm. In *SIGMOD*.
[51] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *VLDBJ* (2019).
[52] Cheng Li, Hao Chen, Chaoyi Ruan, Xiaosong Ma, and Yinlong Xu. 2021. Leveraging NVMe SSDs for Building a Fast, Cost-effective, LSM-tree-based KV Store. *ACM TOS* (2021).
[53] Jianchuan Li, Peiquan Jin, Yuanjin Lin, Ming Zhao, Yi Wang, and Kuankuan Guo. 2021. Elastic and Stable Compaction for LSM-tree: A FaaS-Based Approach on TerarkDB. In *CIKM*.
[54] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, Ke Yi, and Robin Jun Yang. 2010. Tree Indexing on Solid State Drives. *PVLDB* 3, 1-2 (2010), 1195–1206.
[55] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *USENIX ATC*.
[56] Junkai Liang and Yunpeng Chai. 2021. CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency. In *ICDE*.
[57] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. *FAST* (2016).
[58] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. *FAST* (2016).
[59] Chen Luo and Michael J Carey. 2019. On performance stability in LSM-based storage systems (extended version). *VLDB* (2019).
[60] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* (2020).

[61] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *EDBT*.

[62] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *SIGMOD*.

[63] Qizhong Mao, Steven Jacobs, Waleed Amjad, Vagelis Hristidis, Vassilis J Tsotras, and Neal E Young. 2021. Comparison and evaluation of state-of-the-art LSM merge policies. *VLDB Journal* (2021).

[64] Claire Mathieu, Carl Staelin, Neal E Young, and Arman Yousefi. 2014. Bigtable merge compaction. *arXiv preprint arXiv:1407.3008* (2014).

[65] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *VLDB* (2020).

[66] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In *ACM SOCC*.

[67] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[68] Ivan Luiz Picoli, Philippe Bonnet, and Pinar Tözün. 2019. LSM management on computational storage. In *DaMoN*.

[69] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. *SOSP* (2017).

[70] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *PVLDB* 10, 13 (2017), 2037–2048.

[71] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *SIGMOD*.

[72] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *VLDB* (2021).

[73] ScyllaDB. 2022. ScyllaDB. *https://github.com/scylladb/scylladb* (2022).

[74] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. *SIGMOD* (2012).

[75] Radu Stoica and Anastasia Ailamaki. 2013. Improving Flash Write Performance by Using Update Frequency. *PVLDB* (2013).

[76] Risi Thonangi and Jun Yang. 2017. On Log-Structured Merge for Solid-State Drives. *ICDE* (2017).

[77] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management.. In *EDBT*.

[78] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. *EuroSys* (2014).

[79] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. 2020. AC-key: Adaptive caching for LSM-based key-value stores. In *USENIX ATC*.

[80] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. *USENIX ATC* (2015).

[81] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 305–318.

[82] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. *PVLDB* (2020).

[83] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. 2017. A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores. *MSST* (2017).

[84] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *TOS* (2017).

[85] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. 2019. Geardb: A gc-free key-value store on hm-smr drives with gear compaction. In *USENIX FAST*.

[86] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *USENIX ATC*.

[87] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *ACM SOCC*.

[88] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. 2017. Building an Efficient Put-Intensive Key-Value Store with Skip-Tree. *TPDS* (2017).

[89] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: throughput-oriented programmable processing in memory. *HPDC* (2014).

[90] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. *SIGMOD* (2018).

[91] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *USENIX FAST*.

[92] Weitao Zhang, Yinlong Xu, Yongkun Li, and Dinglong Li. 2016. Improving write performance of LSMT-based key-value store. In *IEEE ICPADS*.

[93] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. 2018. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. *HotStorage* (2018).

[94] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *USENIX FAST*.

[95] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices. In *DaMoN*.