Compaction Size and Tail Latency in LSM-based Key-Value Stores

Giorgos Xanthakis*
Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH)
Heraklion, Crete, Greece
gesalous@ics.forth.gr

Antonis Katsarakis
Huawei Research
Edinburgh, United Kingdom
antonioskatsarakis@gmail.com

Georgios Saloustros*
Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH)
Heraklion, Crete, Greece
gesalous@ics.forth.gr

Angelos Bilas*
Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH)
Heraklion, Crete, Greece
bilas@ics.forth.gr

Abstract

Log-structured merge-tree (LSM) engines such as RocksDB achieve high write throughput by buffering updates in memory and propagating them to disk through background compactions. However, once intermediate levels become saturated, the engine must synchronously compact large volumes of data before it can accept new writes, leading to multi-second write stalls even at moderate overall storage utilization.

We present *Rabbit*, a compaction design that bounds the amount of compulsory compaction work on the ingest path. Rather than relying on compaction scheduling or deferral, *Rabbit* changes the granularity at which compaction is performed and prioritizes low-amplification key ranges, allowing the engine to unblock writes after compacting a bounded amount of data.

Our evaluation using a RocksDB-based prototype shows that *Rabbit* performs subsecond write stalls in our evaluated workloads without increasing write amplification relative to the default leveled compaction policy.

CCS Concepts: • **Information systems → DBMS engine architectures**.

Keywords: Key-Value Stores, LSM-trees,Tail Latency

ACM Reference Format:
Giorgos Xanthakis, Antonis Katsarakis, Georgios Saloustros, and Angelos Bilas. 2026. Compaction Size and Tail Latency in LSM-based Key-Value Stores. In *6th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3805687.3806260

*Also with the Computer Science Department, University of Crete

CHEOPS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2604-0/2026/04
https://doi.org/10.1145/3805687.3806260

1 Introduction

Log-structured merge-tree (LSM) engines [7, 10, 12, 13] such as RocksDB sustain high write throughput using a multi-level structure. The engine buffers incoming updates in memory and incrementally propagates them to disk across levels. To preserve sorted order, the engine performs compactions that merge-sort data between adjacent levels once a level reaches its size limit. Compactions execute in the background as long as each level provides sufficient free space to absorb incoming data, which allows the engine to run multiple compactions concurrently across different levels.

The engine stalls client writes when it cannot flush the in-memory component to the first on-disk level due to insufficient space. These stalls occur when background compactions fail to keep pace with the incoming write rate. To mitigate this problem, LSM engines rely on techniques such as compaction scheduling and compaction deferral.

Compaction scheduling approaches, such as Silk [1], assign priorities to background operations based on their proximity to the in-memory component. The engine prioritizes memtable flushes and upper-level compactions because they directly affect write availability. It assigns lower priority to compactions deeper in the tree, since they do not immediately block writes. Scheduling effectively prevents stalls while larger levels still provide slack space. However, once intermediate levels reach their configured size limits, scheduling no longer suffices. At that point, the engine must complete all pending compactions that block progress toward the top of the tree, regardless of priority. In typical LSM configurations, intermediate levels reach capacity when the system uses approximately 10 percent of its total storage budget.

Compaction deferral [7, 14, 15] complements scheduling by allowing the engine to temporarily exceed the configured size of individual levels. The engine tracks excess data using per-level overflow counters that record how many bytes exceed each level's target size. Current systems do not rebalance overflow across levels. Deferral absorbs short write bursts effectively and prevents immediate stalls. Under sustained write-heavy workloads, however, deferral increases compaction sizes and amplifies stall duration. Once the engine reaches the overflow limit, it must compact substantially larger amounts of data before it can resume writes.

These stalls arise when intermediate levels become saturated and block progress toward the lowest level. In this state, the engine must compact data across multiple levels before it can flush the next memtable, forcing large, synchronous compactions on the write path. As a result, write availability collapses once utilization crosses a configuration-dependent threshold, even though sufficient total free space remains.

Prior work [1, 7, 15] primarily mitigates write stalls through compaction scheduling or deferral, for example by prioritizing upper-level compactions or delaying work until additional resources become available. While these approaches reduce the frequency of stalls, they do not change the amount of data that must eventually compact to unblock writes. As a result, they cannot prevent large, synchronous compactions once blocking occurs.

To illustrate this effect, consider an LSM tree where the first two on-disk levels have reached their size limits, while deeper levels still have ample free space. When the engine attempts to flush a full memtable, it cannot proceed until it completes a chain of compactions across these full intermediate levels. Even though the memtable itself may contain only tens of megabytes, the engine must compact gigabytes of data to create sufficient space. As a result, clients observe multi-second write stalls despite low overall storage utilization.

These observations suggest that eliminating long write stalls requires reducing the amount of data involved in each compaction along the critical path from memory to disk. However, naively reducing memtable or file sizes increases the number of levels and significantly amplifies write amplification. Prior work either accepts this trade-off or focuses on scheduling, which does not reduce the total volume of data that must be compacted. In contrast, our approach reduces compaction size on the critical path while preserving overall write amplification, without relying on scheduling or deferring compaction work.

In this paper, we present *Rabbit*, which bounds the amount of compulsory compaction work required to unblock writes. *Rabbit* achieves this by reducing the granularity of compaction and prioritizing key ranges that incur low amplification when compacted. By changing the unit of compulsory work rather than its scheduling, *Rabbit* removes the utilization cliff observed in RocksDB-class engines.

Rabbit begins from a simple observation: smaller memtables and smaller Sorted String Tables (SSTs) files reduce the amount of data that must be compacted to unblock a memtable, which directly reduces stall latency. However, applying this change naively increases the number of LSM levels and substantially amplifies write amplification. *Rabbit* addresses this tension by decoupling compaction granularity from overall write amplification.

Specifically, *Rabbit* uses small memtables and small SST files in the common write path, which reduces the size of compactions that block memtable flushes. To offset the resulting increase in write amplification, *Rabbit* opportunistically creates key ranges in the first on-disk level that exhibit low compaction amplification. These ranges allow the system to compact data with significantly lower amplification early in the LSM tree.

Rabbit then exploits this reduced amplification by increasing the growth factor of the second level by up to 32×. This design enables *Rabbit* to maintain a similar number of levels and overall write amplification as state-of-the-art systems, while substantially reducing the amount of data involved in each compaction along the critical path.

Overall, our work makes the following contributions:

- We identify a utilization cliff in LSM engines caused by unbounded compaction work on the write critical path.
- We propose *Rabbit*, an LSM design that bounds critical-path compaction size while preserving write amplification.
- We demonstrate that *Rabbit* significantly reduces write stall latency under write-heavy workloads.

2 Background

LSM key-value stores sustain high write throughput by buffering updates in memory and incrementally compacting data across a multi-level on-disk structure [4, 12]. Each level is typically configured to be a fixed growth factor larger than the previous one, amortizing write costs by merging sorted runs in the background. As long as each level provides sufficient free space, compactions can proceed asynchronously without affecting the foreground write path.

2.1 Write stalls and the critical path

Writes stall when the in-memory component cannot flush to disk due to insufficient space in the first on-disk level [11]. At that point, the system must complete one or more compactions along the path from memory to deeper levels to free space. These compactions form a critical path that directly blocks progress of the ingest pipeline.

A key observation is that the duration of a write stall does not depend on the size of the flushing memtable, but on the total amount of data that the engine must reorganize along the critical path. When intermediate levels are full, the engine

may need to move gigabytes of data across multiple levels to compact a small memtable, which results in multi-second stalls even on fast flash devices.

2.2 Low utilization, large stalls

In typical leveled LSM configurations [3–5, 12], intermediate levels reach their target size when the database occupies only a small fraction of total storage capacity. For common growth factors (e.g., 8–10×), this occurs at roughly 10% overall utilization. Beyond this point, compactions become interdependent: freeing space in an upper level requires first compacting into a lower level that is already at capacity.

As a result, a memtable flush may trigger a chain of dependent compactions spanning several levels. Although the database as a whole has ample free space, the write path remains blocked until the entire chain completes. This creates a sharp utilization cliff (a sudden transition from low to extreme stall latency), where stall latency increases abruptly rather than gradually as the system fills.

2.3 Bytes-to-unblock as a stall metric

To characterize this behavior, we focus on the amount of data that must compact before a stalled memtable flush can proceed, which we refer to as bytes-to-unblock. This metric captures the minimum compulsory work on the critical path, independent of scheduling or background parallelism.

Empirical measurements show that once intermediate levels are full, bytes-to-unblock grows from tens of megabytes to multiple gigabytes, even though the memtable size remains unchanged. This growth directly translates into long write stalls, as the engine must synchronously wait for all dependent compactions to complete.

2.4 Limitations of existing mitigation techniques

Prior work mitigates write stalls primarily through compaction scheduling, throttling, or deferral. Scheduling approaches prioritize upper-level compactions to delay stalls, while deferral mechanisms allow levels to temporarily exceed their size limits. These techniques are effective when there is slack space in the LSM tree, but they do not reduce the minimum amount of data that must be compacted once intermediate levels reach capacity.

Consequently, when the system operates beyond the utilization cliff, existing designs must still compact large volumes of data on the critical path to unblock a single memtable flush. Scheduling can change when this work occurs, but not how much work is unavoidable.

2.5 Limits of Scheduling and Deferral Under Saturation

Compaction scheduling techniques assign priorities to background work so that flushes and upper-level compactions execute early. These techniques are effective while the LSM retains slack space in intermediate levels. In that regime, the engine can advance the dependency chain ahead of time and overlap compactions with foreground writes, reducing stall frequency.

However, once intermediate levels approach their size limits, the system enters a saturated regime. At that point, freeing space in level L_i requires completing compactions into L_{i+1}, which may itself be full. Progress is then governed by dependency constraints rather than scheduling policy. Scheduling still determines which ready compaction executes next, but it cannot reduce the minimum compulsory amount of data that must be reorganized before a memtable flush can proceed.

In the best case, perfect scheduling reduces a stall to the duration of a single critical compaction chain. Yet the size of that chain remains large if compaction units (SST files) are large.

Deferral-based techniques temporarily allow levels to exceed size targets by accumulating compaction debt. Deferral is effective for absorbing bursts, but under sustained write pressure it increases the size of future mandatory compactions, potentially increasing tail stall latency once limits are reached.

Overall, prior work reduces stall frequency and shifts work in time, but does not provide an explicit bound on the amount of compulsory work required to unblock a flush. Bounding stall latency requires bounding the compaction unit on the critical path.

3 Design

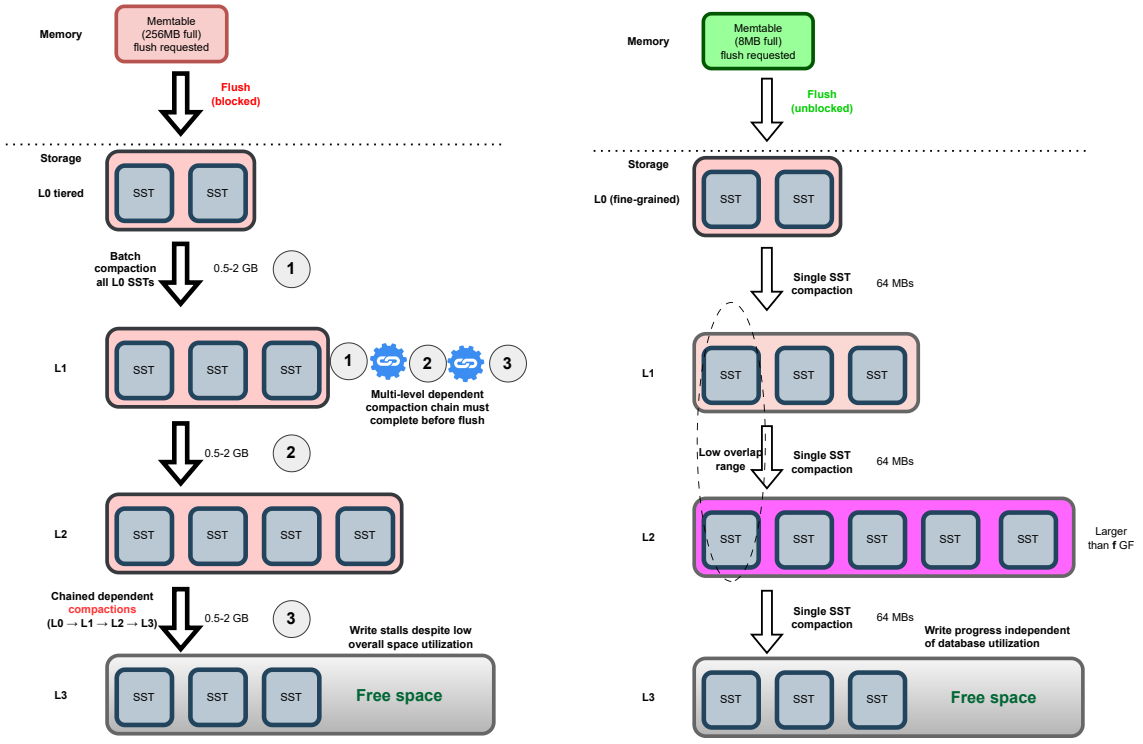
Design overview. *Rabbit* is designed to bound the amount of work required to make progress on the ingest path. It achieves this through two complementary mechanisms. First, it reduces the granularity of compaction by using small memtables and SSTs, limiting the size of individual compaction units. Second, it prioritizes compactions based on overlap, selecting SSTs whose expansion into the next level is small. Together, these ensure that each compaction operates on a bounded amount of data, even when the LSM tree is highly utilized. Figure 1(b) contrasts the blocking compaction chain in a conventional leveled design with *Rabbit*'s bounded critical-path compactions.

3.1 Objective: Bounding Critical-Path Work

Let S denote the target SST file size on the ingest path, f the growth factor between adjacent levels, and k the number of blocking levels on the critical path at the time of a flush.

We define *compaction granularity* G as the maximum number of input bytes of any single compaction that a memtable flush may have to wait for in order to make progress. In other words, G represents the compulsory unit of work per level along the critical path.

In a conventional leveled LSM, compacting one file of size S in level L_i typically overlaps with $\Theta(fS)$ bytes in level



(a) RocksDB (b) *Rabbit*

Figure 1. In conventional LSM designs (left), flushing a memtable when intermediate levels are full requires compacting large amounts of data across multiple levels, resulting in multi-second write stalls. In *Rabbit* (right), compactions on the critical path operate on small units, bounding the amount of data that must be reorganized to unblock writes, largely independent of database size.

L_{i+1}. The compulsory compaction unit per level is therefore $\Theta(S(1+f))$ in expectation. If k consecutive levels are saturated, the bytes that must be reorganized before a flush can proceed are:

$$\texttt{bytes-to-unblock} = O(k \cdot G), \quad \text{where } G \approx S(1+f).$$

Rabbit's objective is to bound G on the ingest path, thereby bounding the worst-case bytes-to-unblock, while preserving competitive long-run write amplification.

3.2 Constraints

Any practical solution must respect three constraints common in production LSM deployments:

(1) The growth factor f cannot be arbitrarily reduced. Lowering f reduces compaction width and thus granularity, but increases write amplification and often increases metadata and read costs by deepening the tree.

(2) File sizes cannot be uniformly shrunk across all levels. Reducing S everywhere increases the number of levels and lengthens dependency chains, partially offsetting latency gains.

(3) Memory is typically controlled via the memtable and L0 budgets. Operators regulate memory usage through write buffer sizes and L0 thresholds. Increasing these budgets can delay stalls or reduce amplification, but is often infeasible in multi-tenant or memory-constrained environments.

These constraints imply that bounding critical-path granularity requires a more selective approach than globally tuning f or S.

3.3 Tension between granularity and amplification

In conventional LSM designs, compaction granularity and write amplification are tightly coupled. Large SSTs amortize merge costs across levels and keep amplification low, but they also inflate the amount of data involved in each compaction on the critical path. Conversely, small SSTs reduce stall latency by limiting compaction size, but increase

amplification by violating the expected size ratios between levels.

This coupling explains why prior work either accepts long stalls or trades them for higher amplification. Existing designs implicitly treat compaction granularity as fixed and optimize around it.

3.4 Decoupling Critical-Path Granularity from Global Amplification

Conventional leveled LSMs couple two roles: (i) the compaction unit that may block a flush, and (ii) the parameters that determine long-run write amplification.

Rabbit separates these roles. First, it uses a small SST size S along the ingest path and in the first blocking levels. This bounds the maximum compaction unit that a flush may synchronously wait for.

Second, it preserves competitive long-run amplification by using asymmetric growth: it increases the effective growth factor between upper levels (e.g., between L_1 and L_2) while keeping conventional growth deeper in the tree. This maintains overall capacity scaling without uniformly increasing granularity.

Finally, *Rabbit* prioritizes low-overlap key ranges when selecting compactions on the critical path. By favoring ranges whose overlap with the next level is small, it reduces the effective compulsory work per flush without globally changing f. Together, these mechanisms bound critical-path granularity while preserving the asymptotic amplification properties of leveled LSMs.

4 *Rabbit*

4.1 Small-granularity ingest path

Reducing compaction granularity. *Rabbit* reduces the size of individual compaction units by using small memtables and SST files on the ingest path. In our implementation, memtables are reduced from 256 MB to 8 MB, and SST sizes range from 8–64 MB, compared to 64–256 MB in conventional configurations.

This ensures that each compaction is triggered by a small input and operates on a limited amount of data. Instead of batching multiple sorted runs before compaction, *Rabbit* flushes and propagates data incrementally, allowing progress to be made through a sequence of small compactions. While small SSTs reduce the size of compaction inputs, the total work of a compaction still depends on how much data they overlap in the next level. *Rabbit* therefore complements small compactions with an overlap-aware selection policy.

4.2 Preserving amplification with asymmetric growth

Shrinking SST sizes naively would either increase the number of levels or violate the growth factor between adjacent levels, both of which increase write amplification. *Rabbit* avoids this by introducing asymmetry in the upper part of the tree.

Specifically, *Rabbit* uses a larger growth factor between the first two on-disk levels than between deeper levels. This allows the tree to absorb fine-grained compactions near the ingest path without increasing its overall depth. As a result, *Rabbit* maintains the same number of levels as conventional designs, despite operating with much smaller SSTs on the critical path.

4.3 Opportunistic low-amplification compactions

Overlap-aware compaction selection. A larger growth factor at the top of the tree increases overlap between adjacent levels, leading to higher amplification. *Rabbit* mitigates this by exploiting heterogeneity in key-range overlap.

For each candidate SST s in level L_i, *Rabbit* estimates the compaction cost as the number of bytes in level L_{i+1} whose key range overlaps with s:

$$\text{cost}(s) = \sum_{x \in L_{i+1}} \text{overlap}(s, x)$$

Where $\text{overlap}(s, x)$ denotes the number of bytes in SST x whose key range intersects with s. *Rabbit* prioritizes SSTs with minimum overlap cost. This ensures that each compaction expands into a bounded amount of data in the next level, reducing the effective overlap factor α relative to the growth factor f.

In practice, this partitions SSTs into two categories: low-overlap SSTs that incur low amplification when compacted, and high-overlap SSTs that would expand into large portions of the next level. *Rabbit* prioritizes low-overlap SSTs when available, using them to make progress with minimal compaction work.

High-overlap SSTs are deferred but not starved. As lower-cost candidates are exhausted, they eventually become the minimum-cost options and are compacted as part of normal operation. This ensures progress without accumulating unbounded compaction debt.

This policy bounds the amount of data each compaction expands into, ensuring that progress can be made with limited and predictable work even under high utilization.

4.4 What *Rabbit* changes and what it does not

Rabbit makes minimal changes to the overall LSM architecture: It changes the granularity and ordering of compactions near the write path. It modifies the size relationship between the first two on-disk levels. At the same time, *Rabbit* preserves key properties of conventional LSM engines: The total number of levels remains unchanged. Write amplification remains comparable to state-of-the-art systems. Memory usage does not increase. Together, these mechanisms ensure that compaction work remains bounded and predictable, preventing large expansions that would otherwise lead to stalls under high utilization.

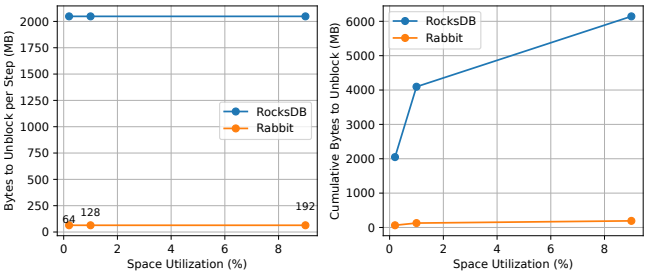


Figure 2. Bytes-to-unblock as a function of storage utilization.

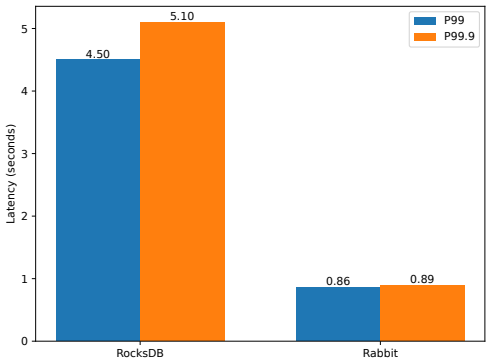


Figure 3. P99 and P99.9 tail latencies percentiles for RocksDB and *Rabbit*.

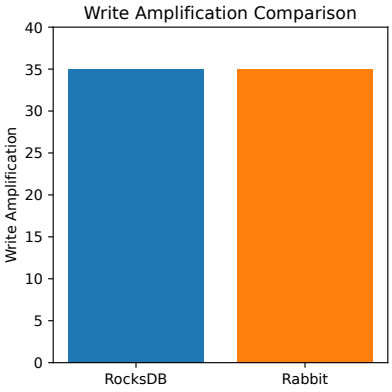


Figure 4. Write amplification for RocksDB and *Rabbit*.

5 Experimental Methodology

We evaluate *Rabbit* using a prototype built on top of RocksDB. We conduct our experiments on a dedicated machine equipped with two Intel(R) Xeon(R) CPU E5-2630 v3 CPUs running at 2.4 GHz, 256 GB of memory, and a Samsung 2TB NVMe flash SSD. The operating system is Ubuntu Linux 20.04. We configure RocksDB and *Rabbit* with 4 device levels, L_1 is 100 MB and growth factor is 8, while using leveled organization [9]. We configure both systems to use direct I/O [8] as it saves considerable CPU and makes I/O predictable compared to buffered I/O.

Both systems use identical memory budgets (256 MB) [6], background thread counts (4). The only differences are *Rabbit*'s fine-grained ingest path, opportunistic range selection, and asymmetric growth configuration.

Our workloads are write-heavy (YCSB [2] Load A), designed to stress the ingest pipeline by rapidly filling intermediate levels. We focus on steady-state behavior after the database grows beyond the point where intermediate levels reach capacity.

We report three metrics. First, bytes-to-unblock, defined as the amount of data that must compact before a stalled memtable flush can proceed. Second, write stall duration, measured as the time during which client writes block. Third, write amplification, computed as the ratio of total bytes written to storage to user-level bytes ingested.Space utilization is the ratio of user-level data in the system to total storage capacity.

6 Evaluation

6.1 Eliminating the utilization cliff

Figure 2 plots bytes-to-unblock as a function of space utilization. In the baseline RocksDB configuration, bytes-to-unblock increases sharply once intermediate levels reach capacity, growing from hundreds of megabytes to multiple gigabytes at approximately 10% utilization. This sharp increase corresponds to the utilization cliff described in Section 2. Once intermediate levels are full, each additional 10% increase in storage utilization increases bytes-to-unblock by a roughly constant increment (about 200 MB in this configuration).

Rabbit removes multi-second write stalls in our evaluated workloads, while the default RocksDB configuration exhibits extended stall periods once utilization exceeds 10%. Across all utilization levels, bytes-to-unblock remains bounded by a small multiple of the SST size per level. Even when all intermediate levels are full, unblocking a memtable flush requires compacting only a limited amount of data on each level, rather than triggering a chain of large dependent compactions.This result confirms that bounding critical-path compaction size is sufficient to eliminate long write stalls.

6.2 Tail latency and Write Stalls

Figure 3 shows tail latency behavior during execution as the database grows beyond the utilization cliff. Each percentile is computed across all operations in the workload, including those that experience stalls. In the baseline RocksDB configuration, P99 latency increases sharply once intermediate levels reach capacity, reflecting the onset of multi-second write stalls. P99.9 latency exhibits an even more pronounced increase, indicating that a significant fraction of writes experience extreme stalls.

In the baseline RocksDB configuration, stalls are infrequent at low utilization but become increasingly severe once intermediate levels reach capacity. After this point, individual stall events last multiple seconds, reflecting the need to compact large volumes of data across all levels before a memtable flush can proceed. Stall duration remains roughly constant with dataset size, but each stall remains large. At 10% stalls stabilize and RocksDB has alternating periods of 5 second stalls and normal operation, as the engine compacts large amounts of data to unblock writes.

In contrast, *Rabbit* maintains sub-second P99.9 latency across all utilization levels, even as the database grows beyond the point where intermediate levels are full. Although stall events still occur, their duration remains limited and does not increase considerably as utilization grows. This behavior confirms that bounding the amount of compulsory compaction work on the critical path is sufficient to prevent catastrophic write stalls, even when intermediate levels are full.

6.3 Write amplification

Figure 4 compares write amplification for RocksDB and *Rabbit*. In our experiment, *Rabbit* does not increase write amplification relative to the default leveled compaction policy of RocksDB. This result suggests that low amplification key ranges appear in the same levels as they would under the default policy. Although *Rabbit* may perform additional compactions to unblock memtable flushes, these compactions do not increase overall write amplification in our workloads.

7 Conclusion

This paper shows that long write stalls in LSM-based key-value stores are caused by unbounded compulsory compaction work on the write path's critical chain, not by insufficient throughput or poor scheduling. Existing techniques shift compaction work in time but do not reduce the amount of work required to unblock writes.

Rabbit eliminates long write stalls by bounding critical-path compaction size while preserving write amplification and memory usage. By decoupling compaction granularity from global amplification, Rabbit delivers predictable write performance even at low overall utilization, providing a practical path toward stall-free LSM engines.

8 Acknowledgments

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the project EUPEX (Grant Agreement ID: 101033975) and Horizon Europe Framework Programme for Research and Innovation through the project DAFAB (Grant Agreement ID: 101128693). We thank the anonymous reviewers for their helpful feedback and suggestions.

References

[1] 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. *ACM Trans. Comput. Syst.* 36, 4, Article 12 (may 2020), 27 pages. doi:10.1145/3380905

[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[3] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. doi:10.1145/3035918.3064054

[4] 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 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. doi:10.1145/3183713.3196927

[5] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proc. VLDB Endow.* 15, 11 (jul 2022), 3071–3084. doi:10.14778/3551793.3551853

[6] Jason Evans. 2018. jemalloc. http://jemalloc.net/.

[7] Facebook. 2018. RocksDB. http://rocksdb.org/.

[8] Facebook. 2018. RocksDB Direct I/O. https://github.com/facebook/rocksdb/wiki/Direct-IO. Accessed: April 2, 2026.

[9] Facebook. 2018. RocksDB Leveled Compaction. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction. Accessed: April 2, 2026.

[10] Google. 2023. LevelDB. https://github.com/google/leveldb.

[11] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. 2020. BoLT: Barrier-optimized LSM-Tree. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 119–133. doi:10.1145/3423211.3425676

[12] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[13] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. doi:10.1145/3132747.3132765

[14] Heejin Yoon, Jin Yang, Juyoung Bang, Sam H. Noh, and Young-ri Choi. 2024. Advocating for Key-Value Stores with Workload Pattern Aware Dynamic Compaction. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems* (Santa Clara, CA, USA) *(HotStorage '24)*. Association for Computing Machinery, New York, NY, USA, 124–130. doi:10.1145/3655038.3665955

[15] Jinghuan Yu, Sam H. Noh, Young-ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow between Components in Log-Structured Key-Value Stores for Improved Performance. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, Article 5, 16 pages.