



WHITE PAPER · DATA STORAGE ENGINES

LSM-Tree Engines in Production

Compaction, write amplification and tuning

White Paper 2 · Data Storage Engines series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

LSM-Tree Engines in Production: Compaction, Write Amplification and Tuning

White Paper 2 · Data Storage Engines series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Data Storage Engines, paper 2 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract.....	4
2. Introduction	4
2.1 Why LSM-trees dominate write-heavy storage	4
2.2 Scope.....	4
2.3 How this paper builds on paper 1	4
3. LSM-tree fundamentals in production.....	4
3.1 Memtable, write-ahead log and sorted runs	4
3.2 Read, write and space amplification.....	5
3.3 The RUM trade-off in LSM terms	5
3.4 The production landscape	5
4. Compaction strategies	6
4.1 Leveled compaction	6
4.2 Tiered and universal compaction.....	6
4.3 Hybrid and partial compaction	6
4.4 The compaction design space	6
4.5 Granulated compaction.....	6
5. Reducing write amplification	7
5.1 Key-value separation.....	7
5.2 Tiering as a write-amplification lever	7
5.3 Deletes and tombstones	7
5.4 Zoned storage	7
5.5 Heterogeneous storage tiers	8
6. Read path and filters.....	8
6.1 Bloom filters and their memory allocation	8
6.2 Range filters.....	8

6.3 Block cache	8
6.4 Learned indexes	9
7. Tuning in production	9
7.1 The main levers	9
7.2 Write stalls	9
7.3 Robust tuning under uncertainty.....	10
7.4 Automated, learned and LLM-assisted tuning	10
8. Cloud-native and disaggregated LSM engines	10
8.1 Separating compute and storage.....	10
8.2 Compaction offload and compaction as a service.....	10
8.3 LSM-trees on object storage	11
9. Comparative analysis.....	11
10. Open challenges and future opportunities.....	11
10.1 Predictable tail latency and write stalls	11
10.2 Tuning complexity and workload drift	12
10.3 Safety of LLM-assisted tuning	12
10.4 Economics of compaction offload	12
10.5 Tiered and heterogeneous storage	12
10.6 Range deletes and privacy deletion	12
10.7 Verification and integrity	12
10.8 Learned and adaptive structures	13
11. What this means for platform decisions	13
References	13

1. Abstract

Log-Structured Merge-trees (LSM-trees) are the storage layer behind many write-heavy databases and key-value stores. Their behavior in production depends less on the basic data structure than on compaction: the background process that merges sorted files. This paper examines how compaction choices set the balance between write, read and space amplification in RocksDB-family engines. It reviews compaction strategies, techniques that reduce write amplification, the read path and its filters, and the tuning problem, including robust, machine-learned and LLM-assisted tuning. It then covers cloud-native designs that move compaction off the database node or place the whole tree on object storage. The paper closes with open challenges, framed as research directions and opportunities, and with notes for platform decisions.

2. Introduction

2.1 Why LSM-trees dominate write-heavy storage

The LSM-tree was proposed to index data with a high rate of inserts at low I/O cost [1]. It buffers changes in memory and merges them into larger sorted files on disk in batches. Writes therefore become sequential, which suits both hard disks and flash. This property has made LSM-trees the storage layer of many NoSQL systems [2] and of several distributed SQL engines (Section 3.4).

RocksDB is the reference point for much of this work. Its developers describe how their main optimization target moved over eight years, from write amplification to space amplification and then to CPU use [3], [4]. That shift is a useful map of the problem. Each resource becomes the constraint once the others are under control.

2.2 Scope

This paper concentrates on RocksDB-family engines: RocksDB itself, engines derived from it or inspired by it, and research prototypes built on it. It does not report benchmark figures. Where a paper reports results, we describe them qualitatively and attribute them to the authors.

2.3 How this paper builds on paper 1

Paper 1 placed LSM-tree engines in Class A of a five-class taxonomy and named write amplification and compaction CPU as their bottleneck. This paper opens that class up. It explains where the bottleneck comes from, which design levers move it, and what remains unsolved.

3. LSM-tree fundamentals in production

3.1 Memtable, write-ahead log and sorted runs

A write first goes to a write-ahead log (WAL) for durability and to an in-memory buffer, the memtable. When the memtable fills, it becomes immutable and is flushed to disk as a Sorted String Table (SSTable). Files are never modified in place. New data is merged into new files, and old files are deleted once nothing refers to them.

On disk, data is organized into levels that grow by a size ratio T . With L levels, total capacity grows exponentially with L , so L stays small even for large datasets. In RocksDB, level 0 holds freshly flushed files whose key ranges may overlap. Each deeper level is a single sorted run, split into files with non-overlapping key ranges [8].

3.2 Read, write and space amplification

Three overheads describe an LSM-tree in production. Write amplification is the ratio of bytes written to storage to bytes written by the application. It rises because compaction rewrites the same entry several times as it moves down the levels. Read amplification is the number of files or blocks consulted for a logical read. Space amplification is the ratio of space used to the size of the live data, and it rises with obsolete versions, tombstones and temporary space during compaction.

These overheads interact. Compacting more eagerly reduces read and space amplification but increases write amplification. Compacting lazily does the reverse. Write amplification matters beyond throughput, because it consumes device bandwidth and, on flash devices, adds to media wear.

3.3 The RUM trade-off in LSM terms

The RUM conjecture holds that an access method that bounds two of read, update and memory (space) overheads must give up ground on the third [5]. LSM-trees make this trade-off explicit through a small set of parameters. In the standard cost model, leveling with size ratio T costs each entry about T rewrites per level, so write cost grows with $T \cdot L$. Tiering lets up to T runs accumulate per level before merging, so write cost grows with L but a lookup may probe up to $T \cdot L$ runs [6], [7].

Monkey showed that Bloom filter memory should not be spread evenly across levels [6]. Allocating more bits per key to smaller levels reduces the sum of false-positive rates, which bounds the expected cost of a lookup for a given memory budget. Dostoevsky showed that much of the merging at smaller levels adds write cost while doing little for lookups or space, and proposed lazy leveling: tiering at smaller levels and leveling at the largest [7]. Both results recast compaction tuning as navigation of a continuous design space rather than a choice between two fixed policies.

3.4 The production landscape

Several production engines share this structure while making different choices. Pebble, the storage engine of CockroachDB, is written in Go and inspired by LevelDB and RocksDB. It implements a subset of RocksDB's features, omits universal compaction, and adds level-0 sublevels and flush splitting to reduce read amplification under heavy writes [9].

CockroachDB's documentation describes an "inverted LSM", where compaction falls behind and upper levels hold more data than lower ones, leading to degraded reads [9].

TiKV uses RocksDB as its underlying key-value engine [10]. YugabyteDB's DocDB runs a customized fork of RocksDB, one instance per tablet, and disables the RocksDB WAL because the Raft log already provides durability [11]. SlateDB is an embedded LSM engine that writes its sorted tables to object storage instead of local disk, batching writes to reduce request costs [12].

4. Compaction strategies

4.1 Levelled compaction

Levelled compaction keeps one sorted run per level beyond level 0. When a level exceeds its target size, the engine merges one or more of its files with the overlapping files in the next level [8]. Each level then contributes at most one candidate file per key, and obsolete versions are discarded soon after they are overwritten.

RocksDB's dynamic level sizing sets level targets from the actual size of the last level rather than from the top down [8]. With a size ratio T , roughly $(T-1)/T$ of the data then sits in the last level, which bounds space amplification. The cost is write amplification, since each entry is rewritten as it passes through every level.

4.2 Tiered and universal compaction

Tiered compaction lets several sorted runs accumulate before merging them into a larger run. RocksDB's universal compaction is a tiered strategy. Its documentation notes lower write amplification than levelled compaction, at the cost of more sorted runs to read and spikier compaction I/O [8]. Full merges of large runs can temporarily need space close to the size of the data, so universal compaction exposes a size-amplification limit that triggers a full compaction when overhead grows too large [8].

Tiering suits ingest-heavy workloads such as logs, events and time series, and is less suited to frequent range scans.

4.3 Hybrid and partial compaction

Most production engines are hybrids. Level 0 in RocksDB is effectively tiered, because flushed files overlap, while deeper levels are levelled. Lazy leveling generalizes this by tiering all levels except the largest [7]. RocksDB can also run intra-level-0 compactions, which merge level-0 files with each other to reduce read amplification at the cost of extra writes [8].

Compaction granularity is a separate choice. A full merge compacts a whole level at once. A partial merge compacts one file, or a few, with the overlapping files below. Partial merges spread work over time and limit temporary space, but the selected files rarely align with the files below, so some non-overlapping data is rewritten [14].

4.4 The compaction design space

Sarkar et al. decompose any compaction strategy into four primitives: the trigger that starts a compaction, the data layout across levels, the granularity of each compaction, and the data-movement policy that chooses which data to move [13]. They evaluate a range of strategies on a common RocksDB-based platform. Their findings show that no single strategy is best across write throughput, lookup and range performance, space amplification and delete performance. They also note that most engines hard-code one strategy, chosen by an engineer, which leaves much of the space unused [13].

4.5 Granulated compaction

Spooky addresses the weaknesses of both full and partial merges [14]. It partitions the largest level into equal-sized files and aligns the boundaries of smaller levels to them. Each compaction then merges a group of files that overlap exactly. The authors report lower space amplification than full merges and lower write amplification than partial merges at the same

time. Writing fewer, larger files also avoids mixing data with different lifetimes on the SSD, which reduces the device's own garbage-collection work [14].

5. Reducing write amplification

5.1 Key-value separation

Compaction rewrites values as well as keys. When values are large, most compaction I/O moves values that have not changed. WiscKey proposed storing values in a separate log and keeping only keys and value pointers in the LSM-tree [15]. Compaction then moves small entries, which reduces write amplification.

Separation shifts costs elsewhere. A read needs an extra lookup to fetch the value. Range scans may read values scattered across the value log. Space used by overwritten values is reclaimed only by garbage collection, which itself rewrites data. DiffKV refines the idea: keys stay fully sorted, values are kept partially sorted to protect scans, and the degree of separation depends on value size [16]. The authors report balanced performance across writes, reads and scans on mixed workloads.

RocksDB's integrated BlobDB brings key-value separation into the main engine [17]. Values above a size threshold are written to blob files during flush and compaction, so they follow the same consistency rules as SSTables. Garbage collection runs as part of compaction and relocates valid values from the oldest blob files. The documentation is clear about the trade-off: a more aggressive garbage-collection cut-off reclaims more space but adds write amplification [17].

5.2 Tiering as a write-amplification lever

Tiering reduces write amplification directly, by merging each entry fewer times (Section 4.2). Combining tiered upper levels with a leveled last level keeps much of that saving while preserving read and space behavior for the bulk of the data [7]. The choice should be re-examined as the read-to-write ratio changes.

5.3 Deletes and tombstones

An LSM-tree deletes by writing a tombstone. The deleted data stays on disk until a compaction carries the tombstone down to the level that holds it. Tombstones therefore add write and space amplification, slow reads that must skip them, and leave no bound on when the data is physically removed. Lethe, and its extended version Lethe+, add delete-aware compaction policies that bound delete persistence latency to a user-set threshold [18]. They also add a layout that supports range deletes on a secondary key by dropping whole pages. The authors motivate this partly by privacy rules such as GDPR, which require timely deletion [18].

Range tombstones remove many keys with one entry, but they interfere with point lookups, which must check whether a covering range delete exists. GLORAN, a 2025 preprint, proposes a global index over deleted ranges so that lookups find relevant deletions without scanning unrelated entries [19]. Pebble takes a practical approach: delete-only compactions can drop whole SSTables that fall inside a range deletion [9].

5.4 Zoned storage

Flash SSDs run their own garbage collection beneath the block interface. When an LSM-tree's file deletions do not align with the device's erase blocks, the device copies live data internally,

adding a second, hidden layer of write amplification. Zoned Namespace (ZNS) SSDs expose erase-block boundaries and sequential-write rules to the host [20]. Because LSM-trees already write large immutable files sequentially, they map naturally onto zones. Bjørling et al. modified RocksDB to place files in zones and report higher write throughput and lower tail read latency than on a conventional SSD with identical hardware [20].

ZNS moves responsibility to the host, which must group files with similar lifetimes into zones and reclaim zones itself. This links compaction design to device behavior, as Spooky's file sizing also does [14].

5.5 Heterogeneous storage tiers

Data in an LSM-tree has a natural age gradient: recent data sits high in the tree and old data at the bottom. RocksDB's experimental tiered-storage feature uses this, assigning a temperature to the last level so that a custom file system can place it on cheaper media [17]. PrismDB goes further, migrating and compacting data between a fast and a low-cost NVMe tier using a cost-benefit policy adapted from log cleaning [21]. The authors report better throughput and tail latency than RocksDB on flash at equivalent hardware cost. Both approaches show that placement and compaction need to be designed together.

6. Read path and filters

6.1 Bloom filters and their memory allocation

A point lookup may need to check every level. Per-file Bloom filters let the engine skip files that do not contain the key, at the cost of memory. The false-positive rate falls exponentially as bits per key rise, so the memory budget for filters is a central tuning decision. Monkey's result, that filter memory should favor smaller levels, remains the basis for analytical tuning [6].

As SSDs have become faster, filter probing itself can become a CPU bottleneck. Chucky replaces the per-run Bloom filters with a single succinct cuckoo filter that also records which run holds each key [22]. One probe then answers which run to read, at the cost of encoding run identifiers compactly within the filter's fingerprints.

6.2 Range filters

Bloom filters do not help range queries. A short range scan must still seek into every sorted run. REMIX records a globally sorted view across multiple table files, so a range query can binary-search once and then read keys in order [23]. SNARF is a learned range filter: it models the key distribution, maps keys into a compressed bit array and answers range-emptiness queries from it [24]. The authors report lower false-positive rates than earlier range filters at equal memory, and faster execution in RocksDB on some read-only workloads.

GRF builds a single global range filter for an LSM-tree, following Chucky's idea of storing run identifiers inside the filter [25]. Its authors identify correctness under multi-version concurrency control and efficient updates as the main obstacles, and address both with a technique they call shape encoding. These designs show a shift from per-file filters toward structures that span the tree.

6.3 Block cache

The block cache holds data, index and filter blocks in memory. RocksDB's tuning guidance sizes it as a fraction of the memory budget, leaving the rest to the operating-system page

cache [27]. Index and filter blocks compete with data blocks for the cache. Compaction also matters here: it writes new files, so cached blocks for the data it rewrote are no longer useful.

6.4 Learned indexes

Learned indexes predict a key's position from a model of the key distribution (paper 1, Section 4.1). LSM-trees suit this idea because SSTables are immutable: a model trained for a file stays valid until compaction replaces it. Bourbon uses piecewise linear regression to learn key positions within files and a cost-benefit analysis to decide when learning is worthwhile [26]. The authors report faster lookups than a production LSM baseline on their datasets. The key insight is that learning must be priced against file lifetime, because short-lived files at upper levels may not repay the training cost.

7. Tuning in production

7.1 The main levers

RocksDB and Pebble expose many options, but a few groups account for most of the trade-offs.

- **Memory and flushing.** Memtable size and the number of memtables set how much data is buffered before a flush. Larger memtables reduce write amplification but use more memory and lengthen recovery [27].
- **Level shape.** The size ratio between levels, the target size of the first level and dynamic level sizing set write and space amplification [8].
- **Compaction style and concurrency.** The choice between leveled and universal compaction, the number of background jobs and sub-compactions set how fast compaction keeps up [8], [27].
- **Compression.** RocksDB's guidance is to use a fast codec for upper levels and a stronger one for the last level, where most data lives [27].
- **Filters and cache.** Bits per key for filters, block-cache size and whether index and filter blocks are cached [27].
- **I/O smoothing.** A rate limiter on compaction and flush I/O reduces interference with foreground reads [27].

7.2 Write stalls

When flush and compaction cannot keep up with writes, RocksDB slows or stops writes to protect the tree. Its documentation lists three triggers: too many immutable memtables, too many level-0 files, and too many bytes waiting for compaction [27]. A stall affects the whole database, even if only one column family triggered it. Raising the thresholds hides the backlog rather than removing it.

ADOC studies write stalls across several storage devices and argues that they come from data overflow, when one component receives data faster than it can pass it on [28]. It tunes thread counts and batch sizes at run time to balance data flow, and the authors report shorter stall durations than RocksDB's auto-tuned configuration. CockroachDB's "inverted LSM" is the same imbalance seen from the read side [9].

7.3 Robust tuning under uncertainty

Classical tuning assumes that the workload is known. In practice, read-to-write ratios, key distributions and data volumes drift. Endure selects compaction policy, size ratio and memory allocation to maximize worst-case throughput within a neighborhood of the expected workload [29]. An uncertainty parameter sets the size of that neighborhood, from conservative to optimistic. The authors report that robust tunings perform better than nominal ones when the workload shifts, with little loss when it does not. The extended journal version introduces a unified LSM design that generalizes existing ones, and concludes that robustness comes from modeling uncertainty in the tuning process, not from the design alone [30].

7.4 Automated, learned and LLM-assisted tuning

Several groups now apply machine learning to LSM tuning. RusKey uses deep reinforcement learning to choose a compaction policy per level online, together with a modified LSM-tree that makes switching policies cheap [31]. CAMAL combines active learning with classical cost models, tunes parameters separately to reduce training cost, and updates its models as data grows [32]. Both report gains over RocksDB baselines on their workloads.

Large language models (LLMs) are the newest entrant. Thakkar et al. ask whether LLMs can tune and configure LSM-based key-value stores, using RocksDB as the target [33]. ELMo-Tune-V2, a 2025 preprint, extends this to a full cycle: LLM-based workload synthesis, iterative configuration refinement from benchmark feedback, and real-time adjustment [34]. The authors report improvements over default configurations. Two cautions apply. Default configurations are a weak baseline, and an LLM can propose settings that are valid syntactically but unsafe operationally. These methods are best treated as assistants whose proposals are validated before they reach production.

8. Cloud-native and disaggregated LSM engines

8.1 Separating compute and storage

Cloud deployments separate compute nodes from storage services. Nova-LSM splits an LSM-tree store into components that communicate over RDMA, scatters SSTable blocks across storage components and parallelizes compaction across dynamically constructed key ranges [35]. Meta's engineers describe moving RocksDB from local SSDs onto a disaggregated file system [36]. They identify metadata overhead and tail latency as the main gaps and describe how they addressed them.

8.2 Compaction offload and compaction as a service

Compaction is a major background consumer of CPU and I/O that competes with foreground work, which makes offloading it attractive. RocksDB's remote compaction lets a database send compaction jobs to another process or host, which reads the input files, writes output to a temporary location and returns metadata for the primary to install [17]. CaaS-LSM goes further, running compaction as a stateless service with a control plane for scheduling and load balancing and a layer for handling transient failures [37]. The authors built it on RocksDB and evaluated it with two distributed databases in a storage-disaggregated setting.

8.3 LSM-trees on object storage

Object storage offers durable, elastic capacity, but with higher latency and a cost per request. SlateDB places the whole tree on object storage [12]. It batches writes into memtables that are flushed as SSTables to limit request costs, and lets callers choose between waiting for durability and returning sooner. Reads rely on block caches, Bloom filters and local caches of SSTables. This design moves the classic trade-off: write amplification now has a direct monetary cost per request, and latency depends on the batching interval.

9. Comparative analysis

Table 1 compares compaction strategies and related design choices. The ratings are qualitative and relative to each other. Actual behavior depends on parameters such as size ratio, value size and key distribution.

Strategy or design choice	Write amplification	Point-read amplification	Range-read amplification	Space amplification	Best-fit workloads
Leveled, partial (file-granular) merge	High	Low	Low	Low	Read-heavy and mixed workloads; space-constrained deployments
Leveled, full-level merge	Medium	Low	Low	High (transient)	Smaller datasets with spare temporary capacity
Tiered / universal	Low	Medium	High	High	Write-heavy ingest, logs, events and time series
Hybrid (tiered upper levels, leveled last level)	Medium	Low	Medium	Low	Mixed writes and point reads with changing ratios
Granulated merge (Spooky)	Medium	Low	Low	Low	Large leveled datasets on SSD where write and space both matter
Key-value separation (WiscKey, BlobDB)	Low	Medium	Medium to High	Medium to High	Large values; write-heavy updates of large objects
Delete-aware compaction (Lethe)	Medium	Low	Low	Low	Delete-heavy workloads and deadlines for physical deletion

Table 1: Qualitative comparison of compaction strategies and design choices.

10. Open challenges and future opportunities

10.1 Predictable tail latency and write stalls

Challenge. Compaction debt builds silently and surfaces as write stalls or degraded reads, often under a load spike. **Research direction.** Work such as ADOC treats stalls as a data-flow imbalance and adjusts resources at run time [28]. **Opportunity.** Engines could expose

compaction debt as a first-class signal and schedule compaction against latency objectives, not only against size thresholds.

10.2 Tuning complexity and workload drift

Challenge. Engines expose many interacting options, and a tuning that fits today's workload may not fit next month's. **Research direction.** Robust tuning plans for a range of workloads [29], [30], and learned tuners adapt online [31], [32]. **Opportunity.** Combining robust baselines with bounded online adjustment could give operators predictable behavior without constant manual retuning.

10.3 Safety of LLM-assisted tuning

Challenge. LLM-generated configurations can be plausible but harmful, for example by trading durability or memory headroom for throughput. **Research direction.** Current work uses benchmark feedback loops to refine proposals [33], [34]. **Opportunity.** Guardrails such as option allow-lists, invariant checks and staged rollout would let teams use LLMs as advisers while keeping humans and tests in the approval path.

10.4 Economics of compaction offload

Challenge. Offloading compaction frees the database node but adds network transfer, remote CPU and failure modes. **Research direction.** Remote compaction in RocksDB and compaction-as-a-service designs address scheduling and error handling [17], [37]. **Opportunity.** Cost models that price compaction per byte moved and per request, especially on object storage [12], would help decide when offload pays for itself.

10.5 Tiered and heterogeneous storage

Challenge. Storage is becoming more varied: ZNS SSDs, low-cost QLC flash, object storage and, in prospect, memory pooled over Compute Express Link (CXL). **Research direction.** ZNS work aligns files to zones [20], PrismDB co-designs placement and compaction across tiers [21], and RocksDB places its last level by temperature [17]. **Opportunity.** CXL-attached memory could hold memtables, filters or caches beyond local DRAM, but we found no peer-reviewed LSM-specific evaluation to cite. This remains an open research area.

10.6 Range deletes and privacy deletion

Challenge. Tombstones leave no bound on when data is physically removed, and range tombstones slow point lookups. **Research direction.** Delete-aware compaction bounds persistence latency [18], and global range-delete indexes reduce lookup interference [19]. **Opportunity.** Engines that can state and enforce a deletion deadline would support privacy obligations directly, rather than relying on periodic full compactions.

10.7 Verification and integrity

Challenge. Compaction rewrites data continuously, so a bug or hardware fault can corrupt data that was correct when written. RocksDB's developers conclude that corruption must be detected earlier and at every layer [3], [4]. **Research direction.** Disaggregation and compaction offload widen the set of components that handle data [36], [37]. **Opportunity.** End-to-end checksums across offloaded compaction, and verifiable compaction outputs, are natural areas for further work.

10.8 Learned and adaptive structures

Challenge. Learned indexes and filters must be retrained when compaction replaces files, and their benefits depend on the key distribution. **Research direction.** Bourbon prices learning against file lifetime [26], and SNARF and GRF bring learned and global designs to range filtering [24], [25]. **Opportunity.** Compaction itself could build models as it writes files, since it already reads every key in order.

11. What this means for platform decisions

No single compaction strategy fits every workload, so engine choice and tuning follow from the read, write and space profile of each workload. In Bassirah's platform work, distributed databases that rely on LSM storage include YugabyteDB, which Bassirah supports through independent open-source engineering, and CockroachDB or TiKV-based systems, which appear here as ecosystem references, while ClickHouse, a Strategic Technology Partner, uses its own MergeTree engine, which merges sorted data parts in the background [38]. S3-compatible object storage, including MinIO AIStor, a Strategic Technology Partner, is the relevant layer for object-storage LSM designs such as SlateDB [12]. In each case, settings are validated on a representative workload before adoption.

References

1. O'Neil, P., Cheng, E., Gawlick, D. and O'Neil, E. (1996). The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 351–385. (<https://doi.org/10.1007/s002360050048>)
2. Luo, C. and Carey, M. J. (2020). LSM-based Storage Techniques: A Survey. *The VLDB Journal*, 29(1), 393–418. (<https://doi.org/10.1007/s00778-019-00555-y>)
3. Dong, S., Kryczka, A., Jin, Y. and Stumm, M. (2021). Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. *USENIX FAST*. (<https://www.usenix.org/conference/fast21/presentation/dong>)
4. Dong, S., Kryczka, A., Jin, Y. and Stumm, M. (2021). RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage*, 17(4), Article 26. (<https://doi.org/10.1145/3483840>)
5. Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A. and Callaghan, M. (2016). Designing Access Methods: The RUM Conjecture. *EDBT*. (<https://doi.org/10.5441/002/edbt.2016.42>)
6. Dayan, N., Athanassoulis, M. and Idreos, S. (2017). Monkey: Optimal Navigable Key-Value Store. *SIGMOD*. (<https://doi.org/10.1145/3035918.3064054>)
7. Dayan, N. and Idreos, S. (2018). Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. *SIGMOD*. (<https://doi.org/10.1145/3183713.3196927>)
8. RocksDB project (2026). RocksDB Wiki: Leveled Compaction; Universal Compaction. Official documentation, accessed 26 October 2026. (<https://github.com/facebook/rocksdb/wiki/Leveled-Compaction>; <https://github.com/facebook/rocksdb/wiki/Universal-Compaction>)
9. Cockroach Labs (2026). CockroachDB Documentation: Storage Layer; Pebble README. Official documentation, accessed 26 October 2026.

- (<https://docs.cockroachlabs.com/docs/stable/architecture/storage-layer>;
<https://github.com/cockroachdb/pebble>)
10. TiKV project (2026). TiKV Deep Dive: RocksDB. Official documentation, accessed 26 October 2026. (<https://tikv.org/deep-dive/key-value-engine/rocksdb/>)
 11. YugabyteDB, Inc. (2026). DocDB Storage Layer; Enhancing RocksDB for Speed and Scale. Official documentation and engineering blog, accessed 26 October 2026. (<https://docs.yugabyte.com/stable/architecture/docdb/>;
<https://www.yugabyte.com/blog/enhancing-rocksdb-for-speed-scale/>)
 12. SlateDB project (2026). SlateDB: an embedded storage engine built on object storage. Official repository and documentation, accessed 26 October 2026. (<https://github.com/slatedb/slatedb>; <https://slatedb.io>)
 13. Sarkar, S., Staratzis, D., Zhu, Z. and Athanassoulis, M. (2021). Constructing and Analyzing the LSM Compaction Design Space. PVLDB, 14(11), 2216–2229. (<https://doi.org/10.14778/3476249.3476274>)
 14. Dayan, N., Weiss, T., Dashevsky, S., Pan, M., Bortnikov, E. and Twitto, M. (2022). Spooky: Granulating LSM-Tree Compactions Correctly. PVLDB, 15(11), 3071–3084. (<https://doi.org/10.14778/3551793.3551853>)
 15. Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2016). WiscKey: Separating Keys from Values in SSD-conscious Storage. USENIX FAST. (<https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu>)
 16. Li, Y., Liu, Z., Lee, P. P. C., Wu, J., Xu, Y., Wu, Y., Tang, L., Liu, Q. and Cui, Q. (2021). Differentiated Key-Value Storage Management for Balanced I/O Performance. USENIX ATC. (<https://www.usenix.org/conference/atc21/presentation/li-yongkun>)
 17. RocksDB project (2026). RocksDB Wiki: BlobDB; Remote Compaction; Tiered Storage (Experimental). Official documentation, accessed 26 October 2026. (<https://github.com/facebook/rocksdb/wiki/BlobDB>;
<https://github.com/facebook/rocksdb/wiki/Remote-Compaction>;
[https://github.com/facebook/rocksdb/wiki/Tiered-Storage-\(Experimental\)](https://github.com/facebook/rocksdb/wiki/Tiered-Storage-(Experimental)))
 18. Sarkar, S., Papon, T. I., Staratzis, D., Zhu, Z. and Athanassoulis, M. (2023). Enabling Timely and Persistent Deletion in LSM-Engines. ACM Transactions on Database Systems, 48(3). Extended version of Lethe (SIGMOD 2020). (<https://disc.bu.edu/papers/tods2023-sarkar>)
 19. Wang, F., Mo, D. and Luo, S. (2025). Don't Forget Range Delete! Enhancing LSM-based Key-Value Stores with More Compatible Lookups and Deletes. arXiv preprint 2511.06061. (<https://arxiv.org/abs/2511.06061>)
 20. Bjørling, M., Aghayev, A., Holmberg, H., Ramesh, A., Le Moal, D., Ganger, G. R. and Amvrosiadis, G. (2021). ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. USENIX ATC. (<https://www.usenix.org/conference/atc21/presentation/bjorling>)
 21. Raina, A., Lu, J., Cidon, A. and Freedman, M. J. (2023). Efficient Compactions between Storage Tiers with PrismDB. ASPLOS. (<https://doi.org/10.1145/3582016.3582052>)
 22. Dayan, N. and Twitto, M. (2021). Chucky: A Succinct Cuckoo Filter for LSM-Tree. SIGMOD. (<https://doi.org/10.1145/3448016.3457273>)
 23. Zhong, W., Chen, C., Wu, X. and Jiang, S. (2021). REMIX: Efficient Range Query for LSM-trees. USENIX FAST. (<https://www.usenix.org/conference/fast21/presentation/zhong>)
 24. Vaidya, K., Chatterjee, S., Knorr, E., Mitzenmacher, M., Idreos, S. and Kraska, T. (2022). SNARF: A Learning-Enhanced Range Filter. PVLDB, 15(8), 1632–1644. (<https://doi.org/10.14778/3529337.3529347>)

25. Wang, H., Guo, T., Yang, J. and Zhang, H. (2024). GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(3). (<https://doi.org/10.1145/3654944>)
26. Dai, Y., Xu, Y., Ganesan, A., Alagappan, R., Kroth, B., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2020). From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. *USENIX OSDI*. (<https://www.usenix.org/conference/osdi20/presentation/dai>)
27. RocksDB project (2026). RocksDB Wiki: Setup Options and Basic Tuning; Write Stalls. Official documentation, accessed 26 October 2026. (<https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning>; <https://github.com/facebook/rocksdb/wiki/Write-Stalls>)
28. Yu, J., Noh, S. H., Choi, Y. and Xue, C. J. (2023). ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. *USENIX FAST*. (<https://www.usenix.org/conference/fast23/presentation/yu>)
29. Huynh, A., Chaudhari, H. A., Terzi, E. and Athanassoulis, M. (2022). Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *PVLDB*, 15(8), 1605–1618. (<https://doi.org/10.14778/3529337.3529345>)
30. Huynh, A., Chaudhari, H. A., Terzi, E. and Athanassoulis, M. (2024). Towards Flexibility and Robustness of LSM Trees. *The VLDB Journal*, 33(4). (<https://doi.org/10.1007/s00778-023-00826-9>)
31. Mo, D., Chen, F., Luo, S. and Shan, C. (2023). Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proceedings of the ACM on Management of Data (SIGMOD)*, 1(3). (<https://doi.org/10.1145/3617333>)
32. Yu, W., Luo, S., Yu, Z. and Cong, G. (2024). CAMAL: Optimizing LSM-trees via Active Learning. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(4). (<https://doi.org/10.1145/3677138>)
33. Thakkar, V., Sukumar, M., Dai, J., Singh, K. and Cao, Z. (2024). Can Modern LLMs Tune and Configure LSM-based Key-Value Stores? *ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*. (<https://doi.org/10.1145/3655038.3665954>)
34. Thakkar, V., Lin, Q., Prasetyo, K. K. A., Wisjunandono, R. H., Kistijantoro, A. I., Rachmadi, R. F. and Cao, Z. (2025). ELMo-Tune-V2: LLM-Assisted Full-Cycle Auto-Tuning to Optimize LSM-Based Key-Value Stores. *arXiv preprint 2502.17606*. (<https://arxiv.org/abs/2502.17606>)
35. Huang, H. and Ghandeharizadeh, S. (2021). Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. *SIGMOD*. (<https://doi.org/10.1145/3448016.3457297>)
36. Dong, S., Shankar P, S., Pan, S., et al. (2023). Disaggregating RocksDB: A Production Experience. *Proceedings of the ACM on Management of Data (SIGMOD)*, 1(2). (<https://doi.org/10.1145/3589772>)
37. Yu, Q., Guo, C., Zhuang, J., Thakkar, V., Wang, J. and Cao, Z. (2024). CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(3). (<https://doi.org/10.1145/3654927>)
38. ClickHouse (2026). MergeTree Table Engine. Official documentation, accessed 26 October 2026. (<https://clickhouse.com/docs/engines/table-engines/mergetree-family/mergetree>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.