Increase Merge Efficiency in LSM Trees Through Coordinated Partitioning of Sorted Runs

Qizhong Mao*, Vagelis Hristidis†
Computer Science and Engineering, University of California, Riverside, USA
Email: *qmao002@ucr.edu, †vagelis@cs.ucr.edu

***Abstract*—The performance of an LSM-tree-based system heavily relies on the compaction strategy employed. Two main categories of compaction strategies exist: leveled and stack-based. Leveled compaction offers several advantages. Firstly, its incremental merge style enables breaking down large compactions into smaller sub-compactions through partitioning. This partitioning enhances parallelism during compaction execution, reduces write stalling, and improves disk utilization. Additionally, for specific workloads like sequential insertions, it allows moving entire files to lower levels without the need for rewriting them, thus saving disk I/O. These moves are known as trivial-moves. On the other hand, stack-based policies typically lack support for these desired properties. Their large compactions either perform no partitioning or rely on naive partitioning methods, resulting in limited opportunities for parallelism and trivial-moves.**

The goal of this paper is to facilitate the compaction advantages of leveled strategies in stack-based systems, hence creating a hybrid strategy that combines the advantages of both worlds. To achieve this, we propose two novel coordinated partitioning algorithms, namely *Local-Range* and *Global-Range*. These algorithms can be applied to any stack-based compaction strategy to enhance parallelism during compactions and create more opportunities for trivial-moves, resulting in improved overall compaction cost. We extend RocksDB to support partitioning on stack-based strategies and conduct a comparative analysis against several baselines using various workloads. The experimental results demonstrate that the Global-Range partitioning method significantly enhances compaction performance with minimal overhead.

***Index Terms*—LSM, compaction, merge, stack-based, leveled, partition**

I. Introduction

Many existing LSM systems fall into two categories: stack-based and leveled. Examples of the former include AsterixDB [5], Bigtable [1], Cassandra [2], and HBase [3]. Stack-based compaction strategies (defined in [15]) excel for high ingestion rate scenarios. They generate fewer but larger files, simplifying resource management by single-threaded compactions without excessive file handlers in memory. However, these strategies suffer drawbacks: 1) Long compaction times can halt data ingestion (a.k.a. *write stalls*); 2) Low disk utilization due to demanding temporary disk space; 3) Canceling compactions is costly, often discarding merged data.

Leveled compaction is employed in LevelDB [4] and RocksDB [6], utilizing partitioned compaction to maintain small files. Data is incrementally merged across sorted runs (levels), favoring multiple quick small compactions over larger ones. This approach enables parallel compaction execution, enhancing system flexibility. Leveled compactions easily accommodate *trivial moves*, optimizing cases like ascending keys. Disk utilization tends to be higher due to smaller compactions not requiring extensive temporary space. Abort costs are lower compared to stack-based compactions. Some workloads might avoid rewriting files during compaction. Stack-based LSM excels in write performance, while leveled LSM excels in read performance and disk use. Previous research [15] explored Binomial and MinLatency stack-based methods, achieving optimal write efficiency while minimizing read amplification.

In this paper, we highlight the benefits of leveled policies integrated with stack-based policies. We achieve heightened compaction parallelism and enable trivial-moves through effective compaction partitioning algorithms. Clever partitioning of records enables parallelization of large compactions for enhanced write performance, or serialization to improve disk use. Trivial-moves support additionally reduces disk I/O for specific workloads.

Partitioning data across components in stack-based compaction presents challenges. We aim for minimizing overlaps between partitions in different components to avoid the *chaining problem* (Section III-B). However, components must independently partition, unaware of other components' boundaries. An efficient partitioning algorithm must minimize overhead in decision-making and overlap detection.

We propose a suite of coordinated partitioning algorithms for aligned partition boundaries across sorted runs, fostering higher parallelism. The Local-Range and Global-Range algorithms recursively split the key space in halves until partitions contain the desired record count. Local-Range bases key space on the smallest and largest actual keys in a component, while Global-Range employs the smallest and largest possible values for the entire database.

We assess these partitioning techniques using RocksDB within stack-based strategies. Our approaches are contrasted with size-based benchmarks, which establish new partitions upon reaching a maximum data size. Notably, Global-Range demonstrates better write efficiency.

In summary, this work offers the following contributions:

1) We introduce the challenge of enhancing parallelism in stack-based compactions. We specifically identify the *chaining problem* that limits the effectiveness of size-based partitioning. (Section III)

2) We propose two coordinated partitioning approaches: Local-Range and Global-Range partitioning. These methods effectively address the chaining problem and enable multiple parallel sub-compactions and trivial-moves. (Section IV)
3) We implement the suggested partitioning algorithms within RocksDB and assess their write and read performance in comparison to No-partitioning and Size partitioning using RocksDB's UniversalCompaction.
4) We conduct extensive experiments on synthetic datasets, demonstrating the superiority of our proposed methods over existing baselines. (Section V)

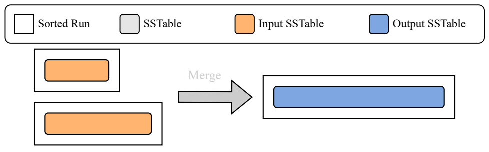
The remainder of the paper is structured as follows. Section II provides essential background information on LSM tree compaction, while Section VI delves into the outcomes and insights gained. Section VII explores related work, and we conclude in Section VIII.

II. Background

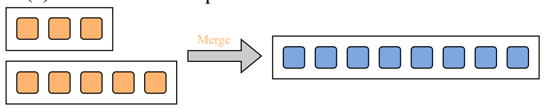
An LSM tree comprises memory and disk components, with overlapping key ranges in disk components, potentially requiring searching through all disk components for point queries. A disk component, also known as a *Sorted Run*, has strictly ordered keys and can be implemented as one or multiple *Sorted-String-Table*s (SSTables), based on the compaction strategy. Our previous work [15] compared stack-based strategies to the leveled strategy, examining write, read, and transient space trade-offs. The following sections provide concise summaries of these compaction types and their differences.

A. Stack-based Compaction

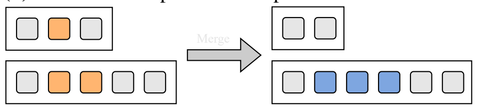
In a typical stack-based LSM tree, each sorted run corresponds to a single SSTable, and stack-based compactions (a.k.a. *full merge* [11]) combine multiple sorted runs into one (Figure 1a). Large SSTables can result from stack-based compactions, with better write performance than leveled compactions. However, they exhibit lower disk and CPU utilization due to limited parallelization.



(a) Stack-based compaction of 2 sorted runs / SSTables.



(b) UniversalCompaction of 2 partitioned sorted runs.



(c) Leveled compaction of 3 SSTables between 2 sorted runs.

Fig. 1: Stack-based compaction v.s. Leveled compaction.

In stack-based compactions, merged SSTables generally overlap, which is common for random key insertion. However, certain workloads with sorted keys, like time-correlated or bulk-loaded data, might not require compaction. Traditional stack-based compactions overlook this and merge already sorted data, causing unnecessary disk I/O. *UniversalCompaction* [7, §Universal Compaction] (Figure 1b) addresses this with limited support for partitioning, though using suboptimal size-based partitioning.

B. Partitioned Leveled Compaction

Leveled compaction in LevelDB [4] and RocksDB [6] divides large sorted runs into smaller, non-overlapping SSTables (Figure 1c). A sorted run is a list of SSTables with sorted, non-overlapping key ranges, maintaining $l_i \leq u_i < l_{i+1} \leq u_{i+1}$ for i-th SSTable key ranges. Leveled compaction (a.k.a. *partial merge* [11]) selects only some SSTables in a run to keep each compaction small. Typically, a maximum SSTable size limit θ is set, creating $\lceil \frac{S}{\theta} \rceil$ SSTables with size S, except for the last. The partitioning, which is size-based, is straightforward with minimal computation overhead and has proven to be efficient, hence its widespread adoption in LevelDB and RocksDB.

Both LevelDB and RocksDB support *trivial-move* operations [7, §Compaction Trivial Move], where non-overlapping SSTables are directly moved, minimizing I/O. Sequential workloads benefit from this in fast compactions.

III. Problem Definition

A. Motivation: Expensive Compactions

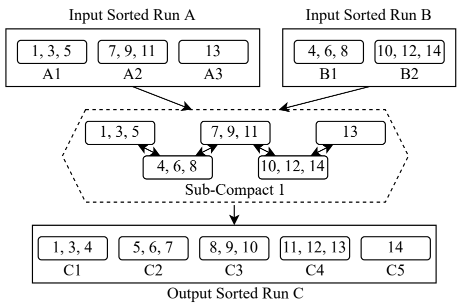
While most stack-based compaction strategies outperform the partitioned leveled compaction approach in writes [15], they face the following issues:

- Large compactions tie up substantial CPU, memory, and disk resources for extended periods, often leading to *write stalls* that hinder or halt new incoming writes.
- Compactions are restricted to single-thread execution because of the requirement for a single output SSTable. This might not fully utilize advanced hardware such as SSDs to achieve enhanced write throughput. The chaining problem described also limits parallel thread compaction.
- For specific workloads, such as sequential data ingestion with monotonically increasing keys, partitioned leveled compaction can move files without rewriting, conserving substantial disk I/O. However, typical stack-based strategies do not capitalize on trivial-moves, as they usually rewrite existing SSTables.

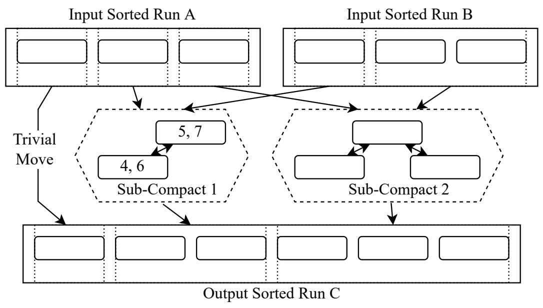
Addressing these issues involves partitioning sorted runs in stack-based LSM trees. Instead of one SSTable per sorted run, each run can be divided into one or more separate SSTables, akin to the partitioned leveled LSM tree. With a well-designed partitioning algorithm, larger compactions can be divided into smaller sub-compactions. Each sub-compaction processes only a subset of input sorted run SSTables. When multiple sub-compactions are scheduled in one compaction, they can be executed concurrently by multiple threads, enhancing overall write throughput and reducing compaction duration. Additionally, non-overlapping SSTables within a compaction can be moved trivially to the output sorted run, conserving disk I/O and further boosting write performance.

B. Chaining Problem in Parallel Compactions

The size-based partitioning method can be easily ported for stack-based LSM trees. However, this approach often leads to the chaining problem, where most or all input SSTables need to be grouped in a single sub-compaction. In Figure 2, when merging sorted runs A and B with keys $\{1, 3, 5, 7, 9, 11, 13\}$ and $\{4, 6, 8, 10, 12, 14\}$, respectively, and a maximum SSTable size limit ($\theta = 3$), size-based partitioning allows only one sub-compaction due to the inability to split the components into disjoint groups. Conversely, Global-Range partitioning (explained in Section IV) enables efficient scheduling of two sub-compactions and one trivial-move for enhanced compaction of these sorted runs.



(a) Size Partitioning



(b) Global-Range Partitioning

Fig. 2: Size partitioning v.s. Global-Range partitioning.

C. Trivial-Move: I/O Efficiency

RocksDB's UniversalCompaction allows trivial moves under strict conditions. A trivial move is feasible when a compaction selects only one SSTable, and it does not overlap with any other in the output sorted run [7, §Compaction Trivial Move]. This often means all level 0 SSTables do not overlap [7, §Universal Compaction]. Due to stack-based compaction's merging of consecutive SSTables, an SSTable will merge if between two overlapping SSTables. For instance, merging SSTables with key ranges $[1, 10]$, $[21, 30]$, and $[6, 15]$ mandates the second SSTable's merge despite non-overlap.

IV. Proposed Local-Range and Global-Range Partitioning

In this section, we extend stack-based compaction with partitioning (Section IV-A). An overview of Local-Range and Global-Range partitioning is in Section IV-B. Importantly, proposed partitioning does not affect compaction's sorted runs selection, hence minimal impact on read and write amplification. It could also reduce transient space amplification by merging individual partitions.

A. Partitioned Stack-based Compaction

A leveled compaction involves either a trivial-move with one SSTable or sub-compactions with multiple overlapping SSTables. In a stack-based compaction, SSTables that do not overlap within the same compaction can be present due to the fact that all SSTable in the input sorted runs must be selected. To enhance stack-based compaction efficiency, supporting trivial-moves within compaction (instead of rewriting all SSTables) is beneficial. In a partitioned stack-based compaction, input SSTables are grouped to avoid overlap. Each group undergoes either a trivial-move (if only one SSTable) or a sub-compaction.

By analyzing input SSTable metadata, we create non-overlapping groups for parallel execution of trivial-moves or sub-compactions. Parallel execution via multi-threaded sub-compactions boosts write throughput but requires more CPU, memory, and disk usage, which depends on hardware. To mitigate chaining issues outlined in Section III-B, we introduce *Local-Range* and *Global-Range* partitioning algorithms. These algorithms partition SSTables according to predefined key ranges, effectively generating more sub-compactions.

B. Local-Range and Global-Range Partitioning

Just like k-d trees [10], Local-Range and Global-Range partitioning utilize binary space partitioning in the one-dimensional key space. The distinction lies in how they define the key space for binary splitting: Local-Range employs the key range of SSTables to be merged, while Global-Range uses the entire key space. Both algorithms iteratively partition keys until all partitions are smaller than θ.

Figure 3 shows the partitioning algorithms (Size, Local-Range, Global-Range, No-partitioning) applied to the same sorted run. Local-Range partitioning divides the key range [4, 254] at mean $\lfloor \frac{4+254}{2} \rfloor = 129$, then further splits left and right partitions at $\lfloor \frac{4+129}{2} \rfloor = 66$ and $\lfloor \frac{129+254}{2} \rfloor = 191$. Global-Range partitioning's first split is at mean of key space [0, 255], always $\lfloor \frac{0+255}{2} \rfloor = 127$, followed by splits at $\lfloor \frac{0+127}{2} \rfloor = 63$ and $\lfloor \frac{128+255}{2} \rfloor = 191$. Both algorithms halt when all partitions are no larger than 10.

When comparing these algorithms: No-partitioning results in a single partition covering most of the key space. Size partitioning forms partitions with comparable file sizes, irrespective of keys. Local-Range partitioning ensures similar key span sizes among SSTables. Global-Range partitioning aligns SSTables within fixed bounds, always 2^b wide in a key space of size 2^B, where $0 \leq b \leq B$.

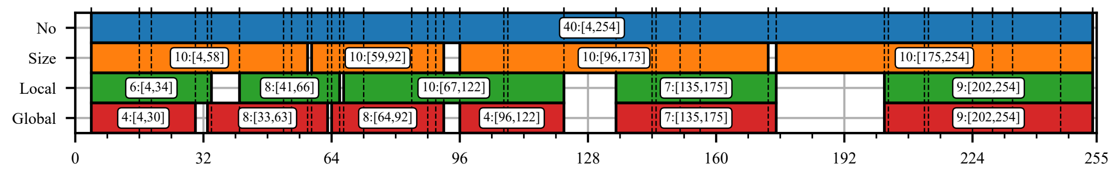


Fig. 3: Overview of No-partitioning, Size, Local-Range partitioning and Global-Range partitioning on a sorted run of 40 8-bit unsigned integer keys. Maximum SSTable size limit $\theta = 10$. SSTable size as the number of keys, the smallest key and the largest key are annotated in each rectangle representing an SSTable. Actual keys are shown in dashed vertical lines.

When merging multiple partitioned sorted runs, Global-Range partitioning often generates more sub-compactions due to shared boundaries. Figure 4 illustrates a randomly generated example of partitioned stack-based compactions involving three sorted runs. Size and Local-Range partitioning result in a single sub-compaction each, whereas Global-Range partitioning yields three sub-compactions and one trivial-move.

While Global-Range partitioning enhances parallelism through multiple sub-compactions, it often generates more SSTables. In the provided example, Size partitioning results in 10 SSTables post-compaction, while Local-Range and Global-Range partitioning yield 16 and 15 respectively. In practice, both Local-Range and Global-Range produce 1.5 to 2 times more SSTables compared to Size partitioning with the same θ. A surplus of SSTables can elevate maintenance overhead and potentially hinder range query performance due to increased file scanning requirements.

V. Experiments

A. Experimental Setup

To ensure fairness, we selected RocksDB as the testbed for comparing partitioned stack-based and leveled compactions. RocksDB is the sole key-value store engine supporting both compaction types. We incorporated the proposed partitioning algorithms for UniversalCompaction in RocksDB. To prevent redundant iterations of all keys during the partitioning phase of compaction, we have implemented a dedicated lightweighted histogram to estimate the distributions of the merging keys.

We compare four stack-based partitioning algorithms (No-partition, Size, Local-Range, and Global-Range) with the default Leveled compaction. To evaluate write and read performance, we conduct two sets of experiments covering *unique-random* and *zipf* distributuions over *UInt* keys: with *unique-random* distribution, keys are random with no duplicates; with *zipf* distribution, keys are skewed using a Zipfian distribution with $q = 0.5$. In both sets, `uint64_t` numbers in big-endian binary format are used, values are random 1016-byte byte strings, and each flushed SSTable's key range could span the entire key space.

Both sets consist of loading 41,943,040 key-value pairs during a loading phase (approx. 40 GB). Subsequently, 100,000 point queries (`Get`) and 10,000 range queries (`Iterator`) of size 1,000 are performed. Additionally, 10,000 keys pre-warm the table cache before executing any point or range queries.

The tests are performed on an AWS EC2 *c5.4xlarge* instance with 16 CPU cores and 32 GB memory. All RocksDB options are kept at default, except *max_subcompactions*, which is set to 16 to match the available CPU cores. Leveled compaction uses Size partitioning by default.

B. Experimental Results with UInt Keys

Random Key Distribution: In a uniform distribution of keys, sorted runs span the entire key space $[0, 2^{64} - 1]$ uniformly. Global-Range partitioning outperforms No-partitioning and Size partitioning by 8% and 13% respectively (Figure 5). Size partitioning is slower than No-partitioning as it can't schedule sub-compactions and involves additional overhead from scanning multiple SSTables due to the chaining issue.

In this key distribution, keys exhibit greater uniformity, favoring Global-Range partitioning's binary-split function in generating partitions with closely aligned key range and data sizes. Consequently, while Global-Range partitioning produces slightly more SSTables than Size partitioning—by just 8% (Figure 6)—Local-Range partitioning generates 70% more SSTables than Size partitioning, resulting in elevated resource management overhead.

Global-Range and Local-Range partitioning yield the highest and second-highest compaction throughput, respectively (Figure 7). These algorithms optimize multithreading for merging large data sets. In contrast, Size partitioning and No-partitioning show comparable behavior, with Size partitioning unable to coordinate sub-compactions. Leveled compaction demonstrates superior multi-threaded throughput but records the lowest overall compaction throughput due to elevated write amplification and subsequent data merging.

Despite the increased number of SSTables, Global-Range partitioning maintains superior performance in point query and range query (Figure 8). Notably, Leveled compaction excels in read performance due to fewer sorted runs resulting from increased data merging.

Among these distribution types, Global-Range partitioning excels. It delivers superior write performance, the highest compaction throughput, and reduced read query latencies, albeit with a mere 8% increase in SSTables to manage.

Skewed (Zipf) Key Distribution: The performance in write, compaction, and read is comparable to the random key distribution, as shown in Figures 9, 10, and 11. No-partitioning has slightly lower range query latency due to reduced file scanning. Remarkably, Local-Range partitioning decreases SSTables by 13%, while Global-Range partitioning increases them by 38% compared to the random key distribution (Figure 12). Overall,

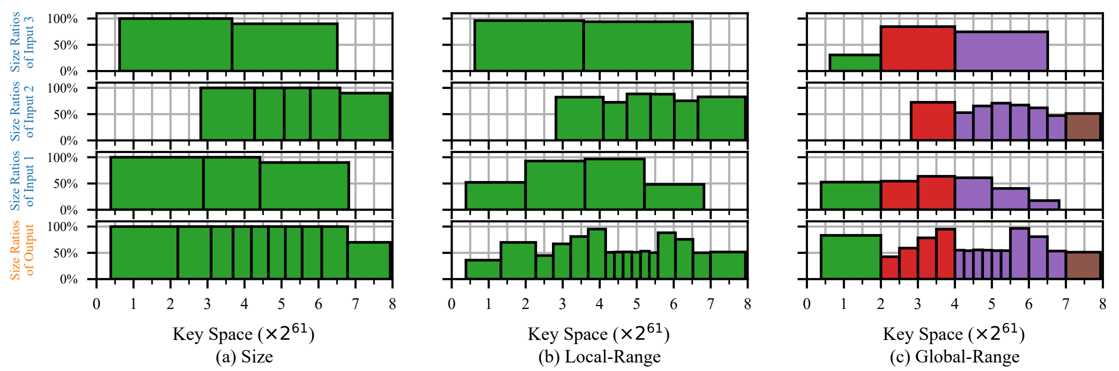


Fig. 4: Examples of merging with different partitioning algorithms. Top three rows show SSTable size ratios relative to θ in three input sorted runs. The bottom row displays size ratios of the output sorted run from merging these inputs. Same-colored SSTables form sub-compactions. Keys are consistent across rows. Sorted runs 1–3 originate from prior flushes and compactions.

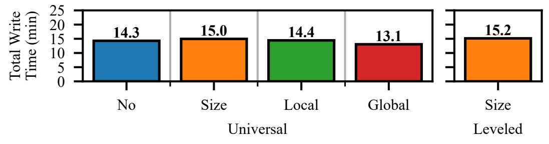


Fig. 5: Random UInt: Total write time.

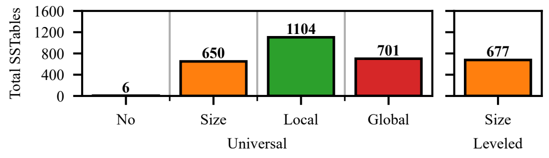


Fig. 6: Random UInt: SSTable statistics.

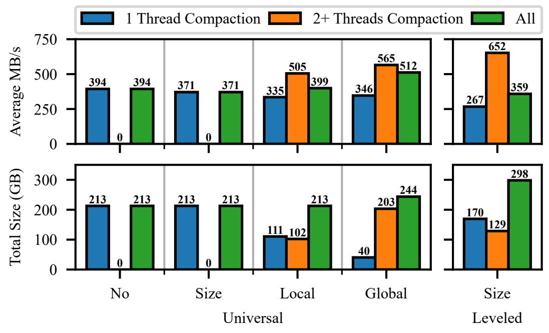


Fig. 7: Random UInt: Compaction statistics.

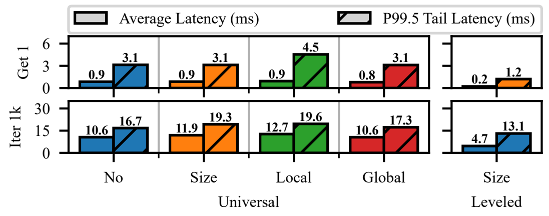


Fig. 8: Random UInt: Read query latency.

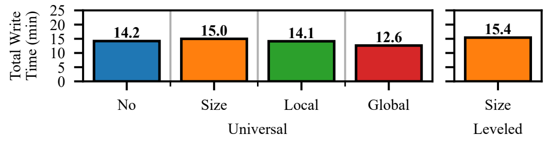


Fig. 9: Skewed UInt: Total write time.

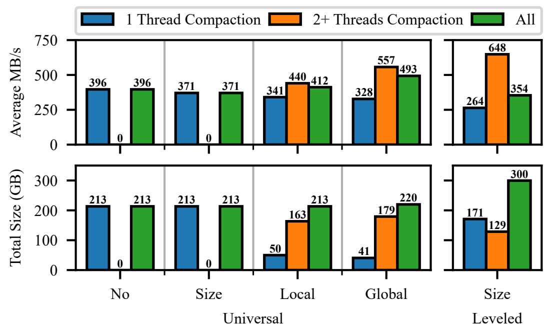


Fig. 10: Skewed UInt: Compaction statistics.

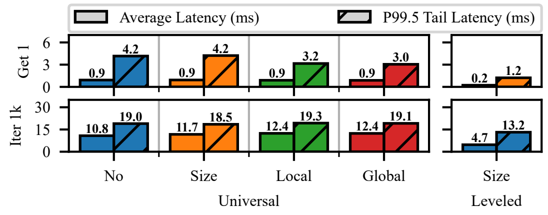


Fig. 11: Skewed UInt: Read query latency.

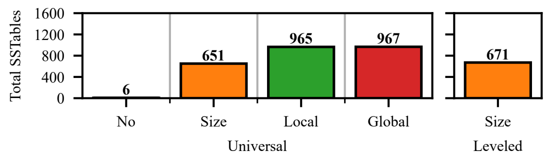


Fig. 12: Skewed UInt: SSTable statistics.

Global-Range partitioning excels in write performance, with a minor trade-off in read performance and increased SSTables.

VI. Discussion

The proposed Local-Range and Global-Range partitioning algorithms effectively increase LSM compaction parallelism in most tested workloads. Global-Range partitioning achieves the highest write throughput and compaction throughput, but can result in more SSTables and degrade range query performance. Local-Range partitioning also achieves high parallelism, creates fewer SSTables, and is suitable for systems with many range queries. To reduce the number of SSTables, small partitions can be grouped together or the maximum SSTable size limit can be increased. For sequential or semi-sequential key workloads, Size partitioning is recommended due to its minimal SSTables, memory, and disk overhead.

A. Limitations and Future Work

Dynamic resource allocation: When using a large number of threads for compactions, the CPU can become a bottleneck, resulting in reduced write throughput. In our work, we address this issue by setting a hard limit on the maximum number of sub-compactions through an option. However, it would be beneficial to have an algorithm that dynamically adjusts sub-compactions based on the number of available CPU cores. Additionally, remote compaction techniques such as those discussed in [8], [13] can significantly alleviate CPU pressure, particularly in shared storage systems.

Hybrid stack-based and leveled compaction strategy: The proposed partitioning algorithms can be applied to leveled compactions, resulting in hybrid stack-based and leveled compaction strategies. This approach preserves the benefits of both stack-based and leveled compactions, ensuring high write throughput, low read latencies, and efficient disk utilization. Additionally, these partitioning algorithms can also be explored for our previously proposed Binomial compaction strategy, which offers comparable read performance to leveled compaction while improving write performance.

VII. Related Work

Partitioning Algorithms: Various partitioning algorithms have been proposed for LevelDB and RocksDB, primarily focusing on the leveled compaction policy. Some of these algorithms can also be applied to stack-based policies. For instance, the partitioning algorithm introduced in PE File [12] is application-dependent and utilizes a binary split at the median of keys, resembling a specialized version of Size partitioning. Zhang et al. [21] and SifrDB [16] adopt a similar partitioned stack-based design as the Size partitioning. Another approach is the LWC-tree [20], [19], which uses vertical grouping for partitioning, similar to the one used in Leveled compaction. This technique also facilitates load balancing across database nodes. PebblesDB [17] employs range-based partitioning, where a randomly chosen key serves as a guard for each range in a level. WB-tree [9], LSM-trie [18], and HashKV [14] employ hash-based partitioning to evenly distribute keys into SSTables, especially for workloads that do not require range query support. It is important to note that most of these approaches are not applicable to stack-based compaction strategies, have limited support for range queries, and may encounter the chaining issues.

VIII. Conclusions

In this work, we introduce two partitioning algorithms, namely Local-Range and Global-Range partitioning, specifically designed for stack-based compaction strategies. We conduct a comparative study involving these two partitioning algorithms, along with the existing Size partitioning and No-partitioning approaches, using the widely-used RocksDB. Our experimental results demonstrate that the proposed algorithms can significantly enhance compaction throughput, achieving improvements of up to 30%. Furthermore, they also exhibit substantial enhancements in overall write throughput, with gains of up to 20% over the No-partitioning or Size partitioning approaches when leveraging RocksDB's Universal Compaction. Remarkably, despite these performance enhancements, the proposed algorithms demonstrate comparable or even superior point query latency and only a minimal increase of less than 10% in range query latency.

IX. Acknowledgements

This work was partially supported by NSF grants IIS-2227669, IIS-1901379 and IIS-1838222.

References

[1] "Bigtable," 2019. [Online]. Available: https://cloud.google.com/bigtable
[2] "Cassandra," 2019. [Online]. Available: http://cassandra.apache.org
[3] "HBase," 2019. [Online]. Available: https://hbase.apache.org
[4] "LevelDB," 2019. [Online]. Available: https://github.com/google/leveldb
[5] "AsterixDB," 2020. [Online]. Available: https://asterixdb.apache.org
[6] "RocksDB," 2020. [Online]. Available: https://rocksdb.org
[7] "RocksDB Wiki," 2020. [Online]. Available: https://github.com/facebook/rocksdb/wiki
[8] M. Y. Ahmad and B. Kemme, "Compaction management in distributed key-value datastores," *Proc. VLDB Endow.*, vol. 8, no. 8, pp. 850–861, Apr. 2015.
[9] H. Amur, D. G. Andersen, M. Kaminsky, and K. Schwan, "Design of a write-optimized data store," Georgia Institute of Technology, Tech. Rep., 2013.
[10] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
[11] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto, "Spooky: Granulating LSM-tree compactions correctly," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 3071–3084, 2022.
[12] C. Jermaine, E. Omiecinski, and W. G. Yee, "The partitioned exponential file for database storage management," *The VLDB Journal*, vol. 16, no. 4, pp. 417–437, Oct. 2007.
[13] J. Li, P. Jin, Y. Lin, M. Zhao, Y. Wang, and K. Guo, "Elastic and stable compaction for LSM-tree: A FaaS-based approach on TerarkDB," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 3906–3915.
[14] Y. Li, H. H. Chan, P. P. Lee, and Y. Xu, "Enabling efficient updates in KV storage via hashing: Design and performance evaluation," *ACM Transactions on Storage (TOS)*, vol. 15, no. 3, pp. 1–29, 2019.
[15] Q. Mao, S. Jacobs, W. Amjad, V. Hristidis, V. J. Tsotras, and N. E. Young, "Comparison and evaluation of state-of-the-art LSM merge policies," *The VLDB Journal*, vol. 30, no. 3, pp. 361–378, 2021.
[16] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A unified solution for write-optimized key-value stores in large datacenter," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 477–489.
[17] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 497–514.
[18] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 71–82.
[19] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*, 2017, pp. 1–13.
[20] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie, "Building efficient key-value stores via a lightweight compaction tree," *ACM Transactions on Storage (TOS)*, vol. 13, no. 4, pp. 1–28, 2017.
[21] W. Zhang, Y. Xu, Y. Li, and D. Li, "Improving write performance of LSMT-based key-value store," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2016, pp. 553–560.