BucketLSM: Breaking the Compaction Scalability Barrier in LSM-Based Key-Value Stores

Jaewan Park*, Kyungwook Min*, Sungjin Byeon*, Taewan Noh*, Hyungi Park*, Xubin He†
Hong-Yeon Kim‡, Youngjae Kim*,§
*Sogang University, †Temple University, ‡ETRI
{brian7567, minsky, sjbyeon, heize0502, hyun0712, youkim}@sogang.ac.kr, xubin.he@temple.edu, kimhy@etri.re.kr

***Abstract*—LSM-tree-based key-value stores serve as the storage backbone for a wide range of cloud services, where write throughput and read latency directly determine quality of service. These systems adopt an append-only write strategy that buffers writes in memory and flushes them to disk as separate files, achieving high write throughput at the cost of accumulating multiple versions of the same key across files. A background process called compaction periodically merges these files to eliminate redundant entries and sustain read efficiency. However, at Level-0 (L_0)—where flushed files first land on disk—files are allowed to have arbitrarily overlapping key ranges. This overlap forces L_0–L_1 compaction to merge nearly all files from both levels as a single coarse-grained task, fundamentally limiting compaction parallelism. When L_0–L_1 compaction cannot keep pace with incoming writes, L_0 files accumulate, triggering severe write stalls and read latency spikes. This paper presents BucketLSM, an LSM architecture that partitions L_0 into non-overlapping buckets to unlock scalable compaction parallelism. BucketLSM splits MemTables at flush time according to bucket boundaries, creating structurally independent compaction units. Key mechanisms include (i) *BucketFlush* for boundary-aligned file generation, (ii) *BucketCompaction* for fine-grained priority-based scheduling, and (iii) *Dynamic Bucket Rebalancing* to adapt boundaries under shifting workloads. Evaluated on RocksDB v10.6, BucketLSM achieves up to 2.6× throughput improvement over the original RocksDB, reduces write stalls by up to 16.7× compared to baseline RocksDB, and improves read performance under high write pressure—all without requiring additional hardware.**

***Index Terms*—Log-Structured Merge-Tree (LSM-tree), Key-Value Store, Compaction Parallelism, Write Stall, Fine-Grained Compaction**

I. Introduction

Log-Structured Merge-trees (LSM-trees) [1] store data as key–value pairs rather than structured table schemas, enabling high-throughput ingestion of large-scale, unstructured data across diverse services and applications. Owing to this property, LSM-trees have become the core I/O engine in a wide range of systems, including cloud object storage, blockchains, vector databases, and relational database storage engines [2]–[6]. Widely used LSM-based engines and systems include RocksDB [7], LevelDB [8], MyRocks [9], and Milvus [10], which serve as storage backbones for performance-critical services such as distributed databases, caching layers, and log stores in modern cloud infrastructures. Consequently, the performance of LSM-based key–value stores (LSM-KVSs) strongly affects the latency, throughput, and quality of service (QoS) of the applications that rely on them.

LSM-trees adopt an append-only design instead of in-place updates to achieve high write throughput. Incoming writes are first buffered in an in-memory MemTable and then flushed to disk as immutable SST files using sequential I/O, effectively converting random writes into sequential access patterns. Unlike higher levels, Level-0 (L_0) uniquely allows arbitrary key-range overlap among SST files, enabling MemTables to be flushed directly without merging with existing on-disk data and thereby reducing write latency. However, as multiple versions of the same key are scattered across overlapping L_0 SST files, read operations must probe many files, causing read latency to increase rapidly as the number of files grows. This behavior reflects a fundamental trade-off in LSM-trees between write efficiency and read performance.

Motivation. To mitigate the above problem, LSM-based key–value stores perform background compaction, which merge-sorts SST files to eliminate duplicate keys and reduce file counts. Due to key-range overlap in L_0, however, L_0–L_1 compaction must merge nearly all SST across both levels as a single coarse-grained task, structurally limiting compaction parallelism. When the write arrival rate exceeds the compaction throughput, L_0 SST files accumulate, leading to recurring read latency spikes and write stalls.

Prior work [3], [11]–[18] has proposed various approaches to mitigate these issues, but most either sacrifice one dimension of performance or require additional hardware.

First, scheduling- and tuning-based techniques improve the coordination between internal and client operations. SILK [11] introduces an I/O scheduler that prioritizes latency-critical flushes and L_0-L_1 compactions over higher-level compactions. ADOC [12] dynamically adjusts thread counts and MemTable, SST file sizes to balance dataflow across LSM-tree components. These techniques can reduce the frequency of write stalls, yet they do not eliminate the fundamental structural bottleneck rooted in L_0 overlapping.

Second, other approaches primarily target write throughput by relaxing compaction constraints or deferring compaction work. DownForce [13] temporarily permits intra-level key-range overlap to enable non-blocking pipelined compaction across levels. PebblesDB [14] allows overlapping key ranges across levels to avoid costly SST merges, and TRIAD [3]

§Corresponding author.

selectively defers compaction when key-range overlap is low. While these techniques improve write performance by delaying or relaxing compaction, they do so at the cost of increased key-range overlap, which degrades read efficiency or introduces additional compaction debt that must be resolved later.

Third, hardware-assisted approaches offload or accelerate compaction using GPUs [15], DPUs [16], NVM [17], or host-SSD collaboration [18]. While effective in their target environments, these approaches rely on specialized hardware and device-specific optimizations, limiting their general applicability across diverse deployment configurations.

Key Insights and Technical Contributions. In conventional LSM-trees, L_0 allows key-range overlap among SST files during flush, resulting in many files broadly overlapping across the entire key space (Section III). *This L_0 overlapping structure unnecessarily expands the search scope for reads and forces L_0–L_1 compaction to operate as a single coarse-grained task, leading to write stalls.* Motivated by the observation that these limitations stem from the structure of L_0, where SST files are first created, this paper proposes BucketLSM, a new LSM design that improves both read and write performance without requiring additional hardware.

To this end, BucketLSM introduces a new LSM structure that partitions the L_0 key space into multiple non-overlapping buckets. Each bucket owns a disjoint key range, enabling L_0–L_1 compaction to proceed in fine-grained, independent units and fundamentally expanding compaction parallelism that is structurally constrained in existing RocksDB-based designs.

To fully realize the potential of this bucketized L_0 while delivering high parallelism and stable performance, BucketLSM incorporates a set of tightly integrated, system-level mechanisms. Specifically, it implements (i) a *bucket-aware compaction scheduler* that prioritizes buckets based on L_0 file pressure and compaction cost to rapidly mitigate read-critical hotspots (Section V-B); (ii) a *dynamic bucket management framework* that continuously adapts bucket boundaries to skewed and evolving key distributions via split and merge operations, sustaining parallelism over time (Section V-C); and (iii) a *redesigned L_0 stall policy* that aligns read and write control with bucket-level access semantics rather than global L_0 state (Section V-D). Together, these mechanisms transform bucketization from a static partitioning strategy into a practical, high-performance LSM design that structurally overcomes the compaction scalability limits of existing systems.

BucketLSM is implemented in RocksDB v10.6 and evaluated under diverse workloads with evolving key distributions and mixed read/write scenarios. The results show that BucketLSM improves both write and read performance over three baselines, achieving up to 2.6× higher throughput.

This paper makes the following contributions:

- We identify the L_0 overlapping structure in LSM-trees as a long-term bottleneck for both write and read performance.
- We show that key-range overlap in L_0 fundamentally limits both intra- and inter-compaction parallelism in L_0–L_1 compaction, forming a structural scalability barrier.

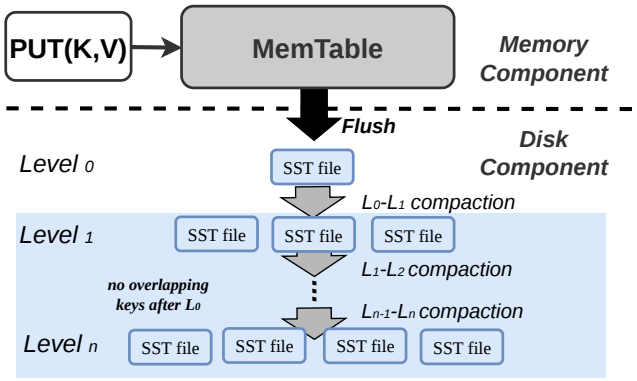


Fig. 1: LSM-tree architecture with append-only writes and multi-level compaction.

- We propose BucketLSM, a new LSM design that restructures L_0 into non-overlapping buckets, decomposing a single coarse-grained L_0–L_1 compaction into structurally independent, fine-grained compaction units.

II. Background: LSM-Tree and Write Stalls

A. LSM-Tree Structure and Read/Write Path

Fig. 1 illustrates the core structure and operation of an LSM-tree. An LSM-tree is a key–value storage architecture that achieves high write throughput using an append-only write path. Incoming PUT(k, v) requests are first buffered in an in-memory MemTable and, once full, flushed to disk as sequential SST files. These SST files are initially placed in Level-0 (L_0).

An LSM-tree consists of multiple levels with progressively increasing size limits. When a level exceeds its size threshold, background compaction is triggered to merge SST files into the next level. Compaction merge-sorts SST files to discard obsolete key versions, reclaim storage space, and maintain read efficiency. From L_1 onward, SST files are organized to maintain non-overlapping key ranges.

The cost of read operations in an LSM-tree is directly proportional to the number of SST files probed. Both point queries (GET) and range queries (SCAN) search SST files whose key ranges may contain the target key k, incurring index lookups and data accesses. As the number of SST files covering a given key increases, the I/O overhead grows, leading to higher GET(k) and SCAN(k) latency.

B. Root Causes of Write Stalls

In LSM-tree–based key–value stores, when the incoming write rate persistently exceeds the processing capacity of flush and compaction, unflushed data accumulates in memory and uncompacted SST files pile up on disk. As a result, GET and SCAN latencies increase sharply, and the system may face memory pressure and disk space exhaustion. To preserve system stability under such conditions, RocksDB deliberately throttles or halts incoming writes and prioritizes pending flush and compaction tasks, a mechanism known as write stall.

In RocksDB, write stalls are triggered under three conditions. First, L_0 stall occurs when the number of SST files in L_0

exceeds a threshold, preventing excessive read latency caused by severe file overlap. Second, MemTable stall is triggered when the number of MemTables reaches a system-defined limit, capping memory usage to prevent uncontrolled growth. Third, pending compaction bytes stall occurs when the total volume of unprocessed compaction work across all levels exceeds a threshold, avoiding disk space exhaustion due to excessive duplicate data.

III. The Status Quo: What's the Story in L_0 Overlapping?

In LSM-tree–based key–value stores, L_0 is the level where SST files are first created on disk and is intentionally designed as the only level that allows key-range overlap among SST files. This design choice minimizes write-path latency: during a flush, MemTables are written to SST files without considering the key ranges of existing L_0 SST files. Enforcing non-overlapping key ranges at flush time would require merging or re-sorting with existing SST files, increasing PUT latency. Accordingly, RocksDB defers expensive operations such as sorting and merging to L_1 and above, using L_0 as a staging area for fast write absorption.

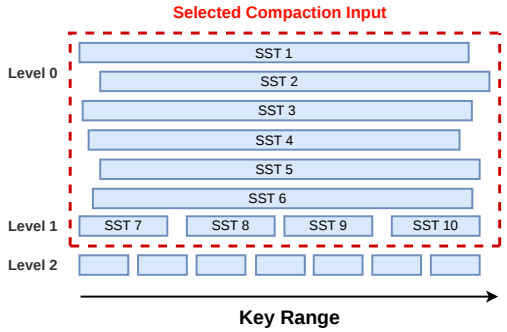


Fig. 2: Overlapping key ranges of L_0 SST files.

As a result, L_0 naturally accumulates SST files with widely overlapping key ranges, as shown in Fig. 2, in contrast to higher levels where non-overlapping key ranges are maintained. Consequently, GET and SCAN operations must probe many SST files in L_0, increasing I/O overhead and read latency.

Moreover, once the number of L_0 SST files exceeds a threshold, RocksDB triggers an L_0 stall to mitigate severe read latency degradation, temporarily halting incoming writes and significantly degrading PUT throughput.

In summary, key-range overlap in L_0 is a deliberate design choice to minimize write latency, but it ultimately leads to GET and SCAN latency spikes and reduced PUT throughput. Beyond these direct effects, however, L_0 overlap imposes a deeper structural limitation on compaction scalability, which we analyze in the following section.

IV. The Hidden Cost of L_0 Overlapping: A Compaction Scalability Barrier

In LSM-trees, when considering both write throughput and read latency, the key control variable is the number of SST files in L_0. As the L_0 file count grows, GET and SCAN operations must probe an increasing number of SST files, repeatedly incurring costly I/O for each file.

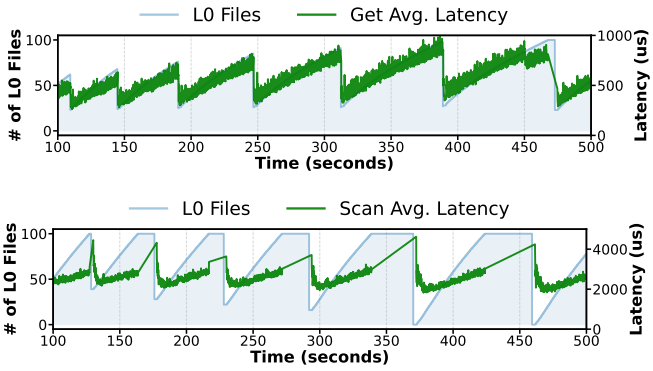


Fig. 3: Correlation between L_0 file growth and read latency. The *top* and *bottom* plots show GET and SCAN latency, respectively.

To illustrate this correlation, we run baseline RocksDB (v10.6) under the YCSB-GET and YCSB-SCAN workloads with 16B keys and 1KB values (detailed configuration in Section VI-A). Fig. 3 shows that both GET and SCAN latency increase as the number of L_0 SST files grows.

To prevent this degradation, RocksDB triggers write stalls once the number of L_0 SST files exceeds a threshold, throttling incoming writes. As a result, the L_0 file count directly governs both PUT throughput and the tail latency of GET and SCAN operations. Ideally, the system should keep the number of L_0 SST files low and stable even under sustained write pressure. In practice, however, RocksDB struggles to maintain this state because L_0–L_1 compaction cannot keep pace due to key-range overlap in L_0.

Our analysis indicates that key-range overlap in L_0 is the root cause preventing the effective reduction of L_0 SST files. We analyze this problem from two perspectives:

- (i) coarse-grained compaction that leads to excessively long compaction times (Section IV-A)
- (ii) structurally limited scalability of L_0 compaction parallelism (Section IV-B)

A. Coarse-grained Compaction

We measure compaction behavior on baseline RocksDB under the `fillrandom` workload, which inserts 100 GB of random key-value pairs(detailed configuration in Section VI-A).

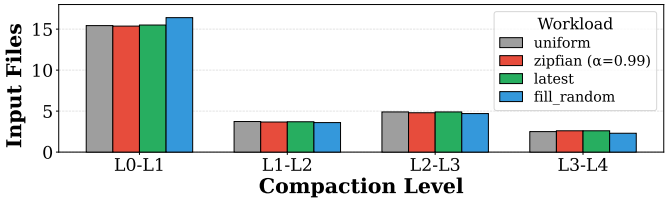


Fig. 4: Average number of SST files participating in compaction across levels under different key distributions.

Fig. 4 shows the compaction input size at each level. L_0–L_1 compaction involves more than 10 SST files on average, with a total input size of 1.014 GB per compaction task. Because all L_0 SST files have overlapping key ranges, L_0–L_1 compaction must include all SST files from both L_0 and L_1 as inputs to a single task, as illustrated in Fig. 1.

In contrast, compaction between higher levels (L_i–L_{i+1}, $i \geq 1$) operates on SST files that maintain non-overlapping key ranges on each level, allowing each task to involve only a small number of overlapping files across two levels and typically merges 3-4 files with a total input size of 0.179 GB on average.

This coarse-grained compaction structure introduces two fundamental problems.

Problem 1: Delayed Reduction of L_0 SST Files. The number of L_0 SST files cannot be reduced until a single L_0–L_1 compaction task completes. While compaction is in progress, MemTable flushes continue, causing new SST files to accumulate in L_0. As a result, the L_0 file count keeps increasing until compaction finishes, directly leading to reduced PUT throughput due to L_0 stalls and spikes in GET/SCAN latency. Although the L_0 file count drops sharply when compaction completes, the same accumulation pattern repeats over long periods, resulting in unstable system performance.

Problem 2: Delayed Upper-Level Compaction and Poor Thread Utilization. New SST files are generated in L_1 only after L_0–L_1 compaction completes, which in turn determines when L_1–L_2 compaction can be triggered. During a large, monolithic L_0–L_1 compaction, upper-level compactions remain idle because no new SST files are produced to satisfy their compaction trigger conditions. Once L_0–L_1 compaction completes, a large batch of SST files is injected into L_1, causing a cascade of upper-level compactions.

This behavior leads to inefficient compaction thread utilization. During L_0–L_1 compaction, only a single thread is actively used, whereas multiple threads become simultaneously busy after completion due to cascading upper-level compactions.

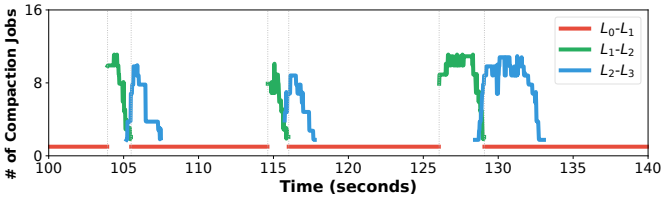


Fig. 5: Sequential execution of L_0–L_1 compaction and delayed upper-level compactions.

Fig. 5 illustrates this imbalanced utilization under the `fillrandom` workload. By concentrating compaction work into short time windows, this pattern reduces overall compaction efficiency, accumulates compaction debt, and ultimately triggers pending compaction stalls that further degrade PUT throughput.

B. Scalability Limits of L_0 Compaction Parallelism

A natural approach to rapidly reducing the number of L_0 SST files is to parallelize compaction. RocksDB supports compaction parallelism along two dimensions: (i) *intra-compaction parallelism*, which splits a single compaction task into multiple subtasks executed concurrently, and (ii) *inter-compaction parallelism*, which executes multiple independent compaction tasks in parallel. However, L_0–L_1 compaction is structurally constrained in both dimensions, regardless of the number of CPU cores or available threads.

1) Scalability Limits of Intra-Compaction Parallelism: As analyzed in Section IV-A, L_0–L_1 compaction involves a single, excessively large task. A natural way to accelerate such a task is to split it into multiple subtasks and execute them in parallel. Subcompaction in RocksDB [19] is a representative implementation of this approach.

Subcompaction partitions the global key range into P segments and assigns the merge-sort for each segment to one of P threads. However, because it fundamentally decomposes a single compaction task, all subtasks must complete to preserve compaction atomicity before output SST files can be committed. As a result, the slowest thread (the straggler) determines the overall compaction time. Effective parallelism therefore requires balanced workload distribution across threads. Our analysis shows, however, that achieving performance scaling proportional to the degree of parallelism via balanced partitioning is fundamentally infeasible.

> **Intra-Compaction Parallelism Paradox.** Effective parallel merge-sort requires evenly distributing key-value pairs across R segments. However, when S SST files have overlapping key ranges, no single file's metadata reveals how keys are distributed across the *global* key space. Determining the exact number of key-value pairs in each segment r requires performing binary searches across all S SST files to locate segment boundaries. These operations effectively reduce to the original problem, as they closely resemble the core steps of the merge-sort itself. In other words, the preprocessing needed to *enable* balanced parallelization effectively *performs* the very computation it aims to parallelize.

To circumvent this paradox, RocksDB's subcompaction adopts an approximate strategy. It sorts only the index keys recorded in the sparse index of each SST and uses an even partitioning of the sorted index-key array to define segment boundaries. However, because sparse indexes capture only per-SST key distributions, this approximation fails to accurately reflect the global key distribution across SST files.

Table I reports the execution-time variance of L_0-L_1 compaction under increasing subcompaction parallelism on the `fillrandom` workload (Section VI-B).

TABLE I: Execution-time variance of L_0–L_1 compaction tasks under increasing subcompaction parallelism.

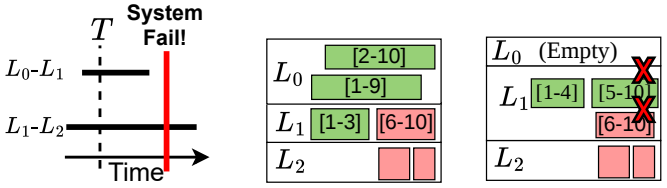
Threads (#)	**2**	**4**	**8**	**16**
Std. Dev. (s)	0.607	17.289	27.833	29.540

The results indicate that subcompaction-based parallelism fails to scale effectively beyond a certain level of parallelism. As the number of threads increases, the variance in the execution time of L_0-L_1 compaction tasks grows sharply, indicating that the parallel work is not evenly balanced across threads. Due to this imbalance, the overall compaction completion time becomes dominated by a small number of long-running tasks, and increasing subcompaction parallelism therefore fails to translate into sustained improvements in compaction throughput.

2) Scalability Limits of Inter-Compaction Parallelism: Inter-compaction parallelism executes multiple compaction tasks concurrently. It can parallelize compactions over disjoint SST groups with non-overlapping key ranges within the same level or execute compactions from different levels simultaneously. However, L_0–L_1 compaction is constrained in its degree of inter-compaction parallelism along two dimensions.

Constraint 1: No Parallel Execution across Multiple L_0–L_1 Compactions. In general, the key ranges of all SST files in L_0 overlap with those of all SST files in L_1, preventing L_0–L_1 compaction from being decomposed into multiple independent tasks based on key ranges. In higher levels, where SST files maintain non-overlapping key ranges, the system can identify disjoint compaction tasks within the same level and schedule them in parallel. This is not possible between L_0 and L_1, as all SST files overlap in key range.

Constraint 2: No Parallel Execution between L_0–L_1 and L_1–L_2 Compactions. Because all L_0 SST files overlap with all L_1 SST files, L_0–L_1 compaction cannot run concurrently with L_1–L_2 compaction. The root cause of this constraint lies in the consistency requirements of L_1.



(a) Compaction timeline (b) State at time T (c) State after recovery

Fig. 6: Inconsistency caused by concurrent L_0-L_1 and L_1-L_2 compactions under failure.

Fig. 6 illustrates why concurrent L_0–L_1 and L_1–L_2 compactions can violate LSM-tree consistency. Because all L_0 SST files overlap with all L_1 SST files, the two compactions inevitably share L_1 files as inputs, as shown in the timeline (a). If L_0–L_1 compaction commits while L_1–L_2 is still in progress (b), a subsequent system crash forces recovery to roll back the incomplete L_1–L_2 compaction, restoring its original L_1 inputs alongside the already-committed L_0–L_1 outputs. As a result, L_1 contains SST files with overlapping key ranges (c), violating the non-overlapping invariant required at L_1 and above.

To prevent this issue, RocksDB enforces a strict ordering in which L_0–L_1 compaction is scheduled only after L_1–L_2 compaction completes, structurally prohibiting parallel execution across levels.

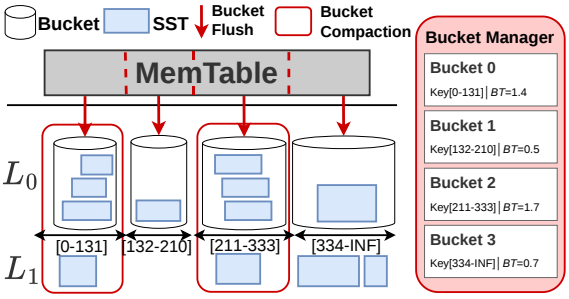


Fig. 7: Bucket-based data organization and compaction workflow in BucketLSM.

V. Design of BucketLSM

To eliminate the structural compaction bottleneck identified in Section IV, BucketLSM proposes a new LSM design that structurally expands L_0 compaction parallelism.

A. Design Overview of BucketLSM

Fig. 7 presents the overall architecture of BucketLSM. BucketLSM is an LSM design that partitions L_0 into multiple non-overlapping buckets. During a flush, the MemTable is split according to bucket boundaries and written into separate SST files, each belonging to a specific bucket. As a result, L_0 SST files are organized into bucket-partitioned files from their creation time.

To deliver high parallelism and stable performance with this bucket-based L_0 structure, BucketLSM incorporates three key mechanisms: (i) a scheduling policy that prioritizes which buckets to compact, (ii) a bucket management scheme that adjusts bucket boundaries in response to evolving key distributions, and (iii) stall policies that regulate read and write flows in accordance with the bucketed structure.

- **BucketCompaction-Scheduler** aims to stabilize PUT, GET, and SCAN latency by minimizing read latency and the risk of write stalls. Specifically, it prioritizes buckets with a larger number of L_0 SST files and, when buckets have the same file count, selects the bucket with a smaller compaction input size to reduce compaction latency.
- **BucketManager** adapts to skewed and evolving key-access distributions to mitigate load imbalance across buckets. By continuously monitoring the key distribution of PUT operations, it splits buckets experiencing concentrated writes and merges buckets with little write activity, sustaining compaction parallelism over time.
- **Bucket-Stall** redefines the L_0 stall condition to reflect the bucketized structure. While RocksDB triggers write stalls based on the total number of L_0 files, read performance in BucketLSM is governed by the per-bucket file count. Accordingly, Bucket-Stall controls stalls based on the maximum L_0 file count across buckets, ensuring stable read performance while avoiding unnecessary degradation of write throughput.

B. BucketCompaction-*Scheduler*

BucketCompaction treats buckets as independent units of L_0–L_1 compaction. Because an LSM-tree must concurrently perform flushes and upper-level compactions in addition to L_0–L_1 compaction, compaction must be scheduled under limited CPU cores to optimize PUT, GET, and SCAN throughput.

To this end, keeping the number of L_0 SST files low in each bucket is critical for reducing stall risk and preserving read performance. BucketCompaction-Scheduler therefore schedules compactions based on two priorities. First, it selects the bucket with the largest number of L_0 SST files, as such buckets increase read latency for their key ranges and raise the likelihood of L_0 stalls. Prioritizing these buckets quickly reduces their file counts.

Second, when multiple buckets have the same number of L_0 files, BucketCompaction-Scheduler prioritizes the bucket with the smaller BucketCompaction size. The size of a BucketCompaction is defined as the total size of the bucket's L_0 SST files and the overlapping L_1 SST files. Executing smaller, fine-grained BucketCompaction tasks first more rapidly reduces the number of buckets with high L_0 file counts, thereby mitigating GET and SCAN performance degradation.

Moreover, while smaller BucketCompaction tasks are being processed, L_1–L_2 compactions may concurrently occur on larger BucketCompaction candidates. Such upper-level compactions reduce the size of the overlapping L_1 input for other buckets, thereby shrinking the input sizes of the pending BucketCompaction candidates(especially the larger ones). As a result, scheduling BucketCompaction tasks in increasing order of size enables faster overall progress across multiple candidates with the same L_0 file count.

C. BucketManager

At initialization, BucketManager partitions the entire key space into N equal-sized buckets based on the key distribution of the first flushed MemTable. The initial bucket count N is a configurable parameter; by default, it is set to the number of background compaction threads allocated to the key-value store. However, realistic workloads exhibit skewed and time-varying key access patterns for PUT operations [2]. If bucket boundaries remain fixed at their initial key ranges, SST creation gradually concentrates on a subset of buckets, forming hot buckets, while other buckets receive few or no SST files and become cold.

Such load imbalance across buckets reduces parallelism. Hot buckets rapidly accumulate L_0 SST files and dominate compaction activity, while cold buckets have little or no compaction work. As a result, the number of compactions that can run concurrently falls below the total number of buckets, diminishing the parallelism benefits of the bucketized design.

Therefore, when imbalanced data ingestion across buckets is detected, bucket boundaries are dynamically rebalanced. The rebalancing decision is determined as follows. Table II summarizes the notation used in the formulation.

$$BT_i = \frac{F_W(B_i)}{F_{W(ideal)}} = \frac{N \cdot F_W(B_i)}{\sum_{j=1}^{N} F_W(B_j)} \tag{1}$$

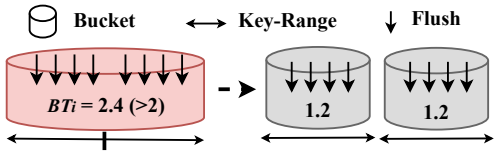


Fig. 8: Bucket split triggered by load imbalance. (Left: Before split, Right: After split)

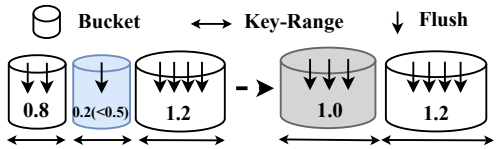


Fig. 9: Bucket merge triggered by low utilization (Left: Before merge, Right: After merge)

where

$$F_{W(ideal)} = \frac{\sum_{j=1}^{N} F_W(B_j)}{N}$$

In Equation 1, BT_i (Bucket Temperature) denotes the temperature of the i-th bucket and ranges from 0 to N. Under a perfectly uniform key distribution, each bucket receives an equal share of flush data, yielding $BT_i = 1$ for all buckets. Values significantly above 1 indicate hot buckets receiving disproportionate write traffic, while values near 0 indicate cold buckets.

BucketManager uses BT_i to decide whether to split (Fig. 8), merge (Fig. 9), or retain a bucket. Note that bucket split and merge operations are applied only to the target bucket that has been compacted and thus contain no remaining L_0 SST files.

1) Bucket Split: A bucket split divides a bucket B_i into two buckets and is triggered when $BT_i > 2$. During a split, BucketManager must determine how to partition the key range of the original bucket. To mitigate the impact of extreme key skew, it uses the median of the median keys of SST files flushed into the bucket during window W as the split boundary. After the split, the two resulting buckets each inherit half of the original bucket's flush volume, i.e., $F_W(B_i)/2$. Fig. 8 illustrates an example where a bucket with $BT_i = 2.4$ is split.

2) Bucket Merge: A bucket merge combines a bucket B_i with one of its adjacent buckets into a single bucket and is triggered when $BT_i \leq 0.5$. Specifically, B_i is merged into the colder of its two neighboring buckets, i.e., the bucket with the smaller BT. After the merge, both the key ranges and the flush volumes $F_W(B_i)$ of the two buckets are aggregated. Fig. 9 illustrates a case where a bucket with $BT_i = 0.2$ is merged into a neighboring bucket with $BT = 0.8$. Due to split and merge operations, the number of buckets may temporarily deviate from

TABLE II: Notation Used for Bucket Load Measurement

Symbol	Description
N	Total number of buckets
W	Sliding window covering the most recent W flush operations
F_W	Total amount of data flushed during window W
$F_W(B_i)$	Total size of SST files flushed into bucket B_i during window W

its initial value N. However, when splits increase the bucket count beyond N, the average BT_i decreases; conversely, when merges reduce the bucket count below N, the average BT_i increases. Through this feedback mechanism, as the workload's key distribution stabilizes, the number of buckets converges toward a balanced state where each bucket receives a roughly equal share of write traffic.

D. Redesigning L_0 Stall Condition for BucketLSM

BucketLSM flushes a MemTable into multiple SST files, resulting in a larger number of L_0 SST files than in conventional RocksDB. However, because these files are isolated by bucket with disjoint key ranges, GET and SCAN operations probe only the SST files belonging to the bucket that contains the target key. This structural change necessitates a redesign of the L_0 stall condition.

The purpose of an L_0 stall is to bound the number of SST files that must be probed during GET and SCAN, thereby preserving read performance. In RocksDB, where L_0 SST files have wide and overlapping key ranges, the stall condition is based on the total number of L_0 files, as all files may be involved in every read.

Applying this criterion directly to BucketLSM leads to two problems. First, even when SST files are evenly distributed across buckets, a stall may be unnecessarily triggered based solely on the total file count, degrading write throughput. Second, simply increasing the stall threshold fails to detect cases where SST files concentrate in a specific bucket, allowing read latency for that key range to degrade unchecked. Thus, the conventional stall policy cannot simultaneously guarantee read performance and write throughput.

To balance GET/SCAN latency and PUT throughput, BucketLSM triggers Bucket-Stall when the maximum number of L_0 SST files within any bucket exceeds a threshold. When SST files are evenly distributed across buckets, this policy avoids unnecessary stalls even with many total L_0 files, since each read probes only a small number of SST files. Conversely, when SST files concentrate on a specific key range, the per-bucket file count increases and the stall is triggered before read latency degrades.

As a result, by aligning the stall condition with the bucketed access pattern of GET and SCAN, BucketLSM maintains a balance between write throughput and read latency.

VI. Evaluation

A. Experimental Setup

We implemented BucketLSM on top of RocksDB v10.6.0. [1] The system setup for the experiments is shown in Table III.

To evaluate and analyze the effectiveness of BucketLSM, we use *db_bench* to run five write-intensive workloads that include GET and SCAN operations.

- **fillrandom.** A PUT-only workload provided by RocksDB that inserts 100GB of data.

[1]https://github.com/lass-lab/BucketLSM-CCGrid2026

TABLE III: Experimental Hardware and System Configuration

Component	Specification
CPU	Dual Intel Xeon Gold 5218R (2.10 GHz)
Cores / Threads	40 physical cores / 80 threads
Main Memory	384 GiB DDR4 DRAM
Storage	1 TB Samsung 990 PRO NVMe SSD
File System	F2FS
OS	Linux 6.8.0, Ubuntu 24.04.03 LTS

- **MixGraph1 [2].** A PUT-only MixGraph workload that inserts 100GB of data with time-varying key distributions. We adopt the Prefix_random configuration, which models key-range hotness through the prefix distribution while uniformly distributing keys within each key-range. We chose Prefix_random over Prefix_dist because the latter employs a power-law distribution for intra-range key selection, causing a small number of keys to be selected with high probability. While this accurately models read-dominant workloads where hot keys are repeatedly accessed, it is unsuitable for write-only evaluation: repeated writes to the same keys result in excessive in-place overwrites within the MemTable, preventing sufficient data accumulation to trigger flushes and compactions. With Prefix_random, keys are evenly spread within each key-range, ensuring that writes produce unique keys at a realistic rate while still preserving inter-range locality. The number of key ranges is increased sequentially from 10 to 50, resulting in keys being distributed across 10, 20, 30, 40, and 50 distinct key ranges over time.
- **MixGraph2.** A MixGraph workload executing PUT, GET, and SCAN operations at a ratio of 8:1:1. All parameters except the query mix are identical to MixGraph1.
- **YCSB-GET.** A YCSB workload executing PUT and GET operations at a 7:3 ratio under a Zipfian key distribution (Zipfian alpha = 0.9).
- **YCSB-SCAN.** A YCSB workload executing PUT and SCAN operations at a 9:1 ratio under a Zipfian key distribution (Zipfian alpha = 0.9).

For all workloads, we use 16B keys with a 1KB value size. The system is configured with two 256MB MemTables to buffer writes before flushing. To avoid compaction being limited by thread availability, the number of background threads is set equal to the number of available CPU hardware threads(80).

We compare BucketLSM against three baselines that represent odud75 compaction strategies:

- **Baseline.** RocksDB 10.6.0, without subcompaction. Note that RocksDB 10.6.0 adopts the idea of SILK [11]
- **Subcompaction [19].** A built-in RocksDB feature that parallelizes a compaction by splitting an SST file into subranges. For the throughput comparison in Section VI-C, the subcompaction degree is set to 8, which achieves the highest throughput in the scalability analysis of Section VI-B.
- **DownForce [13].** A recent non-blocking, pipelined compaction framework designed to enhance cross-level parallelism and address thread imbalance during multi-threaded

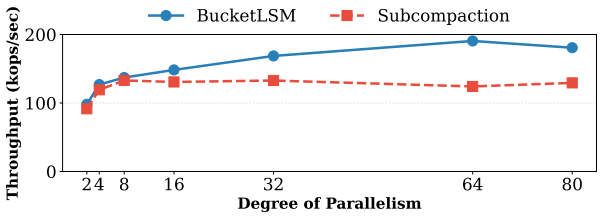


Fig. 10: Throughput comparison while increasing degree of L_0-L_1 compaction parallelism.

compaction.

We exclude ADOC [12] from the comparison because it continuously adjusts both background thread counts and MemTable/SST file sizes at runtime; these concurrent parameter changes make it difficult to isolate the effect of the compaction strategy itself across systems. We note that ADOC's dataflow-level tuning is orthogonal to BucketLSM's structural redesign and could be combined.

B. Scalability of L_0-L_1 Compaction Parallelism

Fig. 10 compares the throughput of subcompaction and BucketLSM as the degree of parallelism in L_0–L_1 compaction increases. Subcompaction parallelizes a single L_0–L_1 compaction by splitting it into multiple subtasks, whereas BucketLSM executes multiple bucket-level L_0–L_1 compactions concurrently as independent tasks.

Subcompaction exhibits limited scalability beyond eight threads. Throughput increases from degree 2 to 8, but then saturates and even degrades at degree 32 and above. This behavior is consistent with the analysis in Section IV-B1: load imbalance and straggler effects inherent to intra-compaction parallelism dominate the overall completion time of a compaction.

In contrast, BucketLSM demonstrates near-linear scalability over a wide range. Throughput steadily increases from degree 2 up to degree 64. This improvement stems from the fact that each L_0–L_1 compaction operates independently on a disjoint bucket key range, preventing delays in one compaction from stalling others.

Scalability in BucketLSM eventually saturates beyond degree 64. This is because the LSM-tree must concurrently perform flushes and upper-level compactions in addition to L_0–L_1 compactions. As the degree approaches near the number of physical cores (80), bucket compaction threads, flush threads, and L_1+ compaction threads become simultaneously active, exceeding available cores. The resulting context-switch overhead limits further performance gains. These results indicate that BucketLSM's structural decomposition of L_0-L_1 compaction improves throughput across a wide range of thread configurations. At low thread counts, the benefit comes from eliminating straggler effects and enabling incremental L_0 drain; at higher thread counts, BucketLSM additionally exploits the independent task structure to scale beyond the limits of intra-compaction parallelism. In practice, the degree of parallelism should be configured with awareness of competing system activities and storage device parallelism limits.

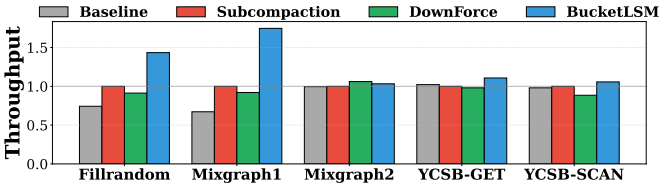


Fig. 11: Normalized throughput under various workloads.

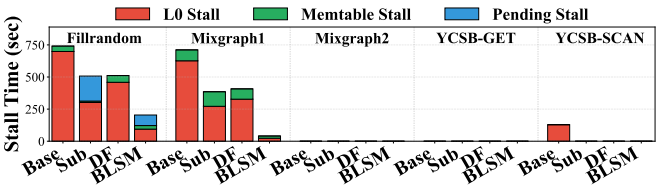


Fig. 12: Cumulative write stall time under various workloads. Base, Sub, DF, and BLSM denote Baseline, Subcompaction, DownForce, and BucketLSM, respectively.

C. Experimental Evaluation of Bucketized L0–L1 Compaction

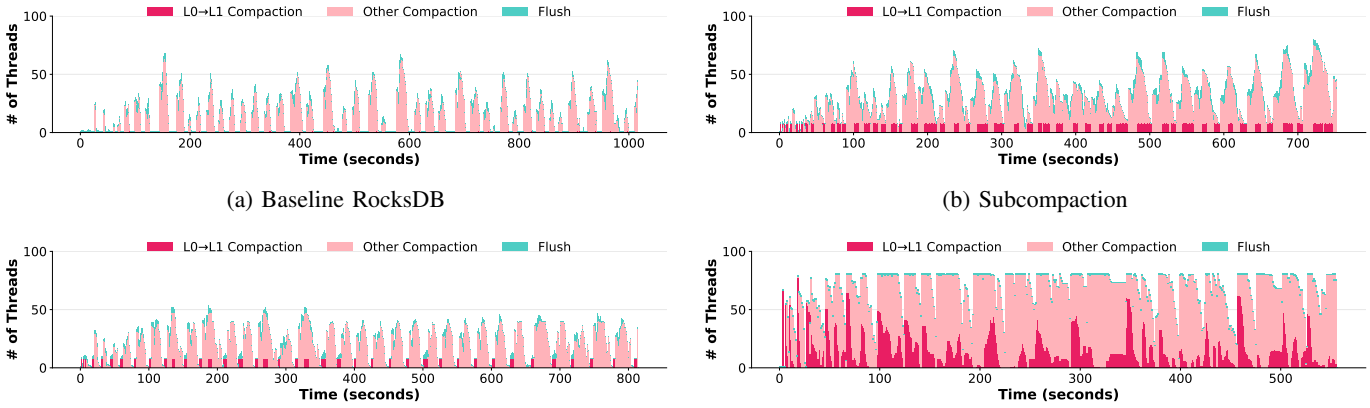
The goal of this section extends beyond a simple throughput comparison to analyze how different compaction parallelization strategies affect performance depending on workload characteristics and to identify which structural bottlenecks dominate in each environment.

Fig. 11 compares the normalized throughput of each system across the five workloads. To explain the observed throughput differences, Fig. 12 presents the cumulative write stall time under the same workloads, categorized into L0 stall, Memtable stall, and Pending stall.

1) Write-heavy Workloads: Fillrandom and Mixgraph1 are write-intensive workloads. In Fillrandom, BucketLSM improves throughput by 1.94× over Baseline and 1.43× over Subcompaction. The gains are even larger in Mixgraph1, reaching 2.60× over Baseline and 1.75× over Subcompaction.

These improvements are largely attributable to the reduction in write stalls, as shown in Fig. 12. The majority of this reduction stems from L_0 stalls: by decomposing L_0-L_1 compaction into fine-grained, independent bucket-level tasks, BucketLSM continuously drains L_0 SST files in parallel, preventing the accumulation that triggers stalls. This shows that each system's throughput is not simply determined by the number of available background threads, but rather dominated by how quickly and continuously L_0 SST files can be reduced under sustained write pressure. Subcompaction increases parallelism within a single compaction task, and DownForce improves parallelism through cross-level pipelining, but both techniques exhibit limitations in scalability due to the coarse-grained L_0-L_1 structure. In contrast, BucketLSM creates structurally independent compaction units from the flush stage, demonstrating that throughput can scale proportionally.

2) Read-mixed Workloads: In MixGraph2, all systems exhibit similar throughput. This is because the MixGraph benchmark processes requests in a single thread, reducing write ingestion rate when read operations are interspersed, and consequently compaction does not act as a performance



(a) Baseline RocksDB

(b) Subcompaction

(c) DownForce

(d) BucketLSM

Fig. 13: Thread utilization comparison: (a) Baseline RocksDB, (b) Subcompaction, (c) DownForce, and (d) BucketLSM.

bottleneck. Also, this confirms that BucketLSM's bucket-level mechanisms are lightweight and introduce nearly no additional overhead when compaction is not the bottleneck.

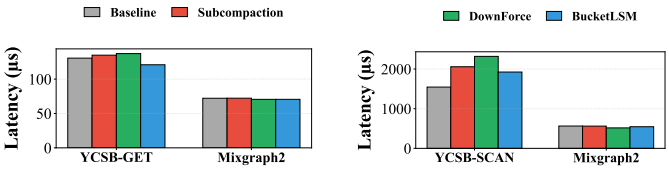
In contrast, YCSB-GET and YCSB-SCAN issue requests from multiple threads, sustaining high write pressure even in the presence of read operations. BucketLSM achieves 1.11× throughput improvement over Subcompaction in YCSB-GET and 1.05× in YCSB-SCAN. Unlike the write-heavy workloads, Fig. 12 shows that write stalls are negligible for most systems in these workloads, indicating that stall reduction alone does not account for the throughput gains. Instead, the improvements are primarily driven by BucketLSM's ability to keep per-bucket L_0 file counts low through continuous parallel compaction, which reduces the number of SST files that each GET or SCAN operation must probe. In a multi-threaded execution environment, lower per-operation latency allows each thread to complete requests faster and issue subsequent operations sooner, directly increasing overall throughput. A detailed latency analysis is presented in Section VI-D3.

D. Diagnosing Bottlenecks in Write-Heavy Workloads

This section identifies the causes of throughput improvement in write-heavy workloads through the analysis of thread utilization patterns and write stalls.

1) Thread Utilization: Figs. 13(a)– 13(d) show background thread utilization over time for the Fillrandom workload.

In Baseline (Fig. 13(a)), most background threads remain idle while a single L_0–L_1 compaction runs for an extended period, resulting in an execution time of approximately 1,000 seconds. Subcompaction (Fig. 13(b)) utilizes up to 20 threads during the L_0–L_1 phase, but exhibits an irregular pattern alternating between high and low utilization periods; execution time decreases to approximately 700 seconds, but this reflects the structural limitation that subcompaction still processes L_0–L_1 compaction as a single coarse-grained task. DownForce (Fig. 13(c)) achieves up to 60 threads through pipelining, but since L_0–L_1 compaction itself remains a single coarse-grained task, it cannot sustain thread utilization beyond a certain level, resulting in an execution time of approximately 800 seconds.



(a) GET latency

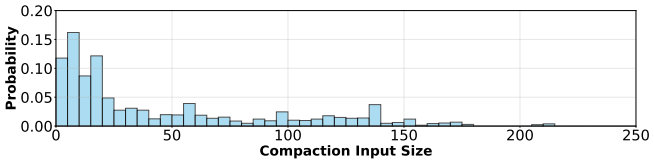
(b) SCAN latency

Fig. 14: Latency comparison for GET and SCAN workloads.

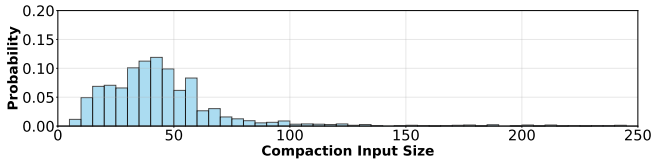
In contrast, BucketLSM (Fig. 13(d)) maintains consistently high thread utilization throughout execution. Because the compaction tasks generated from each bucket are fine-grained and independent, the next task is immediately scheduled upon completion of one task, minimizing idle periods. As a result, the total execution time is reduced to approximately 500 seconds, representing a 2× improvement over Baseline and 1.4× over Subcompaction. These results demonstrate that neither increasing parallelism within a single compaction task (Subcompaction) nor pipelining across levels while retaining the monolithic L_0–L_1 structure (DownForce) can fully overcome the thread utilization bottleneck. By decomposing L_0–L_1 compaction into structurally independent units at the flush stage, BucketLSM eliminates the root cause of idle periods and sustains high utilization throughout execution.

2) Write Stall Analysis: Fig. 12 shows the cumulative stall time for each workload. In Fillrandom, BucketLSM reduces total stall time by 73.3% compared to Baseline, 60.7% compared to Subcompaction, and 61.1% compared to DownForce. The majority of this improvement stems from reduced L_0 stalls. By partitioning L_0 into non-overlapping buckets, fine-grained parallel compaction proceeds continuously, preventing L_0 file accumulation.

In MixGraph1, BucketLSM reduces total stall time by 94% compared to Baseline, with L_0 stall time decreasing by 96%. These results demonstrate that BucketLSM's bucket-based L_0 structure, combined with BucketManager's dynamic bucket adjustment, enables compaction to be continuously performed in parallel with uniform granularity even in environments where key distributions change dynamically.



(a) L_0–L_1 compaction input size with static buckets



(b) L_0–L_1 compaction input size with dynamic buckets

Fig. 15: Comparison of L_0–L_1 compaction input size distributions: (a) static and (b) dynamic bucket boundaries.

3) GET/SCAN Latency: Fig. 14(a), 14(b) compare GET and SCAN latency across the YCSB-GET, YCSB-SCAN, and MixGraph2 workloads.

In YCSB-GET, BucketLSM achieves lower GET latency compared to Baseline. This is because bucket-level parallel compaction can rapidly reduce the number of L_0 SST files within each bucket, even under sustained write pressure that continuously generates L_0 SST files. In Baseline, the coarse-grained compaction targeting all of L_0 causes L_0 SST files to accumulate until compaction completes, requiring a single GET query to probe the indexes of all L_0 SST files. In contrast, BucketLSM partitions L_0 at the bucket level, and independent compactions are performed in parallel for each bucket, keeping the number of L_0 SST files within the bucket containing the target key consistently low. As a result, the impact of increasing L_0 file counts on GET latency is effectively mitigated.

In YCSB-SCAN, BucketLSM exhibits lower SCAN latency than Subcompaction and DownForce, but higher latency than Baseline. Subcompaction and DownForce maintain L_0-L_1 compaction as a coarse-grained single task, causing L_0 SST files to accumulate and forcing SCAN operations to probe many SST files per range query. BucketLSM reduces this probe cost by keeping per-bucket L_0 file counts consistently low through continuous parallel compaction.

However, BucketLSM's sustained parallel compaction activity generates continuous background device I/O, which interferes with foreground SCANs. Prior work has shown that such I/O interference from background operations degrades read latency [20]–[22]. Because Baseline's low compaction parallelism produces minimal background I/O (as reflected in the low thread utilization shown in Fig. 13(a)), its foreground SCANs experience less interference, achieving lower SCAN latency despite maintaining comparatively higher L_0 file counts.

In MixGraph2, low write pressure limits L_0 file accumulation itself, so compaction does not act as a performance bottleneck. In this case, similar latency is observed across all systems, demonstrating that BucketLSM's bucket-based structure does not introduce additional latency overhead even in situations where there is no benefit from compaction parallelism due to low write pressure.

Comparison Summary. The differences among the three parallelization approaches are summarized as follows. Subcompaction divides a single L_0–L_1 task into sub-tasks to exploit intra-compaction parallelism, but scalability is limited beyond 8 threads due to the load imbalance and straggler problems analyzed in Section IV-B. DownForce improves inter-level parallelism through cross-level pipelining, but L_0–L_1 compaction itself remains a coarse-grained single task, and the temporarily permitted intra-level overlap increases subsequent compaction burden. BucketLSM creates structurally independent compaction units from the flush stage and continuously distributes them through scheduling, thereby simultaneously overcoming the limitations of both intra-compaction and inter-compaction parallelism.

E. Effectiveness of Dynamic Bucket Rebalancing

This section analyzes how BucketManager's dynamic bucket boundary adjustment adapts to time-varying key distributions and maintains balanced compaction workloads, and quantifies its overhead. To this end, we compare the L_0–L_1 compaction input size distributions between static and dynamic bucket configurations under the MixGraph1 workload (Section VI-A), which exhibits dynamically shifting key distributions. Fig. 15(a), 15(b) show the probability distributions of L_0–L_1 compaction input sizes for static and dynamic bucket configurations, respectively. In the static bucket configuration, compaction input sizes are widely dispersed, ranging from very small values to over 180MB. Notably, a bimodal distribution is observed, with high frequencies in both the small and very large size ranges. This distribution indicates that as key distributions shift over time, some buckets become hot buckets where large amounts of data concentrate, while other buckets remain cold and receive little to no data. Hot buckets rapidly accumulate SST files, triggering large-scale compactions, whereas cold buckets have minimal compaction work, resulting in only occasional small compactions. This load imbalance across buckets degrades parallelism. While compaction work concentrates on hot buckets, cold buckets have no compaction tasks to execute, reducing the number of concurrent compactions below the total bucket count. Consequently, the effective parallelism degree of the bucket-based structure diminishes.

In contrast, the dynamic bucket configuration shows compaction input sizes concentrated in the mid-range, with significantly reduced frequencies of extremely small or large compactions. This is the result of BucketManager using the BT (Bucket Temperature) metric defined in Section V-C to split hot buckets and merge cold buckets, thereby equalizing load across buckets. When a hot bucket is detected, it is split into two buckets to distribute the write load; cold buckets are merged with adjacent buckets to consolidate unnecessarily fragmented compaction units. Through this equalization, compaction workloads become balanced across buckets, enabling more buckets to perform compactions simultaneously and thus increasing the performance gains from parallelism. In terms of end-to-

end performance, the dynamic bucket configuration achieves 1.12× higher throughput than the static configuration under MixGraph1, confirming that balanced compaction granularity translates directly into improved system throughput.

To verify that dynamic rebalancing does not introduce measurable overhead, we compare the static and dynamic configurations under the fillrandom workload (Section VI-A), which exhibits a uniform key distribution. Under this workload, bucket boundaries remain stable because no bucket exceeds the split or merge thresholds, and both configurations achieve equivalent throughput. This result is expected given the lightweight nature of the rebalancing logic: BucketManager maintains per-bucket metadata entirely in memory, and the BT computation involves only addition, subtraction, and shift operations on flush counters. The per-bucket memory footprint is 240 bytes, making the total memory overhead negligible even with hundreds of buckets. These results confirm that dynamic rebalancing incurs no additional cost even when boundaries are already well-aligned, while providing substantial gains when workload skew demands adaptation.

VII. Conclusion

This paper identifies the compaction bottleneck in LSM-tree–based key–value stores as a structural consequence of allowing global key-range overlap at L_0, which forces L_0–L_1 compaction to execute as a single coarse-grained task and fundamentally limits parallel scalability. To address this limitation, BucketLSM organizes L_0 SST files into non-overlapping buckets at flush time, creating structurally independent compaction units. This design enables scalable inter-compaction parallelism, significantly improves throughput, reduces write stalls, and preserves read performance even under high write pressure.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-00564249), by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02215256), and by NSF-OAC-2311758.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[2] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '20, 2020.

[3] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured Key-Value stores," in *Proceedings of the USENIX Annual Technical Conference*, ATC '17, 2017.

[4] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0. 8.13," 2018.

[5] W. Zhang, E. Xu, Q. Wang, X. Zhang, Y. Gu, Z. Lu, T. Ouyang, G. Dai, W. Peng, Z. Xu, *et al.*, "What's the story in EBS glory: Evolutions and lessons in building cloud block store," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '24, 2024.

[6] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham, "mLSM: Making authenticated storage faster in ethereum," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '18, 2018.

[7] Meta Platforms, Inc., "Rocksdb: A persistent key-value store for fast storage environments." https://rocksdb.org, 2012. Accessed: Dec. 10, 2025.

[8] Google, "LevelDB: A Fast Key-Value Storage Library." https://github.com/google/leveldb, 2011. Accessed: Dec. 10, 2025.

[9] Y. Matsunobu, S. Dong, and H. Lee, "Myrocks: Lsm-tree database storage engine serving facebook's social graph," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.

[10] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the International Conference on Management of Data*, ICMD '21, 2021.

[11] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in Log-Structured merge Key-Value stores," in *Proceedings of the USENIX Annual Technical Conference*, ATC '19, 2019.

[12] J. Yu, S. H. Noh, Y.-r. Choi, and C. J. Xue, "ADOC: Automatically harmonizing dataflow between components in Log-Structured Key-Value stores for improved performance," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '23, 2023.

[13] H. Byun, H. Yoo, and S. Park, "Revisiting multi-threaded compaction in lsm-trees: Enabling compaction pipelining," in *Proceedings of the 54rd International Conference on Parallel Processing*, ICPP '25, 2025.

[14] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the Symposium on Operating Systems Principles*, SOSP '17, 2017.

[15] H. Zhou, Y. Chen, W. Zeng, L. Cui, G. Wang, and X. Liu, "Gpcomp: Using gpu and ssd-gpu peer to peer dma to accelerate lsm-tree compaction for key-value store," *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[16] C. Ding, J. Zhou, J. Wan, Y. Xiong, S. Li, S. Chen, H. Liu, L. Tang, L. Zhan, K. Lu, *et al.*, "Dcomp: Efficient offload of lsm-tree compaction with data processing units," in *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, 2023.

[17] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM," in *Proceedings of the USENIX Annual Technical Conference*, ATC '20, 2020.

[18] K. Kim, H. Chung, S. Ahn, J. Park, S. Jamil, H. Byun, M. Lee, J. Choi, and Y. Kim, "KVACCEL: A Novel Write Accelerator for LSM-Tree-Based KV Stores with Host-SSD Collaboration," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '25, 2025.

[19] RocksDB, "RocksDB Subcompaction Wiki." https://github.com/facebook/rocksdb/wiki/Subcompaction, 2022. Accessed: Dec. 10, 2025.

[20] H. Litz, J. Gonzalez, A. Klimovic, and C. Kozyrakis, "RAIL: Predictable, low tail latency for NVMe flash," *ACM Transactions on Storage*, vol. 18, no. 1, pp. 1–21, 2022.

[21] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of Ultra-Low latency solid state drives," in *10th USENIX Workshop on Hot Topics in Storage and File Systems*, Hotstorage '18, 2018.

[22] S. Yoo, H. Shin, S. Lee, and J. Choi, "A read performance analysis with storage hierarchy in modern KVS: A RocksDB case," in *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium*, NVMSA '22, 2022.