SOS: A High-Performance Distributed Key-Value Store for Large-Scale Online Services

Yingxin Li
Tencent Inc.
Beijing, China
holmesyxli@tencent.com

Kai Liu
Tencent Inc.
Beijing, China
kedixaliu@tencent.com

Han Xie*
Tencent Inc.
Beijing, China
wfxie@tencent.com

ABSTRACT

Large-scale online services—including web search, recommendation, and LLM inference workloads such as Retrieval-Augmented Generation (RAG) and KV-cache offloading—demand storage that handles petabyte-scale data under millisecond tail-latency SLAs. In-memory stores are cost-prohibitive at scale; disk-based systems sacrifice latency for capacity.

We present **SOS**, a distributed key-value store that bridges this gap by guaranteeing *at most one physical disk I/O per operation*, enforced by three co-designed components: a flat in-memory hash index with $O(1)$ lookup; a disk storage engine with a hierarchical variable-size layout that limits internal fragmentation to ≈7.93%; and a write cache with asynchronous flush that fully decouples client-visible latency from device I/O. Distributed strong consistency is provided by Primary-Based Chain Replication with Two-Phase Commit and an in-chain retry mechanism that reduces write failures to near-zero under transient hardware jitter.

Single-node benchmarks show SOS achieves **704** K QPS under uniform YCSB-B (256 client threads) at P99 = 2 ms, and outperforms RocksDB by **10.0**× in throughput and **24.6**× in P99 latency on write-heavy YCSB-A, with write amplification of 1.00× (vs. 14–16×). Deployed at Tencent for several years, SOS manages over 50 PB and 7 trillion records—peaking at 550 M QPS—powering Yuanbao RAG, Hunyuan KV-cache, user portrait and other services with P99.9 read latencies below 5 ms, at a fraction of in-memory cost.

PVLDB Reference Format:
Yingxin Li, Kai Liu, and Han Xie. SOS: A High-Performance Distributed Key-Value Store for Large-Scale Online Services. PVLDB, 19(12): 4237 - 4249, 2026.
doi:10.14778/3827998.3828029

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/holmes1412/sos-vldb2026-supplementary.

1 INTRODUCTION

Modern internet services and AI-serving applications impose storage requirements qualitatively different from classical database workloads. Consider three representative Tencent workloads. First, *user portrait* and *recommendation* services ingest hundreds of terabytes during batch refreshes and serve hundreds of billions of daily lookups under a 20 ms P99.99 SLA. Timely refreshes are crucial to prevent model degradation. Second, *RAG-based search* [23] (e.g., Yuanbao) stores petabytes of text, requiring batch updates and millisecond-scale document retrieval. Third, *Transformer KV-cache offloading* [22, 36] (e.g., Hunyuan LLM) writes intermediate attention tensors to an external store and reads them back on demand during subsequent steps; the round-trip latency of this store lies on the critical path of token generation, and reducing it enables serving larger models within a fixed end-to-end latency budget.

These workloads share a common profile:

- **Massive capacity.** Tens to hundreds of petabytes of key-value objects, ranging from tens of bytes to tens of megabytes.
- **Strict tail latency.** P99.9 random-read latency must remain in the single-digit milliseconds, even under heavy write traffic.
- **High write throughput.** Periodic full-dataset refreshes require sustained bulk-write bandwidth.
- **Cost sensitivity.** At petabyte scale, the prohibitive cost of in-memory storage (e.g., Redis [30]) necessitates the adoption of commodity NVMe SSDs.

The gap. Existing solutions do not simultaneously satisfy all four requirements. In-memory stores [12, 30] deliver sub-millisecond latency but cost roughly 10× more than SSD-based storage [20, 38]. Most production disk-based KV systems are built on LSM-tree engines [3, 14, 27]; these impose write amplification which translates directly to higher tail latency and reduced throughput under mixed workloads. Distributed file and object stores (HDFS [32], HBase [13]) are optimised for sequential or batch access and cannot meet millisecond-level random-access SLAs.

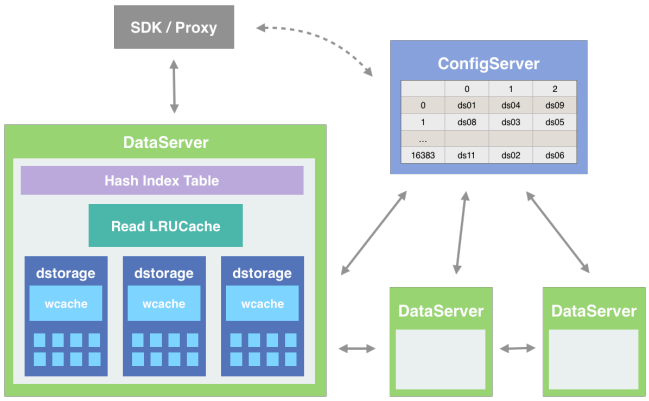

Figure 1: An overview of the SOS architecture.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828029

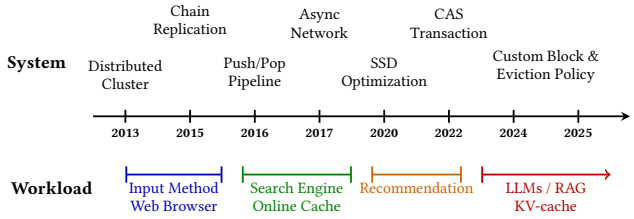

Figure 2: Co-evolution of SOS system capabilities and main serving workloads from 2013 to 2025.

Our approach. We present **SOS** (*Sogou Open Storage*), a distributed key-value store developed entirely in-house at Tencent. The central design invariant is that any read, write, or delete operation requires *at most one physical disk I/O*, enforced by three co-designed subsystems (Figure 1). First, an *in-memory hash index* provides $O(1)$ key-to-offset lookup, eliminating the compaction overhead inherent to LSM-trees. Second, a custom *disk storage engine* uses a hierarchical block-chunk layout and step-sequence scheme to limit internal fragmentation to ≈7.93%. Third, an *in-memory write cache* decouples write latency by responding immediately and flushing data asynchronously. Distributed correctness is provided by *Primary-Based Chain Replication with Two-Phase Commit* (§2.4): Phase 1 validates and reserves resource across the chain while Phase 2 performs the lightweight index commit or rollback, placing failure-prone disk operations in Phase 1 can keep Phase 2 lightweight to minimize commit-path failures. An in-chain retry mechanism further tolerates transient node unavailability without client-visible errors.

SOS originated as a single-node key-value store and has evolved continuously since its distributed deployment in 2013. The evolution of system capabilities and serving workloads—from input method and search engine caching to large-scale recommendation and LLM inference—reflects SOS's growth into a petabyte-scale distributed infrastructure supporting successive generations of internet and AI services at Tencent.

Design principles. These workloads exhibit a common access pattern: point reads and writes with no locality, high-volume periodic writes, and no range-scan requirement. Together, these characteristics make a hash-indexed, SSD-resident design the natural architectural choice. A hash index delivers $O(1)$ point access at the cost of sacrificing key ordering; this trade-off is entirely acceptable here, since recommendation feature retrieval, RAG document retrieval [23] and KV-cache read-back [22] are all addressed by exact key rather than range or prefix traversal. Moreover, even distributed LSM-tree systems expose no global key ordering across shards, so ordering confers no practical benefit in distributed deployments while still incurring its full amplification cost. Crucially, co-designing the hash index, read cache, and asynchronous write cache to maintain a coherent view of on-disk state is non-trivial yet essential: only when all layers agree on the version of each record can the system guarantee at most one physical I/O per operation—enabling disk-resident storage to match the latency of in-memory systems in a distributed setting. These design choices together explain why SOS achieves in-memory performance at SSD cost.

2 SYSTEM DESIGN

SOS is a globally unordered key-value storage system. Data from different users is logically separated by service name within a unified cluster rather than physically isolated (although we can deploy different clusters to meet the true physically isolated requirements). Each key has only one version, with a write timestamp and TTL expiration time, along with an unstructured value. Key-value is the most common format in internet applications, and timestamp-based features enable multi-replica synchronization while expiration time supports cache-like application requirements. This design represents the minimal requirement set for internet online application data storage.

2.1 Architecture

An SOS cluster comprises one or more *ConfigServer* (CS) nodes and N *DataServer* (DS) nodes, storing R replicas per shard. For each piece of data, one DataServer serves as the primary and the remaining $R-1$ as followers. R is fixed at cluster initialization, whereas N can change dynamically, enabling SOS cluster capacity and throughput to scale linearly. ConfigServer manages fault-tolerant replica distribution but stays off the critical path. DataServer handles consistency, indexing, caching, and disk storage. The following sections detail these co-designed modules bottom-up.

SOS provides a command set covering diverse access patterns (Table 1). **PUT** and **DELETE** involve synchronisation across all R replicas, while **GET** and **TEST** perform single-replica point lookups. **MGET** enables batched retrieval from a single node. **PUSH**/**POP** support distributed large-scale batch retrieval: PUSH schedules retrieval tasks across DataServers for sequential disk reads, and POP retrieves prefetched KV pairs, allowing hundreds of millions of records to be scanned globally with high I/O efficiency. **SCAN** provides stateful traversal of all records up to a specified timestamp. The remainder of this paper focuses on the four core commands (PUT, DELETE, GET, TEST) underpinning the read and write paths.

Table 1: SOS Supported Commands

Command	Description	Replicas Involved
PUT	Write/Update a KV pair	R
DELETE	Delete a KV pair	R
GET	Retrieve a value by Key	1
TEST	Check for Key existence	1
MGET	Retrieve multiple values locally	1
PUSH	Batch prefetch value by sequential scan	1
POP	Retrieve prefetched batch KV pairs	1
SCAN	Retrieve all Keys or KV pairs	1

The process for a single request illustrates the collaboration between these components. Clients cache the routing table locally and route requests directly to the appropriate DataServer; a version mismatch triggers a table refresh from ConfigServer.

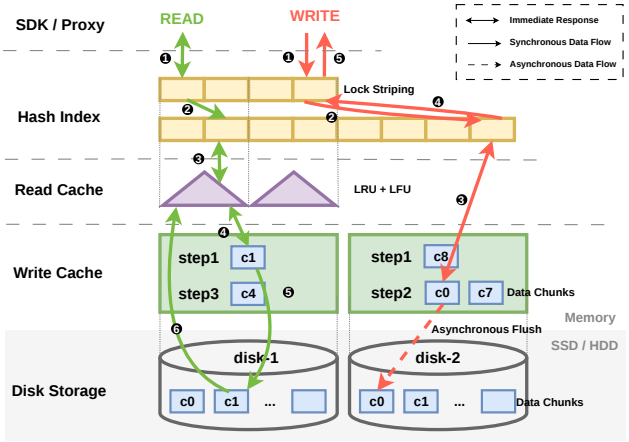

Figure 3: Request processing pipeline and storage hierarchy. Read path (steps ①–⑥): multi-level cache and hash index for low latency. Write path (steps ①–⑤): in-memory write cache returns immediately (④), flushing to SSD asynchronously.

For GET/TEST, the client directly queries the primary DataServer. For PUT/DELETE, the client sends a request with chain metadata to the primary; the primary persists locally, timestamps the write, and forwards to the next follower in the chain. Our two-phase chain-commit is detailed in subsection 2.4.

Within a DataServer, TEST reads only the in-memory index entry (existence, timestamp, TTL) and never touches disk. GET first probes the read cache; on miss, it consults the index for the on-disk location, reads the key-value payload and its record metadata (service ID, timestamp, and TTL), and updates the read cache. PUT/DELETE first probe the index: overwrites and deletes both remove the prior on-disk block and its index entry, invalidate the read cache entry for consistency, and then insert the new block and update the index.

This architecture guarantees at most one disk I/O per request. Since an SSD read of KB-sized data takes approximately 700 μs while intra-datacenter network Round-Trip Time (RTT) is approximately 1 ms, SOS achieves latency comparable to distributed in-memory stores at SSD cost. Additionally, deleted or overwritten blocks are reclaimed synchronously at write time, requiring no compaction or garbage collection, ensuring stable tail latency that meets strict production Service Level Objective (SLO). The implementation of PUT allows massive overwrite performance to be nearly identical to new writes, which is a key advantage over tree-structured indexes that can suffer from write amplification. Production measurements confirming these properties are reported in Section 5.

2.2 Disk Storage

The single-node storage system, called `dstorage`, manages the physical layout of data on SSDs. Internally, `dstorage` is addressed by offsets: the internal "key" is a disk offset, and the internal "value" is the user Key-Value plus metadata. Given an offset and length, it reads and decodes; it can delete at an offset; and it writes user data and returns the actual offset, which upper layers maintain to retrieve the original data.

The primary challenge of `dstorage` is how to manage variable-sized data while minimizing disk space waste, simplifying management, and efficiently reclaiming space after deletions.

Block, Chunk and Step. The `dstorage` solution is illustrated in Figure 4.

The basic unit of data storage is a **block**, which stores exactly one key-value item alongside its metadata: a unique service ID, timestamp and TTL expiration field (these fields are also maintained in the in-memory index; see §2.3). `dstorage` selects the smallest available block size that fits the item, so small KV pairs do not inherit the space overhead of larger size classes.

A **step** sequence determines the sizes of these blocks:

$$S_i = \texttt{block\_size} \times b_j \times 2^k \quad b_j \in \{1, 3, 5, 7\},\ k \geq 0 \tag{1}$$

where `block_size` is determined at cluster initialization.

This design provides finer granularity than power-of-2 allocation, particularly for small objects where most allocations occur.

For uniformly distributed data sizes in the range $[1, N]$, the expected waste ratio is

$$E[W] = \frac{1}{N-1} \sum_{i=1}^{n} \frac{(S_i - S_{i-1})^2}{2S_i}, \tag{2}$$

where $S_0 = 0$ and n is the number of steps covering $[1, N]$. For $N = 64$ MB, our scheme achieves an expected waste ratio of approximately 7.93%, significantly lower than the 25%–30% observed in traditional power-of-2 allocations [2]. Overall, this step sequence balances space efficiency with management complexity, achieving low waste while keeping the number of block sizes manageable.

The minimum `block_size` is bounded below by index addressability: with 32-bit block offsets per `dstorage` instance, a 2 TB partition yields a practical minimum of 512 bytes. For LLM KV-cache, attention tensors have fixed sizes determined by model architecture; setting `block_size` accordingly reduces internal fragmentation to near-zero, reclaiming capacity that would otherwise be lost to padding.

All data is managed in units of **chunks**, with chunk size specified during cluster initialization (e.g., 64 MB). Each chunk carries a

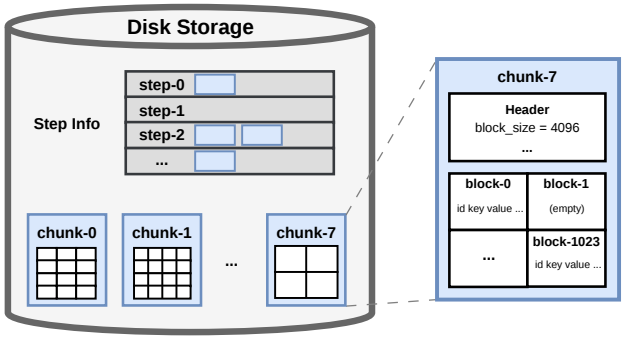

Figure 4: Physical storage layout on disk. Each *Chunk* is serialized into a contiguous sequence of blocks (e.g., 4096-byte blocks in chunk-7). The Header tracks the persistence status, while each individual block stores the Key-Value pair along with a service ID and a timestamp for version control.

header recording its `block_size` and an allocation bitmap tracking which blocks within the chunk are occupied. A disk is initially divided into several chunks; each chunk stores only one block size but is assigned to a step on demand rather than permanently dedicated to one. When all blocks in a chunk are deleted, the chunk is immediately reclaimed for reallocation. For workloads with irregular deletion patterns that leave chunks sparsely occupied, SOS provides an online compaction tool that operators can invoke manually to consolidate live blocks and reclaim underutilized chunks.

This dynamic strategy enables highly efficient disk management. Regardless of whether current business data is large, small, or varies over time, no extra space is wasted for non-existent data sizes. Fixed-length blocks combined with the step sequence design minimize overhead waste for each block, achieving balance between disk fragmentation and space utilization.

Write Cache. SOS supports Direct IO for disk operations and implements its own pagecache-like write-back management mechanism. For write requests, SOS employs a write cache with dedicated backend threads responsible for flushing dirty data when a chunk's utilization ratio exceeds a threshold. In production we enable asynchronous flushing, so client-visible latency does not wait for device persistence.

Write cache management operates at chunk granularity. For a given step, at most one chunk is active in memory. If an incoming write finds no active chunk for its step, a fresh chunk is allocated; similarly, chunks are reclaimed when all data is deleted. During flushing, only modified blocks in a chunk need to be written back to disk by direct I/O.

The primary advantage of this mechanism is that write request latency remains unaffected by disk I/O, which greatly improves write throughput, solving the bottleneck of batch writes in disk storage systems. However, it may introduce distributed inconsistency as a trade-off. For example, if a process exits unexpectedly or a machine suddenly crashes, the node may have unflushed data.

SOS addresses this problem by removing this node from the distributed route table. The cluster will wait for fix until the missing replica is recovered from other replicas. For machine crash, considering systems using PageCache also face disk data inconsistency, removing the node is unavoidable. For unexpected process exits, deploying storage process independently from other business processes or using cgroups to guarantee SOS memory is a useful way to reduce exit probability. Therefore, the significant performance improvements from the write cache justify this acceptable trade-off of the occasional machine removal, which will not affect data consistency or request availability anyway. In a replicated deployment, data durability under node failures is ensured by the multi-replica protocol (§2.4, §3.3): if a node crashes with unflushed write-cache data, the committed version is preserved on surviving replicas, and the missing copy is restored via re-replication. Across years of production operation at Tencent, no data loss attributable to write cache incompleteness has been observed.

Multiple Disk Management. `mdstorage` is the module for a single DataServer to manage multiple physical disks, usually each hosting an independent `dstorage` instance. Disk-level isolation ensures that an I/O error on one disk does not affect reads or writes on others; the DataServer remaps affected blocks to healthy disks and marks itself unhealthy only when the error rate on any disk exceeds a configured threshold (§3). This design also allows background flush threads to operate on each disk independently, improving aggregate write throughput proportionally to the number of disks.

2.3 In-Memory Hash-Based Index

SOS maintains an in-memory index structure to enable efficient key-to-offset lookups in the disk storage module (§2.2).

Hash-based indexing. For a single-node storage engine, common index structures include Log-Structured Merge-tree (LSM-tree) [27] and B+ tree [6], while SOS adopts a hash-based structure. Hash indexes provide $O(1)$ lookup complexity and eliminate write amplification during index updates. The trade-off is the absence of ordering, meaning range queries are not supported. This is acceptable for online serving workloads, where access patterns consist exclusively of point lookups rather than range scans or batch fetching in analysis or machine learning training.

Table 2: Index Structure Comparison

Property	Hash	LSM-tree	B+ tree
Point query	$O(1)$	$O(L \log n)$	$O(\log n)$
Range query	N/A	$O(k + L \log n)$	$O(k + \log n)$
Write amplification	1×	$10 - 30\times$	$1 - 2\times$
Disk I/O per read	1	$[1,\ L]$	$\lceil \log_B n \rceil$
Space Efficiency	High	Medium	Medium/High

Table 2 quantifies the tradeoffs between index structures, where L denotes the number of LSM-tree levels, k is the number of items returned in a range query, and B is the B+ tree branching factor (typically 100–200 for disk-based systems). The write amplification of LSM-tree [11, 25] varies with level count and compaction strategy; the values represent typical production configurations. Specifically, cold-cache point lookups incur reads from up to L data blocks plus index and filter blocks, yielding an effective RAF of 2–12× in production [8].

Compact In-Memory Design. SOS keeps the *entire* index in memory to guarantee at most one disk I/O per operation. This is feasible because the index is extremely space-efficient.

- **Key representation**: Instead of storing full user keys (typically strings), each index entry stores 96 bits extracted from the key's MD5 hash. This reduces space while maintaining negligible collision probability (analyzed below).
- **Timestamp**: 48 bits encode millisecond-precision Unix timestamps for versioning, conflict resolution and Compare-And-Swap transaction.
- **TTL encoding**: 16 bits store expiration time using variable precision, supporting up to 20 years with second-level granularity.
- **Service ID**: 16 bits identify the owning tenant, enabling efficient filtering during SCAN operations.

We now analyze the collision probability for the 96-bit MD5 algorithm by assuming that a cluster consists of H hash index entries. Since the SOS routing mechanism is also based on the key's MD5 hash, we assert that colliding keys must land on the same node. Based on the Birthday Paradox [26], for $H = 2^{42}$ (over 4 trillion)

index entries in an MD5 representation space of size $M = 2^{96}$, the collision probability P_{coll} is approximated as:

$$P_{\text{coll}} \approx 1 - e^{-\frac{H^2}{2M}} = 1 - e^{-\frac{2^{84}}{2^{97}}} = 1 - e^{-2^{-13}} \approx 2^{-13} \quad (3)$$

The more index entries there are, the higher the collision probability. Considering that each DataServer can store at most $2^{32} - 1$ hash index entries, $H = 2^{42}$ means there are at least 1024 DataServers for each replica in the cluster, which is much larger than the scale of actual online clusters. Given that $2^{-13} \approx 0.0122\%$, this collision probability is negligible and acceptable for production deployments. Moreover, the disk storage module verifies the full key on disk as a final safeguard.

Index Layout. SOS resolves hash collisions via separate chaining [21]. The index consists of a bucket array of 32-bit head indices and a pre-allocated pool of 30-byte entries. Using array indices rather than machine pointers keeps the structure position-independent, allowing it to reside in shared memory. On a typical node (256 GB RAM, six 4 TB SSDs), an approximately 120 GB index leaves 137 GB for read caching at 71% disk utilization(accounting for the ≈15% space overhead from §2.2), demonstrating balanced memory–disk resource utilization.

On a typical node (256 GB RAM, six 4 TB SSDs) at 71% disk utilization (accounting for the ≈15% space overhead from §2.2), an ≈138 GB index and a few GB of write cache leave ≈100 GB for read caching, demonstrating balanced memory–disk resource utilization.

This index occupies a single contiguous region fixed at startup, while read and write caches are managed via `jemalloc` for low-fragmentation concurrent allocation; read caches are sized at initialization, with additional OS page cache headroom reserved when DirectIO is disabled.

Index Table Persistence. The index is memory-mapped to disk via `mmap` and periodically flushed to disk using either OS page cache mechanisms or a custom flushing implementation within SOS. In practice, if data is small and frequently modified, flushing the index table might occur before the data's disk I/O is flushed, potentially becoming the performance bottleneck for the entire node; this can be mitigated by mixing workload types on the same node or scaling out capacity.

A potential concern is that index flushes may outpace data flushes; this is safe by design, as any unflushed data block remains readable from the write cache, which is always probed before disk. The index and write cache together present a coherent view of all written data; disk persistence is a background durability operation that does not affect read correctness.

Time-To-Live (TTL) Support. A background routine periodically cleans up expired data. The expire timestamp on the index entry allows TEST or GET operations to quickly filter out expired data without disk access, guaranteeing that expired data is not observed even before physical reclamation.

2.4 Primary-Based Chain Replication with Two-Phase Commit

SOS extends classic chain replication [35] with two key modifications that together improve reliability and read efficiency. In classic chain replication, R replicas form a linear chain: writes enter at the *head*, propagate to the *tail*, and commit there; reads query the tail to ensure linearizability [17].

Protocol Design. SOS modifies chain replication in three key ways.

1. **Head-based routing.** Each key is served by 1 primary and $R-1$ secondary replicas, uniformly distributed across all nodes for load balance. All client operations—reads and writes—route through the primary (head), so clients only observe data committed across all replicas.
2. **Two-phase commit.** Operations execute in two phases:
   Phase 1 (Reserve):
   The head generates a creation timestamp. Each replica validates the routing table version, timestamp, TTL, and CAS conditions; persists data to disk; and reserves an index slot *without* updating the index. This phase propagates forward from head to tail. If any replica fails validation or persistence, the operation is rolled back along the chain in reverse order before reaching the next node, leaving no partially committed state.
   Phase 2 (Commit):
   The tail initiates commit by updating its index to make data visible; commit messages propagate backward to the head, and each replica updates its index upon receiving the success notification.
3. **Read and delete semantics.** Reads query the head and return only index-visible (i.e., successfully committed) data. Deletes follow the same two-phase protocol: Phase 1 validates and propagates the deletion, Phase 2 removes the index entry.

Key Advantages. These modifications yield three key advantages.

Reliability. Decoupling failure-prone operations (validation, disk writes) into Phase 1 and lightweight operations (index updates) into Phase 2 reduces the probability of Phase 2 failures. Recovery is simplified accordingly: Phase 1 failures trigger an in-place rollback of the reserved index slot and persisted disk data, and a rollback response is returned to the upstream node; the rare Phase 2 failures are resolved by persistent retry until the downstream node recovers or is permanently evicted by the ConfigServer heartbeat timeout (§3.2), after which the commit proceeds on a new chain without the node—ensuring no permanent inconsistency can arise.

Linearizability. Shifting reads to the head ensures linearizability through three mechanisms: (1) uncommitted versions are strictly invisible to reads; (2) the linearization point is defined as the moment the head applies the commit and updates its index, enforced by lock striping [15] that serializes Phase 2 index updates; (3) upon routing table changes, ConfigServer broadcasts the new version to all DataServers; any in-flight request carrying a stale version is rejected and prompts a table refresh, bounding the propagation window to within one RTT, which is smaller than the minimum causal gap between dependent client operations.

Operational simplicity. Routing all reads through the primary eliminates the need for clients to track tail identity, reducing client-side complexity. The symmetric role of each node also simplifies the retry logic described in Section 3.1.

Optimization. Beyond the protocol design, SOS optimizes network communication and replica distribution to further improve overall efficiency. DataServer communication uses the C++ Workflow [33] asynchronous framework, which separates network I/O from computation and enables concurrent utilization of all system resources (CPU, network, disk), preventing software-layer scheduling from becoming a bottleneck. Primary and follower replicas are uniformly distributed across nodes subject to the constraints in Section 2.6, avoiding both uneven data distribution and imbalanced request load.

2.5 Compare-And-Swap (CAS) Transactions

SOS supports *Compare-And-Swap* (CAS) semantics [16] built directly on the two-phase commit protocol described in §2.4. Each request carries a `compare_timestamp` that the DataServer compares against the timestamp in its local index:

- **PUT/DELETE:** The operation enforces that data is overwritten only if the existing version is less than or equal to the provided `compare_timestamp`.
- **GET:** The operation retrieves data only if the stored version is *strictly greater* than the `compare_timestamp`.

Atomicity Guarantees in Distributed Environment. Ensuring atomicity is non-trivial because network reordering means that arrival order at one replica does not imply the same order at the next. SOS guarantees atomicity via its two-phase commit:

(1) **Phase 1 (Reserve)**: The CAS request propagates along the chain, reaching each node in order. At each node, the request's `compare_timestamp` is compared against the local index timestamp, and the request proceeds to the next node only if the comparison succeeds.
(2) **Phase 2 (Commit)**: If multiple transactions with the same `compare_timestamp` execute in parallel, only one transaction can successfully commit the reserved data to the index (at the tail node DS_3 in our example) because index updates are serialized by lock striping [15], guaranteeing atomicity at this critical point.
(3) **Failure Handling**: A failure at any phase indicates that new data has been written after `compare_timestamp`. In such cases, the node returns a compare-failure error to the upstream. If failure occurs during Phase 2, SOS's two-phase commit algorithm performs a rollback along the original chain, releasing uncommitted resources and ensuring consistency.

Use Cases. This CAS design has three most common scenarios:

- **Atomic read-modify-write**: the client includes the read timestamp as `compare_timestamp`, preventing lost updates under concurrent writes.
- **Insert-if-not-exists**: setting `compare_timestamp=1` ensures existing data is never overwritten.
- **Conditional retrieval**: clients suppress unchanged-data transfers by setting the last-seen timestamp, reducing bandwidth.

Timestamp Granularity. SOS employs millisecond-precision timestamps for data versioning, while millisecond-precision timestamps

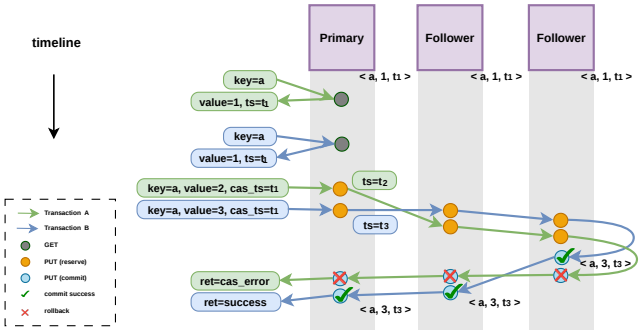

Figure 5: Timeline of concurrent CAS transactions. Transaction A (green) and Transaction B (blue) attempt to update the same key with the same data version (`cas_timestamp` = t_1). Through the Two-Phase Commit protocol across the Primary and Followers, the system enforces linearizability. Transaction B commits first, which make Transaction A rejected with a *cas_error* upon conflict detection ($t_3 > t_1$). Transaction A will roll back and release the reserved block with no visible side effect, ensuring data consistency and atomicity.

are sufficient for version ordering. Given that the network RTT within a datacenter typically exceeds 1 ms, the physical latency naturally serializes causally dependent requests from the same client with a gap larger than the timestamp granularity. Therefore, millisecond precision is sufficient to distinguish versions and order events in most cases even in CAS transactions.

2.6 Routing and Replication

Data sharding and routing are managed by the **ConfigServer**. Its primary responsibilities are to manage the routing table and monitor the liveness of DataServer via a heartbeat mechanism. The routing table is a mapping of data segments and their replicas to specific DataServer nodes. This design ensures that the SOS cluster can scale linearly by the number of DataServer nodes.

Routing Table for Sharding. Unlike consistent-hashing [19] that sacrifice global ordering for load balance, SOS uses a fixed segment partition, enabling deterministic routing with $O(1)$ table lookup. Data partitions into S segments (e.g., $S = 16,384$) with R replicas (1 Primary, $R-1$ Followers), totaling $S \times R$ data segments distributed across N DataServers.

Table 3: Routing Table for 4 Segments, 3 Replicas, and 6 DataServers

Segment	Replica 0 (Primary)	Replica 1	Replica 2
0	DS1	DS2	DS3
1	DS4	DS5	DS6
2	DS2	DS3	DS4
3	DS5	DS6	DS1

Routing table scheduling can be formulated as a constrained optimization problem with the following constraints to ensure balanced load and fault tolerance.

(1) **Primary** replica counts should be balanced across nodes, ensuring distribute read load evenly
(2) **Total** replica counts should be balanced across nodes, ensuring distribute storage and write load evenly
(3) Replicas for a single segment should avoid the **same node**, for fault tolerance
(4) Replicas for a single segment should preferably avoid the **same logical rack** if logical racks are configured, for geographical fault tolerance

The optimization objective for this linear programming problem is to minimize the total number of segments requiring adjustment when nodes change, enabling faster data migration/replication and quicker cluster stabilization.

Data Routing. When DataServer nodes are added or removed, the routing table is updated with a new version number. The client SDK or proxy performs global routing for each request according to the following process.

Algorithm 1 SOS Client Request Algorithm

```
Initial Phase:
table ← configserver.GetRoute()
User Request:
Function ProcessRequest(cmd, key, args):
hash ← MD5(key)
segment ← first_32_bits(hash) mod 16384
nodes ← table[segment]
dataserver ← nodes[0]
response ← dataserver.Send(cmd, table.version, key,args)
if response.error is OldVersion then
    table ← configserver.GetRoute()
end if
return response
```

Logical Racks. As mentioned in the constraints, the routing table can incorporate the concept of a logical rack, or `group_id`, to enhance fault tolerance. The segment allocation algorithm will try to ensure replicas of a single segment are mutually exclusive with respect to their `group_id`.

As an example, consider the original routing table in Table 3. Suppose logical rack constraints are introduced with three groups: $\text{group}_1 = \{\text{DS1}, \text{DS2}\}$, $\text{group}_2 = \{\text{DS3}, \text{DS4}\}$, and $\text{group}_3 = \{\text{DS5}, \text{DS6}\}$. Since each segment violates the logical rack constraint, at least one replica per segment must be relocated. To satisfy the group mutual-exclusion constraint while minimizing data movement, the routing table is adjusted as shown in Table 4, where the modified replica assignments are highlighted in **bold**.

After adjustment, replicas of each segment are placed on the DataServers that belong to different logical groups, ensuring that no two replicas of the same segment reside within the same logical rack. The adjusted routing table achieves this lower bound, thereby minimizing the number of replica movements.

Assigning the same `group_id` to all machines within a physical rack or datacenter can guarantee stable service and data integrity when a physical disaster affects an entire rack or datacenter.

Table 4: Routing Table with Logical Rack Constraints

Segment	Replica 0 (Primary)	Replica 1	Replica 2
0	DS1	**DS5**	DS3
1	DS4	**DS2**	DS6
2	DS2	**DS6**	DS4
3	DS5	**DS3**	DS1

Similarly, a `group_id` can be used for geographical fault tolerance. For an R-replica cluster, configuring R distinct logical groups ensures that a complete copy of the data is present in each logical region. This design enables low-latency local reads and seamless cross-region disaster recovery. Note that different datacenters within the same city may have an RTT of approximately 2ms, meaning a 3-replica chain operation typically adds approximately 4ms of additional write latency.

ConfigServer Availability. ConfigServer is not a performance bottleneck as clients cache the routing table locally and contact it only at initialization or on a version mismatch. For fault tolerance, the routing table is persisted and replicated to a standby; upon failure, the standby is promoted via a DNS update, affecting only clients actively fetching a fresh table. In practice, ConfigServer failures have been exceedingly rare and have caused no measurable availability impact across years of production operation.

3 AVAILABILITY AND RUNTIME MANAGEMENT

SOS handles node failures across three temporal phases: transient unavailability via in-chain retry (§3.1), preventing client-visible errors; permanent exclusion via health checking (§3.2), removing failed nodes from the routing table; and redundancy restoration via re-replication (§3.3), restoring full fault tolerance. This layered design balances rapid response to transient faults against the cost of full re-replication.

3.1 In-chain Retry Mechanism

When a node required by a request fails to complete the operation within the specified timeout or returns a direct failure, the request initiator will attempt a retry. Since a single request might involve one or more nodes, and the multi-replica data synchronization protocol is implemented using a chain-based approach, this retry can be initiated either by the client-side SDK or by a DataServer currently involved in the chain operation.

The retry mechanism in chain operations significantly enhances write availability. When a downstream node is temporarily unavailable, the upstream node caches and retries the write, ensuring eventual consistency without blocking clients. The symmetric node roles in chain replication greatly simplify this retry logic across the system.

Write And Delete. The request initiator (either the client or an upstream DataServer) that detects an unavailable node will leverage the SOS routing table to perform a target switching retry to another replica node within the same data shard, thereby preventing the entire chain operation from failing. This significantly improves

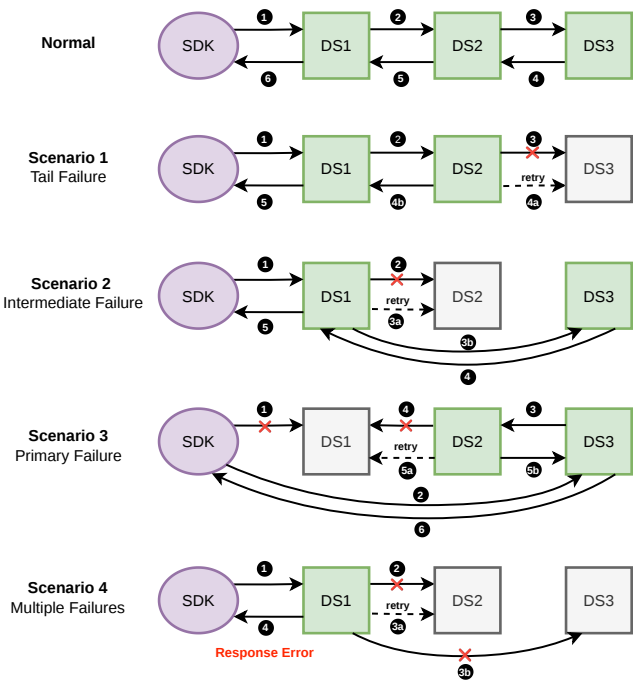

Figure 6: Chain retry scenarios. Scenarios 1–3: single-node failure (grey) succeeds via retry; background retry (4a/5a) and commit (4b/5b) proceed in parallel. Scenario 4: multi-node failure causes error.

availability: if a cluster has N nodes, the user success rate might drop to $\frac{N-1}{N}$% during periods of jitter, but the incorporation of this retry mechanism can boost the success rate to nearly 100%.

When write or delete requests synchronize via chain in SOS, each node operates equivalently with the following logic:

- If the request contains two or more addresses: Place unavailable address back in request packet and try another address for synchronization.
- If the request contains only one address (i.e., the next node is the chain tail): Place the current request into its internal retry module and immediately return success to the preceding node.

The retry module inside each DataServer continuously attempts to send the data to the problematic downstream node until it either recovers or is permanently evicted by the cluster heartbeat mechanism. This process is essential for ultimately guaranteeing data consistency across all replicas. This ensures the overall chain operation is completed successfully from the perspective of upstream nodes.

If the temporarily unavailable node is the head node of the chain, the retry mechanism is initiated by the client. The client reorganizes the chain: it moves the original unavailable head node to the tail, selects a different node as the new head, and reissues the request.

Read and Test. Retry for read-only requests is simpler. The request primarily targets a single node; if the head node fails, the client attempts another replica node holding the data. The request is considered failure only after all R replicas have been attempted and failed.

Table 5: Retry Behavior by Command Type

Property	PUT / DEL	GET / TEST
DataServer Per Attempt	R	1
Max Attempts	$\max(1, R-1)$	R
Fault Tolerance	$1/R$	$(R-1)/R$

Note: Max attempts represents the total number of request attempts (initial + retries). For $R = 1$, PUT/DEL operations have exactly one attempt.

Fault Tolerance. Thus, the retry mechanism allows write and delete requests to tolerate one temporarily unavailable replica out of R, requiring $R-1$ operational replicas for completion. For read-only requests, only one available replica among R is sufficient to complete the operation.

In practice, this mechanism is crucial for recommendation systems where reads can tolerate partial failures. Recommendation systems typically require <100 ms application-layer latency, translating to <20 ms storage requirements. If data isn't retrieved within 20 ms, returning empty is preferable to blocking. After deploying SDK with 20 ms timeout retry, failure rates during hardware or network jitter dropped from 4% to 0%.

3.2 Failure Detection and Health Checking

When a node remains unavailable beyond a certain duration, its permanent exclusion from the global routing table becomes necessary. Failure detection is the DataServer's internal mechanism for identifying its own local failures, while health checking is the ConfigServer's mechanism for detecting unhealthy DataServers across the cluster. Note that failure detection only marks DataServers as unhealthy; ConfigServer is responsible for handling eviction for the cluster.

Failure Detection. Internally, a DataServer can exist in several states: Initializing (preparing to join), Running (normal operation), and Unhealthy (self-detected local failure). If a DataServer marks itself as unhealthy through its internal failure detection, it continues operating until the cluster-level command permanently removes it from service.

A DataServer typically hosts multiple physical disks. SOS is equipped with single-disk fault tolerance logic to mitigate localized storage failures before node-level eviction. When a disk fails or exhibits excessive I/O errors, the DataServer takes immediate action:

- Read: Attempt to read from the target disk location. If an I/O error occurs, return an error code for upper-layer handling (implying retry to another replica).
- Write: Temporarily write to other healthy disks to maintain service continuity and update the location to index table.

When the count of I/O errors on any disk reaches a configured threshold, the DataServer marks itself as unhealthy. This proactive self-reporting enables early detection before complete disk failure.

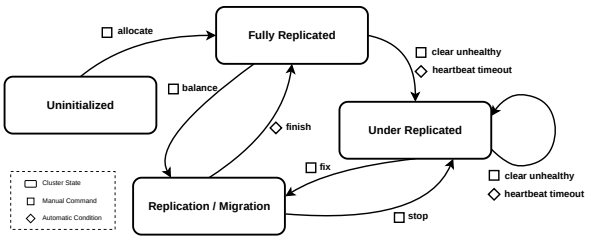

Figure 7: Cluster State Transitions. Rectangles (□) represent manual commands, while diamonds (◇) denote automatic system conditions.

Health Checking. Beyond self-detection, ConfigServer discovers unhealthy nodes by maintaining a heartbeat mechanism with every DataServer. Specifically, ConfigServer records the last heartbeat timestamp for each DataServer and refreshes it upon each received heartbeat. If a DataServer fails to update its heartbeat within the user-configured timeout value, ConfigServer considers the node failed and initiates eviction, implementing an eventually perfect failure detector in the sense of Chandra and Toueg [4].

Cluster Eviction. The eviction procedure can be triggered either automatically by heartbeat timeout or manually by operators. If DataServers are detected as problematic through monitoring or alerts, operators can immediately mark the node as unhealthy and issue an eviction command to ConfigServer. In both cases, the eviction procedure is:

(1) Removal: ConfigServer removes the unhealthy node from the routing table.
(2) Primary promotion: For each segment whose Primary resided on the evicted node, ConfigServer promotes a surviving Follower. Eviction is rejected if it would eliminate all replicas of any segment.
(3) Version broadcast: ConfigServer increments the routing table version number and broadcasts it to all DataServers.
(4) Client refresh: Client requests routed using the outdated table version will be rejected by DataServers, prompting clients to fetch the updated table.

After eviction, the cluster enters an under-replicated state: affected segments operate with fewer than R replicas, and chain replication for these segments involves at most $R-1$ DataServers, maintaining consistency among the remaining copies. Before the replica rebuilding process (§3.3) restores full redundancy, the amount of write requests for each node is not affected. However, those DataServer nodes that were promoted as primary need to handle more read requests until replica rebuild.

3.3 Data Re-replication and Cluster Balancing

A cluster operates in four states: *Uninitialized*, *Fully Replicated* (Healthy), *Under-replicated*, and *Migration/Replication* in Progress. Except for the Uninitialized state, the cluster can provide normal service during all other states.

State transitions involve some coordinated operations: the ConfigServer adjusts the routing table, while DataServers physically migrate data at segment granularity. All transitions are manually triggered, except for automatic eviction due to heartbeat timeout.

Cluster Perspective. When a new DataServer initializes, it sends heartbeats to the ConfigServer, which registers it as an "Idle" node. Upon receiving a command (Initialize, Fix, or Scale-out), the ConfigServer validates the operation:

- The operation is valid for the current cluster state.
- Heartbeats of existing and new nodes are stable.
- No unavailable retry sources: To guarantee consistency (Section 3.1), operations pause if a retry-initiating node is unavailable; unavailable targets are ignored.

Regarding the last condition: waiting for active node retries to complete ensures consistency. Since successful retries resolve quickly, this safety check rarely hinders normal cluster operations.

Once validated, ConfigServer updates the routing table according to the constraints in Section 2.6, then sends migration commands to each DataServer that needs to migrate data out, specifying target DataServer assignments per segment. ConfigServer only instructs DataServers to send data for segments where they serve as primary, ensuring migration tasks are distributed as evenly as possible, thereby reducing the time required for the migration process.

DataServer Perspective. SOS manages migration at the granularity of *Segments*. Upon receiving the migration command, the DataServer checks the data segments to be sent, establishes connections to the target DataServers, and begin transmission.

- **Scale-out (Expansion):** The cluster moves data to new nodes to free up space. Once data is copied to the target, the source node deletes its local copy, so that when capacity is nearly exhausted, expansion can immediately support normal write services. To ensure consistency, the source node forwards all requests for migrating segments to the target machine. The forwarded write request carries the current timestamp, so it will not be overwritten by the migrated data with the original timestamp.
- **Replica Repair (Fix):** The source node copies data to restore missing replicas while retaining its own copy; Write and delete requests are forwarded as in Scale-out.

Once a DataServer completes its migrations, it reports to the ConfigServer. When all nodes report success, the ConfigServer broadcasts the new routing table and version to all DataServers. Clients will discover the outdated version number through DataServer responses and fetch the new routing table from ConfigServer.

4 EVALUATION

We focus on single-node storage engine performance to isolate SOS's core design invariants; distributed KV systems introduce confounding variables (replication protocol, network topology, cluster tuning) that preclude controlled comparison on a shared testbed. We compare against RocksDB as the de facto disk-based KV baseline; Redis is excluded because its single-threaded model makes thread-count comparisons structurally unfair, and since SOS guarantees at most one disk I/O per operation, any gap would primarily reflect network-framework differences rather than storage efficiency.

4.1 Experimental Setup

Hardware. All single-node experiments are conducted on a server with an Intel Xeon Platinum 8374C (132 cores), 382 GB DRAM, and a Samsung PM9A3 NVMe SSD (3.84 TB, 4 KB native LBA). The Linux I/O scheduler is set to none.

Software Configuration. To evaluate the efficiency of our disk engine, we strictly limit the **Read Cache to 4 GB** (≈ 1% of the dataset) and enable **DirectIO**. The **Write Cache Ratio** defines the threshold of dirty blocks within a chunk that triggers asynchronous flushing and is set to 25%, thereby reducing random disk I/O and maximizing write throughput for update-heavy scenarios. SOS partitions the SSD into two `dstorage`, each managed by **2 background flush threads**, yielding 4 flush threads in total. For a fair comparison, RocksDB (v7.0) is configured with the same 4 GB BlockCache, Bloom filters enabled, compression disabled, and `max_background_jobs=8`, providing it with at least as many background threads as SOS.

Workloads. We use YCSB [7] with **100 million keys** (20-byte key, 4000-byte value), resulting in a ≈374 GB raw dataset. The primary comparison uses three standard profiles: **YCSB-A** (50% Write), **YCSB-B** (5% Write), and **YCSB-C** (Read-only); specific experiments such as write amplification measurement may use different configurations, as noted.

4.2 Single-Node Performance

We evaluate the baseline performance of SOS under both Zipfian [40] and Uniform distributions by varying client threads from 16 to 256. Figure 8 presents the throughput results; Figure 10 and Figure 9 present the latency breakdown at 64 threads and across thread counts respectively.

Throughput. In read-heavy workloads (YCSB-B/C), SOS exhibits near-linear scalability under Zipfian distribution, peaking at 256 threads at a throughput of **1.86 M QPS**. At 64 threads, SOS reaches **1.44 M QPS** on YCSB-B and **1.55 M QPS** on YCSB-C—**11.4**× and **10.0**× the throughput of RocksDB respectively (Figure 10(a)). For write-heavy workloads (YCSB-A), SOS saturates at approximately **390–470 k QPS**, limited by 4 background flush threads. RocksDB reaches **69.4–72 k QPS** (**6.5–6.8**× lower).

Latency. All measurements are taken at each system's own throughput. Under Zipfian YCSB-A at 64 threads, SOS handles **6.8**× more requests than RocksDB yet posts P99 of **1.2 ms** vs. **12.4 ms** (**10.6**×,

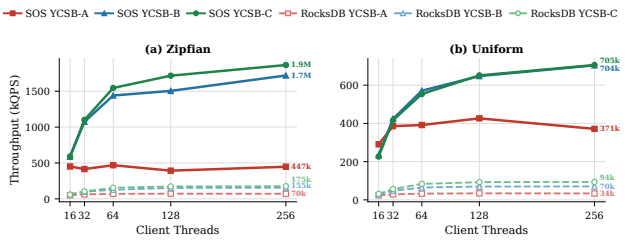

Figure 8: Throughput vs. client threads under Zipfian (a) and Uniform (b) distributions. Solid lines: SOS; dashed: RocksDB.

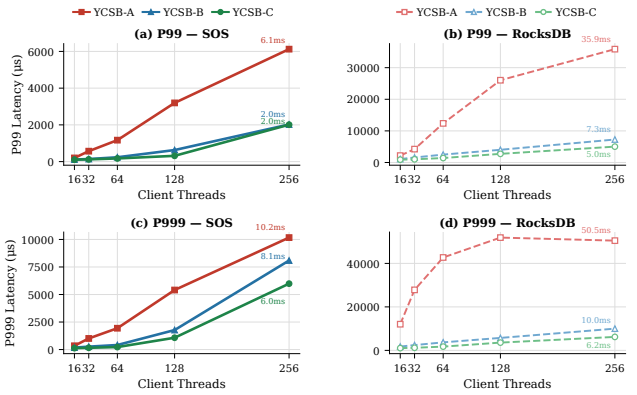

Figure 9: P99 and P99.9 latency under Zipfian distribution at 256 threads. SOS (left, linear); RocksDB (right, larger scale).

Figure 10(b)). For YCSB-C (64 threads), SOS P99 stays below **172 µs** with only ≈1% cache coverage, validating the one-IO design. At 128 threads under YCSB-A, SOS P99.9 is **5.4 ms** (Figure 9(c))—within our 10 ms production SLA (§5)—while RocksDB reaches **51.9 ms** (Figure 9(d)), exceeding it entirely. We note that RocksDB's P99.9 degradation under YCSB-A is proportionally more severe than its P99, further widening the gap shown in Figure 10(b).

Workload Sensitivity. Under Uniform distribution, the 4 GB read cache is effectively cold (≈0% hit rate), exposing the raw I/O efficiency of each engine. The sharpest contrast appears under YCSB-A (random write): SOS sustains **392 kQPS** at a P99 of **1.1 ms**, while RocksDB manages only 33 kQPS at a P99 of **30.6 ms** (**12.0**× throughput gap, **26.7**× P99 gap, Figure 10(c)(d)). Every incoming key lands at an arbitrary position within RocksDB's SSTable hierarchy, maximising compaction fragmentation; SOS writes to the hash-indexed write cache and incurs no such overhead. For read-only YCSB-C, SOS delivers **554 kQPS**—still **3.2**× above RocksDB's best-case Zipfian result (175 kQPS), confirming that the performance gap does not depend on cache locality.

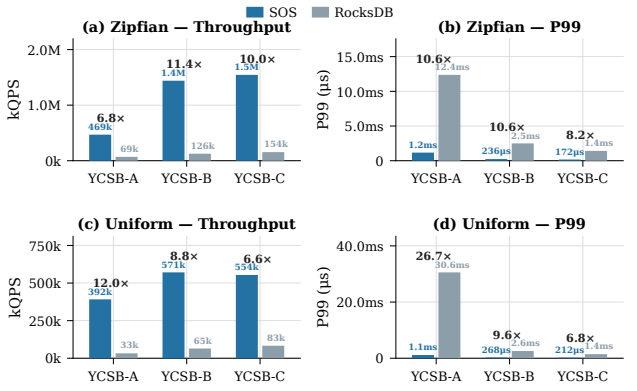

Figure 10: Throughput and P99 at 64 threads. Top: Zipfian; Bottom: Uniform.

4.3 Write and Read Amplification

We measure the Write Amplification Factor (WAF) and Read Amplification Factor (RAF) to quantify each system's I/O overhead. Results are illustrated in Figure 11.

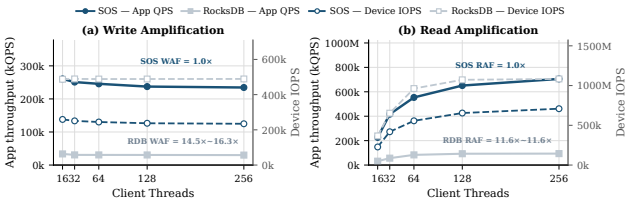

Figure 11: WAF (a) and RAF (b) vs. thread count. Solid: application QPS (left axis); dashed: device IOPS (right axis).

Write Amplification. SOS achieves WAF = 1.00× across all thread counts; RocksDB exhibits 14–16× due to mandatory LSM compaction (Figure 11(a)).

Read Amplification. SOS maintains a constant RAF of 1.00×: every read resolves to exactly one physical I/O. As shown in Figure 11(b), SOS application throughput therefore tracks the device's peak IOPS capacity directly, whereas RocksDB incurs an RAF of ≈11.6× due to its multi-level search process—each cache miss must probe bloom filters and read blocks at potentially every LSM level before returning a result.

4.4 Component Ablation

We present three micro-experiments that probe the contribution of SOS's cache, index layers and variable-sized disk storage in isolation.

Cache sensitivity (Figure 12). The SOS/RocksDB ratio decreases monotonically with hit rate: 25.5× (all-hit) → 10.0× (Zipfian, ≈1% hit) → 6.6× (Uniform, ≈0% hit). The 25.5× gap under all-hit (zero disk I/O) reflects pure CPU overhead; the ratio does not converge to 1× as I/O pressure grows, confirming that CPU overhead and I/O amplification contribute independently.

Index layer (Figure 12). Querying only absent keys removes all disk I/O, isolating the index or filter structure. SOS performs a single hash probe per miss; RocksDB evaluates a per-level Bloom filter chain. Both systems reach ≈29–30 M QPS at 64 threads(**1.047**×), confirming neither index design produces a measurable advantage for negative lookups. This null result is informative: the throughput

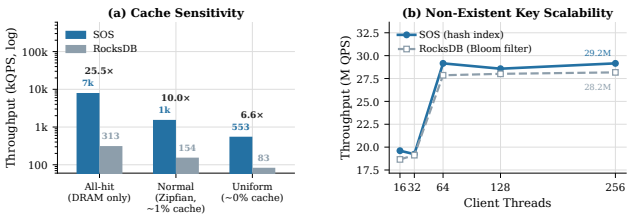

Figure 12: (a) YCSB-C throughput across cache regimes at 64 threads (log scale). (b) Negative-key lookup scalability, 16–256 threads.

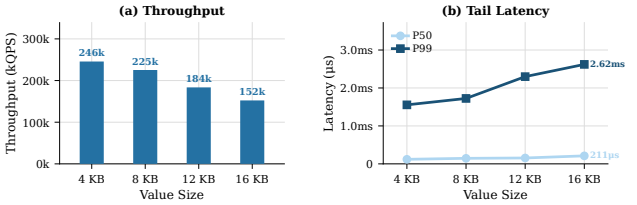

Figure 13: Value-size sensitivity (100% Write, Uniform, 64 threads). (a) Throughput; (b) P50 and P99 latency, both degrading smoothly with payload size.

gains in Section 4.2 arise from I/O amplification and compaction overhead, not index-layer efficiency.

Variable-size payload of disk storage. (Figure 13). As value size increases from 4 KB to 16 KB, throughput and latency degrade smoothly and proportionally to I/O payload size (Figure 13), confirming that the step-sequence block layout introduces no structural overhead beyond the larger per-write transfer and that the one-I/O guarantee holds across the full range of tested object sizes.

4.5 Scalability Analysis

SOS is designed to scale linearly across all dimensions.

- **Vertical scalability.** The $O(1)$ hash lookup and single-I/O guarantee mean single-node throughput grows linearly with thread count until device bandwidth is exhausted, as visible in Figure 8. During node repair or data migration, the affected node runs one additional thread performing sequential segment scans; the impact on foreground latency is bounded and does not affect other nodes.
- **Horizontal scalability.** The routing table provides $O(1)$ node lookup independent of cluster size N. Expanding from N to $N+M$ nodes requires only a routing table version increment and targeted segment migration, with no global re-indexing. During expansion, the routing table has not yet been updated on clients, so in-flight requests complete against the existing layout with no disruption.
- **Replication scalability.** For replication factor R, write cost scales as $O(R)$: each write must be committed to all replicas via chain replication before the client is acknowledged. Read latency is unaffected since reads query only the primary replica, so adjusting R scales read capacity and fault tolerance at a predictable per-hop write overhead. During expansion, the new nodes act as additional chain links; each write incurs one extra network hop per newly added replica, with no contention on existing nodes.

5 PRODUCTION EXPERIENCE

SOS has been deployed at Tencent for several years across over **40 production clusters**, powering diverse services including Yuanbao (AI search and RAG), Hunyuan (LLM inference), Tencent Video, Tencent News, QQ Browser, web search engine, and recommendation systems. Total deployment exceeds 50 PB across all clusters. We highlight three representative cases:

Portrait South Cluster (192 instances, 3-replica). Compared to the legacy storage system, SOS reduces client-observed average read latency from 1.7 ms to **0.7 ms** (a 60% improvement), with P99.9 stable at 5.0 ms. Moreover, SOS significantly optimizes the daily data ingestion pipeline, reducing the end-to-end processing time from **20 hours to 9 hours**, while maintaining a 100% success rate. Figure 14 shows 48-hour traces; P99.9 write latency remains under 9 ms, consistent with single-node benchmarks.

Yuanbao South Cluster (912 instances, 2-replica). As one of the largest deployment, it stores **1.66 trillion keys** ($1,664.9 \times 10^9$) keys across **15.9 PB** of capacity, with the dataset still growing rapidly. It demonstrates SOS's horizontal scalability in practice: DataServers are added with only a routing-table update, and data migration proceeds without service interruption.

Yuanbao East Cluster (320 instances, 3-replica). This cluster sustains a peak SCAN throughput of 62 M QPS, validating the stateful SCAN path at production scale.

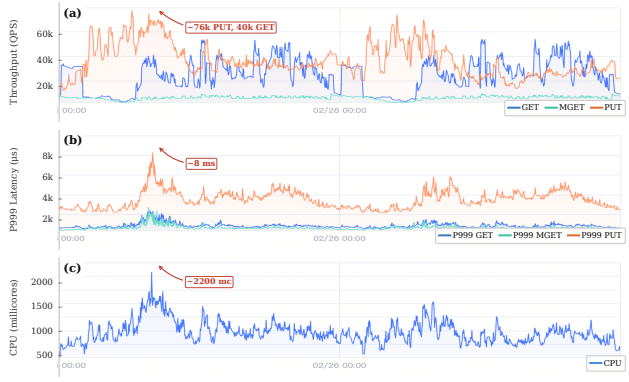

Figure 14: Portrait South Cluster, single instance, 48-hour traces (Feb. 25–26, 2026). Server-side write latency includes 0 to 2 replica hops. (a) QPS peaks: GET ≈50 k, PUT ≈80 k. (b) P99.9: PUT <8 ms; GET <500 µs. (c) CPU < 2,200 mc.

Cost Efficiency. DRAM-based storage (e.g., Redis) is 10×–30× more expensive per GB than NVMe-based SOS in our infrastructure. For production services migrated from Redis to SOS, we observed an average **TCO (Total Cost of Ownership) reduction of 84%** based on internal billing data. This significant saving, while maintaining millisecond-level tail latency, validates SOS's fitness for petabyte-scale internet application and AI serving.

6 RELATED WORK

In-Memory and LSM-tree Stores. Redis [30] and Memcached [12] achieve sub-millisecond latency but cost roughly 10× more per TB than NVMe SSDs. RocksDB [3] is the de-facto disk KV standard, yet incurs WAF of 10–30× [11] and RAF of up to L per cache miss [8]. WiscKey [25], SILK [1], and Titan [29] mitigate compaction I/O without eliminating amplification. SOS achieves WAF = RAF = 1.00× by enforcing at most one physical disk I/O per operation. Anna [37] achieves extreme read scalability via lattice-based coordination-free design but targets eventual consistency, lacking the strong consistency and TTL semantics required here. FASTER [5] shares SOS's one-I/O principle via a hybrid log but is optimized for single-node throughput rather than distributed strong consistency.

Hash-indexed and Distributed Stores. Bitcask [31] and SILT [24] use hash indexing for $O(1)$ lookup but require periodic merge to reclaim space from append-only logs; SOS uses in-place overwrites, eliminating this overhead. HDFS [32] and HBase [13] target sequential workloads; their LSM-tree random reads and GC pauses preclude millisecond SLAs, yet HBase is widely adopted for RAG corpora, where cold lookups experience P99 above 10 ms.

Consistency Protocols and AI-serving Systems. Classic chain replication [35] serves reads from the tail (R hops). CRAQ [34] scales read throughput via multi-replica access, but dirty reads require tail version queries, increasing latency under write contention. DeepSeek 3FS [10] is a recent AI-oriented system targeting massive sequential bandwidth (6.6 TiB/s) for training, adopting CRAQ for its replication. In contrast, SOS optimizes for per-request tail latency (P99.9 < 5 ms) in AI serving via a structured KV interface with strong consistency. By routing all reads to the primary (head), SOS eliminates tail round-trips and version queries entirely—preserving linearizability while trading replica-side read fan-out for consistently lower single-request latency, the primary constraint for our production workloads. Raft-based [28] systems (TiKV [18]) provide strong consistency at multi-RTT commit cost. FoundationDB [39] and Dynamo [9] offer general transactions or eventual consistency, unsuited to microsecond AI-serving SLAs. Hash indexes sacrifice range queries for $O(1)$ point access and 1× amplification—an acceptable trade-off since recommendation, RAG, and KV-cache [22, 36] access patterns require only exact-key lookup, not range traversal.

Limitations. SOS does not support range queries, as the hash index sacrifices key ordering for $O(1)$ lookup. For RAG applications, it requires a separate vector search engine for nearest-neighbor candidate identification. Currently, domain-specific logic—such as mapping multi-turn KV-cache context or encoding composite structures (maps, lists) for recommendations—must be managed by the application or serving framework. Tighter LLM scheduler integration, native composite-type support, and per-tenant QoS to mitigate noisy-neighbor effects are left for future work.

7 CONCLUSION

SOS enforces at most one physical disk I/O per operation via co-designed hash index, step-sequence storage, and asynchronous write cache, achieving WAF = RAF = 1.00×, 1.86 M QPS peak, and sub-250 µs P99 reads. Two-Phase Chain Replication with in-chain retry delivers near-zero write failures. Deployed at petabyte scale across Tencent's AI services at P99.9 < 5 ms, SOS demonstrates that the one-IO principle is a viable blueprint for latency-sensitive large-scale storage.

ACKNOWLEDGMENTS

We thank the SRE teams for their support in deploying and maintaining SOS at scale. We are grateful to the SDK developers who built client libraries for multiple programming languages. We also thank the service teams across Tencent who adopted SOS in production and continuously drove improvements to the system.

REFERENCES

[1] Oana Balmau, Florin Dinu, et al. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proc. USENIX ATC*. 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[2] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. USENIX Summer Technical Conference*. 87–98.

[3] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB KV Store Workloads at Facebook. In *Proc. FAST*. 1–16. https://www.usenix.org/conference/fast20/presentation/cao

[4] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (1996), 225–267. https://doi.org/10.1145/226643.226647

[5] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proc. ACM SIGMOD*. 275–290. https://doi.org/10.1145/3183713.3196898

[6] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. ACM Symposium on Cloud Computing (SoCC)*. 143–154. https://doi.org/10.1145/1807128.1807152

[8] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proc. SIGMOD*. 79–94. https://doi.org/10.1145/3035918.3064054

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. SOSP*. 205–220. https://doi.org/10.1145/1294261.1294281

[10] DeepSeek-AI. 2025. 3FS: The Fire-Flyer File System. https://github.com/deepseek-ai/3FS. Accessed: 2026-03-02.

[11] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2017. Optimizing Space Amplification in RocksDB. In *Proc. CIDR*. http://cidrdb.org/cidr2017/papers/p2-dong-cidr17.pdf

[12] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (2004), 5.

[13] Lars George. 2011. *HBase: The Definitive Guide*. O'Reilly Media.

[14] Sanjay Ghemawat and Jeffrey Dean. 2011. LevelDB: A Fast and Lightweight Key/Value Database Library by Google. https://github.com/google/leveldb. Accessed: 2026-02-01.

[15] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley.

[16] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM TOPLAS* 13, 1 (1991), 124–149. https://doi.org/10.1145/114005.102808

[17] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[18] Dongxu Huang, Qi Liu, Qiyu Cui, et al. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB* 13, 12 (2020), 3072–3084. https://doi.org/10.14778/3407706.3407711

[19] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. STOC*. 654–663. https://doi.org/10.1145/258533.258660

[20] Ana Klimovic, Yawen Wang, Christos Kozyrakis, et al. 2018. Pocket: Elastic Ephemeral Storage for Serverless Applications. In *Proc. OSDI*. 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[21] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. SOSP*. 611–626. https://doi.org/10.1145/3600006.3613162

[23] Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Proc. NeurIPS*. 9459–9475. https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[24] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. SOSP*. 1–13. https://doi.org/10.1145/2043556.2043558

[25] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, et al. 2016. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *Proc. FAST*. 133–148. https://doi.org/10.1145/3033273

[26] Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press. https://doi.org/10.1017/CBO9780511813603

[27] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 1 (1996), 351–385. https://doi.org/10.1007/s002360050048

[28] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc. USENIX ATC*. 305–320. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[29] PingCAP. 2019. Titan: A RocksDB Plugin for Key-Value Separation. https://github.com/tikv/titan. Accessed: 2026-02-01.

[30] Salvatore Sanfilippo. 2009. Redis: An Open Source, In-Memory Data Structure Store. https://redis.io. Accessed: 2026-02-01.

[31] Justin Sheehy and David Smith. 2010. *Bitcask: A Log-Structured Hash Table for Fast Key/Value Data*. Technical Report. Basho Technologies. https://riak.com/assets/bitcask-intro.pdf

[32] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proc. MSST*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[33] Sogou Inc. 2019. C++ Workflow: Sogou's C++ Server Engine. https://github.com/sogou/workflow. Accessed: 2026-03-01.

[34] Jeff Terrace and Michael J. Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *Proc. USENIX ATC*. 11–24. https://www.usenix.org/conference/atc09/presentation/terrace

[35] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Availability and Strong Consistency. In *Proc. OSDI*. 91–104. https://www.usenix.org/conference/osdi-04/chain-replication-supporting-high-availability-and-strong-consistency

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. 2017. Attention Is All You Need. In *Proc. NeurIPS*. 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[37] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2019. Anna: A KVS for Any Scale. *IEEE Transactions on Knowledge and Data Engineering* 31, 2 (2019), 344–358. https://doi.org/10.1109/TKDE.2018.2879262

[38] Xinjing Zhang et al. 2022. IonCache: A High-Performance KV Storage Engine with Log-structured Merge-Tree and Flash-aware Cache. *Proc. VLDB Endow.* 15, 11 (2022), 2533–2546. https://doi.org/10.14778/3551793.3551812

[39] Jingyu Zhou, Meng Xu, Alexander Shraer, et al. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proc. SIGMOD*. 2653–2666. https://doi.org/10.1145/3448016.3457559

[40] George K. Zipf. 1949. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Cambridge, MA.