Benchmarking, Analyzing, and Optimizing Write Amplification of Partial Compaction in RocksDB

Ran Wei*
Boston University
weiran22@bu.edu

Zichen Zhu*
Boston University
zczhu@bu.edu

Andrew Kryczka
Meta
andrewkr@meta.com

Jay Zhuang
Snowflake
jay.zhuang@yahoo.com

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Log-structured merge (LSM) trees are widely used because they offer high ingestion throughput via appending incoming data in a memory buffer, which, when filled up, is *flushed* to storage as a sorted run. To reduce space amplification and facilitate queries, LSM-trees periodically merge data on storage to form larger but fewer sorted runs, with a process termed *compaction*. In commercial LSM-based systems like RocksDB, sorted runs are often stored as a set of small files, allowing *partial compaction* (i.e., of a subset of a sorted run), reducing the worst-case compaction latency without increasing the amortized compaction cost. Since a sorted run consists of multiple files, the decision of which file to compact (*file-picking policy*) significantly impacts the system's write cost, quantified by Write Amplification (WA). Thus, finding the optimal file-picking policy remains an open research question.

In this paper, we focus on the four different file-picking policies offered by RocksDB (*MinOverlappingRatio*, *RoundRobin*, *OldestLargestSeqFirst*, and *OldestSmallestSeqFirst*). While some heuristics to guide which compaction policies to use are given in the literature and the documentation of RocksDB, we highlight that an up-to-date, in-depth, and comprehensive analysis and guidelines for the partial compaction policies in RocksDB is needed, as earlier guidelines are now *obsolete* in the presence of new optimizations/implementations of RocksDB. Further, we investigate the headroom for improvement by comparing each policy with the optimal WA, obtained by enumerating all possible picking decisions. From our comprehensive experimental results, we distill 10 key observations, which rectify obsolete heuristics/observations and also provide valuable insights for LSM-based systems that employ partial compactions.

1 INTRODUCTION

LSM-trees are everywhere. Log-structured Merge-trees (LSM-trees) [27, 31] offer a high ingestion rate and low query latency with bounded space amplification and are thus widely adopted in industry, including RocksDB [20] at Facebook, LevelDB [22] and BigTable [9] at Google, Accumulo [4], AsterixDB [3], Cassandra [5], HBase [6], IoTDB [39], ScyllaDB [36] at Apache, WiredTiger [43] at MongoDB, X-Engine [24] at Alibaba, and Dynamo [15] at Amazon. LSM-trees are also used as the underlying storage engines of relational database systems (e.g., RocksDB in TiDB [23], MyRocks [19] and Pebble [11] in CockroachDB [10]). Variants of LSM-trees supporting spatial and graph data have also been developed [29, 37].

In LSM-trees, incoming writes are batched into a memory buffer, which is sorted and flushed to disk whenever it fills up. On-disk data are organized as *sorted runs* in multiple levels where the capacity of each level grows by a constant *size ratio*. When a level is saturated, a *compaction* is triggered to merge it with the overlapping sorted data from the next level, thus amortizing the cost of having a fully sorted last level (where most data resides) and supporting fast reads.

Compactions in LSM-trees lead to Write Amplification (WA). To propagate data entries to the last level, they are repeatedly read and written via iterative compactions, leading to *Write Amplification* (WA). Formally, WA is defined as the ratio between the bytes written to storage and the bytes ingested to the database [7]. As such, WA is directly related to the amount of data written to disk, and thus WA reflects the overall write cost and device endurance. Specifically, higher WA leads to higher write effort and shorter device endurance. We highlight that different compaction strategies have substantially different WA [12, 13]. Considering an LSM-tree with a *size ratio* T, when comparing the two fundamental compaction approaches, *leveling* has $T\times$ higher WA than *tiering*, while offering $T\times$ lower read amplification (RA) and thus faster read queries.

Note that a *full compaction* (i.e., one that involves an entire level) would take a large amount of time, especially when targeting the deeper levels of the LSM-tree. This results in large latency spikes during compaction as well as temporary storage amplification since older data are kept in their original sorted runs until the compaction finishes. To address this, industry-level LSM-tree key-value storage engines like LevelDB, RocksDB, and Pebble typically employ *partial compactions* to amortize the cost of a *large* compaction into several *small* compactions. These systems partition each sorted run into smaller non-overlapping files so that a single file can be picked for merging at compaction time [34]. The key decision of partial compaction is which file to pick, a question that is understudied despite having substantial implications in systems performance and overall behavior and is the focus of this study.

In this work, we focus on RocksDB because it has both practical and research value for the following reasons. First, it is one of the most popular open-source LSM engines and has the widest adoption in industry [21] and thus our observations benefit all users of RocksDB, which include several organizations and all applications/systems built on top of it. Second, the partial compaction policies supported by RocksDB are a superset of those employed by other LSM engines. Specifically, LevelDB only supports *RoundRobin* and Pebble only supports *MinOverlappingRatio*, while RocksDB offers more flexible file-picking policies

*The first two authors have equal contribution and are listed alphabetically.

other than the above two. Note that both LevelDB and Pebble are based on older versions of RocksDB, which leads to higher WA. As a result, the new RocksDB design is the key motivation for revisiting existing guidelines.

Challenge 1: The WA difference between existing file picking policies is not fully investigated. Although the amortized WA for the partial compaction remains asymptotically the same as the full compaction, the actual WA varies a lot when selecting different files to compact for specific workloads. Earlier studies [27, 34] mostly focus on the difference between partial and full compactions and do not investigate all the available file-picking policies. Further, the impact of the workload (e.g., update distribution) and the underlying hardware (e.g., which storage device) on WA remains unclear, leaving LSM users without clear guidelines on which picking policy to choose for certain workloads.

Challenge 2: New file-picking policies and optimizations warrant a new study. Notably, the *RoundRobin* partial compaction policy [18], short as *RR*, has been re-implemented in RocksDB to ensure each key in a level can be uniformly compacted [26], which yields a substantial WA reduction for update-intensive workloads, as we will show in this paper. In addition, RocksDB refines the existing file splitting mechanism [17] in a compaction to reduce WA based on the key boundaries at the *grandparent* level. That is, in a compaction from level i to $i+1$, level $i+1$ is the parent level, and if level $i+2$ has files that overlap with this compaction in the key space, level $i+2$ is the grandparent level. The new splitting mechanism produces files in the parent level (unless it is the deepest level) that overlap less data in the grandparent level. This optimization reduces the size of future compactions involving those files, and may affect the benefit of existing file-picking policies. Previous analysis of the impact of compaction on WA [34] assumed both a different set of file-picking policies and a rudimentary form of file splitting (which was the only one supported at the time), thus a new refreshed study is warranted.

Challenge 3: The WA improvement headroom between existing policies and an ideal one is still unclear. While modeling the impact of compaction and, specifically file-picking policies is very hard, we can experimentally find the optimal compaction and file-picking policies. Specifically, by replaying a specific ingestion workload and enumerating all possible decisions, we can obtain the minimum WA for this workload. However, due to the exponential search space, there is no earlier work that quantified the minimum WA and the improvement headroom.

Benchmarking, Analyzing, and Optimizing WA of Different Partial Compaction Policies in RocksDB. To answer these questions, we conduct a detailed experimental study to identify the impact of varying file-picking policies on WA. We experiment using RocksDB (v8.8.1) that supports the up-to-date *RoundRobin* policy and incorporates a new file splitting mechanism to reduce overlapping ratio with deeper levels. In addition, we also implement a brute-force search algorithm to find the best file-picking policy throughout all compactions for any given workload. While the brute-force approach can only be used as an offline study due to the exponential search space, we can still quantify the difference between the optimal strategy and other file-picking policies for partial compaction. Specifically, we compare the optimal file-picking decisions with *MinOverlappingRaio*, and we are able to explain the instability issue of *MinOverlappingRatio* using observations drawn from the comparison result.

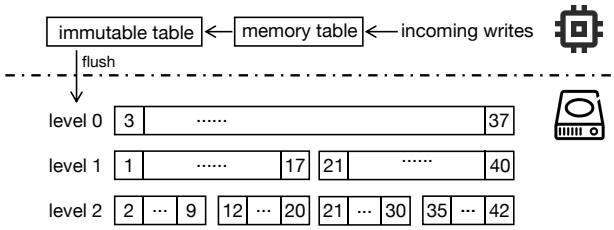

Figure 1: Classical LSM-tree architecture.

To validate our observation, we further refine *MinOverlappingRatio* and develop *RefinedMOR* that reduces the average WA by 2% and offers higher stability. Overall, we conduct extensive experimentation to derive insights for partial compaction useful for RocksDB and other LSM-based systems.

Contributions. We summarize our contributions as follows.

- We identify that prior tuning guidelines and observations need to be updated considering new implementations/optimizations in RocksDB. To address this, we benchmark *MinOverlappingRatio*, *RoundRobin*, *OldestLargestSeqFirst*, and *OldestSmallestSeqFirst* under different ingestion workloads.
- To investigate the headroom for the optimal WA, we design a brute-force search algorithm, which enumerates all possible file picks in each compaction to find the minimum WA for a given workload. Since the search space is exponentially large, we prune it with an incremental approach that allows for efficiently finding the minimum WA for small workloads. We summarize our findings by distilling 10 observations.
- Given the offline algorithm that finds the best files to be compacted for specific workloads, we perform an apples-to-apples comparison between the most frequently used *MinOverlappingRatio* policy and the optimal one. This allows us to explain why *MinOverlappingRatio* may have unstable WA and lead to larger WA compared to the optimal one.
- We further validate our observation on the *MinOverlappingRatio* policy by refining it as *RefinedMOR* with applying our observations, and we experimentally show that *RefinedMOR* reduces the average WA by 2.2% and the quartile deviation by 75.1%, compared to *MinOverlappingRatio* in 40GB workloads.
- Our artifacts[1] that include the benchmark and the baseline system with the optimal searching strategy and *RefinedMOR*, are publicly available for exploration and reproducibility.

2 BACKGROUND

In this section, we review the LSM-tree background and introduce existing partial compaction policies in RocksDB. We also elaborate on the most recent design and implementation of compactions in RocksDB to explain why prior guidelines in the system's documentation and research literature need to be refreshed.

LSM-tree Basics. In LSM-trees, each insert, update, and delete is treated as a new key-value entry (the value of each delete is a *tombstone* [33] marker). All these incoming entries are first buffered in a memory table, and when the memory table reaches a predefined capacity, it is *sealed* as an *immutable* memory table and is added to the flush queue. Note that updates and deletes may trigger in-place changes if older entries with the same key exist. When flushing an immutable memory table on disk, all the key-value pairs in this table are sorted and attached with

[1] https://github.com/BU-DiSC/Benchmark-Analyze-Optimize-Partial-Compaction-in-RocksDB-Codebase

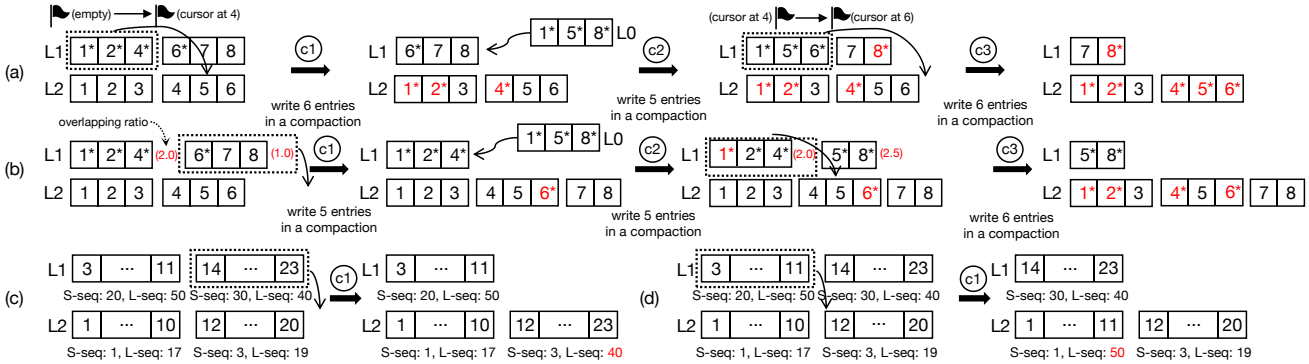

Figure 2: Examples of four partial compaction policies (a) *RR*, (b) *MOR*, (c) *OLSF*, (d) *OSSF* in RocksDB. *key denotes a duplicate key, and a red key indicates that an older version is discarded during compaction. S-seq (resp. L-seq) represents the smallest (resp. largest) sequence number in a file (all the keys are tagged with a sequence number during flushing). When there are multiple compactions, we use (c1), (c2) to represent the first and the second compaction, and so forth.**

a monotonically increasing *sequence number*. Data on disk is organized into multiple levels where the capacity of each level grows exponentially with a constant size ratio (noted by T).

In a *leveling* LSM-tree, where each level forms a single *sorted run*, newly flushed data from the memory buffer may have to be merged with existing data in level 0 (abbreviated as L0). Similarly, when level i reaches a predefined capacity, all the data in level i are merged with existing data in level $i+1$ to form a larger sorted run in level $i+1$, and older entries with the same key are discarded. This process is termed *compaction*. For ease of notation, compactions triggered at level i will be noted as L<i> compactions (e.g., *L0 compactions*). In contrast, a *tiering* LSM-tree allows up to $T-1$ sorted runs per level, which triggers fewer compactions, since incoming sorted runs from level i are simply appended in level $i+1$ without compaction. This leads to having more sorted runs per level and, thus, a higher read cost since every sorted run requires its own binary search.

Navigating the Read vs. Write Trade-Off. Overall, in leveling LSM-trees, the incoming entries are written in each level $O(T/2)$ times on average, and the overall WA can reach $O(T \cdot L/2)$, where L is the number of levels. On the other hand, in tiering LSM-trees, WA is $O(L)$ since each sorted run will only be merged once per level. By selecting between tiering vs. leveling and tuning the size ratio, we can navigate the read/write trade-off of the LSM-tree design space [12, 30, 34]. In addition to this, practical LSM-tree systems also support *hybrid compaction strategies*. For example, in RocksDB's default setting, the shallowest level on disk (L0) follows a tiering design that allows up to four sorted runs, while all other levels follow the leveling strategy, also termed *1-leveling* [34]. Figure 1 shows an example of a classical leveling LSM-tree with three levels.

File-based Partial Compaction. Classical leveling and tiering LSM-trees employ *full* compaction, in which all data in a level has to be involved in one compaction. When deeper (i.e., larger) levels are compacted, the compaction takes a substantially longer time to complete, and there could also be a temporarily high spike for storage space occupation since old data that is serving concurrent queries cannot be removed until the compaction and the in-flight queries finish. If a crash interrupts such a long-running compaction, it will restart post-recovery.

To alleviate the latency spike, the space occupation spike, and crash-recovery inefficiency, leveling LSM-trees usually store a sorted run as multiple *SST* (Sorted String Table) files with a predefined size, which enables a file-based partial compaction [16]. By this decomposition, each compaction in a leveling LSM-tree just needs to pick one SST file in level i (if level i fills up) to merge with SST files in level $i+1$ that have overlapping key ranges. In literature, when a compaction in level i is triggered (i.e., we need to pick a file in level i to compact), level $i+1$ is often referred as the *parent* level, and level $i+2$ is referred as the *grandparent* level for every compaction compacting data from level i to level $i+1$. As such, a full compaction between two levels is fragmented into several small compactions where each compaction only touches a few files. Since the partial compaction strategy effectively amortizes the compaction cost without harming the query performance, it is widely used in both academic LSM-tree prototypes (e.g. Spooky [14], and FGKV [38], MirrorKV [41]) and industry LSM-tree systems (e.g., LevelDB, RocksDB, and Pebble).

File-Picking Policies in RocksDB. File-picking (a.k.a. partial compaction) policies control which SST file of a level to compact. We detail the four key file-picking policies below.

(a) `RoundRobin` (*RR*) picks a file in a level in a round-robin manner in the key space. We introduce the classical design and discuss later a newer implementation in RocksDB. For each level, RR maintains a *cursor* that represents the largest key, which was lastly compacted in this level. Initially, this cursor is empty, in which case we always pick the first file in RR. After compacting the first file, the cursor is moved to the largest key involved in the last compaction. In Figure 2(a), we see that since the initial cursor is empty when L1 fills up for the first time, the first file at L1 is picked to compact. After compacting the first file, the cursor moves to key 4, the largest key in the last compaction. When L1 fills up again, *RR* will choose the first file (from left to right) with a maximum key larger than the cursor (i.e., the first file that contains keys 1, 5, and 6 in the example).

(b) `MinOverlappingRatio` (*MOR*) picks the file that has the minimum overlapping ratio. The overlapping ratio of a file in level i is defined as the fraction between the total bytes of files in level $i+1$ that overlap with this file and the size of that file. For example, Figure 2(b) shows that when L1 first fills up, the overlapping ratio for the first and the second file is, respectively, 2.0 and 1.0. For the first file, the key range is 1 to 4, which overlaps with two files at L2, and thus, the

ratio is 2.0 (each file has the same size in this example). Similarly, since the second file at L1 only overlaps with one file at L2, the overlapping ratio is 1.0. As a result, the second file is compacted because it locally minimizes the number of bytes written. *MOR* is also called *LO+1* (picking the least overlapping file with the next level) [34].

(c) `OldestLargestSeqFirst` (*OLSF*) picks the file of which the latest key-value pair is the oldest. As the largest sequence number (abbreviated as *L-seq*) of a file indicates the most recently inserted entry in this file, *OLSF* essentially picks the file that has minimum L-seq. For example, in Figure 2(c), L1 has two files of which the L-seq of the first and the second file is respectively 40 and 50. As the L-seq of the second file is smaller (i.e., 40 < 50), the second file is picked to compact under *OLSF* policy. *OLSF* is also termed as the *coldest* picking policy.

(d) `OldestSmallestSeqFirst` (*OSSF*) picks the file of which the smallest sequence number (abbreviated as S-seq) is the oldest among all other files in the same level. In the example of Figure 2(d), the S-seq of two files at L1 is 20 and 30, and since 20 < 30, the first file is picked to compact to the next level. *OSSF* is also called the *oldest* file-picking policy.

Tombstone-based File-Picking Policies. In RocksDB, there is another partial compaction policy, *CompensatedSize*, which favors larger files compensated by the number of deletes (i.e., tombstones) in this file. Other tombstone-based policies (e.g., Lethe [33]) also exist in academic LSM prototypes. In the presence of tombstones, compactions are not necessarily triggered by level capacity. With new privacy protection laws (e.g. GDPR [1] and CCPA [2]) being enacted, LSM-trees require a *Time-To-Live* (TTL) parameter to specify the longest time a tombstone (a deleted item) that can exist in the database. With this constraint, each tombstone is associated with a TTL, and once the tombstone has expired with respect to the TTL, several compactions are enforced to compact it into the last level (where the tombstones can be safely discarded). TTL-based compaction policies are not part of the production RocksDB, and since they would increase the complexity of the compaction process and introduce a trade-off among space amplification, WA, and privacy, we focus on the existing approaches.

Newly Implemented RoundRobin. Figure 2(a) demonstrates a classic *RR* implementation in LevelDB. Another version of *RR* [35] picks the file using the file rank instead of the key cursor. Specifically, the first file is picked in the first compaction in this level, the second file is picked in the second one, and so on so forth. However, these two implementations actually fail to compact the key space in a round-robin manner. In Figure 2 (a), even though the round-robin cursor is moved to 4 after the first compaction, the file being picked in the next compaction has the minimum key of 1, causing key 1 to be included in two consecutive compactions. As shown by prior work [26], ensuring each key is uniformly compacted can yield lower WA. To uniformly compact keys within a level, RocksDB re-implements *RR* via a splitting mechanism [18] which enforces that every compaction that outputs in level i has to split the file based on the existing cursor of level i. In the new implementation, after a compaction from level i finishes, the cursor in level i moves to the smallest key of the successive file of the picked file in the last compaction, and the next compaction from level i will pick the first file (from left to right) that is *larger than or equal to* the cursor. By skipping the cursor past the gap between the last picked file and its successor,

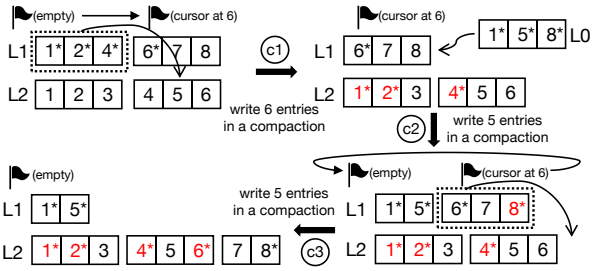

Figure 3: The latest version of RoundRobin in RocksDB.

new keys arriving into that gap in level i will not be compacted immediately. For example, as shown in Figure 3, the first L1 compaction moves the cursor to key 6, and the next L0 compaction splits the file at key 6 to ensure that in the next L1 compaction, the file starting with key 6 can be chosen and thus keys at L1 can be uniformly compacted in a round-robin manner.

Since the scope of a compaction is at the granularity of files, an existing file in the output level must be included if its key-range overlaps the picked file from the input level. A given compaction can include at most two output-level files that contain a mix of keys that overlap and do not overlap the picked file. Rewriting keys not overlapping with the selected file also contributes to additional WA. To mitigate this, the newest version of *RR* allows multiple consecutive files to be selected together, as long as the overall bytes involved in this compaction do not exceed a user-defined threshold. This input-level expansion reduces the total amount of data that is unnecessarily rewritten at the output level. Selecting multiple files in *RR* can thus reduce WA at the same level by slightly increasing the compaction latency (still much smaller than the full compaction since the number of bytes in a compaction is bounded). When two or more files are selected, the cursor still changes into the smallest key of the successive file of the last picked file.

Splitting Files at Boundaries in Grandparent Levels During Compactions. During compactions, RocksDB merges all the input key-value pairs and discards obsolete ones to generate new files. In RocksDB, newly generated files are not restricted to be exactly as large as the user-specified target file size (noted by fs). Instead, they can vary to align new files with the key boundaries in a *grandparent* level for smaller write amplification in future compactions to the grandparent level. This optimization does not apply to files generated at the deepest level as there is no grandparent level. Thus, the file size at the deepest level should be exactly the same as the target one. The splitting mechanism works as follows. Suppose we are populating a buffer that temporarily contains all the key-value pairs of the next file to be generated. There are three conditions when RocksDB stops appending new entries:

- **Case 1:** when the buffer size reaches the user-defined maximum compaction bytes or $2 \cdot fs$. In this case, newly generated files never cross any grandparent boundaries.
- **Case 2:** when the next key crosses a grandparent boundary and the buffer size reaches a threshold (less than the target file size). The threshold is given by $((fs + 99)/100) \cdot (50 + min(n \cdot 5, 40))$ where n is the number of accumulated crossed boundaries, as shown in Figure 4(a). This condition allows the newly generated files to overlap with fewer files at the grandparent level.

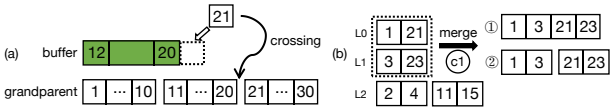

Figure 4: (a) An example of a new entry crossing the boundary in the grandparent level. When a new key 21 is being added to the buffer, and it is larger than the largest key of the second file in the grandparent level, the buffer will be split here. (b) An example of case 3. Without this condition, the merged file resembles case ①, overlapping with two L2 files, whereas applying it results in case ② with only one L2 file overlap, reducing future L1 compaction WA.

- **Case 3:** when the next key crosses more than two grandparent boundaries and the current buffer is larger than $fs/8$. Splitting the file before the next key avoids producing files that overlap a *skippable* file in the grandparent level, as shown in Figure 4(b).

3 BENCHMARKING FILE-PICKING POLICIES

In this section, we first benchmark existing picking policies in RocksDB under different ingestion workloads (see §3.1). Then, we quantify the optimal WA using a smaller workload to find the improvement headroom. By **[R]**, we refer to a *rectified* observation of prior work, while **[N]** marks a *new* observation.

Environment. All experiments are executed on a Rocky Linux server with 375GB main memory and two Intel Xeon Gold 6230 2.1GHz processors (each having 20 cores with virtualization enabled and 28160K of L3 cache). By default, we use a 350GB Optane P4510 P4800X SSD with direct I/O disabled as our disk storage.

RocksDB Configuration. By default, the write buffer size and the target file size are both set to be 64MB, and the size ratio T is set as 4 (which means that `max_bytes_for_level_base`, the capacity of L1, is 256MB). We use `vector` as the default buffer implementation. Although other implementations like `skiplist` or `hash-skiplist` allow in-place updates, which can facilitate future queries [25], `vector` only appends writes without building an index for reads or de-duplication. Our workloads are write-only, so we are stressing the ingesting and flushing part of the system. Note that different buffer implementations have little impact over WA since a buffer flush always eliminates duplicates. Further, we turn off compression and dynamic compaction to magnify the impact of different file-picking policies on WA.

Experimental Setup. For each workload (characterized by the number of ingestions, the key and value size, the proportion of updates, and the update distribution), we execute the experiment for ten *runs* as follows: in each run, we randomly generate the workload with the same characteristics and run the ingestion experiment to collect WA. Specifically, WA is calculated as the fraction between the overall written bytes from background RocksDB threads (for flushing and compactions), and the size of all key-value pairs passed through `Put()` calls. For recovery purposes, a key-value pair passed in the `Put()` function is appended into the Write-Ahead-Log (WAL) before returning an `Ok` status. We exclude the bytes written to WAL when calculating WA.

Using the WA data we collected over ten runs for the same type of workload, we report five statistic measures of WA in most experiments: min, max, average, the first quartile Q_1, and the third quartile Q_3. An illustrative example of displaying these five metrics using a box plot can be found in Figure 6(a). While we mainly use the average WA to compare different picking policies,

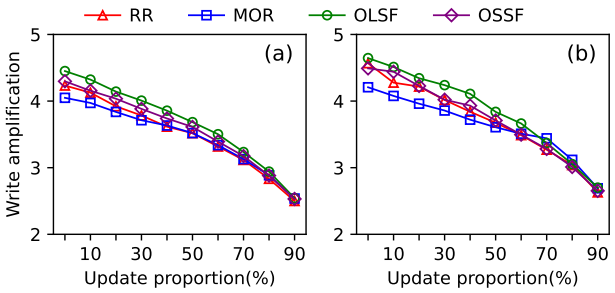

Figure 5: (a) the average WA and (b) the maximum WA for a specific update proportion in ten runs.

min and max can also reflect the best and the worst WA for one type of workload, and the difference between Q_3 and Q_1 can be used to quantify the quartile deviation, i.e., $(Q_3 - Q_1)/2$.

One could question whether measuring WA as discussed with ten different instances of a workload could lead to different minimum WA. Our experiments show that the optimal WA remains stable even if we run the experiments using ten randomly generated workloads as long as the workload properties are fixed.

In addition, in our experiments, we usually have an LSM-tree with three or more levels. All the data in L1 or deeper, which is also the majority of unique key-value pairs, must have been written at least twice: to L0 and L1. As such, for illustration purposes, we set `ymin` as 2 in many figures. For different workloads that lead to different WA, we move the min and max plotted WA accordingly and resize the figure to maintain the same scale.

3.1 Benchmarking Existing Policies

Varying the Proportion of Updates. We keep the workload constant at $5M$ operations, each associated with a 1KB entry, and then adjust the percentage of update operations from 0% to 90%. Updates are generated following a uniform distribution.

> **Observation 1 [R]: *OSSF* does NOT have the lowest WA (neither the average nor the maximum) for random updates. *This is revising the statement "If your updates are random across the key space, write amplification is slightly better with [OSSF]".***

While all four curves are very close to each other in Figure 5, we can still see that the purple curves with diamond markers (*OSSF*) lie above either the blue or the red curve (*MOR* or *RR*, respectively) in terms of both the average and maximum WA. Specifically, for workloads with 50% updates, the average WA of *OSSF* is larger than *RR* and *MOR* by 2.67% and 2.63%, and the maximum WA (i.e., the worst-case WA) becomes 0.90% and 2.81% larger. Moreover, subsequent experiments that modify the update distribution and workload scale consistently show the same pattern. As shown in Figures 6 and 7, *OSSF* (represented by the purple box) consistently fails to achieve the lowest average (indicated by the crossing mark in a box) or the lowest maximum (upper boundary of the box).

> **Observation 2 [N]: File-picking policies have minor impact on WA for update-intensive workloads.**

Figure 5 also shows that when the update fraction is large (especially when the update proportion is larger than 60%), all four

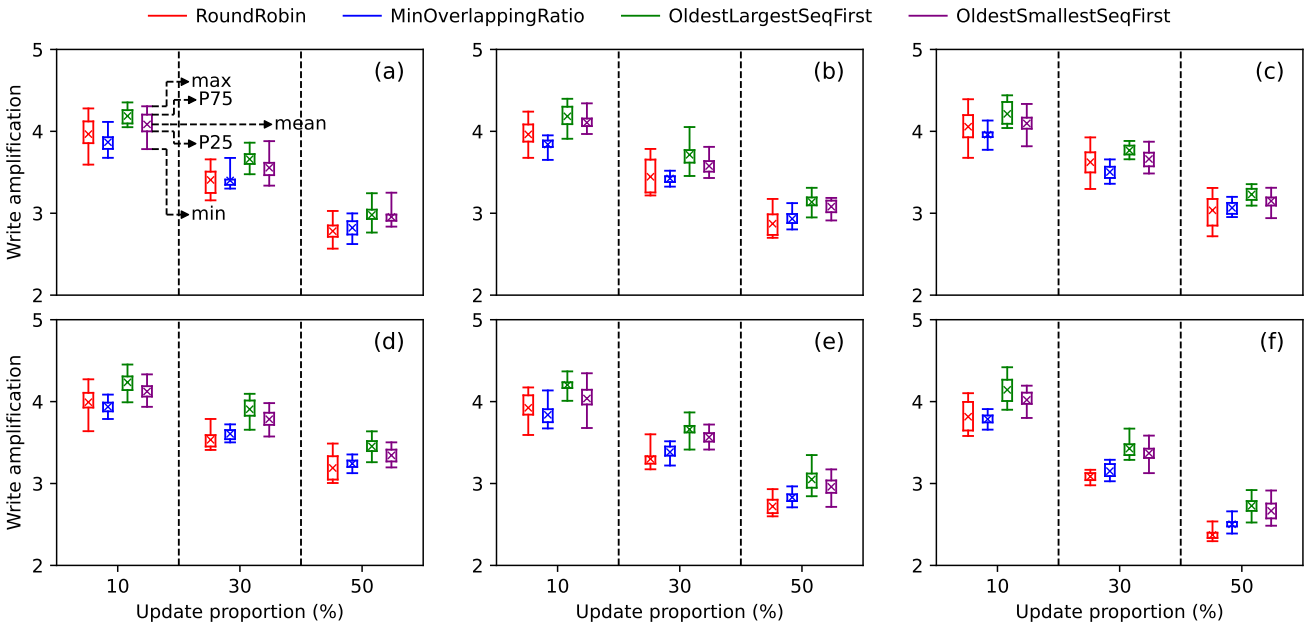

Figure 6: WA of four file-picking policies for workloads with different update distributions. (a) normal distribution with $\mu = 0.5, \sigma = 0.5$, (b) normal distribution with $\mu = 0.5, \sigma = 1$, (c) normal distribution with $\mu = 0.5, \sigma = 2$, (d) Zipfian distribution with $\alpha = 0.8$, (e) Zipfian distribution with $\alpha = 1.0$, (f) Zipfian distribution with $\alpha = 1.2$.

file-picking policies have lower and similar WA. While we only plot the mean and maximum, the other three statistic measures of WA (e.g., min, Q_1, and Q_3) also have very similar values. This is because, with increasing updates, entries with duplicate keys can be updated in buffer flushes and L0 compactions. Specifically, an L0 compaction for update-intensive workloads writes fewer entries than the ones it received by discarding duplicates, while L0 compactions in insert-intensive workloads write more data to disk because there are fewer duplicates to eliminate. Furthermore, update-intensive workloads generate fewer L1 compactions, leading to a lower WA. As more entries can be updated during buffer flushes and L0 compactions, more entries are absorbed in L1 before L1 capacity is reached. Taking *MOR* as an example, the average number of L1 compactions for workloads with 90% and 10% updates is 25 and 35, respectively. In other words, the number of L1 compactions increases by 40% with fewer updates, thereby explaining why the average WA of 10%-update workloads is 40% higher than that of 90%-update workloads.

Varying the Update Distribution. For a mixed workload with inserts and updates, inserts are unique by default, and thus, only updates can involve duplicate keys. Note that earlier work [34] examines the impact of a *PrefixZipf* distribution for unique inserts, where the prefix of inserted keys follows a Zipfian distribution and all inserted keys remain unique. We observed similar results to previous work (i.e., the file-picking policy has minimal impact on PrefixZipf inserts), so we omit them here and focus instead on the update distribution. Since file-picking policies differ little for update-intensive workloads, we select workloads with light and medium update fractions (10%, 30%, 50%) to verify the impact of a skewed update distribution on WA. Specifically, we examine two distributions: the normal distribution, where larger σ indicates lower skew, and the Zipfian distribution, where larger α indicates higher skew, as shown in Figure 6. Before we delve into the differences across the four picking policies, we highlight a general pattern: *the average WA decreases with a higher skew.* Specifically, a smaller σ from Figure 6(c) to Figure 6(a) and a larger α from Figure 6(d) to Figure 6(f) both reduce the average WA. The rationale behind this pattern is similar to increasing the update proportion – more ingestions can be absorbed in flushes and L0 compactions before triggering L1 compactions. In the case of *MOR*, for workloads comprising 50% updates with $\alpha = 0.8$ in a Zipfian distribution, 32 L1 compactions are triggered. However, when $\alpha = 1.2$, there are only 22 L1 compactions on average, around two-thirds when compared with the number of compactions for $\alpha = 0.8$.

> **Observation 3 [R]: *OLSF* does NOT have significantly lower WA than *RR* and *MOR* for skewed updates. *This is revising the statement "Try [OLSF] if you only update some hot keys in small ranges".***

While RocksDB suggests that *OLSF should be preferred if some keys in small ranges are frequently updated*, we do not observe *OLSF* dominating *RR* and *MOR* in Figure 6(a), Figure 6(b), or Figure 6(c). In these figures, updates follow a normal distribution, where keys in the center of the key domain ($\mu = 0.5$) are frequently updated, using our recently developed key-value workload generator [47]. In Figure 6(a), the most skewed normal distribution that we tested, the reduction in percentage in terms of the average WA of *RR* and *MOR* over *OLSF* is 5.3% and 7.6% for 10%-update workloads, 6.9% and 7.1% for 30%-update workloads, 6.7% and 5.3% for 50%-update workloads. Further, for the Zipfian distribution where frequently updated keys are not *concentrated in small ranges* but randomly dispersed in the key space, when the update distribution becomes more skewed (i.e., α increases from 0.8 to 1.2) from Figure 6(d) to Figure 6(f), the WA reduction of *RR* and *MOR* over *OLSF* is up to 13% and 8.7% among tested workloads.

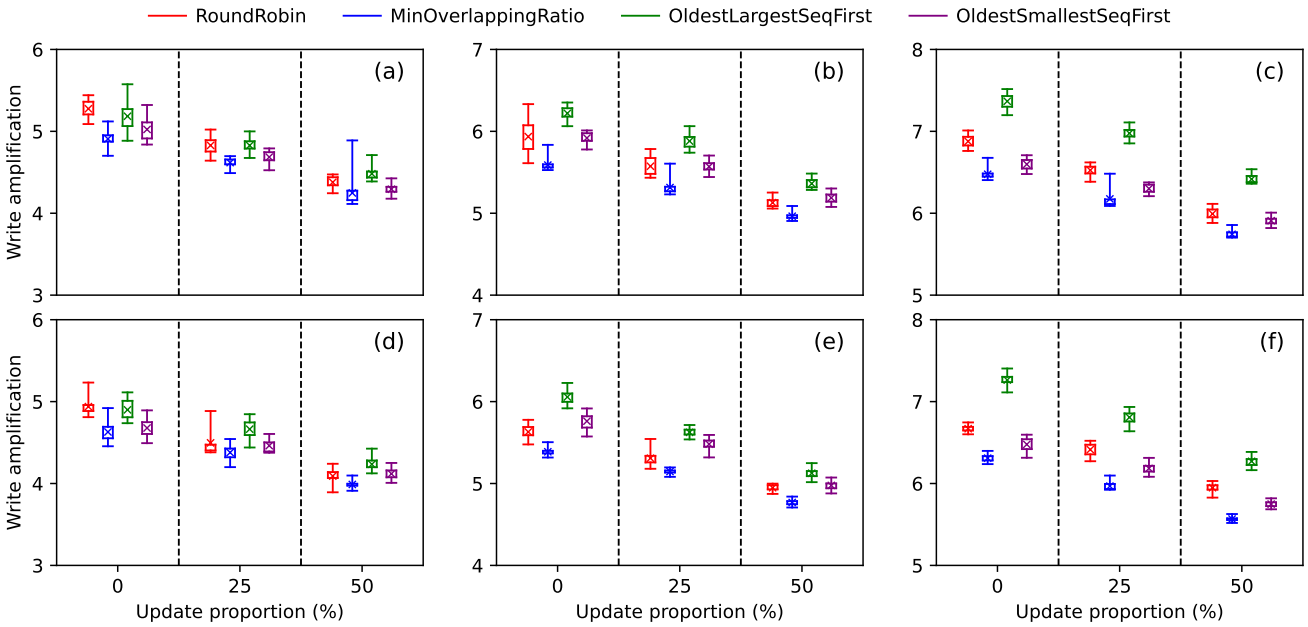

Figure 7: WA of four file-picking policies at different scales. (a) 20M ingestion of 512-byte entry size, (b) 40M ingestion of 512-byte entry size, (c) 80M ingestion with 512-byte entry size, (d) 10M operations with 1KB entry size, (e) 20M ingestion with 1KB entry size, (f) 40M ingestion with 1KB entry size. The y-min and y-max change across different columns.

> **Observation 4 [R]: *RR* leads to a lower average WA than *MOR* when updates exhibit a higher skew. *This is revising the statement "Try [RR] if you only update some hot keys in small ranges".***

If we only compare *RR* and *MOR* when varying the update distribution, we see an interesting trend: the average WA of *RR* decreases faster than *MOR* when the update distribution becomes more skewed, as shown in Figure 6. For instance, in the normal distribution, when σ changes from 2 to 0.5, the average WA of *RR* decreases by 5.99% and 8.39% for 30%-update and 50%-update workloads, while the average WA of *MOR* only decreases by 2.93% and 7.95%, as we can see by comparing Figure 6(a) with Figure 6(c). A similar pattern can be captured when we compare Figure 6(d) and Figure 6(f) for the Zipfian distribution. The average WA of *RR* is smaller than the one of *MOR* when the update distribution exhibits a higher skew. We credit this benefit of *RR* to the fact that frequently updated keys stay longer in L1. In the presence of update skew, frequently updated keys are inevitably re-written multiple times in L1 because every flushed buffer contains the duplicate key. Every L0 compaction discards obsolete entries and writes new entries with the same key. More specifically, each L0 compaction generates a file containing the latest version of the frequently updated key. Ideally, this key stays in L1 for as long as possible to be overwritten by future L0 compactions. If this key is compacted to L2, it is written – along with its value – at least one more time before it can be replaced by newer versions. Since *RR* waits for the cursor to iterate over the whole key space, *MOR* has a higher probability of compacting the most frequently updated key to deeper levels than *RR*, which leads to a higher WA when updates have higher skew.

Scalability. In this experiment, we scale up the workloads from 10GB to 40GB by increasing the number of operations and the entry size. We use uniform updates and measure the WA under three update proportions (0%, 25%, and 50%). The experimental results are summarized in Figure 7, where in the upper row, we vary the number of operations from 20M to 80M with 512-byte entries, and in the bottom row, the number of operations from 10M to 40M with 1KB entries. As mentioned earlier, the majority of unique key-value pairs are written more times as the LSM-tree becomes taller. Thus, regardless of the policy, WA naturally increases for larger workloads since they lead to taller LSM-trees, as shown in Figure 7. In addition, when comparing Figure 7(a) with Figure 7(d), or comparing Figure 7(b) with Figure 7(e), we observe that larger entry size typically has lower write amplification if they have the same workload size. Since the above two observations on workload scale and entry size are mentioned in earlier study [34], we do not list them as new observations here.

> **Observation 5 [N]: *MOR* scales better than other policies by exhibiting lower average WA and higher stability for uniform update distribution.**

Among all tested workloads in Figure 7, *MOR* always has the lowest average WA among the existing file-picking policies. For example, in 40GB workloads with 1KB entry size and 25% updates, the average WA of *MOR* is 5.97, while the average WA of *OSSF*, *RR*, and *OLSF* is 6.18, 6.41, and 6.81, respectively. While we observe that the average WA of four policies follows a specific order (*OLSF* > *RR* > *OSSF* > *MOR*) for all 40GB workloads, this order does not necessarily hold for 10GB and 20GB workloads (except that *MOR* always has the lowest WA). For example, in 10GB workloads with 100% inserts, for both 512-byte and 1KB entry size, we get *RR* > *OLSF* > *OSSF* > *MOR*. We also see that *MOR* has higher stability with smaller quartile deviation and smaller min-max distance. For example, in Figure 7(f), the quartile deviation of *MOR* is 0.052, 0.062, and 0.020, which are the smallest deviations among four policies under three workloads.

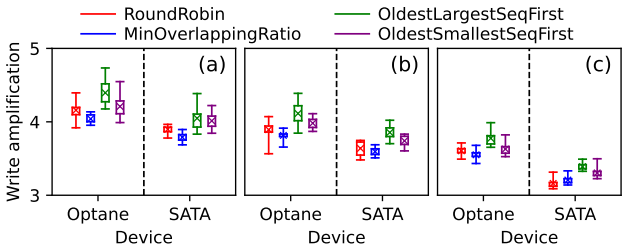

Figure 8: WA of 5GB workloads with different update proportions on two storage devices (i.e., an Optane SSD and a SATA SSD). (a) 90% insert and 10% update, (b) 70% insert and 30% update, (c) 50% insert and 50% update.

As *RR*, *OLSF*, and *OSSf* all have high instability in most tested workloads, we refrain from making definitive observations about their order with respect to WA.

Varying Storage Devices. We also examine the impact of storage devices on WA when we vary the update proportion. We use two different storage devices, an Optane P4800X SSD and a SATA S4610 SSD (the Optane SSD is 17× faster than the SATA SSD).

> **Observation 6 [N]: Slower SSDs have lower WA due to compaction stall in L0, which leads to larger space amplification and higher query latency.**

Figure 8 unveils that the WA when using the SATA SSD is much smaller than that on the Optane SSD, by up to 16.59%. We attribute this WA difference to the compaction stall in slower SSDs. Specifically, since the L0 compaction mostly overlaps with the ongoing L1 compaction, the L0 compaction has to wait until the ongoing one is completed. Since RocksDB keeps digesting upcoming ingestion, L0 files accumulate. This is similar to the experiment with insert-intensive workloads where the L1 compactions may take a long time to complete. For example, when running the 30%-update workload, there are on average 10.5 files in L0 compactions for the slower SSD, and 8.5 files for the Optane SSD. While accumulating more L0 files can reduce WA, this does not directly reflect the execution time because we are using different storage devices. Figure 9 shows the execution time of each workload using two devices, where SATA SSD substantially increases the latency compared to Optane SSD. Additionally, more files in L0 also lead to more duplicates and higher space consumption. Further, since files in L0 may overlap, point queries have to probe every file until the desired key is found. While we aim to benchmark and reduce WA in this paper, downgrading the SSD device does not pay off due to excessive space and read costs.

Varying Target File Size and Size Ratio. Now we vary the target file size (noted by fs) from 16MB to 128MB with two size ratios $T = 4, 10$. Note that fs only affects the output file size in compaction, which means that the file size in L0 remains the same as the buffer size, 64MB. We run 5GB workloads with 0% and 50% updates to examine the impact of fs and T on WA. In addition, as the number of levels, L, also affects WA, we ensure that L is the same for different T. Specifically, when ingesting 5GB with either 0% or 50% updates, our LSM trees have 4 levels.

> **Observation 7 [N]: The target file size does not have significant impact over the average WA.**

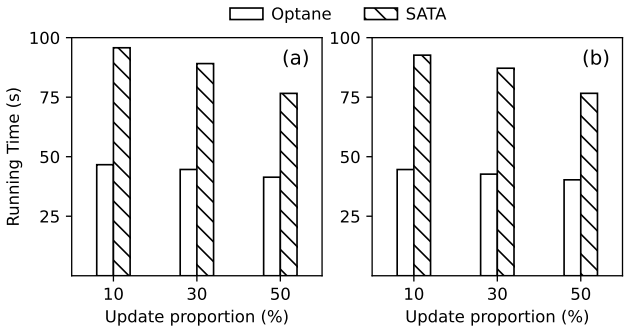

Figure 9: Running time of two policies on different workloads with an Optane SSD and a SATA SSD. (a) RR, (b) MOR.

When we compare the average WA in Figure 10, we do not observe a significant change when fs varies between 16MB and 128MB for all the file-picking policies. For example, when we set $T = 10$, the difference in the average WA for *MOR* between fs = 16MB and fs = 128MB is no more than 3% for workloads with 0% and 50% updates. However, in the extreme case that the fs is unlimited, every level forms a file, which downgrades into full-level compaction. Taking Figure 10(a) as an example. With fs = 5GB, partial compaction behaves very similar to full compaction, although they are exactly the same due to the splitting mechanism in Figure 4. In this case, the average WA of *RR* and *MOR* is, respectively, 11.72%, 12.85% larger than the ones with $fs = 128MB$. As for the size ratio T, we see that the average WA also increases for all policies with larger T. For example, fixing fs as 32MB, the average WA of *MOR* with $T = 4$ is 7.41%, 4.99% smaller than the ones with $T = 10$ for 100% inserts and 50% inserts. This observation is consistent with the existing write cost model $O(T \cdot L/2)$ and an earlier experiment study [34].

3.2 Exploring Optimal WA

This section explores the optimization space of existing file-picking policies by comparing them with the optimal WA, which is obtained by enumerating all possible file-picking decisions in all compactions with some pruning to reduce the search space. We highlight that the enumeration space is exponential to the number of compactions. For example, for a 5GB workload, the optimal WA cannot be fully enumerated within 4 hours. Thus, we restrict the workload scale and re-configure some knobs to adapt to a smaller workload.

Setup for Small Workloads. The enumeration method repeatedly runs the same workload to find the optimal compaction strategy. To accelerate the process, we choose a relatively small workload: $2M$ operations with 50% insert and 50% update. Each operation is associated with a pair of an 8-byte key and a 56-byte value (i.e., 64-byte entry). For the LSM-tree setting, we set the write buffer, the target file size to 8MB, and the size ratio between levels to 4. In this setup, the workload typically triggers 5 L1 compactions. Just like earlier experiments, we randomly generate ten workloads with the same workload characteristics. We then run our algorithm to search for the minimum WA for each of them to examine its stability.

Searching the Optimal WA. The naïve approach to finding the optimal WA is to enumerate all possible cases by replaying the workload an exponential number of times. We propose a multi-step searching framework to reduce the search space. Specifically, at each iteration, we note the minimum bytes we found so far

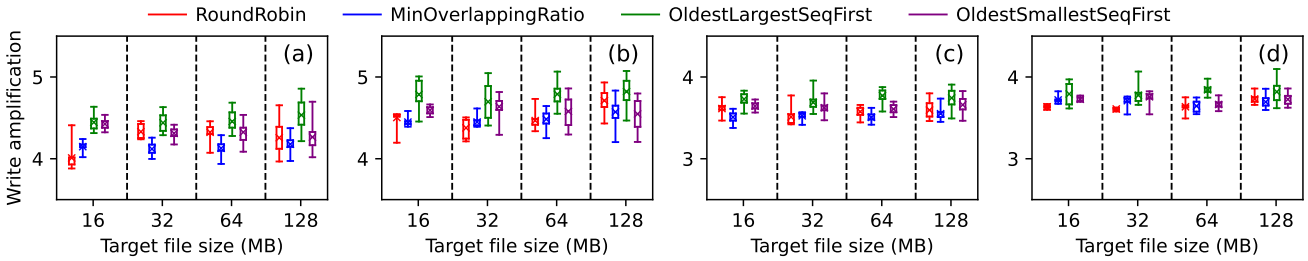

Figure 10: WA of different target file sizes and different size ratio on two 5GB workloads. (a) 100% insert with $T = 4$, (b) 100% insert with $T = 10$, (c) 50% insert, 50% update with $T = 4$, (d) 50% insert, 50% update with $T = 10$.

Table 1: Each row represents the optimal WA searched in 20 workloads of the same update proportion.

Update proportion	Min	Q_1	Mean	Q_3	Max
0%	3.582	3.583	3.583	3.584	3.584
20%	3.335	3.336	3.336	3.336	3.337
40%	3.026	3.027	3.027	3.028	3.029

as $bytes_{min}$, which can be initialized as the minimum number of written bytes among all four existing picking policies. During each trial in the enumeration, we add up the number of written bytes after we start executing this workload in this trial and the bytes left in the workload to ingest, and if the result is greater than $bytes_{min}$, we stop this trial because future compactions will necessarily lead to a larger WA. Note that this multi-step searching only works for `vector` implementation in the write buffer since there are no in-place updates in the buffer. When we find fewer written bytes to execute the workload, we update $bytes_{min}$ and continue searching until all possibilities are enumerated. Further, we see that RocksDB triggers a *trivial move* when a file does not overlap with any file in the next level, which also avoids re-writing data. The original searching framework, noted by *skip* strategy, does not pick any file when a trivial move is triggered. We also design an alternative *non-skip* searching strategy that disables trivial moves and tries to enumerate all possible files to compact or move when a level is full.

Observation 8 [N]: Trivial move should always be prioritized over compactions since *non-skip* and *skip* searching find virtually the same WA.

Although the *non-skip* strategy has larger search space and potentially leads to lower WA, we observe little difference between the *skip* and the *non-skip* strategy. Specifically, the largest difference among our tested workloads for the same measure is just 0.07%. As the two strategies find virtually the same WA, we omit the detailed comparison between them and only show the WA of *skip* in Table 1. This is consistent with the current strategy for trivial moves in RocksDB: for any file picking policy, trivial moves are always prioritized over compactions. In the following experiments for optimal WA, we use the *skip* strategy by default.

Comparing the Optimal WA with Existing Policies. We now use a slightly larger workload – $2.5M$ operations – to examine the headroom of existing policies by comparing them with the optimal WA obtained by the skip strategy. To quantify the improvement headroom for existing policies, we now compare them

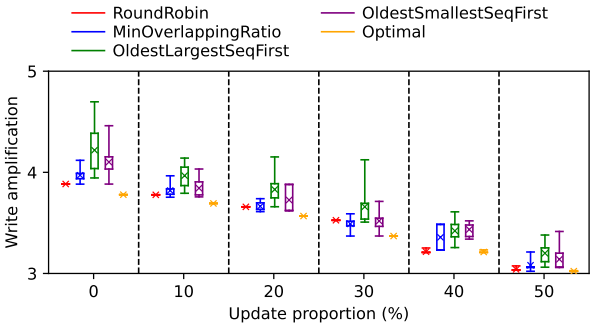

Figure 11: WA of four file-picking policies and the optimal one for workloads with different update proportions.

with the optimal WA by varying the update fraction from 0% to 50% in 6 different workloads, as shown in Figure 11.

Observation 9 [N]: An optimal policy with minimal WA offers higher stability than existing policies.

Figure 11 shows that the optimal WA is also the most stable compared to all other existing policies in all tested workloads. Specifically, in the worst case, the max WA is only 0.06% larger than the min WA for the optimal WA. The consistency of the optimal WA confirms the reliability of our experimental setup, that is, even if we use 20 different workloads, the WA collected from these 20 workloads is still comparable as long as these 20 workloads are randomly generated using the same workload characteristics. Now, we focus on the headroom for WA improvement of existing policies. In fact, we can see that the min WA of *RR*, *MOR* is already very close to the optimal one. For example, with 40% updates, the optimal WA is only 0.09%, 4.31% smaller than the min WA of *RR* and *MOR*, which reflects the improvement headroom for *RR* and *MOR*. In terms of *minimum WA*, there is not a significant headroom between existing policies and the optimal one. However, the instability of *RR* and *MOR* moves the *average WA* farther than the optimal one. For example, for 0%-update workloads, the max WA is up to 6.06% larger than the min WA for *MOR* in Figure 11. Although *RR* remains stable in small workloads, *RR* becomes more unstable for larger workloads, as we see in Figure 7. The above observations suggest that future research on file-picking policies should focus on investigating and alleviating the instability of *RR* and *MOR*.

4 A REFINED MinOverlappingRatio

To investigate the instability of *MOR*, we study *MOR* with a smaller workload and compare it with the optimal strategy. Based

on the observed difference, we further refine *MOR* with minor changes and examine its performance.

***MOR* has unstable WA.** We scale down the workload so that we can better trace the compaction history and analyze the source of instability of *MOR*. We run two 100%-insert workloads: one has $8M$ 8-byte entries and the other has $2M$ 64-byte entries. As shown in Figure 12, WA can be quite different when we run the same workload ten times. On the other hand, we observe that for small workloads, there is only a limited number of possible WAs in 50 runs. For example, there are only four possible WAs in Figure 12(a) and five possible WAs in Figure 12(b). The minimum WA in *MOR* among all possible WAs is nearly the same as the optimal one. However, as *MOR* strictly selects the minimum overlapping ratio and sometimes the overlapping ratio of the best file to compact is only slightly larger than the minimum one (i.e., $< 5\%$), *MOR* can lead to unstable WA for the same workload.

Explanation with Traces. By comparing the compaction history between *MOR* and the optimal one, we are able to explain the instability issue of *MOR* using a real example traced in the log after we run the 100%-inserts workload in Figure 12, as shown in Figure 13. When we are executing 100%-insert workload, *MOR* and the optimal one behave very similarly until the LSM-tree state changes into Figure 13(a), where there are four files in L1 that have the same overlapping ratio. To be exact, the overlapping ratios of these files actually differ by less than 1%, which we treat as the same. Since we randomly generate ten workloads using the same workload characteristics of 100% unique inserts and *MOR* strictly selects the file that has the minimum overlapping ratio, these four files in L1 have a very similar probability of being selected. Although the decision of which file to compact in L1 does not make a large difference in terms of the WA for the current compaction, this decision impacts the long-term WA, which explains why *MOR* may eventually lead to substantially different WA. In fact, according to the traced log of the optimal policy, the last file is always picked to compact in the example of Figure 13. Let us examine the traces:

Observation 10 [N]: Picking different files that have similar WA (overlapping ratio) in the current compaction can lead to substantially different final WA.

- When the first file in L1 is picked, the structure after this compaction is shown in Figure 13(b). When an L0 compaction is triggered, the number of entries within key range 1~10 from L0 is not sufficient to compose a 64MB file, more entries in key range 10~20 are used. Based on the splitting condition mentioned in Figure 4(a), the first new file in L1 will have the key range of 1~20, and the next two new files have the same key range as before. Note that these two new files are larger than 64MB due to the aforementioned file splitting mechanism, which allows files to exceed the target file size (fs) as long as their key ranges do not cross the grandparent boundaries. In this scenario, even though the aggregated number of bytes within ranges 21~30 and 31~40 both exceed 64MB, these two ranges only produce two separate files. The remaining keys between 41~55 form a small file, as shown in Figure 13(c).
- When the last file in L1 is picked, the structure after this compaction is shown in Figure 13(d). Following similar analysis above, we can obtain Figure 13(e): the number of entries for key range 1~10, 11~20, 21~30 is large enough so three files with the same key range are generated that are larger than 64MB, and the rest of the keys form two files of key ranges 30~50, and 51~55.

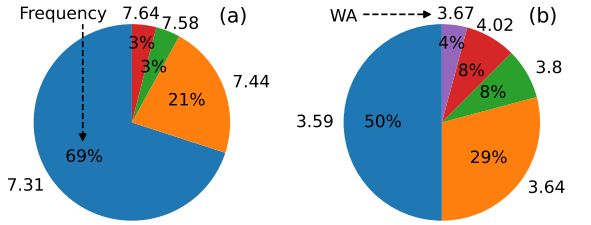

Figure 12: WA of *MOR* in 50 runs for two workloads: (a) $8M$ inserts of 8-byte entries, (b) $2M$ inserts of 64-byte entries. The number outside each sector represents the measured WA, and the number inside each sector represents the relative frequency of the associated WA in 50 runs.

Although the WA in LSM-tree states (c) and (e) are the same, files with a smaller overlapping ratio that lead to smaller long-term WA in (c) are fewer than (e). Similar analysis can be applied for state (c) and (e). For example, in state (c), if we pick the file with key range 20~30, we do not have files with a small overlapping ratio for the next L1 compaction because L0 compactions have already merged the file with key range 31~40. In contrast, if we compact the key range 31~40, we can still compact the key range 21~30 after the compaction L0, which has an even smaller overlapping ratio.

Refining *MOR*. We observe that compacting a file leaves a *hole* in the key space, which may force future compactions that output to this level to use more entries in the file following the one previously compacted. As such, the optimal policy tries to keep the files with a small overlapping ratio as much as possible when picking a file to compact. In the example of Figure 13(a), the optimal order to compact files (identified by key range) in L1 is 31~40, 21~30, 11~20, and 1~10. This suggests that, if there are several files that have similar overlapping ratios to the minimum one, we should treat all of them as candidates and pick the last file in this level (if it exists in the candidates) or a file of which the successive one has the largest overlapping ratio. This prevents future compactions from the upper level eliminating existing small overlapping ratios at this level. As such, we adjust *MOR* using a threshold th ($0 < th < 1$) as follows. We select all the files of which the overlapping ratio is no larger than the minimum one by th, and we re-rank them based on a newly defined ratio. Specifically, with $files[j].r$ representing the overlapping ratio of the j^{th} file in this level, we use $th \cdot files[j].r - files[j+1].r$ to replace $files[j].r$, as shown in Algorithm 1. The adjusted algorithm is termed *RefinedMOR*.

Algorithm 1: RefinedMOR($files, th$)

```
1 f ← MOR(files)
2 for j ← 0 to files.size() − 1 do
3     if files[j].r < f.r · (1 + th) then
4         if j == files.size() − 1 then
5             return files[j]
6         files[j].r ← th · files[j].r − files[j + 1].r
7 return MOR(files)
```

While RocksDB does not provide an interface for custom selection strategies, we implement *RefinedMOR* by modifying *MOR* [42].

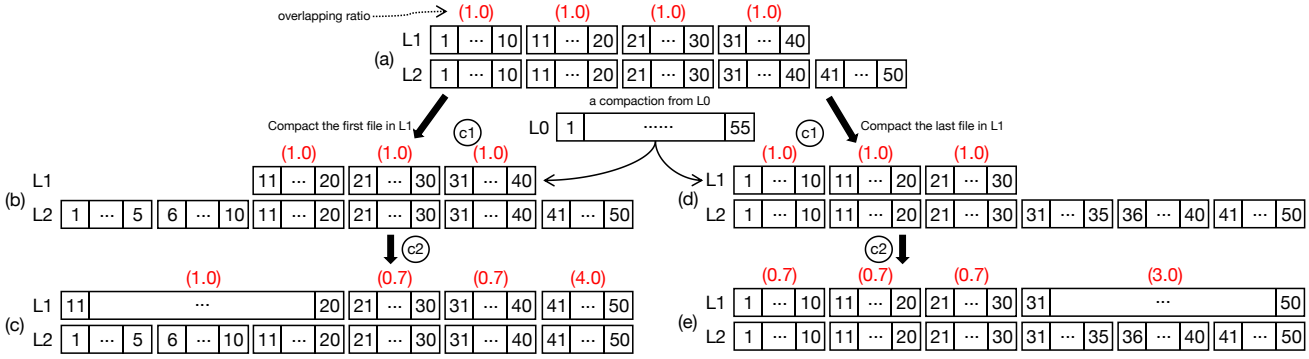

Figure 13: A traced example in RocksDB that explains the instability of *MOR*. We start with state (a). In this state, files at L2 have similar key ranges to L1 because files at L2 are initially *trivial moved* from L1 without triggering a compaction. When L1 becomes full again, we need to pick a file to compact, given that four files at L1 have nearly the same overlapping ratio. We now emulate two cases – picking the first or the last file – which respectively results into state (b) and (d).

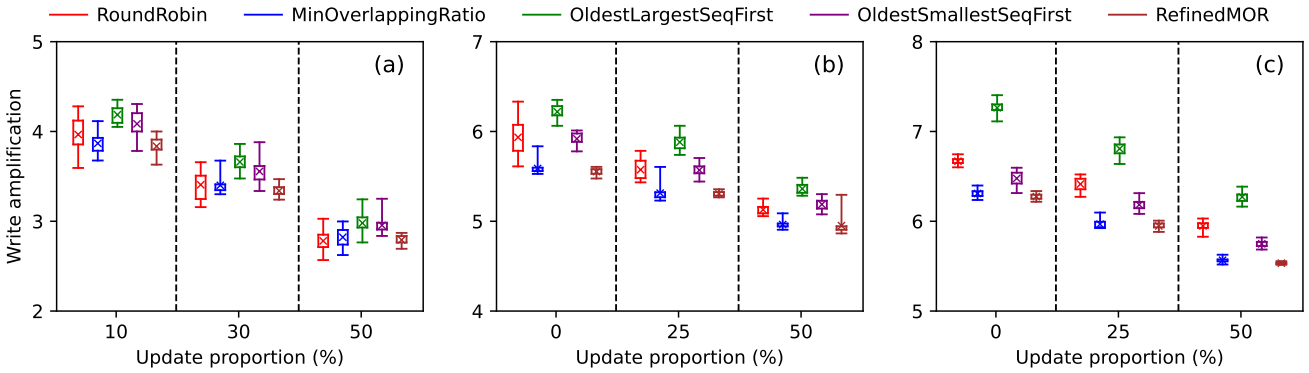

Figure 14: WA of four file-picking policies and *RefinedMOR* on different workloads. (a) 5GB data of $5M$ 1KB entries, and normal update distribution with $\sigma = 0.5$, (b) 20GB workload of $40M$ 512-byte entries, (c) 40GB data of $40M$ 1KB entries.

***RefinedMOR* outperforms *MOR*.** We now evaluate the WA of *RefinedMOR* with $th = 0.05$. First, we re-use the workloads presented in Figure 12 to conduct a comparative analysis with *MOR*. For *RefinedMOR*, the measured WA is **always 7.31** and **3.59** for $2M$ 64-byte inserts and $8M$ 8-byte inserts in 50 runs. Compared to *MOR* in Figure 12, *RefinedMOR* not only achieves the minimum WA of *MOR* but also demonstrates significantly higher stability. We further expand our comparison of *RefinedMOR* against other file-picking policies using larger workloads, as illustrated in Figure 14. In Figure 14(a), we test a 5GB workload where updates follow a normal distribution. Despite that *RR* still outperforms *MOR* for skew updates, as previously mentioned, *RefinedMOR* exhibits both lower and more stable than *MOR*. In particular, *RefinedMOR* decreases the average WA of *MOR* by 0.84%, 1.83%, and 1.07% for workloads with 10%, 30%, and 50% updates, respectively. This superiority of *RefinedMOR* over *MOR* is also evident in larger workloads. In the experiment with 20GB and 40GB workloads with uniform updates, *RefinedMOR* exhibits the smallest average WA among all five policies. For instance, in the 40GB workload, comparing the average WA of *RefinedMOR* with that of *MOR*, it is 0.55%, 0.33%, and 0.60% lower for 0%, 25%, and 50% updates, respectively. It's worth noting that, before *RefinedMOR* is included for comparison, *MOR* already has the smallest WA among the four existing file-picking policies in these workloads. Furthermore, we observe that *RefinedMOR* demonstrates greater stability, as indicated by its smaller quartile deviation. The difference between Q_1 and Q_3 of *RefinedMOR* is consistently the smallest among all the existing picking policies for every examined workload. Specifically, for 40GB workloads with 50% updates, the quartile deviation for *RR*, *MOR*, *OLSF*, *OSSF*, and *RefinedMOR* is 0.052, 0.020, 0.071, 0.046, and 0.012, respectively. Synthesizing the experimental findings concerning *RefinedMOR*, it becomes evident that even a slight refinement of *MOR* using our observations can simultaneously reduce WA and improve stability. As *RefinedMOR* is merely a rudimentary version that uses our observation, we encourage further research on a more refined *MOR*.

Instability of Other Policies. In addition to *MOR*, we also observe that *RR*, *OLSF*, and *OSSF* have a large deviation, attributed to the new splitting mechanisms presented in Figure 3 and Figure 4. Specifically, *RR* may generate a very small file due to a RR cursor (Fig. 3). For example, when testing a 40GB workload with fs =64MB, files with size as small as 98KB are generated in L1 due to the cursor in L2, which increases the number of required L1 compactions to cover the whole key domain. For both *OLSF* and *OSSF*, we see that the smallest sequence number can be dispersed in any newly generated file during compaction, and thus, when workloads with the same characteristics are generated, different files can be picked from a similar LSM-tree state. Note that the current level that triggers a compaction can also be a grandparent level for compactions from shallower levels, which interplays with the grandparent splitting mechanism in Figure 4 In such cases, the grandparent splitting mechanism may

significantly affect how files are partitioned as long as different files are picked, which further magnifies the difference on WA. Thus, *OLSF* and *OSSF* that rely on sequence numbers also suffer from instability. Specifically, for a 10GB workload with 100% inserts, when the grandparent splitting mechanism is off, the quartile deviation of *OLSF* (*OSSF*) decreases by 24% (6.6%) while WA increases by 11.3% (12.5%). Overall, although the new split mechanisms reduce WA for all file-picking policies, this comes with the cost of higher WA deviation.

5 RELATED WORK

Compaction Analysis and Benchmark. An earlier LSM-based survey [28] has pointed out that the partial compaction scheme allows a smart file-picking policy, which could potentially have a smaller WA by avoiding re-writing cold keys for skewed updates. However, no quantitative study shows the WA reduction percentage for different picking policies in that survey. Another experimental study [34] compares several existing partial compaction policies and other full-level compaction strategies, but its observations do not reveal the difference across several file-picking policies in the partial compaction scheme. Besides, the version of RocksDB used in the existing experimental study has not implemented the current *RR* and a complete splitting mechanism, which may render existing experimental results obsolete. While another mathematical WA estimation mechanism [26] claims that *RR* has marginally lower WA than *MOR*, we still need an up-to-date and thorough evaluation to compare different file-picking policies. This paper fills this research gap with extensive experiments and further explores the potential headroom for improving existing file-picking policies.

Compaction Granularity. In addition to partial compaction, there are also other compaction granularities (e.g., level, sorted run, block, or even a mixed version). For example, a Block Compaction [40] scheme is proposed to delay each file-based partial compaction and trigger a compaction across several data blocks, reducing WA and alleviating cache invalidation. Similarly, LDC (Lower-level Driven Compaction) [8] also allows data slices that are bounded by selected key ranges to be involved in a compaction. To accelerate each partial compaction, NVLSM [45] exploits the byte-addressability of NVM for finer slicing granularity. Besides, multiple compaction granularities [14, 38] can co-exist to leverage the trade-off between space amplification and WA in LSM-trees. This study still applies as long as there is partial compaction, and thus, file-picking policies.

Compaction Scheduling. Multiple compactions can be pipelined and parallelized to exploit the SSD concurrency [46]. A more advanced compaction scheduler, like Compaction-as-a-service [44], can manage all the compactions across several LSM-tree instances. When all the compactions are scheduled properly and executed instantly, the file-picking policy is orthogonal to the compaction scheduler. However, delayed compactions [32] could also lead to a trade-off between space amplification and write amplification. Similar observations have been captured earlier in this paper (i.e., files are accumulated in L0 on a slower SSD).

6 CONCLUSIONS

In this paper, we present an experimental study on the impact of partial compaction policies on Write Amplification (WA) in RocksDB. We present 10 observations, which enrich and rectify conclusions and guidelines in existing studies. Further, we find the optimal WA by multi-step searching and compare the optimal file-picking decisions with *MOR*. We present an in-depth analysis to explain why *MOR* can be unstable, and we show that *Refined-MOR*, a simple refinement of *MOR* based on our observations, can reduce both the average WA and the quartile deviation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work is funded by the National Science Foundation under Grant No. IIS-2144547, a Facebook Faculty Research Award, and a Meta Gift.

REFERENCES

[1] 2016. Regulation (EU) 2016/679 of the European Parliament and of the council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC. *Official Journal of the European Union (Legislative Acts)* (2016), L119/1 – L119/88.
[2] 2018. California Consumer Privacy Act. *Assembly Bill No. 375, Chapter 55* (2018).
[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916. https://doi.org/10.14778/2733085.2733096
[4] Apache. 2023. Accumulo. *https://accumulo.apache.org/* (2023).
[5] Apache. 2023. Cassandra. *http://cassandra.apache.org* (2023).
[6] Apache. 2023. HBase. *http://hbase.apache.org/* (2023).
[7] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461—-466. http://dx.doi.org/10.5441/002/edbt.2016.42
[8] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. 2022. Adaptive Lower-Level Driven Compaction to Optimize LSM-Tree Key-Value Stores. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 34, 6 (2022), 2595–2609. https://doi.org/10.1109/TKDE.2020.3019264
[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1—-4:26. https://doi.org/10.1145/1365815.1365816
[10] CockroachDB. 2024. CockroachDB. *https://github.com/cockroachdb/cockroach* (2024).
[11] CockroachDB. 2024. Pebble. (2024). https://github.com/cockroachdb/pebble
[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94. https://doi.org/10.1145/3035918.3064054
[13] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520. https://doi.org/10.1145/3183713.3196927
[14] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084. https://www.vldb.org/pvldb/vol15/p3071-dayan.pdf
[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220. https://doi.org/10.1145/1323293.1294281
[16] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf
[17] Facebook. 2018. Splitting a File at Key Boundaries of Grandparant Files in RocksDB. (2018). https://github.com/facebook/rocksdb/blob/8.9.fb/db/compaction/compaction{_}outputs.cc{#}L323
[18] Facebook. 2022. RoundRobin in RocksDB. (2022). https://github.com/facebook/rocksdb/blob/b75438f9860e3cff5e713917ed22e0ac394a758c/include/rocksdb/advanced{_}options.h{#}L61
[19] Facebook. 2023. MyRocks. *http://myrocks.io/* (2023).
[20] Facebook. 2024. RocksDB. *https://github.com/facebook/rocksdb* (2024).
[21] Facebook. 2024. RocksDB Users. (2024). https://github.com/facebook/rocksdb/blob/main/USERS.md
[22] Google. 2021. LevelDB. *https://github.com/google/leveldb/* (2021).
[23] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang,

Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[24] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 651–665. https://doi.org/10.1145/3299869.3314041

[25] Shubham Kaushik and Subhadeep Sarkar. 2024. Anatomy of the LSM Memory Buffer: Insights \& Implications. In *International Workshop on Testing Database Systems (DBTest)*. https://doi.org/10.1145/3662165.3662766

[26] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 149–166. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lim

[27] Chen Luo and Michael J. Carey. 2018. LSM-based Storage Techniques: A Survey. *CoRR* abs/1812.0 (2018). arXiv:1812.07527 https://arxiv.org/abs/1812.07527

[28] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[29] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230. https://doi.org/10.14778/3415478.3415546

[30] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *CoRR* abs/2308.0 (2023). https://doi.org/10.48550/ARXIV.2308.07013

[31] 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. https://doi.org/10.1007/s002360050048

[32] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed Compaction for the LSM-Tree. *Int. J. Parallel Program.* 45, 6 (2017), 1310–1325. https://doi.org/10.1007/s10766-016-0472-z

[33] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. https://doi.org/10.1145/3318464.3389757

[34] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. https://doi.org/10.14778/3476249.3476274

[35] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. LSM Compaction Analysis. *https://github.com/BU-DiSC/LSM-Compaction-Analysis* (2021).

[36] ScyllaDB. 2024. Online reference. (2024). https://www.scylladb.com/

[37] Jaewoo Shin, Jianguo Wang, and Walid G Aref. 2021. The LSM RUM-Tree: A Log Structured Merge R-Tree for Update-intensive Spatial Workloads. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 2285–2290. https://doi.org/10.1109/ICDE51399.2021.00238

[38] Hui Sun, Guanzhong Chen, Yinliang Yue, and Xiao Qin. 2023. Improving LSM-Tree Based Key-Value Stores With Fine-Grained Compaction Mechanism. *IEEE Trans. Cloud Comput.* 11, 4 (2023), 3778–3796. https://doi.org/10.1109/TCC.2023.3329646

[39] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 2 (2023), 195:1—-195:27. https://doi.org/10.1145/3589775

[40] Xiaoliang Wang, Peiquan Jin, Bei Hua, Hai Long, and Wei Huang. 2022. Reducing Write Amplification of LSM-Tree with Block-Grained Compaction. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 3119–3131. https://doi.org/10.1109/ICDE53745.2022.00279

[41] Zhiqi Wang and Zili Shao. 2023. MirrorKV: An Efficient Key-Value Store on Hybrid Cloud Storage with Balanced Performance of Compaction and Querying. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 4 (2023), 249:1—-249:27. https://doi.org/10.1145/3626736

[42] Ran Wei, Zichen Zhu, Andrew Kryczka, Jay Zhuang, and Manos Athanassoulis. 2024. Benchmark, Analyze, Optimize Partial Compaction in RocksDB. (2024). https://github.com/BU-DiSC/Benchmark-Analyze-Optimize-Partial-Compaction-in-RocksDB-Codebase

[43] WiredTiger. 2021. Merging in WiredTiger's LSM Trees. *https://source.wiredtiger.com/develop/lsm.html* (2021).

[44] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 124:1—-124:28. https://doi.org/10.1145/3654927

[45] Baoquan Zhang and David H C Du. 2021. NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction. *ACM Transactions on Storage* 17, 3 (2021), 23:1—-23:26. https://doi.org/10.1145/3453300

[46] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined Compaction for the LSM-Tree. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 777–786. https://doi.org/10.1109/IPDPS.2014.85

[47] Zichen Zhu, Arpita Saha, Manos Athanassoulis, and Subhadeep Sarkar. 2024. KVBench: A Key-Value Benchmarking Suite. In *International Workshop on Testing Database Systems (DBTest)*. 9–15. https://doi.org/10.1145/3662165.3662765