

Scaling Embedded Key-Value Storage with a Local Partitioning Layer

Douglas Pereira Luiz¹, Odorico Machado Mendizabal¹

¹Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brazil

douglas.pereira@posgrad.ufsc.br, odorico.mendizabal@ufsc.br

Abstract. *Embedded key-value stores such as LMDB, LevelDB and RocksDB are widely used due to their low deployment complexity, and efficient local persistence. However, operations within a single database are constrained by the synchronization and resource management mechanisms of the underlying storage engine, limiting concurrency under highly parallel workloads. This paper presents a partitioning layer that distributes data across multiple storage engine instances while exposing a unified key-value interface. The architecture uses fine-grained key-to-partition assignment through a routing index and partition-level synchronization. It also supports hash-based lock-striping strategy and optional workload-aware repartitioning. The design is storage-engine agnostic and requires no modifications to existing embedded databases. Experiments with YCSB workloads and multiple storage engines reach up to 2× the throughput of standalone instances on single-key operations, and show the costs of range scans, repartitioning, and extra resource use.*

1. Introduction

Embedded key-value engines are attractive building blocks for data-intensive systems because they provide low-latency local persistence, simple deployment, and efficient storage utilization. Engines such as LMDB [Chu 2026], LevelDB [Dean and Ghemawat 2026], and RocksDB [Meta Platforms, Inc. 2026] are widely used as storage backends for applications requiring compact software stacks and high local I/O performance. To ensure consistency under concurrent access, these engines implement concurrency control mechanisms such as single-writer models, fine-grained locks, or multi-versioning. As concurrency increases, however, these mechanisms can limit throughput even when operations access independent portions of the dataset.

A common approach to increasing parallelism in database and storage systems is data partitioning. However, applying existing partitioning strategies to embedded storage poses distinct challenges. Most prior approaches target in-memory data structures or require intrusive modifications to storage engine internals, limiting portability and reuse with production-ready backends [Luiz and Mendizabal 2024, Curino et al. 2010, Quamar et al. 2013, Goulart et al. 2023, Didona and Zwaenepoel 2019, Hong et al. 2016].

To address these limitations, this work presents a partitioned key-value store layer for embedded databases. The proposed engine-agnostic architecture builds a unified key-value store over multiple embedded database instances, partitioning data so that operations on different partitions can proceed concurrently without being constrained by a

single engine instance. The design combines fine-grained key placement, runtime repartitioning, and atomic operations across one or more partitions, preserving the advantages of embedded databases while improving throughput under concurrent workloads.

The framework evaluates two data-to-partition routing strategies: *Indexed Partitioning*, which uses a dynamic routing index with optional workload-aware repartitioning to co-locate frequently accessed keys, and *Lock-Striping Partitioning*, which employs static hash-based routing, eliminating routing metadata, key migration, and coordination overhead. Experiments with YCSB workloads under varying concurrency levels and durability modes (asynchronous and synchronous) using LMDB and LevelDB show that local partitioning improves scalability for single-key operations, achieving up to $2\times$ higher throughput than standalone database instances under write-intensive workloads. The evaluation also highlights trade-offs, including higher latency for multi-key range scans and increased memory and disk usage due to multiple database instances.

2. Related Work

Reducing synchronization overhead through finer-grained locking has been widely studied in concurrent systems. Lock striping associates locks with subsets of the key space, allowing unrelated operations to execute concurrently without the complexity of per-key synchronization [Herlihy and Shavit 2012]. More broadly, scalable multicore systems show that reducing shared synchronization domains is essential for high throughput under concurrent workloads [Mao et al. 2012, Lim et al. 2014], motivating the synchronization strategy adopted in this work.

Shared-nothing architectures reduce contention by isolating independent workloads [DeWitt and Gray 1992]. In contrast, embedded key-value stores are typically deployed as a single logical storage instance [Dean and Ghemawat 2026, Meta Platforms, Inc. 2026, Chu 2026]. LevelDB and RocksDB use Log-Structured Merge Trees (LSM-trees) [O’Neil et al. 1996], while LMDB adopts a memory-mapped B+ Tree with copy-on-write updates, providing efficient reads and snapshot isolation [Chu 2026].

Data partitioning is a well-established technique for improving scalability and parallelism in storage systems. Early distributed databases used horizontal partitioning to balance workloads across nodes [DeWitt and Gray 1992]. Distributed key-value stores such as Dynamo [DeCandia et al. 2007], Cassandra [Lakshman and Malik 2010], and HBase [George 2011] employ partitioned architectures based on consistent hashing and shard assignment. Workload-aware approaches, including Schism [Curino et al. 2010] and SWORD [Quamar et al. 2013], further exploit access patterns to reduce cross-partition interactions while maintaining balanced resource utilization. However, these approaches target distributed systems rather than embedded storage engines.

For local key-value storage, prior work has shown that partition-aware execution can significantly improve performance. *Size-aware sharding* separates requests by execution cost to improve throughput and tail latency in in-memory key-value systems [Didona and Zwaenepoel 2019]. Similarly, *soft partitioning* dynamically rebalances execution resources over a shared in-memory data structure under skewed workloads [Hong et al. 2016]. In [Luiz and Mendizabal 2026], the authors investigated asynchronous and workload-aware repartitioning for partitioned key-value systems. That

work showed that graph-based workload characterization, asynchronous repartitioning, and proactive monitoring can reduce scheduler disruption and improve performance under synchronization-intensive workloads. As with the other approaches discussed in this section, this strategy was evaluated in a multithreaded in-memory prototype, limiting its applicability to existing key-value store implementations.

In contrast to previous work, this paper presents a partitioning layer designed as a general approach for embedded key-value databases. The proposed architecture distributes data across multiple local instances of storage engines, providing a storage-engine-agnostic mechanism for partitioned execution over existing embedded databases.

3. Adaptive Partitioning Layer

The proposed strategy is implemented as a partitioning layer that exports a unified key-value store while internally managing multiple embedded storage instances. Unlike purely logical partitioning approaches, each partition can be mapped to its own storage-engine instance, enabling physical data separation.

Operations execute directly in the calling thread, with concurrency controlled by reader-writer locks associated with partition assignments. This design allows requests to different partitions to proceed concurrently while preserving atomicity for single- and multi-key operations. By distributing requests across multiple engine instances, the approach reduces the impact of restrictive concurrency-control mechanisms, such as single-writer designs, coarse-grained synchronization, and contention-inducing operations like LSM-tree compactions.

3.1. Partitioned Storage Model

The strategy assigns a dedicated storage instance to each partition. Storage management consists of a set of n storage engines, $\mathcal{S} = S_0, S_1, \dots, S_{n-1}$, a corresponding set of shared locks, $\mathcal{L} = L_0, L_1, \dots, L_{n-1}$, and a storage-partition mapping structure, M_s , protected by the shared lock μ_m . Given a key $k \in \mathcal{K}$, $M_s(k)$ returns the partition responsible for storing the key and its associated lock. The mapping is implemented as a dedicated embedded database instance and is used for both request routing and partition-level synchronization.

Figure 1 illustrates a write operation in the partitioned storage model. Double-ended arrows indicate access to a storage engine instance. The green dashed region denotes the period in which the caller holds μ_m , while the blue region denotes when it holds L_p . These lock regions overlap to ensure consistency, following a hand-over-hand locking approach. The operation begins by acquiring μ_m to read the key-to-partition mapping from M_s . After obtaining the target partition, the corresponding partition lock is acquired before releasing μ_m , preventing changes to partition assignments during lock acquisition. The write is then executed, and the partition lock is released.

Read operations follow the same protocol, returning immediately if no partition exists for the requested key. Instead of writing to S_p , they retrieve the corresponding value. Scan operations first perform a range scan over M_s to obtain the required key-partition pairs, acquire the corresponding partition locks in a fixed order, execute the reads, and finally release the locks.

An optional dynamic repartitioning mechanism adapts data placement during execution. When enabled, a background thread, called the *partitioner*, periodically redis-

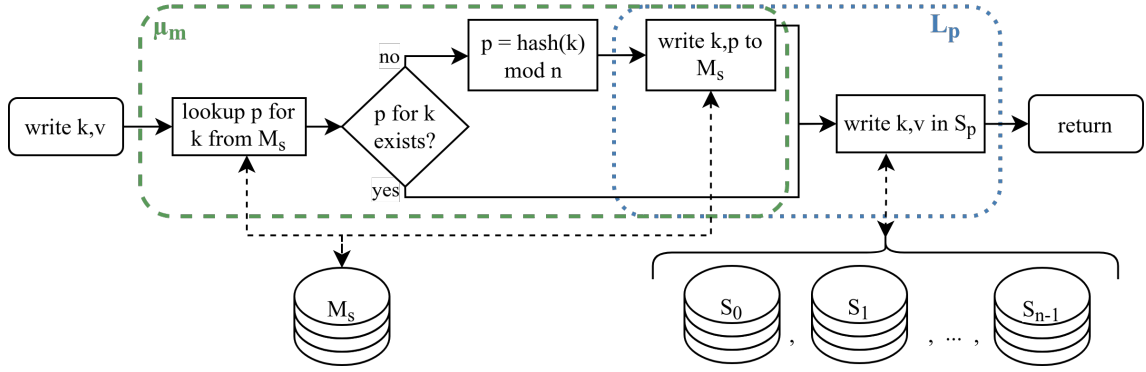


Figure 1. Writing operation through the key-value store layer.

tributes keys according to the workload observed over a configurable monitoring window. During this tracking period, each operation is recorded by a *tracker*, which builds a graph relating accessed keys. The partitioner then partitions this graph to improve data placement, as illustrated in Figure 2. The goal is to co-locate frequently accessed keys and balance the workload across storage instances. Repartitioning is transparent to application threads and updates the storage-partition mapping as keys are migrated.

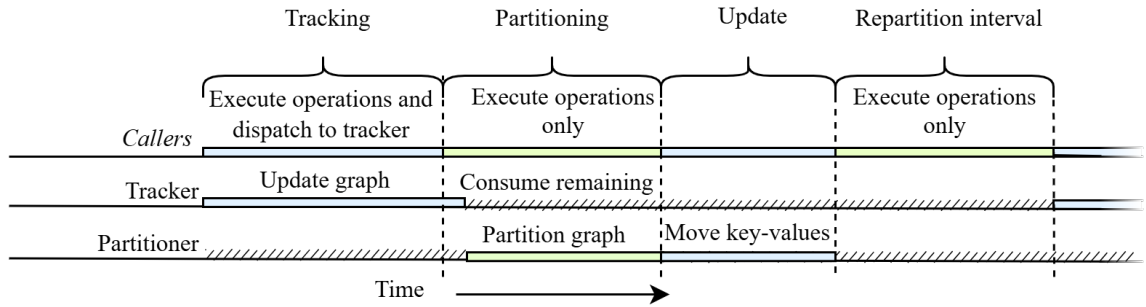


Figure 2. Dynamic repartitioning cycle.

The workload is modeled as a weighted undirected graph, where each vertex represents a key and its weight corresponds to the number of operations involving that key. Edges connect keys accessed together in the same operation, with weights indicating the frequency of co-access. The graph is then partitioned to balance the computational load while minimizing cross-partition operations, defining how key-value pairs should be migrated. The proposed implementation uses the multilevel graph partitioning algorithm provided by METIS [Karypis and Kumar 1998]. To keep the load continuously balanced, the partitioner periodically processes the co-access graph collected during the monitoring window, and a separate migration step moves the affected pairs between engine instances and updates the routing index.

As an alternative to indexed partition assignment, the layer supports a lock-stripping strategy that eliminates explicit key-to-partition mappings. The partition for a key k is computed as $h(k) \bmod n$, where h is a hash function and n is the number of partitions. This approach reduces metadata overhead and simplifies request routing, since the target partition is determined without consulting a mapping structure. Its main drawback is higher overhead for multi-key operations: because key placement is computed rather

than tracked, range scans must examine all partitions and merge their results.

A key feature of the proposed model is the physical distribution of data across multiple storage-engine instances. During initialization, each instance can persist its data in a different directory, allowing partitions to be placed on separate storage devices. This enables device-level I/O parallelism when concurrent operations access different partitions. The approach is especially beneficial under strict durability requirements, where writes must be persisted before completion, as distributing partitions across independent devices can reduce I/O contention and improve throughput.

4. Experimental Evaluation

The partitioning layer was implemented in C++ and is available in a dedicated GitHub repository.¹ The repository includes the source code, datasets, scripts, and experimental results required to reproduce the evaluation.

We evaluated four configurations: direct engine access without partitioning; indexed partitioning without repartitioning (*index*); indexed partitioning with repartitioning every 10 s over a 1 s monitoring window (*repart*); and lock-striping partitioning (*ls*). The layer was configured with one or three SSDs. In the multi-device setup, partitions were assigned to SSDs in a round-robin fashion, placing partition i on SSD $(i \bmod d)$, where d is the number of devices. All configurations were evaluated with synchronous and asynchronous writes, and RocksDB results are included for comparison. The evaluation examines how throughput and latency scale with concurrency and when physically partitioning data across multiple storage-engine instances outperforms standalone engines and logical partitioning alone.

The experiments were conducted with three embedded storage engines: LMDB (Lightning Memory-Mapped Database) [Chu 2026], LevelDB [Dean and Ghemawat 2026], and RocksDB [Meta Platforms, Inc. 2026]. LMDB is a memory-mapped B+ tree database that provides ACID transactions and efficient reads, while LevelDB and RocksDB are log-structured merge-tree (LSM) key-value stores optimized for high write throughput. These engines were selected because they represent distinct storage organizations and concurrency-control strategies while being widely used in practice. Their adoption in real-world systems makes them suitable for evaluating the proposed layer. The implementation is modular and extensible, allowing additional storage engines to be incorporated with minimal changes.

4.1. Workloads

The request sets were derived from the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al. 2010]. We evaluated three workloads. *Workload A* (update-heavy) consists of 50% reads and 50% updates with a highly skewed Zipfian distribution ($\theta = 0.99$), representing scenarios such as session stores. *Workload D* (read-latest) contains 95% reads and 5% inserts, modeling social-media workloads in which users frequently access recent updates. *Workload E* (scan-heavy) includes 95% scans and 5% updates, representing applications that perform range queries, such as retrieving posts within a discussion thread.

¹<https://github.com/repartkv-cmyk/repart-kv>

To provide a gradual increase in load, workload-specific *thinking times* were inserted between requests. For Workload A, the thinking time was $100\ \mu s$ with asynchronous writes and $10\ ms$ with synchronous writes. For Workloads D and E, the values were $20\ \mu s$ and $0.2\ ms$, respectively. The longer delay in Workload A prevented the standalone engines from saturating prematurely due to its high write intensity.

4.2. Test Environment and Parameterization

The experiments were conducted on a workstation with an Intel Xeon E5-2683 v4 processor (16 physical cores at 2.10 GHz, 40 MB cache), 64 GB of DDR4-2400 ECC memory, three 128 GB SATA III SSDs, and Ubuntu 24.04 LTS (Linux kernel 6.14.0-37-generic).

Each experiment used a pre-populated database with 1 million key-value pairs, each storing a $1\ kB$ value. The number of worker threads varied from 4 to 16 (4, 8, 12, and 16). Each worker repeatedly waited for the configured thinking time before issuing the next workload operation. All partitioned configurations used 32 partitions. When enabled, repartitioning was performed every $10\ s$ using a $1\ s$ workload tracking window. A baseline configuration without repartitioning was also evaluated. Each scenario was executed five times, and the reported results are the average across runs.

4.3. Results

This section presents the performance evaluation of the proposed partitioning layer using benchmarks, comparing it with off-the-shelf key-value storage engines. Each worker thread records the start and end time of every operation to measure latency, while an auxiliary thread samples throughput every $100\ ms$. Each experiment was repeated five times, collecting metrics over $30\ s$ after pre-populating the database with 1,000,000 key-value pairs. The evaluation focuses on the maximum throughput achieved by each configuration and its impact on latency. Latency-throughput curves identify the saturation point and throughput limit; latency is the median. Throughput can fall from hardware exhaustion and synchronization costs, as threads wait on locks, switch context, or thrash the cache.

4.3.1. Update-Heavy Workload

Experiments suggest performance increase potential on write-intensive workload of single-key operations. Figure 3a shows results with LMDB as storage engine, and additionally RocksDB, under workload A. The results suggest that the partitioning layer can greatly improve performance, as it pushed the throughput up to $150k\ ops/s$, a huge improvement compared to pure LMDB and RocksDB, as both saturated around $70k\ ops/s$, indicating a throughput improvement of about $2\times$. The layer did not seem to benefit from multiple storage devices or repartitioning under this workload using LMDB. Also, *lock striping* seems to have some latency and throughput advantage, likely due to its much lighter partition assignment strategy. Nevertheless, even though the pure engine saturated the earliest between the tested configurations, it presented the lowest latency medians in lower loads, suggesting a trade-off between throughput and latency in layer utilization.

In Figure 3b, that shows the same setting but with LevelDB as the storage engine, the layer also seemed to greatly improve throughput, achieving almost $60k\ ops/s$, leaving

pure LevelDB far behind on less than $22k\ ops/s$. Also, using LevelDB as storage drastically changed the perceived impact of multiple storage devices and repartitioning: index, repartitioning and lock striping presented throughput higher in about 30%, 18% and 50%, respectively.

Comparing index to repartitioning, both with one storage device, it can be seen that repartitioning increased throughput by about 30%. Using 3 storage devices, the throughput gains were near 14%, however latencies on higher throughput also decreased close to RocksDB. Even though the performance improvement with LevelDB was up to 160% throughput increase, RocksDB achieves even higher throughput. However, lock striping latencies were between $5\ \mu s$ and $10\ \mu s$, significantly lower than RocksDB, that were at least 50% higher on lower load and over 100% higher under higher loads.

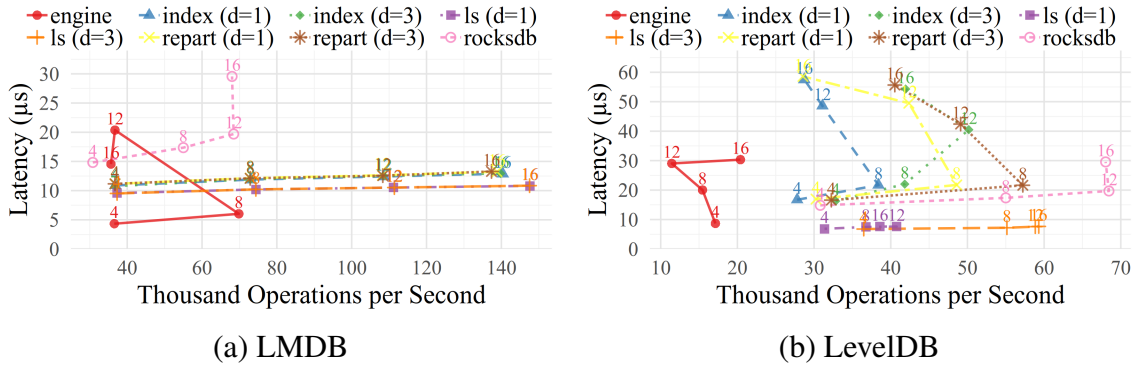


Figure 3. Throughput versus median latency for workload A with asynchronous writes and thinking time equal to $100\ \mu s$. Points indicate worker counts.

Overall, the results indicate that the proposed partitioning layer is particularly effective for update-heavy workloads. While standalone storage engines saturated at $\approx 70k\ ops/s$, the partitioned configurations sustained more than $140k\ ops/s$, corresponding to an improvement of roughly $2\times$. This gain comes at the cost of increased operation latency, rising from $\approx 5\ \mu s$ to $10\ \mu s$. Nevertheless, the latency increase remains modest in absolute terms, suggesting that the proposed approach provides a favorable throughput-latency trade-off for highly concurrent update-intensive workloads.

4.4. Read Latest Workload

The results with the read-latest workload, presented in Figures 4a and 4b, indicate performance gains over LevelDB and RocksDB under this single key read dominated workload. LevelDB throughput median was at most $124\ ops/s$, and RocksDB achieved almost $212k\ ops/s$, while repartitioning, the layer settings with the lowest performance, achieved a similar throughput. With index or lock striping, the throughput was $240k\ ops/s$ and $302k\ ops/s$, with a single SSD, and $270k\ ops/s$ and $315k\ ops/s$, with 3 storage devices, corresponding to an increase in performance of at least 77%, and at most 154%.

Lock striping achieved the highest throughput and lowest latencies, but did not consistently outperform standalone LMDB, reaching similar performance as shown in Figure 4a. The remaining partitioned configurations did not exceed $280k\ ops/s$, about 16% below both lock striping and standalone LMDB. Nevertheless, the partitioning layer with LMDB still outperformed RocksDB, delivering at least 32% higher throughput and up to 57% improvement while maintaining median latencies about $3\times$ lower.

With LevelDB, repartitioning degraded performance relative to indexed partitioning, while lock striping again achieved the best results. A likely explanation is the behavior of Workload D. Because writes continuously insert new keys, hash-based partitioning naturally distributes requests across partitions, reducing persistent hotspots. In addition, the workload favors recently inserted records, causing the hot set to change rapidly over time. Consequently, workload information collected during the tracking period quickly becomes outdated, limiting the benefits of adaptive repartitioning. Under these conditions, the overhead of tracking accesses, maintaining partition mappings, and migrating keys can outweigh the gains from improved data placement.

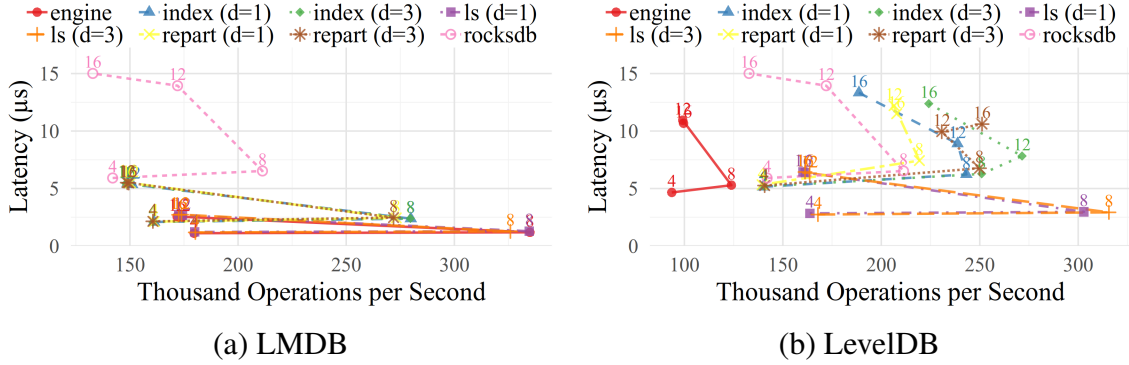


Figure 4. Throughput versus median latency for workload D with asynchronous writes ($20 \mu s$ thinking time). Points indicate worker counts.

4.4.1. Range Scans Workload

Experiments with Workload E reveal a significant latency penalty for multi-key operations. The lock-striping configuration exhibited median latencies above $2000 \mu s$ and throughput below $3000 ops/s$. Unlike indexed partitioning, lock striping does not maintain a key-to-partition index, requiring range scans to query all partitions.

Figures 5a and 5b show that the indexed and repartitioning configurations incurred latencies about $4\times$ higher than the corresponding standalone engines for both LevelDB and LMDB. With LevelDB, throughput dropped to at most half that of the standalone engine. The effect was even more pronounced with LMDB, whose standalone version scaled to 16 workers with almost no latency degradation.

Despite this limitation, the indexed and repartitioning configurations remained competitive when using LMDB as the backend. Compared with LevelDB and RocksDB, the partitioned LMDB consistently achieved lower latencies and higher throughput. In particular, it outperformed standalone RocksDB by up to 35% in throughput while maintaining latencies about 10% lower. These results indicate that, although range scans increase the cost of partitioned execution, the choice of storage engine remains a key factor in overall performance.

4.5. Synchronous writes

Ensuring that writes are atomically persisted is costly but essential for consistent recovery after failures or restarts. We therefore evaluated the partitioning layer with synchronous

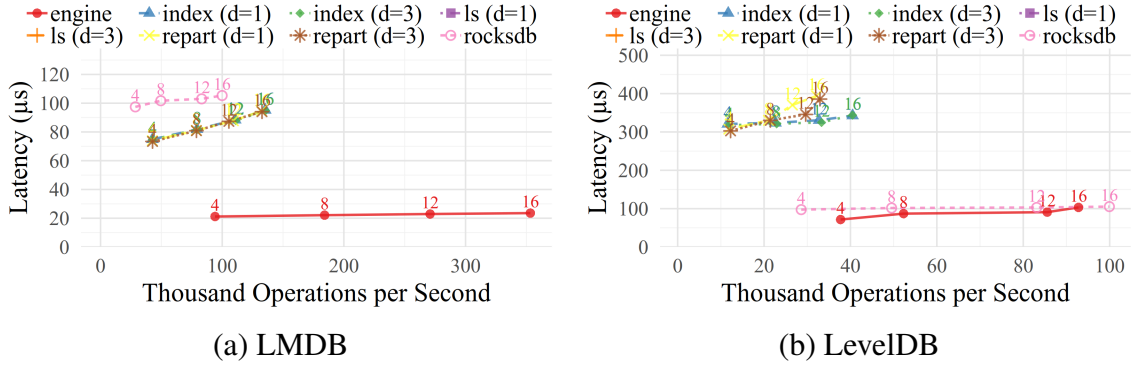


Figure 5. Throughput versus median latency for workload E with asynchronous writes ($20 \mu s$ thinking time). Points indicate worker counts.

writes. The results show that the layer benefits substantially from multiple storage devices, particularly under write-intensive workloads.

For Workload A, standalone LMDB achieved less than 300 ops/s . Using the partitioning layer with one SSD roughly doubled throughput, while three SSDs increased it to nearly $4\times$ the baseline, exceeding standalone RocksDB by 58% (Figure 6). With LevelDB engine, although the standalone LevelDB already outperformed RocksDB, the partitioning layer improved throughput by about 40% with multiple storage devices.

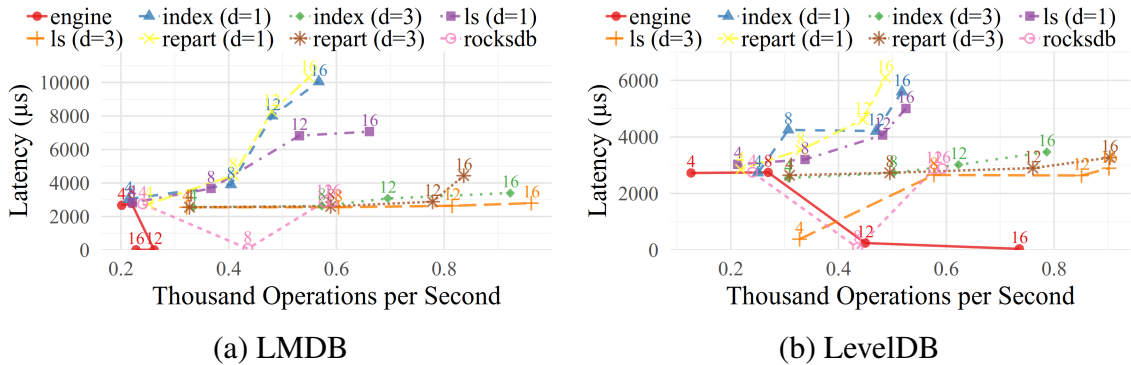


Figure 6. Throughput versus median latency for workload A with synchronous writes and thinking time equal to 10 ms . Points indicate worker counts.

Even in Workload D, which contains only 5% writes, multiple SSDs improved performance. Figures 7a and 7b show that the partitioned configurations consistently achieved higher throughput than their single-SSD counterparts. Relative to standalone LMDB, the layer tripled throughput while maintaining similar latencies with lock striping. The indexed and repartitioning configurations incurred a latency penalty but still achieved substantially lower latencies than RocksDB.

With LevelDB, lock striping reduced median latency and, when using multiple storage devices, achieved throughput comparable to the standalone engine. In contrast, the indexed and repartitioning configurations did not surpass standalone LevelDB, while RocksDB maintained at least 20% higher throughput than partitioned LevelDB settings.

Though repartitioning with multiple storage devices yielded significantly higher throughput than stock LMDB on workload D with synchronous writes, the performance

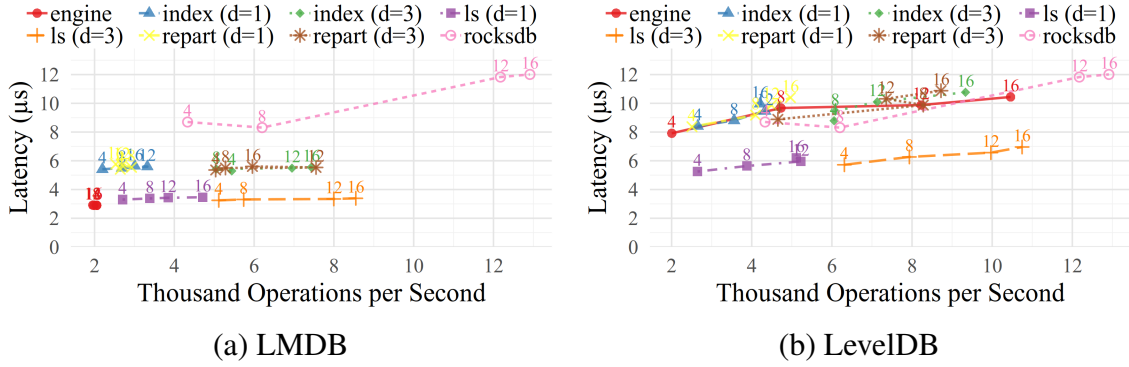


Figure 7. Throughput versus median latency for workload D with synchronous writes (0.2 ms thinking time). Points indicate worker counts.

on scans dominated workload seems to be the weakest point of the proposed layer (Figure 8). Lock striping latencies are higher than 3000 μs , in a similar behavior to the one observed with asynchronous writes. Even though indexed partitioned LMDB could surpass the out-of-the-box engine, no other configuration could increase performance. Additionally, standalone RocksDB reaches higher throughput, extending farther along the horizontal axis of the latency–throughput curve as the number of workers grows.

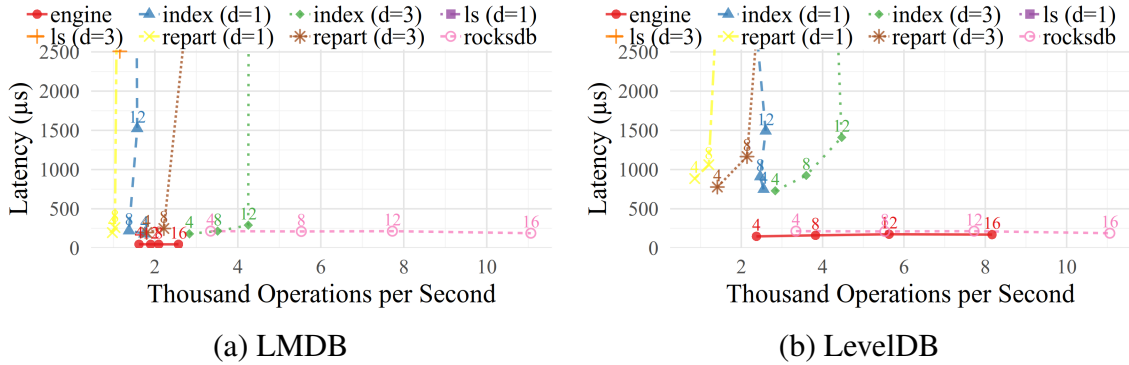


Figure 8. Throughput versus median latency for workload E with synchronous writes (0.2 ms thinking time). Points indicate worker counts.

4.6. Memory and storage costs

Beyond throughput and latency, partitioning also affects resource consumption. We therefore measured disk usage and memory footprint during execution. Memory utilization was collected by a dedicated thread that sampled the process resident set size (RSS) every 100 ms. Figures 9a and 9b show the average storage utilization for YCSB Workload D with asynchronous writes; the remaining workloads exhibited similar trends.

The storage overhead depends strongly on the underlying storage engine. As shown in Figure 9b, LevelDB’s disk usage increases substantially with the number of partitions, with some configurations requiring nearly twice the storage of the standalone engine. The lock-striping configuration consistently incurs lower overhead because it does not require an additional storage instance for the partition mapping. A likely explanation is that distributing data across multiple independent LSM-tree instances reduces

compaction efficiency, causing obsolete entries and metadata to persist longer. A detailed investigation of this effect is left for future work.

A different behavior is observed for LMDB in Figure 9a. Although LMDB requires substantially more storage than LevelDB in absolute terms, the relative overhead introduced by partitioning is much smaller. This observation is consistent with the compaction hypothesis, since LMDB’s B+ Tree architecture does not rely on LSM-tree compaction. Consequently, the storage overhead of maintaining multiple database instances is less pronounced.

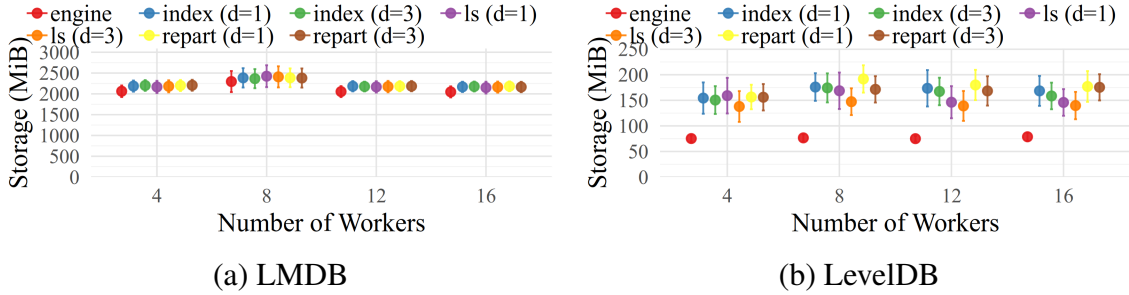


Figure 9. Median and standard deviation of storage usage under workload D with asynchronous writes ($20 \mu s$ thinking time).

Figures 10a and 10b show memory consumption for Workload D with asynchronous writes. Error bars represent the standard deviation across repetitions. As shown in Figure 10b, LevelDB’s memory footprint increases considerably with the number of partitions due to the overhead of maintaining multiple engine instances. In contrast, Figure 10a shows that partitioning has a much smaller impact on LMDB, although its absolute memory consumption remains higher than LevelDB in all configurations.

Overall, these results indicate that the resource overhead of partitioning depends strongly on the underlying storage engine. Although the proposed middleware can significantly improve performance, it also increases memory and storage consumption. These overheads are modest for LMDB but substantially higher for LevelDB, highlighting the importance of storage-engine-specific trade-offs when deploying the partitioning layer.

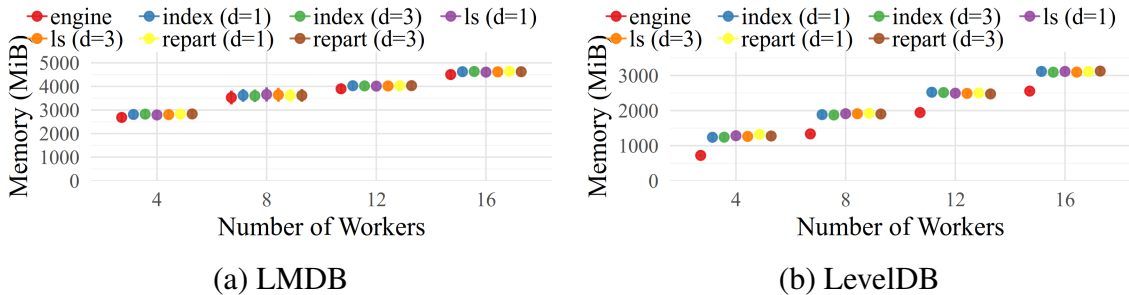


Figure 10. Median and standard deviation of memory usage under workload D with asynchronous writes ($20 \mu s$ thinking time).

4.7. Discussion

The experimental results show that partitioning can substantially improve the performance of embedded key-value stores for workloads dominated by single-key operations. By

distributing data across multiple storage engine instances, the partitioning layer reduces contention and increases concurrency among independent operations. Across the evaluated write-intensive and mixed workloads, all partitioned configurations consistently outperformed standalone engines, demonstrating that partition-level isolation improves throughput without requiring changes to the underlying databases.

A second observation is that the effectiveness of adaptive repartitioning depends on workload stability. Repartitioning was beneficial when access patterns exhibited sufficient locality to justify key migration. However, under workloads with rapidly changing hot sets, such as YCSB Workload D, it often provided little benefit and sometimes introduced additional overhead. In these cases, naturally balanced access patterns and the limited predictive value of historical workload information outweighed the gains from improved data placement. The results also indicate that hash-based partition assignment offers an attractive balance between simplicity and performance. The lock-striping configuration frequently achieved the highest throughput and lowest latency by eliminating metadata lookups and migration overhead while still distributing single-key operations effectively. This suggests that sophisticated partitioning mechanisms are unnecessary when workloads are already well balanced.

The benefits of multiple storage devices were workload-dependent. Significant improvements were observed only with synchronous persistence, where distributing partitions across independent devices reduced I/O contention. Under asynchronous writes, the impact was generally negligible, indicating that CPU, synchronization, or memory bottlenecks dominated performance. For multi-key operations, the experiments reveal an important limitation of the proposed approach. The partitioning layer is best suited to single-key workloads. Frequent range scans incur additional routing overhead and require accessing multiple storage instances, particularly with hash-based partitioning, where scans must examine every partition.

Finally, the partitioning layer improves performance at the cost of additional memory and storage resources. This overhead is modest for LMDB but considerably higher for LevelDB, indicating that the efficiency of the approach depends on the architecture of the underlying storage engine.

5. Conclusion

This paper presented a partitioning layer to improve concurrency in embedded key-value storage engines. The approach combines data partitioning, optional workload-aware repartitioning, and partition-level synchronization to increase parallelism while preserving a unified key-value interface. Its modular design allows existing storage engines to be used without modification. The layer was evaluated with LMDB and LevelDB, with comparisons against standalone configurations and RocksDB. The results show that partitioned execution substantially improves performance for workloads dominated by single-key operations, while workloads with frequent range scans incur additional coordination overhead due to accesses across multiple partitions. Overall, the proposed partitioning layer provides a practical and effective way to improve the scalability of embedded key-value storage engines under concurrent workloads, although no single configuration is optimal for all workload characteristics.

References

- Chu, H. (2026). Lightning memory-mapped database (lmdb). <https://symas.com/lmdb>. Ubuntu package liblmdb-dev, version 0.9.31.
- Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of SoCC '10*, page 143–154.
- Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1–2):48–57.
- Dean, J. and Ghemawat, S. (2026). Leveldb. <https://github.com/google/leveldb>. Ubuntu package libleveldb-dev, version 1.23.
- DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., and Vogels, W. (2007). Dynamo: amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220.
- DeWitt, D. and Gray, J. (1992). Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98.
- Didona, D. and Zwaenepoel, W. (2019). Size-aware sharding for improving tail latencies in in-memory key-value stores. In *USENIX NSDI*, NSDI’19.
- George, L. (2011). *HBase: The Definitive Guide*. O’Reilly Media.
- Goulart, H., Trombeta, J., Franco, Á., and Mendizabal, O. (2023). Achieving enhanced performance combining checkpointing and dynamic state partitioning. In *SBAC-PAD*.
- Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.
- Hong, B., Kwon, Y., Ahn, J. H., and Kim, J. (2016). Adaptive and flexible key-value stores through soft data partitioning. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 296–303.
- Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, pages 359–392.
- Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40.
- Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. (2014). Mica: a holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI’14, page 429–444, USA. USENIX Association.
- Luiz, D. and Mendizabal, O. (2024). Lightweight asynchronous repartitioning for local state partitioned systems. In *Proceedings of the 25th Brazilian Symposium on High Performance Computing Systems*. SBC.
- Luiz, D. P. and Mendizabal, O. M. (2026). Stall-free asynchronous state repartitioning with a proactive workload tracking window. *Concurrency and Computation: Practice and Experience*, 38(3):e70581.
- Mao, Y., Kohler, E., and Morris, R. T. (2012). Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer*

- Systems*, EuroSys '12, page 183–196, New York, NY, USA. Association for Computing Machinery.
- Meta Platforms, Inc. (2026). Rocksdb. <https://rocksdb.org>. Ubuntu package librocksdb-dev, version 8.9.1.
- O’Neil, P., Cheng, E., Gawlick, D., and O’Neil, E. (1996). The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385.
- Quamar, A., Kumar, K. A., and Deshpande, A. (2013). Sword: Scalable workload-aware data placement for transactional workloads. In *International Conference on Extending Database Technology (EDBT)*.