Lessons from Operating a Large-Scale Distributed Database with a Replicated Key-Value Engine

Jung-Sang Ahn Navaneetha Krishnan Thanka Nadar* Sami Ben-Romdhane
eBay Inc.

Abstract

Modern distributed databases frequently rely on key-value storage engines as durable backends for replicated state machines. While prior work extensively studies storage engine internals, far less attention has been given to the system-level constraints that arise when such engines are deployed at scale under strict resource limits. In production, storage design directly impacts tail latency, availability, recovery time, and multi-tenant stability.

This paper presents operational insights from running a log-structured merge-tree (LSM-tree)-derived key-value engine as the storage foundation of eBay's cloud-native distributed database, which serves trillions of requests per day. We describe practical challenges and operational responses in snapshot binding and installation, scan overhead caused by visibility checks and predicate evaluation, CPU-constrained compaction, write amplification and noisy-neighbor effects in multi-tenant environments, and programmable compaction for unified maintenance. These experiences show that when the storage layer underpins a distributed database, priorities shift from peak throughput to latency stability, predictable recovery, and bounded background work. Some mechanisms in our system rely on properties of our storage engine, but the underlying integration lessons are more broadly applicable to replicated database deployments.

1 Introduction

Modern large-scale services increasingly rely on replicated key-value stores as the foundation of their storage stack. In these systems, the storage engine is not merely a local performance component; it serves as the replicated state machine underlying consensus protocols such as Paxos [29, 30, 32] and Raft [35], in the tradition of state machine replication [40]. As a result, design choices in the storage layer directly affect replication correctness, recovery behavior, tail latency, and overall system stability.

This paper describes our experience operating a production Raft-based distributed database deployed at scale within eBay. The system consists of hundreds of thousands of Raft groups, serves trillions of requests per day, and runs in a geo-distributed, multi-tenant, multi-availability-zone cloud environment orchestrated by Kubernetes [3]. Our team is responsible for the full stack, covering front-end database logic and transaction processing as well as the backend key-value storage engine and its integration with Raft. This end-to-end responsibility provided visibility into cross-layer interactions that are often hidden in modular deployments.

In this architecture, the storage engine is the persistent backend of multiple replicated databases. Local storage decisions therefore become database-level concerns. We encountered four recurring challenges:

- Raft snapshots and log maintenance must be lightweight and predictable, because they determine recovery, replica replacement, and state-transfer time.
- A minimal key-value API is too narrow for database semantics, causing scan amplification from visibility, multi-version concurrency control (MVCC) [10], and predicate filtering.
- Background maintenance competes with foreground traffic and must not become tail latency.
- Write amplification creates shared-disk interference and can turn replicas into noisy neighbors.

The common pattern behind these challenges is that storage behavior becomes visible at the replicated-database layer. A local decision that appears acceptable for a standalone engine can still lengthen recovery, consume shared resources, or turn rare tail cases into observed availability loss. Our experience therefore shifted the optimization target from peak local throughput to latency stability, predictable recovery, and bounded background work.

Over more than a decade of production deployment, we introduced mechanisms including explicit snapshot-log index binding, iterator repositioning to mitigate MVCC amplification, decoupling read latency from compaction, and

*Work done while at eBay; currently at Google.

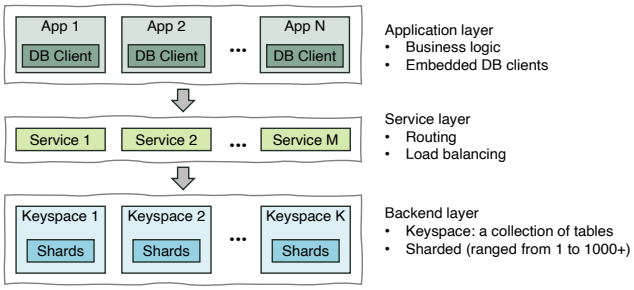


Figure 1: End-to-end architecture.

programmable compaction policies that adapt to workload and replication state. We do not present these mechanisms as isolated new algorithms. Instead, this paper distills our production experience combining these mechanisms in an integrated storage and replication stack. From this experience, we identify cross-layer lessons for operating storage engines as replicated state-machine backends under recovery, replication, and multi-tenant resource constraints. We hope these insights help practitioners avoid common pitfalls in replicated storage deployments and inform future storage-engine designs for large-scale distributed databases.

2 System Overview

Our distributed database platform consists of multiple database services that share a common replication and storage substrate. We first describe the end-to-end deployment architecture. We then describe NuKVCore, the shared library that integrates replication and storage within each database server process.

2.1 End-to-End Architecture

Our platform serves tens of thousands of applications that execute diverse business logic, as illustrated in Figure 1. The database client is provided as a library and embedded directly into each application process, allowing tight integration with application-level retries, timeouts, and consistency semantics.

At the front end, tens of thousands of stateless database service instances act as entry points. Applications maintain connections to multiple service instances and distribute requests among them. The service layer performs request routing and load balancing and forwards requests to the appropriate backend entities. Because it is stateless, the service layer can be elastically scaled and replaced without coordination.

The backend is organized into thousands of keyspaces. A keyspace is the fundamental database unit and represents a collection of tables. Each keyspace is partitioned into shards using either consistent hashing [26] or key-range partitioning, as shown in Figure 2. Shards are disjoint and do not share data. In production, the number of shards per keyspace varies widely depending on workload and data size. Some small keyspaces consist of a single shard, while large keyspaces may

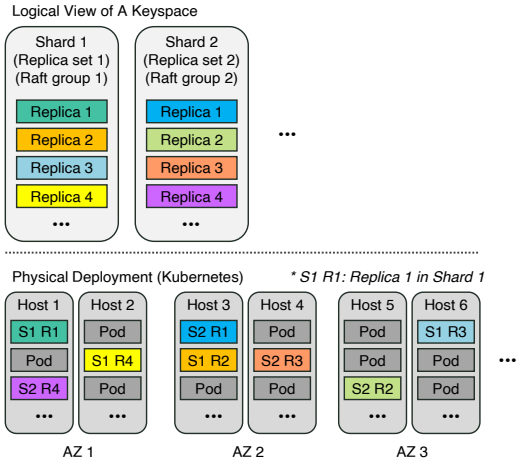


Figure 2: Logical view and physical deployment of a keyspace across multiple AZs (availability zones).

span thousands of shards, with the majority falling in the range of several tens of shards. The shard count of a keyspace is not fixed and can be adjusted online to accommodate workload growth or contraction.

Each shard corresponds to a replica set and forms a single Raft group. One replica serves as the leader and the remaining replicas act as followers. The leader handles all writes and strongly consistent reads, while followers may serve eventually consistent reads. Writes commit through quorum agreement and are considered durable once a majority of replicas acknowledge the entry.

Replicas are deployed on Kubernetes, with each replica running in its own pod. Our deployment runs in eBay-operated data centers rather than a public cloud environment, although the architectural principles described here are not specific to a particular infrastructure provider. Replicas in a replica set are distributed across at least three availability zones. A committed write is therefore persisted in at least two availability zones, allowing the system to tolerate a complete outage of one zone without data loss or service interruption. Anti-affinity rules further ensure that replicas within the same zone are placed on different hosts and racks to limit correlated hardware failures.

While this architecture defines how data is partitioned, replicated, and deployed, all backend database servers rely on a shared replication and storage substrate described next.

2.2 NuKVCore: A Shared Replication and Storage Substrate

Within eBay, we operate multiple database services that target different data models and access patterns, including a document store with SQL-like queries and its secondary index store, a pure key-value store, edge and vertex stores for a graph database, a vector store, and internal metadata stores. Although these systems expose different query semantics,

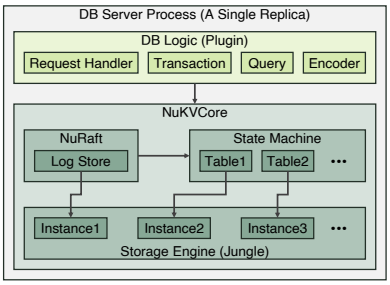


Figure 3: NuKVCore overview.

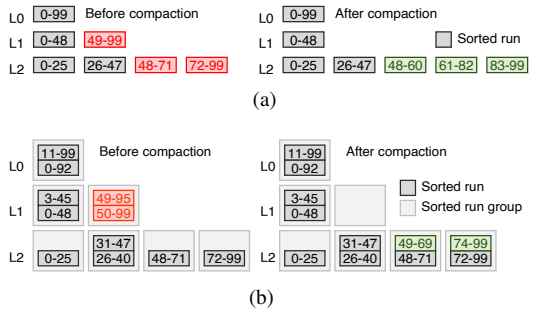


Figure 4: (a) Leveled LSM-tree compaction. (b) Tiered LSM-tree compaction. Red boxes are inputs; green boxes are outputs.

they share common requirements for replication, durability, recovery, and storage management.

Implementing replication and storage logic independently for each database would be inefficient to build and difficult to maintain. We therefore modularized the shared functionality into a common core library called NuKVCore, illustrated in Figure 3. NuKVCore is embedded into each database server process, where each process hosts a replica participating in a Raft group. Database-specific logic plugs into NuKVCore through a well-defined interface and is responsible for request handling, query planning and execution, key and value encoding and decoding, and defining what must be replicated and how committed operations should be applied.

NuKVCore provides two foundational components: replication and local persistence. Replication is implemented using the Raft consensus protocol via NuRaft [5]. Local persistence is provided by Jungle [2, 7], a key-value storage engine based on a tiered or hybrid LSM design [15, 16, 18, 36].

In LSM-based storage engines, sorted runs are immutable

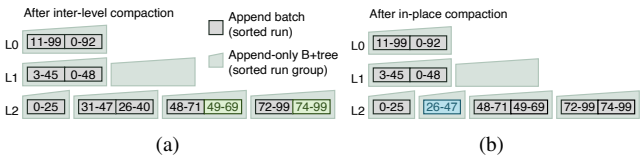


Figure 5: (a) Jungle inter-level compaction. (b) Jungle in-place compaction.

sorted sequences of key-value entries produced by flushing or compaction, typically stored as SSTables, a design originating from Bigtable [12] and widely adopted in systems such as LevelDB [4] and RocksDB [6]. In leveled LSM designs [4, 6], each level is organized as a set of non-overlapping sorted runs. When data from level `N` is compacted into level `N+1`, it is merged with all sorted runs in level `N+1` whose key ranges overlap the input range, and the merged entries are rewritten as new sorted runs, as illustrated in Figure 4(a).

In tiered LSM variants [6, 15, 16, 36], a single level may contain multiple sorted runs whose key ranges overlap. We refer to the set of overlapping runs within a level as a *sorted run group*. At compaction time, sorted runs in a sorted run group at level `N` are merged, and the output is added as one new sorted run to one or more overlapping sorted run groups at level `N+1`, as shown in Figure 4(b). Unlike leveled compaction, this process does not immediately rewrite the existing runs at the next level. This reduces immediate rewrite cost, but shifts part of the cost to reads: point lookups and range scans may need to consult and merge multiple overlapping sorted runs at query time.

Jungle replaces each sorted run group with a single append-only B+tree, also known as a copy-on-write (CoW) B+tree [37, 38], while preserving the logical behavior of a tiered LSM. In this organization, each sorted run in the group is represented as an append batch in the append-only B+tree. New compaction output can therefore be incorporated by adding append batches, without immediately rewriting existing data in the same group. Unlike a conventional tiered LSM, however, reads do not merge overlapping sorted runs at query time. Instead, the append-only B+tree directly indexes the latest entry for each key, so read cost does not grow with the number of append batches in the group.

Jungle performs compaction at two granularities. Inter-level compaction moves data between adjacent levels, as in a tiered LSM design, but writes the output as append batches into the corresponding append-only B+trees at level `N+1`, as illustrated in Figure 5(a). In-place compaction rebuilds a single append-only B+tree to reclaim stale data, as shown in Figure 5(b): live keys are rewritten into a new B+tree covering the same key range, and the old B+tree is discarded. The detailed design and evaluation of Jungle are described in prior work [7]; this paper focuses on the operational lessons that emerge in production deployments.

A single NuKVCore instance may host multiple table-backed storage instances, each backed by an independent Jungle instance representing an isolated LSM-tree. Commonly, one storage instance corresponds to one table, but the mapping is flexible and determined by database-specific layout decisions. For example, one logical table may be mapped to multiple instances, or multiple logical tables may share a single instance.

In addition to table-backed instances, NuKVCore maintains a special-purpose storage instance used as the Raft log

store. The log store persists the ordered sequence of Raft log entries and provides an interface for reading and writing opaque binary payloads associated with log index numbers. We describe the log store requirements and optimizations in Section 3.1.

Write Path When a write request arrives, the database logic constructs the data to be replicated and submits it to NuKVCore as an opaque payload. NuKVCore appends the payload to the Raft log, persists it in the log store, and replicates it to followers. Once the entry is committed by quorum agreement, the database logic deterministically applies the operation through NuKVCore's key-value interface.

A committed log entry may update one or more table-backed storage instances, but the same apply procedure is executed on every replica in the Raft group, ensuring convergence to a consistent shard state.

Read Path For read requests, the database logic issues operations through the same key-value interface without appending new log entries in the common case. Strongly consistent reads are served by the leader, while eventually consistent reads may be handled by followers. In cases where reads trigger state changes, such as metadata updates or materialization of derived data, the request follows the same replication and apply procedure as writes.

3 Key-Value Storage as the State Machine of a Distributed Database

This section focuses on the storage-engine mechanisms required when a key-value engine backs a Raft state machine: durable log storage, snapshots bound to committed log indexes, and efficient bulk installation of those snapshots during recovery and reconfiguration.

3.1 Log Store Mode

The log store maintains an ordered mapping from log index to opaque payload. It must support the following operations:

1. Append of `<log index, payload>` pairs, and occasional overwrite during leader changes.
2. Sequential reads over a contiguous range of log indexes during follower catch-up.
3. Rollback to a specific log index by removing the most recent entries.
4. Compaction up to a given log index by removing the oldest retained entries.

The log store has a much simpler access pattern than the state store: writes are almost entirely sequential appends, reads are mostly sequential during follower catch-up, and deletions occur only at the head or tail for log compaction or rollback. It does not require random point deletion, arbitrary range updates, or secondary indexing.

Persisting log entries through a general-purpose LSM configuration, with the log index as the key, would be functionally

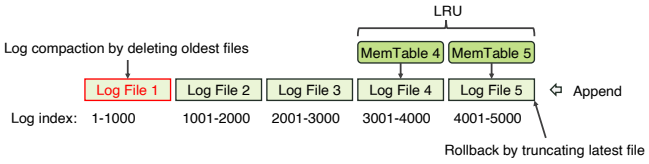


Figure 6: Log store mode.

correct but operationally wasteful. Flushes and merges optimized for general key-value workloads would still run, and rollback or head compaction would require key iteration and per-entry deletion, creating bursty I/O on a latency-sensitive replica. A separate append-only log engine could avoid this overhead, but at our scale of hundreds of thousands of Raft groups it would add another persistence subsystem to configure, tune, debug, and operate. We therefore keep a unified storage substrate while specializing one mode for Raft's append and range-truncation semantics.

The key observation is that an LSM engine already contains the components needed for this workload. Updates are appended to a write-ahead log and indexed in a memtable before later being flushed into sorted runs [4, 6, 12, 34]. For a Raft log store, the durable append path and in-memory index are useful, but the sorted-run hierarchy is unnecessary.

Jungle implements this specialization as a dedicated log-store mode, shown in Figure 6. In this mode, the sorted-run component is disabled, and only the log and memtable layers remain active. Unlike a general LSM-tree, which bounds log growth by flushing data into sorted runs, log-store mode allows log files to accumulate and manages them directly according to Raft log semantics.

Because the log store maintains large volumes of data in production, typically between 5 GB and 100 GB per instance, it is infeasible to keep all corresponding memtables in memory. To bound memory usage, memtables that have not been accessed recently are evicted using an LRU policy. When data from an evicted log file is needed again, the log file is scanned to reconstruct the memtable. Since log reads are sequential by nature, once a log file is loaded, its contents are read in order and fully consumed.

Log files are segmented such that each file covers a contiguous range of log indexes, and the log store maintains lightweight metadata for segment boundaries. Rollback and overwrite are implemented by truncating the tail: segments whose index range lies entirely beyond the target index are dropped, and the last remaining segment is truncated to the byte offset of the target entry when needed. Compaction is implemented symmetrically at the head by deleting entire segments whose maximum index is below the compaction point, enabling immediate space reclamation without key iteration.

Table 1 reports log compaction latency for individual replica log stores of varying sizes and payload characteristics, based on over a month of production metrics. Table 4 provides additional details for these cases and for all other labeled cases reported throughout the paper.

Log stores				Latencies (ms)		
Case	Size	# logs	Interval	p50	p99	p99.9
A	5.1 GB	2.6M	2 hours	0.63	24.70	48.20
B	23 GB	42M	5 hours	1.45	6.68	25.60
C	41 GB	49M	2 hours	3.23	14.82	110.40
D	90 GB	84M	3 hours	2.90	7.00	43.29
E	90 GB	29M	1.5 hours	2.73	6.93	28.20

Table 1: Observed log compaction latency and interval in production for individual replica log stores of varying sizes and payload characteristics, based on over a month of production metrics.

The reported latency is the running time of each background log compaction job, not foreground request latency. In each case, a compaction removes approximately 10% of the total log entries in that replica. For example, Case A performs log compaction roughly once every two hours, and the p99.9 running time of an individual job is 48.2 ms, occupying only a negligible fraction of the two-hour interval. The goal of this design is therefore to keep each background reclamation job short, thereby minimizing interference with foreground state-machine operations.

Across all evaluated configurations, median job running time remains within a few milliseconds, and p99.9 running time stays below 200 ms. This indicates that per-replica log compaction introduces only brief background activity rather than long-running maintenance work.

By specializing the existing log-oriented components of the LSM engine for Raft's append and range-truncation semantics, we meet the durability and ordering requirements of consensus without introducing a separate storage engine. The result is a unified persistence substrate that keeps log maintenance operationally simple and bounded.

3.2 Lightweight Persistent Snapshots

Beyond the log store, a replicated state machine requires persistent snapshot support. In Raft, a snapshot captures the complete shard state at a specific log index. Each snapshot is bound to a log index and represents the state after applying all committed entries up to that point.

Snapshots are primarily used when a replica cannot catch up from retained logs alone. If a follower's most recent log index is no older than the leader's oldest retained log index, it can recover by replaying the missing entries; otherwise, the leader must transmit a snapshot bound to log index `X`, after which the follower resumes replay from `X+1`. The same requirement appears in replica replacement, membership reconfiguration, online resharding, and backup restoration: before normal replication can resume, the replica must first be brought to a consistent state at a specific log index.

Because snapshots may be generated frequently, in some workloads as often as every few tens of seconds, they must be lightweight. Snapshot creation must not block writes, trigger excessive disk I/O, or introduce sudden resource spikes in a latency-sensitive production environment. Snapshots must also be durable: once a snapshot bound to a given log index is created, it must remain accessible across restarts until it is safely obsolete.

Keyspaces				Latencies (ms)		
Case	Size	# keys	Interval	p50	p99	p99.9
A	13 GB	9.8M	4 mins	4.70	68.75	296.37
B	65 GB	56M	30 secs	9.41	22.20	28.49
C	260 GB	355M	20 secs	41.39	422.69	883.91
F	500 GB	1.4B	50 secs	8.15	22.10	157.64

Table 2: Observed snapshot creation latency and interval in production for representative shard sizes (per-replica size), based on over a month of production metrics.

To make snapshot binding precise and inexpensive, each replica maintains a monotonically increasing sequence number that orders mutations applied to its local state stores. Every mutation applied to any table-backed storage instance is assigned a unique sequence number at apply time after commit, ensuring that each sequence boundary corresponds exactly to a state transition produced by applying a committed log entry.

The append-only B+tree structure aligns naturally with this sequence abstraction. Each append batch is associated with a sequence boundary `S`, whose root represents the state of the storage instance at that boundary. When Raft requests a snapshot at log index `X`, the storage layer opens the corresponding snapshot view using the sequence number `S` bound to `X`. Similar to copy-on-write filesystem snapshots such as WAFL [23], ZFS [11], and Btrfs [38], this operation selects a stable root to obtain a point-in-time view without copying the full data set, while avoiding iterative merging across overlapping components.

If Raft requests a snapshot at a sequence number S_2 that lies between two batch boundaries S_1 and S_3, the system opens the snapshot at S_1 and overlays updates in the interval $(S_1, S_2]$. Each storage instance maintains a hidden index ordered by sequence number to enumerate keys updated within that interval. These updates are merged onto the S_1 snapshot using a bounded iterative merge.

When Raft requests snapshot creation at log index `X`, NuKV-Core records a mapping between `X` and the current sequence number `S`. Before persisting this mapping, the storage layer issues an `fsync` to flush all mutations up to `S`. Snapshot creation thus consists only of persisting the log-index-to-sequence binding; no data copying or full scans are required.

Table 2 reports snapshot creation latency for representative shard sizes, based on over a month of production metrics. The reported size corresponds to one replica of a shard, whose replicas have identical data size. As in Table 1, the reported latency is the running time of each background snapshot-creation job, not foreground request latency. These jobs run periodically, with intervals ranging from tens of seconds to several minutes depending on the shard, and in most cases complete within sub-second latency even for large shards.

This lightweight creation path still requires coordination with later storage maintenance. Background compaction must respect snapshot pinning: in-place compaction may remove

Keyspace	Size	# keys	Regular Write	Bottom-Up	Ratio
G	16 GB	26M	00:08:28	00:02:57	0.348
H	80 GB	107M	03:52:15	00:18:18	0.079
I	140 GB	201M	05:53:02	00:33:19	0.094
J	140 GB	295M	03:22:43	00:37:26	0.185
K	450 GB	410M	03:34:49	01:34:07	0.438

Table 3: Per-replica total recovery time (snapshot transfer + log replay) across five production keyspaces, reported in `hh:mm:ss` with their ratio.

stale versions, but versions referenced by active snapshot sequence numbers cannot be discarded. Compaction therefore consults the set of retained snapshot boundaries when determining safe version reclamation.

3.3 Efficient Snapshot Installation via Bottom-Up Index Build

When a snapshot is transmitted, a receiving replica must reconstruct its entire state machine efficiently across all table-backed storage instances. Snapshot transmission occurs as a stream of messages carrying key-value pairs that represent the state of the snapshot bound to a given log index.

A straightforward implementation on the snapshot receiver side would insert each received key-value pair using the regular key-value write API. However, routing bulk installation through the regular mutation path triggers flush work and background merge or compaction activity. Snapshot installation is inherently a bulk-loading workload, and treating it as a sequence of independent point updates results in unnecessary write amplification and poor throughput.

Instead, the receiver constructs storage structures directly from sorted input using a bottom-up index build. This requires the sender to provide records in strict key order with each key appearing exactly once. Because a snapshot may span multiple storage instances, each with its own independent index, these guarantees are enforced per storage instance rather than globally across the entire state machine. The sender therefore streams snapshot data separately for each storage instance, avoiding ambiguity when identical user keys appear in different instances.

The snapshot sender generates snapshot data by opening a snapshot view at the sequence number bound to the requested log index. It then iterates over each storage instance independently in key order at that sequence boundary and emits keys in strict lexicographical order within that instance. Snapshot data is typically further segmented by contiguous key ranges within each instance to control message size. Each message therefore carries a list of key-value pairs corresponding to a specific (storage instance, key-range) slice of the snapshot.

Keys within each storage instance are transmitted in order and appear exactly once, allowing the receiver to accumulate key-value pairs until a predefined size threshold is reached and then construct an append-only B+tree bottom-up, with the received entries forming the leaf nodes. The resulting B+tree is registered into the appropriate storage instance for the corresponding key range. This lets the receiver bypass the regular write path and install the snapshot as a sequential write workload, without triggering flush, merge, or in-place compaction work. Correctness follows from the existing log-index-to-sequence binding: the installed structures represent the state at log index `X`, and subsequent replay starts from `X+1` through the regular write path.

Table 3 compares per-replica total recovery time across five representative production cases. Total recovery time includes snapshot transfer over the network and catch-up of Raft logs accumulated during transfer. Longer snapshot installation also allows more Raft log entries to accumulate during transfer, increasing the subsequent catch-up time roughly linearly with the installation duration. We report recovery time for the regular write path and the bottom-up index build, together with their ratio. In production, bottom-up index build substantially reduces recovery time, with ratios ranging from 0.438 to 0.079 depending on workload characteristics. Despite workload variability, the bottom-up approach consistently shortens recovery time across all evaluated configurations. Because a replica must reach a consistent log index before resuming normal operation, reducing recovery time shortens the critical path of state transfer workflows, including membership reconfiguration, online resharding, and large-scale replica replacement in production deployments.

4 Operating Storage Engines Under Production Resource Constraints

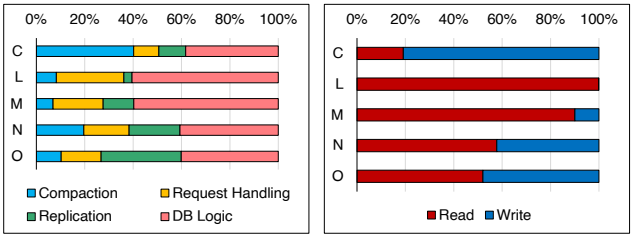
4.1 Small-Node Deployments with Limited CPU

Our production deployment follows a small-resource, large-node architecture: instead of a few high-capacity servers, we deploy many servers with limited resources. This reduces failure blast radius and improves horizontal scalability through finer shard distribution.

The trade-off is strict per-node CPU limits. In production, each replica is provisioned with only two to three CPU cores, most of which are consumed by request processing, replication, and database logic such as transaction handling and query execution. Consequently, the storage engine must operate within a narrow CPU envelope. Background tasks such as compaction are typically limited to roughly one core per replica while maintaining multiple tables concurrently.

Figure 7 shows CPU utilization across five representative workloads, based on over a month of production metrics. Even in the most write-heavy case, compaction accounts for only about 40% of total CPU usage, and in most workloads its share is substantially smaller. Compaction therefore cannot aggressively consume CPU resources, motivating CPU-aware rather than throughput-maximizing policies.

In LSM-based systems, delayed compaction increases disk usage and read cost, reflecting the fundamental read-update trade-offs studied in prior work [8, 15, 39, 44]. While tem-



(a) CPU usage breakdown. (b) Read and write ratio.

Figure 7: CPU utilization breakdown across five keyspaces with different read-write ratios, based on over a month of production metrics.

porary disk growth is tolerable, increased read cost directly affects tail latency. In our production environment, requests exceeding timeout thresholds are treated as failures; thus tail latency directly impacts observed availability. Even moderate read amplification caused by compaction backlog can degrade availability.

Throughput in distributed systems can often be increased by adding shards or replicas. Tail latency, however, does not scale in the same way and is frequently dominated by local resource contention and background maintenance [19, 20]. Under strict CPU limits, protecting tail latency becomes the primary objective.

4.2 Noisy Neighbor Effects and Disk-Level Interference

Our system runs in a Kubernetes-based environment where CPU and memory resources are controlled via cgroups, limiting interference from co-located pods. Disk resources, however, remain shared.

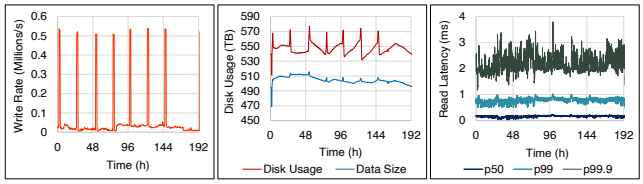
If other pods generate heavy I/O traffic, disk latency can increase significantly, introducing the well-known noisy neighbor problem at the storage level [20,43]. In such environments, background storage activities that generate large volumes of disk I/O can negatively affect both the local replica and neighboring workloads.

Traditional leveled LSM designs perform frequent cross-level merges that substantially increase write amplification and generate large amounts of disk I/O. In multi-tenant clusters, such merge activity can create bursty background traffic and turn a replica into a noisy neighbor, causing latency spikes for co-located workloads.

Reducing write amplification is therefore not merely a performance optimization but a stability requirement in shared environments.

4.3 Decoupling Read Latency from Compaction

Our design addresses both constraints by combining a tiered LSM organization with append-only B+tree structures.



(a) Write Rate. (b) Disk usage. (c) Read Latency.

Figure 8: Eight consecutive days of production behavior for an illustrative keyspace.

First, we adopt a tiered LSM structure to significantly reduce cross-level merge activity and lower write amplification. This reduces background disk I/O and mitigates noisy neighbor effects in multi-tenant deployments.

However, tiered LSM structures allow multiple runs to accumulate within a level, which can increase read cost and introduce latency variability. To eliminate this effect, we leverage the structural property described in Section 2.2: each logical sorted run group is represented as a single append-only B+tree. As a result, reads do not require iterative merging across overlapping runs, and read cost does not grow with compaction backlog.

This shifts the traditional LSM trade-off. Instead of balancing compaction speed against read amplification, we primarily balance compaction frequency against disk usage. Read latency becomes largely decoupled from temporary compaction delays.

Compaction within each B+tree is triggered when stale data exceeds a configurable threshold or at scheduled intervals. Scheduled compaction times are randomized across servers to avoid synchronized load spikes. Compaction aggressiveness is dynamically adjusted based on workload: during peak traffic, compaction is delayed to prioritize foreground requests, allowing temporary space amplification to grow; during off-peak periods, compaction is accelerated to reclaim space. If disk usage approaches a critical threshold, compaction is triggered to prevent disk exhaustion.

Figure 8 shows eight consecutive days of production behavior for an illustrative keyspace. The workload normally exhibits a moderate write rate but experiences a pronounced burst once per day, resulting in a sharp peak in write throughput. During these bursts, the active dataset size remains essentially unchanged; however, compaction is intentionally delayed to prioritize foreground requests. As a result, disk usage temporarily increases during the burst periods and decreases afterward as background compaction catches up. In practice, the resulting space amplification remains modest, typically around 10%. Despite these fluctuations, p50, p99, and p99.9 read latencies remain largely unaffected, demonstrating that read latency is effectively decoupled from temporary compaction backlog.

This adaptive design enables operation within strict CPU budgets while minimizing disk interference and maintaining

stable tail latency and availability.

5 Extending the Key-Value Abstraction for Higher Level Semantics

5.1 Separating Metadata from Values: The Key-Meta-Value Model

Although the storage engine exposes a key-value abstraction, higher level database semantics often require finer-grained access patterns. Depending on the data model and query type, many databases need to read only specific metadata associated with a key in order to decide whether the key should be included in the query result.

Common examples include expiration timestamps and tombstone markers representing logical deletion. In such cases, the database layer may only need to inspect metadata to determine visibility, without accessing the full value payload.

If metadata is stored as part of the value, the storage engine must read the entire value from disk and the database layer must decode it, even when only metadata is required. For large values, this results in unnecessary disk I/O and significant decoding overhead. In latency-sensitive environments, such waste directly impacts tail latency and CPU utilization.

To address this inefficiency, our storage engine adopts a key-meta-value tuple model rather than a simple key-value pair. Metadata is stored separately from the value, allowing the upper-layer database logic to selectively read only metadata, or both metadata and value when necessary.

This design reduces unnecessary disk reads, avoids decoding costs for unused data, and enables early filtering decisions during scans. In workloads where metadata-driven visibility checks are common, such as time-to-live (TTL) enforcement or logical deletion handling, this separation significantly improves read efficiency.

5.2 Order-Preserving Encoding Instead of Custom Comparators

From the storage engine's perspective, keys and values are opaque binary objects. The engine performs comparisons and ordered iteration without interpreting higher-level semantics; encoding and ordering logic are handled entirely by the database layer.

For structured data models, a storage key is typically a composite of multiple attributes:

```
attr1 + attr2 + ... + attrN
```

Iteration over such keys naturally corresponds to:

```
ORDER BY attr1, attr2, ..., attrN
```

Two approaches can support this ordering. One is to serialize the composite key (e.g., JSON, BSON, or Protobuf) and implement a custom comparator that interprets the encoded key during comparisons. The other is order-preserving encoding, where the binary representation is constructed so that lexicographical byte order matches the desired attribute ordering.

In practice, custom comparators introduce significant overhead because each comparison requires a callback and partial decoding. We observed this to be roughly an order of magnitude slower than pure lexicographical comparison, and therefore adopted order-preserving encoding.

In our encoding scheme, each attribute is encoded as: (1) 1-byte type indicator, and (2) variable-length encoded value. The encoded attributes are then concatenated in attribute order.

The type indicator specifies both the type of the following binary representation and the cross-type ordering when an attribute may hold values of different types. For example, `null < integer < binary`. Certain special values such as `null`, negative infinity, or positive infinity may consist solely of a type indicator.

For fixed-width N-bit integers, unsigned values are encoded directly in big-endian byte order, while signed values are first mapped to the unsigned domain by adding $2^{(N-1)}$ and then encoded in big-endian form. Binary values use escape-based termination: bytes from `0x1` to `0xff` are stored as-is, `0x0` is encoded as `0x0 0xff`, and `0x0 0x0` terminates the field.

Floating-point values use dedicated type indicators for special values such as `infinity` and `NaN`; regular numeric values are canonicalized to IEEE 754 representation [25] and then monotonically transformed by inverting all bits for negative values and flipping only the sign bit for positive values. Descending order for an attribute is implemented by flipping all bits of that attribute's encoded representation, including the type indicator. For example, for `ORDER BY attr1 ASC, attr2 DESC`, only the encoded bytes of `attr2` are bit-flipped.

With this approach, most data models expressible as compositions of primitive types can be supported without custom comparator logic. The minor encoding overhead is negligible compared to the runtime overhead of comparator callbacks.

5.3 Mitigating Version-Chain Amplification under MVCC

Database logic often relies on multi-version concurrency control (MVCC) to support transaction isolation in a lock-free manner. In particular, long-running ad-hoc queries that read large portions of the database require a consistent snapshot view of the data.

We implement MVCC by extending the order-preserving encoding scheme described earlier. A hidden special attribute representing the transaction version is appended as the last component of the composite key. This attribute stores a transaction identifier as a 64-bit unsigned integer and is encoded in descending order.

Because the version attribute is encoded in descending order, keys sharing the same user-level attributes are ordered from newest to oldest version. The database logic performs a

prefix iteration over keys with the same user-key prefix and selects the version visible to the current transaction.

Over time, multiple versions accumulate for the same user key. These stale versions are eventually reclaimed during in-place compaction, as described in Section 6.

A pathological case arises when certain user keys are updated frequently, resulting in long version chains. This can significantly degrade read latency. Consider a query such as:

```
user_key >= 1 AND user_key <= 9
```

Assume that the type indicator for a 32-bit unsigned integer user key is `0x08`, and that the type indicator for the version field is `0x0c`. The version field is encoded in descending order, so the entire encoded version attribute, including its type indicator and unsigned big-endian value bytes, is bit-flipped. We also assume that negative infinity and positive infinity are represented by `0x00` and `0xff`, respectively. Under order-preserving encoding, the storage-layer iteration corresponds to scanning all keys in the byte range:

`08 00 00 00 01 00` to `08 00 00 00 09 ff`.

Suppose there are ten distinct user keys in this range, and each key has ten versions. If the database logic selects only the latest version per user key, the storage engine traverses 100 physical keys but returns only 10 logical results.

To mitigate this inefficiency, we use a lower-bound seek primitive provided by the storage engine. The database logic defines a configurable threshold `K`. While iterating over keys sharing the same user-key prefix, if more than `K` versions are encountered, the database logic constructs a boundary key and performs a lower-bound seek to skip the remaining versions of that user key.

For example, if the current key is:

`08 00 00 00 02 f3 ff ff ff ff ff ff ff f6`,

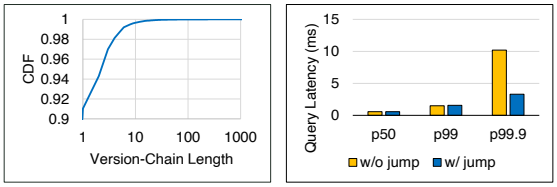
which corresponds to user key 2 and version 9 after descending-order encoding, the database logic constructs the jump target:

`08 00 00 00 02 ff`

and performs a lower-bound seek to that key. Because this target represents the lexicographical upper bound of all versions for that user key, iteration resumes at the next user key without skipping any candidate that satisfies the user-level predicate.

Sequential advancement to the next key costs $O(1)$, whereas a lower-bound seek costs $O(\log N)$. When the number of irrelevant versions that would otherwise be traversed exceeds the logarithmic seek cost, performing a jump becomes beneficial. The threshold `K` captures this crossover point and is tuned empirically based on workload characteristics. The storage engine provides ordered iteration and seek primitives, while version interpretation and jump decisions are handled entirely by the database layer.

Figure 9(a) shows the version-chain length distribution from an illustrative production keyspace snapshot. The majority of user keys have very short chains: approximately 90% have a single version, 99% have fewer than 10 versions, and



(a) (b)

Figure 9: (a) Version-chain length CDF from an illustrative production keyspace snapshot. (b) Range-scan latency comparison with MVCC jump disabled and enabled (`K=10`).

99.9% remain below 30. However, a small but non-negligible tail exists, with some keys accumulating hundreds or even up to 1,000 versions. These long chains typically arise from skewed update patterns or hot keys and are sufficient to amplify range-scan costs.

Because we cannot disable the jump optimization in live production without risking user impact, we evaluated its effect offline by replaying production data. Figure 9(b) compares range-scan latency with jump disabled and with `K=10`. Up to p99 latency, the difference is negligible, reflecting that most keys have short version chains. However, divergence appears at p99.9 and beyond, where jump reduces tail latency by approximately 65%. The magnitude of improvement varies by workload; use cases with stronger update skew or hot-key behavior exhibit more pronounced benefits.

5.4 Predicate-Based Jump for Partial Index Matching

The version-based jump described above can be generalized to predicate-based jump for partial index matching.

Consider a query:

```
A==1 AND C==2
```

The ideal solution is an index on attributes `A` and `C`. However, maintaining a best-matching index for every query is often impractical due to storage and write amplification costs.

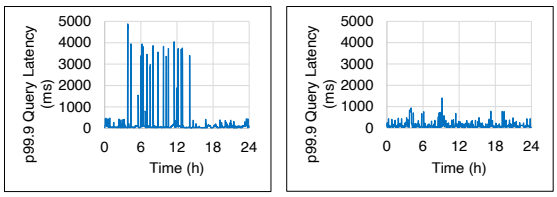
Suppose the available index is composed of attributes `A, B, C`. To serve the query, the system performs a prefix scan for `A==1` and applies a filter on `C`. Because there is no predicate on `B`, the scan must traverse all `B` values for `A==1`. This results in scanning many physical keys while returning few matches.

When attribute `B` has low cardinality and attribute `C` has high cardinality, predicate-based jump reduces unnecessary traversal.

Let the current key correspond to values (`A=a, B=b, C=c`).

1. If `c<2`, the database logic constructs a target key (`A=a, B=b, C=2`) and issues a jump through the storage engine's lower-bound seek primitive.

2. If `c>2`, the database logic constructs a target key (`A=a, B=b, C=+infinity`) and issues a jump. This causes the iterator to advance to the next larger value of `B` for the same `A`, effectively skipping all remaining `C` values for the current `B`. The process then repeats from step 1.



(a) (b)

Figure 10: (a) p99.9 latency spikes. (b) Spikes mitigated with predicate-based jump.

Because each jump target is constructed as a lower-bound boundary within the composite key ordering, no qualifying key is skipped. The same cost trade-off discussed in the MVCC case applies here: a seek becomes beneficial when it avoids traversing a sufficiently large number of irrelevant keys.

Predicate-based jump uses the same threshold-based mechanism. A configurable threshold `K` is defined, and when filter evaluation skips more than `K` consecutive entries during iteration, the database logic issues a lower-bound seek to a predicate-derived boundary key, skipping only entries that cannot satisfy the predicate. The threshold `K` is tuned empirically per workload and is typically guided by the ratio `cardinality(C) / cardinality(B)`, which approximates the expected number of entries scanned per qualifying result.

In practice, however, real production cases are often more complex than this heuristic suggests. Consider a representative schema where attribute `A` is a user ID, `B` is a status field with very low cardinality, and `C` is a last-modified timestamp with high cardinality. For a query of the form `A==a AND C>=c1 AND C<=c2`, most users have only a few distinct `{B, C}` combinations, and iteration overhead is negligible. For these users, predicate-based jump is unnecessary and offers no meaningful benefit.

However, a small number of power users exhibit highly skewed update patterns: for a single value of `B`, the number of distinct `C` values can be extremely large. Under such skew, the query may scan millions of physical entries while returning only a handful of results. As shown in Figure 10(a), this manifests as rare but pronounced spikes at extreme tail latencies (e.g., beyond p99.9). Although such users are rare, their queries can dominate extreme-tail latency and violations of service-level agreements (SLAs), and the large scans they trigger can temporarily consume shared CPU and I/O resources, affecting other requests.

Because this behavior does not occur for the vast majority of users, introducing additional indexes solely to accommodate a small number of power users is not operationally justifiable. Instead, we rely on configuration to selectively enable predicate-based jump with an appropriate threshold `K`. In practice, `K` can be tuned so that the optimization is effectively inactive for typical users with short attribute chains, while activating for skewed workloads associated with power users. As shown in Figure 10(b), enabling predicate-based jump mitigates these tail spikes without affecting the common case. The observed benefit varies significantly across use cases and query patterns, but this approach allows us to mitigate extreme scan amplification without incurring the global cost of maintaining additional indexes.

6 Compaction as a Programmable Control Surface

In a long-running production system, stale versions, expired entries, and logically deleted keys inevitably accumulate, requiring systematic cleanup. While conventional LSM designs treat compaction primarily as a storage-level mechanism for reclaiming space and reorganizing data, we elevate compaction into a programmable control surface. During compaction, each key-value entry is passed through database-defined logic that determines whether it should be retained, discarded, or transformed, allowing multiple background maintenance tasks to be unified within a single storage-driven pass.

6.1 TTL Enforcement and Version Garbage Collection

TTL expiration and MVCC version garbage collection are recurring maintenance tasks in a production database. Both require examining key-level metadata and evaluating a visibility condition to determine whether an entry should remain logically present.

Rather than implementing separate background scanners, we embed both mechanisms into programmable compaction. During in-place compaction of the append-only B+tree, each key-value entry is passed to database-defined logic that decodes metadata and evaluates expiration or version-visibility rules. The logic returns a deterministic keep-or-drop decision, and discarded entries are omitted from the newly constructed tree. Cleanup is thus unified with a storage pass that would occur regardless.

Expiration is enforced logically before physical reclamation. Expired keys are invisible to reads according to replicated metadata, while compaction reclaims space and reduces future read amplification. Replica convergence semantics are defined by database logic, not by the storage engine. Deployments may tolerate temporary divergence under local expiration policies, but strict determinism requires deriving expiration decisions from replicated state rather than local wall-clock time. One implementation is to replicate timestamps through the Raft log, for example using a hybrid logical clock (HLC) [27, 28], so that both logical expiration during reads and physical removal during compaction make identical decisions across replicas. Because compaction removes only entries that are already logically invisible and applies only transformations deterministic with respect to replicated state, physical rewrites preserve logical equivalence across replicas.

Unifying TTL and version GC within compaction eliminates redundant full-database scans, amortizes cleanup cost under existing compaction scheduling, and avoids bursty background I/O, which is critical in CPU-constrained and multi-tenant deployments.

6.2 Lazy Data Format Upgrade

Schema evolution and data format changes are inevitable in long-lived production systems. New features may introduce additional metadata fields, revised encodings, or structural changes to stored values. In a cloud-native distributed database, a format change effectively entails keyspace migration. This goes beyond rewriting stored entries and requires control-plane orchestration, including shard or replica provisioning, online traffic catch-up, and an atomic switchover without downtime. Executing such workflows under live production load is complex, failure-prone, and resource-intensive.

Instead, we perform lazy data-format upgrades through programmable compaction. Each key-value entry carries a format version identifier in its metadata. During compaction, the database-defined callback checks the version and rewrites outdated entries into the current format before inserting them into the reconstructed B+tree, while entries already in the expected format are copied unchanged.

Because migration proceeds incrementally, the system must temporarily support multiple format versions. Both read and write paths recognize and correctly interpret legacy and current formats during the transition period. Only after compaction has rewritten all remaining legacy entries can support for the old format be safely removed.

This design amortizes migration cost across regular compaction cycles and avoids control-plane-driven keyspace migration solely for schema evolution. Upgrade work is bounded by compaction scheduling policies, keeping CPU and I/O usage within operational budgets while enabling continuous format evolution without maintenance windows or disruptive resource spikes.

6.3 Anomaly Detection

Programmable compaction provides a natural control point for anomaly detection. Because compaction scans live entries within a key range and invokes database-defined logic for each key, it can validate metadata and enforce invariants without introducing separate validation passes.

One class of anomalies arises from application bugs. For example, incorrect TTL calculations may generate expiration timestamps far beyond the intended range, effectively creating keys that never expire, or mistakenly use past timestamps that cause entries to expire immediately. During compaction, the database logic inspects expiration metadata and can clamp invalid values, drop the entry, or emit diagnostic signals. Embedding these checks into compaction prevents silent accumulation of logically incorrect data without requiring dedicated background scans.

Compaction also surfaces storage-level corruption. Although replication protects durability, localized disk faults can silently corrupt data on a single replica without immediately triggering failures. Such corruption may remain latent until the affected entries are accessed, at which point recovery becomes more disruptive. Because compaction systematically decodes and rewrites entries, decoding failures or invariant violations can be detected deterministically during background maintenance. The replica can then be marked unhealthy and repaired through Raft recovery. Integrating anomaly detection with compaction unifies correctness validation with storage maintenance while keeping incremental CPU and I/O overhead bounded by compaction scheduling.

7 Lessons Learned

Optimize for the replicated database, not the local storage engine A key lesson is that the storage engine should not be optimized as an isolated component once it becomes the persistence layer of a replicated database. Local storage decisions affect not only local read and write latency, but also replication progress, recovery time, and observed availability at the database layer. An optimization that improves a local storage metric can still be harmful if it increases foreground tail latency, delays recovery, or makes state transfer less predictable.

This shifted our optimization target from the local storage instance to the end-to-end database service built on top of replication. In production, requests that exceed timeout budgets are observed as failures, so extreme-tail latency is not merely a storage outlier but a database availability event. Such events are often dominated by uncommon workload-specific cases, so rarity alone is not a reason to ignore them. We therefore prefer storage policies that provide predictable latency and recovery behavior under tail conditions, even when they sacrifice peak throughput or delay space reclamation.

Treat background work as a budgeted resource Background work is not free simply because it is asynchronous. In a long-running database, maintenance work accumulates continuously, and treating each task as a separate scanner or rewrite path is dangerous at fleet scale. These tasks still consume the same CPU, I/O bandwidth, and cache resources needed by foreground traffic.

In practice, we treat background work as part of the foreground resource budget. Rather than adding independent maintenance passes, we fold cleanup, migration, and validation into programmable compaction, where scheduling and throttling already exist. This also constrains the programming model: callbacks must be deterministic with respect to replicated state, bounded in cost, and safe to retry or abandon during replica rebuild. Maintenance work should be budgeted before it is optimized.

Prefer predictable interference over aggressive reclamation In a multi-tenant deployment, write amplification is not

only a storage-efficiency metric; it is an interference source. CPU and memory can be bounded with container limits, but disk bandwidth and latency remain shared. A compaction policy that looks acceptable for one replica can still destabilize co-located workloads if it generates bursty rewrite traffic.

This changed our view of reclamation. Immediate space reclamation is not always the right production objective; during peak load, it can be better to tolerate temporary space amplification than to create unpredictable I/O interference. Our append-only B+tree design made this trade-off practical by allowing compaction to be delayed without making reads pay for repeated merge work at query time. The broader lesson is that compaction burstiness is an isolation concern, not merely an internal storage-efficiency trade-off.

A minimal key-value API is too low-level, but a semantic storage engine is too high-level A minimal key-value API was too weak for our database services. With only get, set, and iteration, the database layer often had to pay full scan or decode cost even when its own semantics could rule out most entries. These costs were most visible in tail-heavy workloads, where rare amplification patterns dominated p99.9 latency rather than average latency.

Pushing database semantics into the storage engine was not the right answer either, because it would tie the engine to a particular data model, transaction protocol, or query language. The boundary that worked in production was to expose generic control primitives without embedding database-specific semantics. The storage engine need not understand MVCC visibility, TTL rules, or query predicates, but it must expose enough structure for the database layer to avoid unnecessary scan, decode, and rewrite work.

Separate engine-specific mechanisms from transferable principles Our experience should be read at three levels. The engine-specific lesson is lightweight snapshot binding: stable roots in our append-only B+tree provide durable snapshot views for Raft log indices without copying or scanning state. The conceptually portable lesson is read stability under delayed compaction: we achieve it with one indexed B+tree per logical sorted-run group, while a conventional LSM would need an equivalent per-group index or read-path structure. The broadly transferable lessons are the integration principles: specialize Raft log storage, use a bulk path for state transfer, treat compaction as a bounded maintenance surface, and expose generic control primitives without embedding database-specific semantics.

8 Related Work

LSM-trees were introduced as write-optimized indexes that buffer updates in memory and reconcile them through background merges over immutable on-disk sorted runs [34]. A large body of work studies the LSM compaction design space, including read/write amplification, filter memory allocation, and tuning sensitivity under realistic workloads [14, 15, 31]. Recent work further identifies compaction as a major source of performance variance and proposes scheduling or control mechanisms to reduce stalls and tail-latency spikes [9]; production experience from RocksDB similarly shows that operational priorities shift from raw throughput toward space amplification, CPU efficiency, and predictability under mixed workloads [21, 22]. Other work improves LSM read-path and range-query efficiency using range filters and alternative filter organizations [17, 33]. These systems primarily study storage-engine internals, whereas our focus is on operating an LSM-derived engine as the persistence layer of a replicated database under production resource constraints. Distributed databases commonly combine replication, transactions, and a local ordered store: Bigtable introduced sharded tablets backed by memtables and immutable files [12], CockroachDB and TiDB build transactional systems over Raft-replicated key-value substrates [24, 41], and Aurora decouples compute from a scale-out storage service that processes redo and storage operations [42]. State machine replication via Raft defines a replicated log and snapshotting for log truncation and state transfer [35]. Higher-level systems also commonly encode structured data into byte-ordered keys; FoundationDB's tuple layer and Record Layer are representative examples [1, 13]. Finally, production LSM engines expose compaction controls and hooks, and prior experience reports show that compaction policy affects space amplification, CPU cost, and tail latency [21, 22]. Our work focuses on the cross-layer lessons that arise when local storage, consensus, state transfer, and background maintenance are co-designed for a replicated database.

9 Conclusion

Operating a key-value storage engine as the persistent state machine of a large-scale distributed database reshapes traditional storage trade-offs. Local storage decisions surface as database-level latency, recovery, availability, and multi-tenant interference. As a result, production storage design must prioritize predictable tail latency, bounded background work, and fast state transfer over peak throughput or aggressive space reclamation alone.

Our experience shows that the storage layer should provide cross-layer control without absorbing database-specific semantics. Although several mechanisms rely on append-only B+tree structures within our LSM-derived engine, the broader lesson is transferable: once a storage engine becomes the persistent backend of a replicated database, it must be designed as part of the end-to-end system together with consensus, recovery, and resource control.

Acknowledgments

We thank our shepherd Liuba Shrira and the anonymous reviewers for their constructive feedback. We are grateful to the current and former members of the NuData team for their invaluable engineering contributions, operational support, and many discussions that shaped this work.

References

[1] FoundationDB data modeling and tuple layer documentation. https://apple.github.io/foundationdb/data-modeling.html. Accessed 2026.

[2] Jungle. https://github.com/eBay/Jungle.

[3] Kubernetes. https://kubernetes.io.

[4] LevelDB. https://github.com/google/leveldb.

[5] NuRaft. https://github.com/eBay/NuRaft.

[6] RocksDB. https://rocksdb.org.

[7] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining lsm-tree and copy-on-write b+-tree. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2019.

[8] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing access methods: The rum conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[9] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 753–766, 2019.

[10] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[11] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2003.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[13] Christos Chrysafis, Ben Collins, Scott Dugas, Jay Dunkelberger, Moussa Ehsan, Scott Gray, Alec Grieser, Ori Herrnstadt, Kfir Lev-Ari, Tao Lin, Mike McMahon, Nicholas Schiefer, and Alexander Shraer. FoundationDB record layer: A multi-tenant structured datastore. In *Proceedings of the 2019 ACM International Conference on Management of Data (SIGMOD)*, page 1787–1802, 2019.

[14] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 79–94, 2017.

[15] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 ACM International Conference on Management of Data (SIGMOD)*, pages 505–520, 2018.

[16] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 ACM International Conference on Management of Data (SIGMOD)*, pages 449–466, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Niv Dayan and Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 ACM International Conference on Management of Data (SIGMOD)*, pages 365–378, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. Spooky: Granulating lsm-tree compactions correctly. *Proceedings of the VLDB Endowment (PVLDB)*, 15(11):3071–3084, 2022.

[19] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[20] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 77–88, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. Optimizing space amplification in RocksDB. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*, Chaminade, CA, USA, 2017.

[22] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *Proceedings of the 19th*

USENIX Conference on File and Storage Technologies (FAST), pages 33–49, 2021.

[23] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, 1994.

[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A raft-based htap database. *Proceedings of the VLDB Endowment (PVLDB)*, 13(12):3072–3084, 2020.

[25] IEEE. IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019.

[26] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663. ACM, 1997.

[27] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*, pages 17–32, 2014.

[28] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[29] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[30] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, 2001.

[31] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Towards accurate and fast evaluation of multi-stage log-structured designs. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 149–166, 2016.

[32] Yanhong A. Liu, Saksham Chand, and Scott D. Stoller. Moderately complex paxos made simple: High-level executable specification of distributed algorithms. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 1–15. ACM, 2019.

[33] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD)*, pages 2071–2086, 2020.

[34] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[35] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[36] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[37] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3(4):1–27, 2008.

[38] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[39] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. Constructing and analyzing the lsm compaction design space. *Proceedings of the VLDB Endowment (PVLDB)*, 14(11):2216–2229, 2021.

[40] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[41] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD)*, pages 1493–1509, 2020.

[42] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 1041–1052, 2017.

[43] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 61–76, 2007.

[44] Hengrui Wang, Jiansheng Qiu, Fangzhou Yuan, and Huanchen Zhang. Rethinking the compaction policies in lsm-trees. In *Proceedings of the 2025 ACM International Conference on Management of Data (SIGMOD)*, pages 207:1–207:26, 2025.

Case	Data size	# keys	Avg. value size (compressed)	# shards	# server instances (replicas)	Avg. writes	Peak writes	Avg. reads	Peak reads	Avg. CPU util.	Peak CPU util.
A	194 GB	148M	1.30 KB	15	90	1.1K/s	2.4K/s	2.7K/s	5.4K/s	6.7%	180.0%
B	3.6 TB	3.1B	1.16 KB	56	504	18.2K/s	47.4K/s	24.7K/s	34.0K/s	7.1%	52.1%
C	130 TB	177B	735 Bytes	500	3,000	1.1M/s	6.5M/s	1.5M/s	2.3M/s	30.5%	147.0%
D	5.2 TB	5.2B	1.04 KB	30	180	20.5K/s	41.5K/s	19.9K/s	39.3K/s	14.5%	62.9%
E	28.2 TB	9.2B	3.08 KB	64	384	38.5K/s	70.9K/s	71.3/s	1.1K/s	15.4%	63.7%
F	2.0 TB	5.6B	354 Bytes	4	24	1.3K/s	2.5K/s	2.0K/s	12.0K/s	25.8%	108.3%
G	1.8 TB	2.9B	615 Bytes	110	660	1.2K/s	10.2K/s	361K/s	533K/s	5.8%	57.0%
H	239 GB	321M	742 Bytes	3	27	1.9/s	312/s	132/s	5.5K/s	1.5%	82.5%
I	26.8 TB	38.6B	696 Bytes	192	1,152	61.8K/s	213K/s	9.2K/s	27.7K/s	8.9%	54.6%
J	140 GB	295M	474 Bytes	1	6	231/s	443/s	1.1K/s	2.1K/s	15.4%	110.0%
K	64.9 TB	59.2B	1.10 KB	144	864	31.2/s	952/s	337/s	2.5K/s	1.4%	81.0%
L	6.5 TB	10.5B	621 Bytes	375	3,375	5.7K/s	3.9M/s	888K/s	1.8M/s	3.3%	83.8%
M	13.4 TB	29.9B	445 Bytes	40	240	14.2K/s	22.6K/s	244K/s	370K/s	30.9%	132.5%
N	11.2 TB	16.8B	675 Bytes	280	1,680	128K/s	337K/s	229K/s	836K/s	27.3%	122.5%
O	120 TB	65.6B	1.85 KB	450	4,050	127K/s	290K/s	857K/s	2.7M/s	22.1%	128.3%

Table 4: Additional context for anonymized production measurement cases. Case labels match those used in the main text tables and figures. The listed attributes provide coarse workload and configuration context for interpreting heterogeneous production results, not for isolating the causal effect of any individual factor.

A Context for Production Measurements

The production measurements reported in this paper are drawn from anonymized production cases. The case labels in this appendix match the labels used in the main text tables and figures. These cases come from heterogeneous production deployments rather than controlled single-factor experiments. The listed attributes are therefore intended to provide coarse workload and configuration context for interpreting the reported production results. They should not be read as isolating the causal effect of any individual factor. The detailed attributes of the cases are summarized in Table 4.

The reported data size corresponds to the unique logical dataset size of a keyspace, not the aggregate physical disk footprint across all replicas. Each shard consists of multiple server instances, or replicas. In our production deployments, a shard typically has six to nine replicas. Therefore, the total physical disk usage associated with a keyspace is approximately the unique dataset size multiplied by the number of replicas per shard, plus any space amplification from Raft logs and other storage overhead.

The reported value size refers to the size stored by the storage engine after LZ4 compression. Value sizes can vary widely within the same keyspace. In some deployments, multiple applications share a keyspace, and their value-size distributions can differ substantially. For example, even when the average value size is around 1 KB, some applications may mostly store values of roughly 100 bytes, while others may store values larger than 1 MB. For this reason, the reported value-size statistics should be interpreted only as coarse context rather than as a uniform property of all requests in the keyspace.

The read and write TPS values in the appendix are database-level request rates. When interpreting these values at the shard level, the per-shard TPS is the total TPS divided by the number of shards. Writes are replicated within a shard: every committed write to a shard is applied by every replica in that shard. Therefore, each replica observes the same write TPS as the shard-level write TPS. Reads behave differently. Incoming reads are distributed across replicas, so the sum of per-replica read TPS corresponds to the shard-level read TPS. The exact distribution of reads across replicas depends on the use case, consistency requirements, routing policy, leader/follower read configuration, and other operational factors, so we do not assume a single general read-distribution model.

Case	Avg. writes	Peak writes	Avg. reads	Peak reads
A	6.3K/s	20.0K/s	3.3K/s	27.2K/s
B	129K/s	250K/s	102K/s	252K/s
C	6.0M/s	36.1M/s	1.4M/s	13.1M/s
D	122K/s	259K/s	121K/s	248K/s
E	229K/s	422K/s	78K/s	212K/s
F	7.7K/s	15.1K/s	3.7K/s	6.7K/s
G	7.3K/s	61.1K/s	256K/s	553K/s
H	16.9/s	2.6K/s	120/s	4.6K/s
I	457K/s	921K/s	157K/s	330K/s
J	1.5K/s	2.7K/s	1.7K/s	2.5K/s
K	165/s	1.8K/s	324/s	2.4K/s
L	1.6K/s	35.6M/s	893K/s	8.3M/s
M	85.1K/s	132K/s	771K/s	1.3M/s
N	763K/s	1.7M/s	1.0M/s	1.2M/s
O	912K/s	1.9M/s	990K/s	1.5M/s

Table 5: Storage-engine-level request rates for anonymized production cases. Each rate reports the aggregate request rate observed at the storage engine interface, summed across all replicas belonging to the corresponding keyspace.

The database-level request rates should also not be interpreted as the number of operations executed by the storage engine. The mapping between database requests and storage engine operations is not one-to-one. A single database-level write may trigger storage engine reads, for example to check existing state, update secondary structures, or evaluate deduplication opportunities. Conversely, write deduplication may suppress some storage engine writes. For reads, a database-level request may be served from an upper-layer cache without reaching the storage engine, but it may also fan out into mul-

tiple storage engine operations, such as point lookups, range scans, metadata reads, index lookups, or accesses to multiple table-backed instances. As a result, storage engine operation counts are usually higher than database-level request counts, but they can also be lower for cache-heavy or deduplicated workloads. Table 5 reports storage-engine-level request rates for the anonymized cases where such measurements are available.

Finally, CPU utilization is reported relative to the regular Kubernetes resource allocation of each replica. In some cases, peak CPU utilization exceeds 100%. This indicates that the replica temporarily used overprovisioned CPU resources beyond its regular allocation. Such behavior can occur when spare CPU capacity is available on the host, but it should not be interpreted as a guaranteed resource budget.