Terark-DS: A High-Performance and Storage-Efficient Key-Value Separation Storage Engine on Disaggregated Storage

Jianshun Zhang*
Xun Deng*
Fang Wang
Huazhong University of Science and Technology, China
{shunzi,dorence,wangfang}@hust.edu.cn

Jiaxin Ou
Yi Wang
Hao Wang
Jianjun Chen
ByteDance
{oujiaxin,wangyi.ywq,hao.wang,jianjun.chen}@bytedance.com

Peng Fang†
Dan Feng
Huazhong University of Science and Technology, China
{fangpeng,dfeng}@hust.edu.cn

ABSTRACT

Log-structured merge-trees (LSM-trees) are widely adopted in modern storage systems for high write throughput, but suffer from significant write amplification. Key-value (KV) separation mitigates this issue but introduces higher space overhead. To improve cost efficiency and resource elasticity, modern storage systems increasingly adopt compute-storage disaggregated architectures. However, disaggregation increases network overhead for data access, degrading write performance. It also prolongs garbage collection (GC), which increases the space cost of KV-separated LSM-trees.

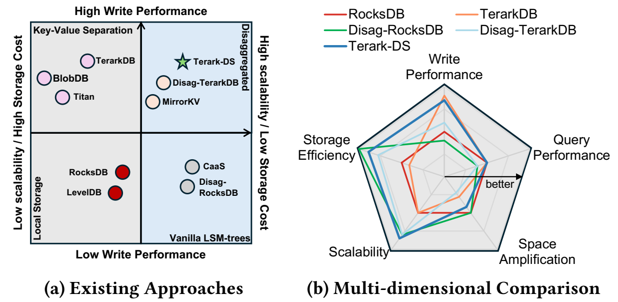
In this paper, we propose Terark-DS, a high-performance and storage-efficient KV separation storage engine on disaggregated storage. To achieve both high performance and low cost, Terark-DS employs differentiated redundancy based on LSM-tree access patterns, adaptive write-ahead logging that switches between serial and parallel modes for different batch sizes, and a network-efficient GC design to accelerate GC execution. Experiments show that Terark-DS outperforms existing disaggregated LSM-trees by 20.4%-63.9% in write throughput while reducing total costs by 22.7%-58.6%.

PVLDB Reference Format:
Jianshun Zhang, Xun Deng, Fang Wang, Jiaxin Ou, Yi Wang, Hao Wang, Jianjun Chen, Peng Fang, and Dan Feng. Terark-DS: A High-Performance and Storage-Efficient Key-Value Separation Storage Engine on Disaggregated Storage. PVLDB, 19(5): 822 - 835, 2026.
doi:10.14778/3796195.3796198

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SZ-NPE/terark-ds.

1 INTRODUCTION

LSM-trees are prevalently used in distributed databases [1–3], file systems [4, 5], and stream processing engines [6] due to their high write throughput, which makes them popular in write-intensive scenarios [7]. However, as storage scales up, write amplification emerges as a critical bottleneck, severely limiting further performance gains [8, 9]. For example, traditional LSM-tree implementations, such as LevelDB [10] and RocksDB [11], rely on frequent compactions to maintain sorted tree structures. Nonetheless, this process causes severe write amplification, often reaching 10× or higher, significantly reducing write throughput by up to 90% [8, 12, 13].



(a) Existing Approaches (b) Multi-dimensional Comparison

Figure 1: Analysis of Disaggregated LSM-trees

To mitigate write amplification, WiscKey [12] introduced key-value (KV) separation by storing only keys and indexes in the LSM-tree and placing values in separate data files. This design alleviates write amplification by reducing the volume of data involved in compaction and has been adopted by industrial systems such as Titan [14], BlobDB [15], and TerarkDB [16]. As reported in a recent study [17], these KV-separated LSM-tree systems achieve 1.73×–4.31× improvements in write throughput, but also incur 34.7%–106.3% higher space amplification due to inefficient garbage collection (GC), which increases storage costs.

Traditional LSM-trees [10, 11] employ a monolithic architecture, with compute and storage tightly coupled within the same physical node, limiting their scalability as data volumes grow due to constrained resource elasticity and escalating compaction overhead. Recently, cloud providers have increasingly adopted compute-storage disaggregation to enhance cost efficiency and resource elasticity, as exemplified by systems like Aurora [18] and PolarDB [19]. However, our evaluations reveal that disaggregation introduces substantial network overhead, reducing LSM-tree write throughput by 34.9%–45.5% and hindering scalability under large-scale workloads, as shown in Figure 1(b). Although KV separation mitigates this performance degradation by reducing network overhead, it exacerbates space amplification, with storage costs increasing by up to 74.1%

*Both authors contributed equally to this research.
†Corresponding Author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796198

under mixed small and large KV pair workloads, further limiting cost-efficiency and scalability in disaggregated environments.

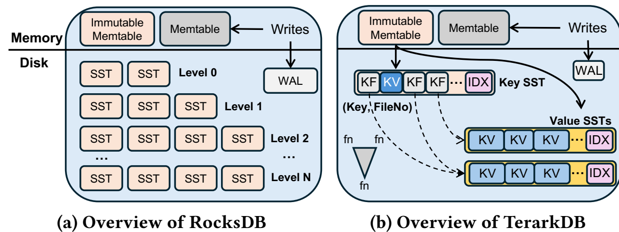
Recent efforts have sought to optimize disaggregated LSM-trees. Disag-RocksDB [20] focuses on compaction parameter tuning and I/O scheduling, while CaaS [21] mitigates performance penalties by offloading compaction tasks to dedicated nodes, but both fail to address write amplification, leading to resource wastage that limits scalability under heavy workloads. As a result, they deliver suboptimal write performance, as shown in Figure 1(a). MirrorKV [22] integrates KV separation into disaggregated architectures to alleviate write amplification. However, its scalability is limited by costly value compaction and additional storage cost, which lead to sub-optimal performance and delays space reclamation, reducing its effectiveness in disaggregated settings.

While prior work has optimized disaggregated LSM-trees, two fundamental challenges remain, as shown in Figure 1(b). **First**, disaggregation severely degrades write performance due to network overhead. On the one hand, internal compaction amplifies network traffic. Our evaluations show that with 3-replication policy, it consumes up to 87.8% of the NIC bandwidth, leaving little for user writes. On the other hand, synchronous WAL writes to remote storage introduce network latency absent in monolithic setups, and our tests reveal this increases write latency by 44.6%. These bottlenecks persist in existing efforts. **Second**, KV separation provides limited benefits under disaggregated architecture and introduces additional space costs. It does not alleviate network overhead caused by disaggregation, and its effectiveness is further limited by inefficient GC. Our experiments show that frequent remote accesses increase GC latency by 2.03×, which increases storage overhead.

To address these challenges, we propose Terark-DS, a high-performance and storage-efficient KV-separated storage engine optimized for disaggregated architecture. Compared to existing approaches, Terark-DS reduces network overhead and enhances write performance through differentiated redundancy strategies and an adaptive WAL writing mechanism. Besides, it reduces overall storage costs via network-efficient GC. As shown in Figure 1(b), Terark-DS is designed to match the performance of KV-separated LSM-trees deployed on local SSDs while reducing storage costs close to those of vanilla LSM-trees, delivering a better performance-cost tradeoff for LSM-trees in disaggregated architectures.

The contributions of this work are summarized as follows:

- **Differentiated Redundancy Strategies**: We propose a differentiated redundancy strategy based on the access characteristics of different files in KV-separated LSM-trees. To guide design choices, we develop analytical models that quantify the trade-off between performance and storage cost.
- **Adaptive WAL Writing**: We identify write-ahead logging (WAL) as a key bottleneck in the critical write path of disaggregated LSM-trees and introduce an adaptive WAL writing strategy that dynamically switches between serial and parallel writes based on task granularity. This design enhances the foreground write throughput of disaggregated LSM-trees.
- **Network-Efficient Garbage Collection**: We propose network-efficient garbage collection (GC) for disaggregated environments. It reduces network overhead via three techniques: *On-Demand Value Fetching* avoids transmitting redundant keys and invalid values; *Batched and Localized GC-Lookup* uses *Flat Index Cache* to reduce RPCs and enable GC offloading. *Adaptive Readahead* merges fragmented data to minimize network round-trips. Together, these techniques lower storage overhead and improve cost-efficiency in disaggregated environments.
- **Comprehensive Evaluation**: We implement Terark-DS based on TerarkDB, and evaluate it on ByteDance's disaggregated storage system. Extensive evaluations across diverse workloads demonstrate that Terark-DS achieves better write performance and lower cost than other disaggregated solutions.



(a) Overview of RocksDB (b) Overview of TerarkDB

Figure 2: Vanilla LSM-tree and Key-Value Separation

2 BACKGROUND

2.1 Log-Structured Merge Trees

Vanilla Log-Structured Merge Trees (LSM-trees) comprise in-memory write buffers (**Memtable**), on-disk write-ahead logs (**WAL**), and sorted string tables (**SST**), as shown in Figure 2(a). Data is first appended to the WAL for durability, then inserted into the Memtable for fast access. When a Memtable fills up, it becomes immutable, and a background flush thread sequentially persists it to disk as SST files, exploiting the advantages of sequential writes. The SST files are organized into a multi-level structure to optimize queries. Background compactions continually merge and sort SST files into new ones at lower levels, keeping all levels except level 0 sorted. Queries first check the Memtable and then search for SST files in level 0. If not found, queries proceed to lower levels (L_1 to L_N). The search terminates once the entry is located, avoiding extra lookups.

2.2 Key-Value Separation

LSM-trees enable high write performance via append-only writes but require frequent compaction to maintain read efficiency, causing write amplification [8, 12, 23]. To address this, WiscKey [12] introduced key-value separation, storing values separately while retaining keys and indexes in the index LSM-tree. This design reduces the size of the LSM-tree and alleviates compaction-induced write amplification. Due to its simplicity and effectiveness, KV separation has become a widely adopted optimization for LSM-trees [17, 24–28], with implementations including Titan [14] and BlobDB [15]. ByteDance has also successfully deployed the KV-separated storage engine, TerarkDB [16, 29], at scale, providing high-performance and cost-effective key-value storage service for cloud-native databases and large-scale stream processing engines [29, 30].

TerarkDB. We use TerarkDB [16, 29] to elucidate key-value separation, as it outperforms both vanilla LSM-trees (e.g., RocksDB) and KV-separated alternatives (e.g., Titan [14]) in write performance. As shown in Figure 2(b), TerarkDB selectively separates large KV pairs (≥ 512B). Small pairs and indexes of large KVs reside in the **Key SSTs**, while large values are stored in **Value SSTs**. Key

Table 1: Tradeoffs of Different Redundancy Strategies

Redundancy Strategy	Fault Tolerance (#failures)	Cost (× original size)		Performance (↑ better, ↓ worse)	
		Storage Volume	Network Bandwidth	User Read/Write	Data Repair
Replication (N)	N-1	N	N	↑	↑
Erasure Coding (M:N-1)	N-1	(M+N-1) / M	(M+N-1) / M	↓	↓

SSTs store the <*Key*, *File Number*> mapping to reference large KVs in Value SSTs, with both SSTs structured as ordered tables with internal indexes for efficient reads. Compared to vanilla RocksDB, TerarkDB limits compaction to smaller Key SSTs, reducing write amplification and achieving higher write throughput [17, 31].

For Value SSTs, space reclamation relies on garbage collection (GC). This process frequently relocates valid data, which can invalidate file number references in Key SSTs. To minimize index update overhead during GC, TerarkDB maintains an **inheritance tree of file numbers** to track relationships between old and new Value SST identifiers. This design enables efficient data redirection with minimal interference to foreground writes. It avoids the resource competition between GC and foreground writes that affects KV-separated alternatives like Titan, ensuring more stable write performance under heavy GC load [17, 32]. While KV separation alleviates write amplification, it introduces higher storage costs than vanilla LSM-trees due to delayed value space reclamation, as shown in Figure 1(a). Approaches such as HashKV [24], DiffKV [25], BlobDB [15], and Scavenger [17] optimize GC to reduce space amplification but remain less space-efficient than vanilla RocksDB.

2.3 Disaggregated Storage

Before exploring the challenges of building storage engines based on disaggregated storage, we first discuss its key characteristics.

2.3.1 *Better Scalability and Elasticity.*

Disaggregated architecture decouples computation and storage resources, enabling dynamic resource allocation and independent scaling.

Efficient and Flexible Data Partitioning. Independent scaling enables dynamic partitioning of storage based on workload characteristics, reducing unnecessary data migration overhead [33]. Computing resources can be dynamically allocated to partitions, improving load balancing and mitigating data hotspots [34, 35].

Elastic and Rapid Deployment. Stateless computing reduces inter-node coordination overhead, enabling rapid demand-driven instantiating and reclaiming of compute nodes to improve elasticity [36]. Disaggregation naturally supports serverless models with stateless, on-demand execution and efficient resource usage [31, 37].

2.3.2 *Higher Bandwidth but Increased Latency.*

Disaggregated storage improves aggregate I/O throughput by allowing multiple compute nodes to access a shared pool of SSDs. This architecture removes the bandwidth constraints of traditional monolithic LSM-tree deployments, where each instance typically uses a single local SSD through a kernel file system. With sufficient network provisioning, disaggregated storage can achieve higher I/O bandwidth by enabling parallel access to more storage devices. Optimizations such as SPDK [38, 39] can further reduce software overhead of I/O stack.

Disaggregated storage incurs unavoidable latency penalties due to the network hops between compute and storage nodes. High-speed network technologies like RDMA and InfiniBand alleviate communication overhead [40], but fail to eliminate remote round-trip costs. Moreover, when nodes span RDMA-inapplicable regions, aggravated network overhead further increases latencies.

2.3.3 *Reduced Hardware Cost and Flexible Redundancy Strategies.*

Disaggregated storage reduces hardware costs through optimized resource allocation and enables flexible redundancy strategies to balance performance and cost, as summarized in Table 1.

Reduced Hardware Cost. By decoupling compute and storage, resource provisioning becomes more efficient. Storage nodes are typically configured with high-capacity and high-bandwidth drives while relying on modest CPU and memory resources, thereby improving resource utilization and reducing hardware costs [41].

Flexible Redundancy Strategies. To ensure data reliability, disaggregated storage systems commonly adopt either **Replication** or **Erasure Coding (EC)**, each offering distinct trade-offs.

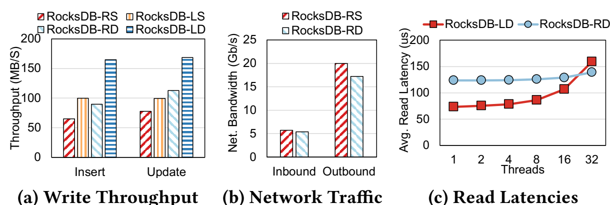
- **Storage and Network Overhead**. Replication maintains N full copies of the data across N failure domains. As shown in Table 1, it incurs $N\times$ storage overhead and requires $N\times$ the original data volume in network bandwidth during data transfer. In contrast, erasure coding (EC) splits the data into M blocks and generates $N-1$ parity blocks, distributing them across $M+N-1$ domains. Both storage and network overhead are thereby reduced to $\frac{M+N-1}{M}\times$ the original data size.
- **Performance Trade-offs**. Replication enables low-latency access via ready-to-use full copies, with minimal computational overhead. In contrast, EC introduces computational costs from encoding/decoding, increasing access latency, particularly for small objects. During recovery, replication fetches data directly from a surviving copy, whereas EC must reconstruct it from multiple blocks, resulting in higher latency.

3 MOTIVATION

Disaggregation enhances scalability and cost-effectiveness but incurs performance degradation due to network overhead. KV separation mitigates this by reducing compaction-related NIC bandwidth contention, though its benefits are limited and increase storage overhead. To understand these challenges, we analyze the performance impact of disaggregation and the trade-offs of KV separation.

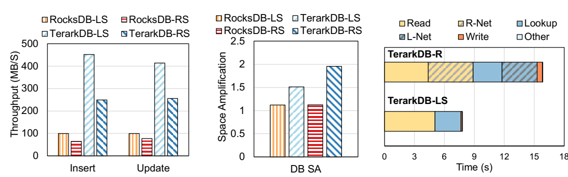
3.1 Disaggregation: Hurting Performance

We evaluate disaggregated LSM-trees using RocksDB [11], a popular LSM-tree implementation, on ByteDance's disaggregated storage system with six storage nodes and 3-way replication. Compute nodes hosting RocksDB instances access remote storage via a 25



(a) Write Throughput (b) Network Traffic (c) Read Latencies

Figure 3: Performance of Disaggregated RocksDB



(a) Write Throughput (b) Space Amp. (c) GC Latency breakdown

Figure 4: Performance of Disaggregated TerarkDB

Gbps network, forming the **RocksDB-RS** configuration (remote storage, WAL enabled), compared against a local instance with synchronous WAL writes (**RocksDB-LS**). To isolate WAL's impact, we add WAL-disabled variants: **RocksDB-LD** (local) and **RocksDB-RD** (remote). The workload uses a realistic mix of small and large KV pairs, reflecting data patterns in cloud-native databases [17] where small values derive from incremental updates and large values from data pages. The benchmark loads a 100 GB dataset, followed by 300 GB of continuous updates to simulate sustained write pressure, and measures read latency under varying thread counts. Additional experimental settings are detailed in Section 5.1.

Figure 3(a) presents the write performance of RocksDB across configurations. Specifically, **RocksDB-RS** exhibits 34.9% lower throughput than the local baseline **RocksDB-LS**. While WAL disablement improves performance, yielding 38.4% higher throughput than **RocksDB-RS**, it still lags **RocksDB-LS** by 45.5%, indicating a persistent network-induced performance gap. For read performance, as shown in Figure 3(c), **RocksDB-RS** shows 1.21× to 1.68× higher latency than **RocksDB-LS** with fewer than 16 threads. As thread count reaches 32, **RocksDB-LS** saturates local SSD bandwidth, causing a sharp latency spike. In contrast, **RocksDB-RS** maintains more stable latency due to the greater aggregate bandwidth of disaggregated storage. These results confirm that disaggregation imposes significant penalties on both write throughput and read latency. We further analyze the root causes below.

3.1.1 Disaggregation Introduces Network Bottlenecks. Disaggregated storage adds a network layer absent in local SSD access. While local NVMe SSDs communicate directly over PCIe, disaggregated storage incurs network overhead due to remote data access. Advanced technologies such as RDMA and InfiniBand can significantly reduce latency, but they cannot fully eliminate the cost of network round-trips. In practice, the adoption of RDMA is often constrained by cross-datacenter or cross-rack network topologies.

Redundancy strategies further exacerbate network overhead. In ByteDance's disaggregated storage systems, a multi-replica policy is typically enforced, and a star-topology write strategy is adopted to minimize write latency. In this strategy, clients concurrently issue RPCs to multiple storage nodes without dependencies. However, individual clients are often limited by their NIC bandwidth (e.g., 25 Gbps), which restricts the efficiency of user data transfer. As shown in Figure 3(b), under I/O-intensive workloads, the surge in RPC traffic from replicated writes rapidly saturates the available bandwidth of NICs. This saturation leads to increased queuing delays for subsequent requests, ultimately degrading system throughput.

3.1.2 LSM-Tree Limitations Aggravated by Disaggregation. LSM-trees achieve high write throughput by leveraging sequential writes and accelerating WAL operations through the page cache. Typically, WAL data is first buffered in the page cache, enabling fast write acknowledgments while preserving crash consistency. Data persistence is enforced via sync operations during user writes, significantly increasing write latency. In disaggregated architectures, managing page cache behavior becomes infeasible. As a result, every WAL record must be persisted via remote procedure calls (RPCs) to the disaggregated storage to guarantee durability. This enforces sync-level durability semantics without explicit synchronization in write options, but it also introduces substantial network overhead. The resulting latency increase on the critical write path limits the effective utilization of disaggregated storage bandwidth.

Moreover, frequent compactions aggravate write amplification, resulting in I/O contention in local deployments and excessive RPC traffic in disaggregated environments. Compaction competes for bandwidth with user requests over limited network bandwidth. As shown in Figure 3(b), write amplification in disaggregated RocksDB drives network utilization close to the 25 Gbps limit. This not only prolongs compaction duration but also intensifies LSM-tree throttling, leading to write stalls and degraded foreground performance. While disabling WAL can relieve network pressure and temporarily boost foreground throughput, delayed compactions still cause write stalls. Consequently, disaggregated deployments continue to suffer inferior performance compared to their local counterparts.

3.2 Separation: Limited Benefits, Higher Costs

Key-value separation has proven effective in local SSD deployments by reducing write amplification [12, 24]. However, its effectiveness in disaggregated storage remains uncertain, as it introduces new trade-offs between performance gains and storage efficiency.

3.2.1 Limited Performance Benefits. To evaluate the performance benefits of KV separation in disaggregated architectures, we assess disaggregated TerarkDB (**TerarkDB-RS**) with the same setups. As shown in Figure 4(a), KV separation improves write throughput by 3.8×, partially offsetting the performance degradation caused by disaggregation. Nonetheless, a 44.9% gap persists compared to local TerarkDB (**TerarkDB-LS**). This is because KV separation does not eliminate the inherent network overhead of disaggregation. Frequent RPCs for user writes introduce additional latency, and NIC bandwidth limitations further constrain throughput. While KV separation alleviates write amplification, its benefits are partly negated by remote access costs and network bottlenecks.

3.2.2 Higher Space Costs. KV separation imposes additional storage overhead [17], which becomes particularly problematic in disaggregated environments. Space amplification arises from inefficient garbage collection (GC) constrained by network latency and

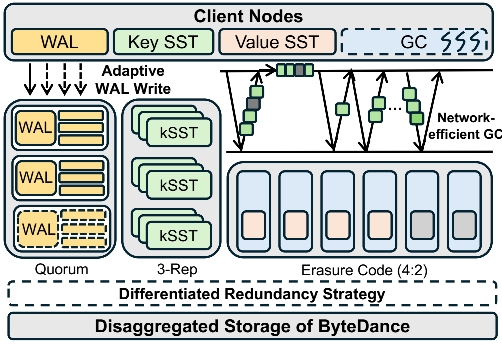


Figure 5: Architecture of Terark-DS

bandwidth limitations, leading to excessive storage consumption. Moreover, redundancy mechanisms such as multi-replica storage further exacerbate the overall storage cost. To quantify these effects, we preload a 100 GB dataset and apply a rate-limited 200 MB/s update workload over 1800 seconds to simulate sustained and consistent write pressure. As shown in Figure 4(b), TerarkDB exhibits a space amplification factor of 1.51× under local configurations (**TerarkDB-LS**), which increases to 1.96× in disaggregated deployments (**TerarkDB-RS**). In contrast, RocksDB maintains a consistently lower amplification of 1.12× across both environments.

To further understand the causes of space amplification, we conduct a fine-grained latency breakdown of the key steps in GC. As shown in Figure 4(c), network overhead accounts for 28.4% and 22.2% of the total GC latency in the read and lookup phases during GC, respectively. This additional latency extends the duration of GC cycles, delays space reclamation, and results in the accumulation of obsolete data, ultimately increasing the overall storage footprint.

4 DESIGN

Terark-DS is a high-performance, storage-efficient key-value engine that adapts LSM-trees to disaggregated architectures by mitigating network-induced performance and storage overheads.

4.1 Architecture of Terark-DS

Terark-DS addresses performance and cost challenges in disaggregated LSM-trees by optimizing redundancy, write-ahead logging (WAL), and garbage collection (GC), as shown in Figure 5. Storage nodes manage WALs, key SSTs (kSSTs), and value SSTs (vSSTs), with compute nodes determining data transmission methods.

Built on KV separation, Terark-DS mitigates the network bandwidth amplification of traditional LSM-trees by restricting compaction to smaller kSSTs. As shown in Figure 5, it employs a **differentiated redundancy strategy** tailored to access patterns of different files: storage nodes apply quorum-based replication for WALs, 3-replication for kSSTs, and erasure coding (4:2) for vSSTs, guided by compute nodes. This design balances performance and cost by minimizing redundancy-induced transmission overhead.

With bandwidth bottlenecks alleviated, latency becomes a critical factor limiting LSM-trees' write performance. Since WAL writes are serialized, network latency exacerbates write delays. Terark-DS introduces an **adaptive WAL writing** strategy, where compute nodes dynamically adjust modes based on write group size at commit time, processing small groups sequentially and large groups in parallel, as shown in Figure 5. This approach improves user-facing write throughput by parallelizing WAL writes.

Storage cost remains a critical concern for KV-separated LSM-trees. Terark-DS proposes a **network-efficient garbage collection** mechanism, depicted in Figure 5, where compute nodes minimize RPC overhead by reducing round-trips and optimizing data transfer granularity, accessing storage nodes with on-demand reads, request batching, and adaptive readahead. This design lowers GC latency during read and lookup phases, accelerating space reclamation and reducing overall storage consumption.

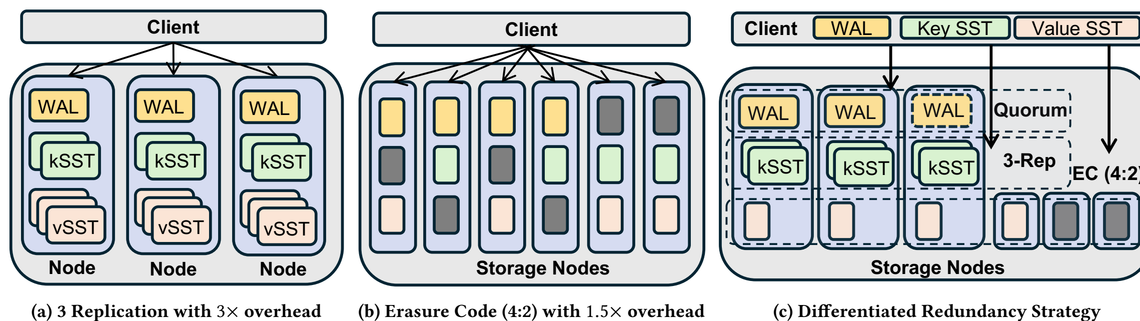
4.2 Differentiated Redundancy Strategy

Most disaggregated systems adopt a single redundancy strategy. As discussed in Section 2.3.3, replication and erasure coding (EC) each offer tradeoffs between performance and cost. Figure 3 shows that disaggregated RocksDB with 3-way replication suffers severe performance degradation due to write amplification and redundancy-induced traffic, which saturates the NIC. KV separation alleviates bandwidth contention but still incurs high storage overhead. EC can improve space and bandwidth efficiency but undermines query performance due to its decoding overhead (as Figure 14 shows).

A uniform redundancy strategy overlooks workload heterogeneity and fails to exploit the distinct access patterns across LSM-tree components. KV separation further amplifies these differences: Key SSTs are small, frequently accessed, and latency-sensitive, whereas Value SSTs are large, infrequently accessed, and throughput-oriented. To better balance performance and storage efficiency, we advocate for a differentiated redundancy design. We begin by characterizing the access behaviors of each file type in KV-separated LSM-trees.

4.2.1 Access Pattern Analysis. A KV-separated LSM-tree organizes data into three distinct file types: (1) **Write-Ahead Logs (WAL)** for crash consistency, (2) **Key SSTables (kSSTs)** storing small key-value pairs and the indexes of large key-value pairs, and (3) **Value SSTables (vSSTs)** storing the large key-value pairs. These files exhibit heterogeneous access patterns:

- **WAL Files** ensure write durability via append-only logging. Reads of WAL files are rare and occur only during crash recovery via large and sequential I/Os. Once the corresponding Memtable is persisted, WALs are promptly deleted, resulting in short lifespans and negligible storage overhead.
- **Key SST Files** handle fine-grained point reads on cache misses, covering both small KVs and index lookups for large KVs. They are also essential for compaction, which involves large sequential I/Os, and participate in GC validity checks (GC-Lookup), exhibiting read patterns similar to user queries. Their storage overhead remains significantly lower than that of Value SSTs.
- **Value SST Files** store large values, with lower access frequency and coarser read granularity than Key SSTs. While Value SSTs are involved in GC with large sequential I/Os, the lower frequency of GC than compaction reduces overall accesses. For instance, under the YCSB-A workload with a 1:1 mix of small and large values, Key SSTs are accessed 46.8% more frequently than Value SSTs. As primary storage for large KV pairs, Value SSTs dominate total storage usage in KV-separated LSM-trees.



(a) 3 Replication with 3× overhead (b) Erasure Code (4:2) with 1.5× overhead (c) Differentiated Redundancy Strategy

Figure 6: Data Layout Under Different Redundancy Strategies.

Distinct access patterns across these file types enable differentiated redundancy strategies, allowing more adaptive trade-offs.

4.2.2 Differentiated Redundancy Strategy. To balance performance and cost, we apply differentiated redundancy to WALs, Key SSTs, and Value SSTs based on their distinct access patterns (Figure 6(c)).

Redundancy Strategy for WAL. WAL resides on the critical write path, with reads occurring rarely during recovery. Given its small footprint and critical role, Terark-DS employs a quorum-based replication strategy with star-topology writes for reliability and performance. In a three-replica setup, a write is committed once persisted on two replicas, with the third updated asynchronously. Read requests query two replicas for consistency. Unlike leader-based consensus protocols requiring synchronous majority coordination, Terark-DS reduces write latency by avoiding centralized control, while ensuring strong consistency, ideal for write-heavy WALs.

Redundancy Strategies for Key SST and Value SST. Key SSTs and Value SSTs exhibit different access patterns, necessitating distinct redundancy policies. We consider three combinations:

- **All Replication**: Provides optimal read and write performance due to fast replica access, but incurs high storage costs. The additional write traffic may saturate the NIC's bandwidth and degrade performance in resource-constrained environments.
- **All Erasure Coding**: Reduces storage cost and network overhead during write operations, but increases read latency due to cross-node access and decoding overhead.
- **Mixed Replication-EC**: Balances performance and cost by replicating Key SSTs for low-latency lookups while applying erasure coding to Value SSTs to reduce storage overhead with slightly higher read latency for infrequent large-value reads.

Key SSTs are involved in every point query, making low-latency access the primary concern. Given their small storage footprint, **3-way replication (R3)** offers fast access with minimal overhead. In contrast, Value SSTs store large values, are accessed less frequently, and typically involve large reads, where storage efficiency is more critical. **Erasure coding (EC)** significantly reduces storage overhead while maintaining acceptable performance for large values. By applying **R3 to Key SSTs** and **EC to Value SSTs**, our design achieves a principled balance between performance and cost.

Although different redundancy strategies enable flexible trade-offs between performance and cost, their effectiveness depends on workload characteristics. Erasure coding suits storage-heavy workloads with infrequent reads, while replication benefits latency-sensitive ones with ample resources. As empirical evaluation cannot cover the full spectrum of workloads, we develop a quantitative model to systematically analyze the performance-to-cost efficiency of various redundancy strategies and guide strategy decisions.

Table 2: Used Notations

Notation	Description
D_k	Total storage size of Key SST files.
D_v	Total storage size of Value SST files.
α	Key-to-Value SST size ratio (D_k/D_v).
L_r^s	Read latency of small I/O under 3-replication (**R3**).
L_r^l	Read latency of large I/O under 3-replication (**R3**).
L_e^s	Read latency of small I/O under EC.
L_e^l	Read latency of large I/O under EC.
K_s	Ratio of EC to R3 read latency for small I/O (L_e^s/L_r^s).
K_l	Ratio of EC-large to R3-small read latency (L_e^l/L_r^s).
p	Fraction of queries accessing only Key SSTs.
S	Cost-effectiveness of redundancy strategy.

4.2.3 Modeling and Analysis. We define **Queries Per Second (QPS)** as the number of read operations completed per second. To evaluate the cost-effectiveness (S) of a redundancy strategy, we introduce the **QPS-to-storage ratio model**:

$$S = \frac{QPS}{Storage\ Cost} = \frac{1/Average\ Latency}{Storage\ Cost} \tag{1}$$

We assume sequential request processing (e.g., single-threaded scenarios) where QPS is inversely proportional to average latency. This model focuses on read throughput because SST writes in LSM-trees are typically large and sequential, resulting in similar write latencies for erasure coding (**R**eed-**S**olomon, RS=4:2) and 3-way replication (**R3**). Since query latency dominates strategy selection, the model incorporates Key/Value SST size ratio, small/large KV access proportion, and corresponding read latencies for EC- and R3-based configurations. Table 2 summarizes the model variables.

We evaluate three redundancy configurations, applying 3-way replication (**R3**) and erasure coding (**RS=4:2**) for Key and Value SSTs. Table 3 summarizes their storage cost, average read latency (Lat_{R3}, Lat_{EC}, Lat_{Mix}), and cost-efficiency (S_{R3}, S_{EC}, S_{Mix}). Each latency metric is derived from the weighted sum of small-KV access latency (from the index LSM-tree) and large-KV access latency (from separated values), with p denoting the proportion of small-KV queries. Latencies differ between R3 and EC configurations for different I/O sizes, with latency terms defined in Table 2.

Table 3: Storage Cost, Average Read Latency and Cost-Efficiency of three redundancy strategies

Strategy	Key SST	Value SST	Storage Cost	Average Read Latency	Cost-Efficiency
All Replication	R3	R3	$3(D_k + D_v)$	$Lat_{R3} = p \cdot L_r^s + (1-p) \cdot \left(L_r^s + L_r^l\right)$	$S_{R3} = \frac{1}{Lat_{R3} \cdot 3(D_k+D_v)}$
All Erasure-Code	EC	EC	$1.5(D_k + D_v)$	$Lat_{EC} = p \cdot L_e^s + (1-p) \cdot \left(L_e^s + L_e^l\right)$	$S_{EC} = \frac{1}{Lat_{EC} \cdot 1.5(D_k+D_v)}$
Mixed Replication-EC	R3	EC	$3D_k + 1.5D_v$	$Lat_{Mix} = p \cdot L_r^s + (1-p) \cdot \left(L_r^s + L_e^l\right)$	$S_{Mix} = \frac{1}{Lat_{Mix} \cdot (3D_k+1.5D_v)}$

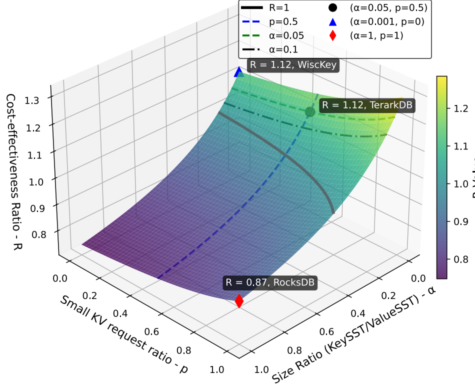


Figure 7: Cost-effectiveness Ratio of Mixed R3-EC and All-EC

Mixed R3-EC vs. All-R3. To investigate the advantage of the Mixed R3-EC strategy over All-R3, we compute the cost-effectiveness ratio $R = S_{Mix}/S_{R3}$ (where $R > 1$ indicates that Mixed R3-EC is more efficient) based on Equation 1. The ratio is defined as:

$$R = \frac{Lat_{R3}}{Lat_{Mix}} \cdot \frac{3(D_k + D_v)}{3D_k + 1.5D_v} = \frac{p \cdot L_r^s + (1-p) \cdot \left(L_r^s + L_r^l\right)}{p \cdot L_r^s + (1-p) \cdot \left(L_r^s + L_e^l\right)} \cdot \frac{2(D_k + D_v)}{2D_k + D_v} \tag{2}$$

Empirical results suggest that large I/O read latencies are comparable under EC and R3 ($L_e^l \approx L_r^l$). Substituting this into Equation (2), we simplify R with the Key-to-Value SST size ratio $\alpha = D_k/D_v$:

$$R \approx \frac{2(D_k + D_v)}{2D_k + D_v} = \frac{2\alpha + 2}{2\alpha + 1} > 1 \tag{3}$$

Since $R > 1$ holds for $\forall \alpha > 0$, the Mixed R3-EC strategy consistently outperforms All-R3 in terms of cost-effectiveness. When the Key-to-Value SST size ratio α approaches zero (i.e., when Key SSTs incur negligible storage cost), the cost-effectiveness ratio R approaches 2, indicating that Mixed R3-EC achieves nearly twice the efficiency of All-R3. Although the benefit diminishes with increasing α (as storage overhead of Key SSTs increases), Mixed R3-EC remains strictly more cost-effective than All-R3 for $\forall \alpha > 0$.

Mixed R3-EC vs. All EC. To compare the All-EC and Mixed R3-EC strategies, we compute the cost-effectiveness ratio R as:

$$R = \frac{Lat_{EC}}{Lat_{Mix}} \cdot \frac{1.5(D_k + D_v)}{3D_k + 1.5D_v} = \frac{L_e^s + (1-p) \cdot L_e^l}{L_r^s + (1-p) \cdot L_e^l} \cdot \frac{D_k + D_v}{2D_k + D_v} \tag{4}$$

We further simplify R using the latency ratios of different I/O sizes under EC and R3, denoted as K_s ($K_s = L_e^s/L_r^s$) and K_l ($K_l = L_e^l/L_r^s$), along with the Key-to-Value SST size ratio α:

$$R = \frac{K_s + (1-p) \cdot K_l}{1 + (1-p) \cdot K_l} \cdot \frac{1 + \alpha}{1 + 2\alpha} \tag{5}$$

Although this ratio depends on multiple factors, both K_s and K_l can be empirically measured. In our setup, we observe $K_s = 1.3$ and $K_l = 1.5$. Using these values, we simplify R and visualize the parameter space in Figure 7. When the SST size ratio α is small (e.g., below 0.1), common in KV-separated systems, R consistently exceeds 1, indicating that Mixed R3-EC outperforms All-EC in cost-effectiveness. Figure 7 also presents two representative cases: systems like WiscKey with full KV separation exhibit small α and benefit significantly from Mixed R3-EC, while systems like RocksDB, with large Key SSTs, favor All-EC due to greater storage savings.

Summary. Our analysis demonstrates that Mixed R3-EC generally achieves superior cost-effectiveness over All-R3 and All-EC, especially in typical KV separation scenarios with small Key SSTs.

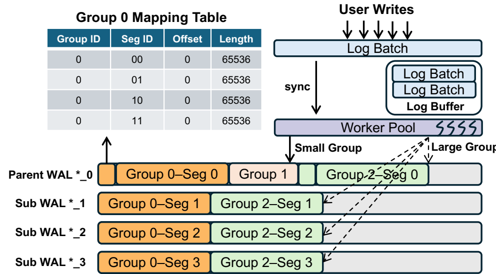
4.3 Adaptive WAL Writing

WAL (Write-Ahead Logging) is essential to LSM-trees and directly affects write latency. While vanilla LSM-trees buffer WAL writes for performance, each batch still requires synchronization to ensure crash consistency [42]. In disaggregated storage, stateless compute nodes must persist WALs remotely via RPC, incurring higher latency and limiting throughput. To address this, we propose an adaptive WAL strategy that optimizes batching and dynamically adjusts parallelism based on workload characteristics, reducing network overhead while preserving consistency.

4.3.1 Batching and Parallel Execution. The conventional group commit mechanism amortizes persistence overhead by batching multiple WAL writes. We enhance this approach by increasing the batch size (e.g., 512 KB), which reduces the frequency of sync writes and improves remote bandwidth utilization. This optimization is particularly effective in KV-separated LSM-trees, where key-value pairs are inherently larger and more amenable to batching. However, larger batches alone cannot fully saturate the available bandwidth, as each commit remains time-consuming and limits throughput.

To address this, we introduce parallel WAL writing by partitioning each batch into multiple segments and leveraging client-side thread pools for concurrent writes. Unlike conventional single-threaded WAL designs, our approach exploits compute-side abundant CPU resources and the inherent concurrency of disaggregated storage to accelerate log persistence. However, disaggregated systems typically enforce single-writer semantics per file to prevent write conflicts. To overcome this, we introduce sub-WALs, where each thread writes its segment to a dedicated file identified by the parent WAL number and segment ID. Finally, the parent WAL merges and persists segment metadata to ensure crash consistency.

As shown in Figure 8, the parent WAL maintains a segment mapping table recording each segment's *Group ID*, *Segment ID*, *Offset*, and *Length*. Each segment is written to a corresponding sub-WAL



Group ID	Seg ID	Offset	Length
0	00	0	65536
0	01	0	65536
0	10	0	65536
0	11	0	65536

Figure 8: Adaptive WAL Writing

file by individual threads, enabling parallel writing and expediting recovery by allowing parallel access to data. It accelerates the write-critical path and maximizes storage-side parallelism, effectively reducing the WAL writing latency for disaggregated systems.

4.3.2 Log Buffering and Adaptive Write. For workloads that do not require persisting WAL synchronously (e.g., MyRocks [43]), Terark-DS employs a log buffer (default size: 1MB) to hold WAL writes temporarily. WAL data is persisted only upon buffer overflow or an explicit flush request, reducing remote I/Os. The log buffer accumulates more WAL data, making it easy to generate large I/O writes that are more friendly to remote storage bandwidth. However, small I/O writes cannot be avoided due to the explicit flush request.

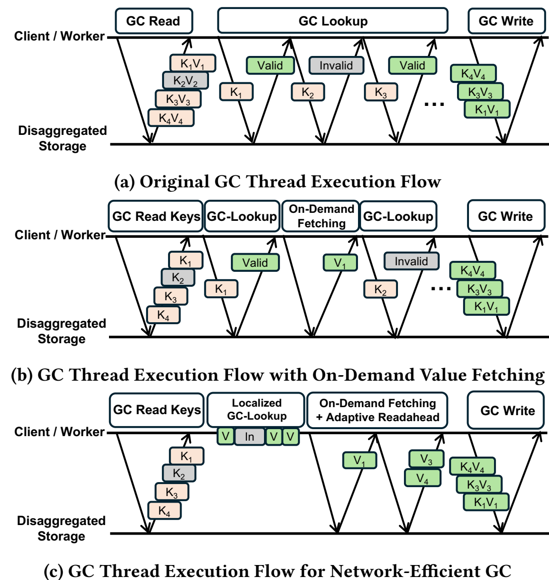
Terark-DS adaptively chooses between serial and parallel WAL writes based on buffered data size. Our evaluation identifies 16KB as a practical per-segment threshold that balances remote bandwidth utilization and thread-scheduling overhead. As shown in Figure 8, small batches (e.g., Group 1, <64KB) are written sequentially to minimize latency, while larger batches (e.g., Group 0 and Group 2) are split into four segments and written in parallel to three sub-WALs and the parent WAL. This mechanism also applies to synchronous WAL writes, enabling dynamic mode selection even under consistency constraints. As batch sizes fluctuate, Terark-DS selects the optimal execution mode, ensuring write latency and improving storage bandwidth utilization for large groups.

By adapting to workload characteristics at runtime, Terark-DS achieves low write latency for small-KV workloads and high throughput for large-KV workloads. In mixed scenarios, it dynamically balances these objectives, effectively leveraging clients' parallelism to ensure both performance and efficiency.

4.3.3 Parallel Recovery and Overhead Analysis. During recovery, Terark-DS first locates the parent WAL via the MANIFEST file, as in RocksDB. When encountering partitioned WALs generated by parallel WAL writes, the process explicitly switches to a parallel replay mode. Each partitioned record in the parent WAL contains a segment mapping table with *Group ID*, *Segment ID*, *offset*, and *length*. Using this metadata, Terark-DS assigns different segments to multiple workers, which concurrently retrieve their designated sub-WAL data ranges. Retrieved segments are reassembled in memory according to their original order (via *Segment ID* sequence), reconstructing the write batches in parallel and accelerating recovery.

The metadata required for parallel replay is minimal. Each segment mapping entry consists of 2 bytes for the *Group ID* and *Segment ID*, as well as 8 bytes each for the *Offset* and *Length*, resulting in negligible overhead relative to the WAL file size. Since WAL files are short-lived and removed once their data is persisted to disaggregated storage, this metadata does not accumulate.

Summary. Terark-DS reduces WAL write latency through batching, parallel execution, log buffering, and adaptive writes. Its design also enables parallel WAL replay during recovery, accelerating system restart with negligible runtime overhead.



(a) Original GC Thread Execution Flow

(b) GC Thread Execution Flow with On-Demand Value Fetching

(c) GC Thread Execution Flow for Network-Efficient GC

Figure 9: GC Execution Flows Under Different Optimizations

4.4 Network-Efficient Garbage Collection

While KV separation and prior work enhance write performance, space amplification remains a critical issue, particularly in disaggregated settings (see Section 3.2). Prior work on TerarkDB targeted local LSM-trees, reducing GC-induced index rewrites to mitigate contention with user writes and minimize I/O overhead [17]. In contrast, our study targets network overhead unique to disaggregated systems, addressing excessive data transfers and round-trips in GC. We analyze GC communication and data flows, proposing optimizations tailored for the disaggregated environment.

GC Execution Flow Analysis. Figure 9(a) illustrates the GC thread execution flow in disaggregated TerarkDB, with three steps:

- **GC-Read:** Candidate KV pairs (e.g., K_1V_1 to K_4V_4) are batch-read from disaggregated storage for validation.
- **GC-Lookup:** Each candidate is validated by querying its version from the remote index LSM-tree. If the version matches, it is marked as *Valid*; otherwise, it is considered *Invalid*.
- **GC-Write:** Validated pairs (e.g., K_1V_1, K_3V_3, K_4V_4) are batched to new remote storage files, maintaining old-to-new mappings.

Original GC faces two network-specific **challenges**:

(1) **GC-Read** transfers excessive invalid KV pairs, with high garbage ratios exacerbating network bandwidth waste.
(2) **GC-Lookup** incurs numerous round-trips with per-key probes, while network latency prolongs the execution time.

Therefore, reducing both network bandwidth waste and round-trips is critical for efficient GC in disaggregated environments.

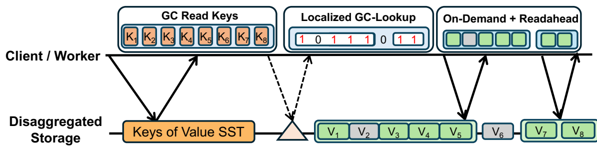


Figure 10: An Example of Network-Efficient GC

4.4.1 On-Demand Value Fetching. To reduce bandwidth waste of invalid KV pair transfers (**Challenge 1**), we propose **On-Demand Value Fetching**, which extends prior local GC techniques to disaggregated storage. By further separating keys and values within each value SST, we aggregate keys in the index block naturally, enabling GC-Read to batch-load candidate keys directly without redundant key storage. GC-Lookup then validates these keys and fetches values of valid entries, avoiding unnecessary transfers for invalid values. As Figure 9(b) shows, it reduces bandwidth waste but introduces a new challenge: each valid key necessitates an RPC to retrieve its value, significantly increasing round-trips.

To reduce network round-trips during *on-demand value fetching*, batch loading is essential. However, the original GC-Lookup, tailored for local storage, issues independent per-key queries in a serialized fashion. This serialization leads to strictly sequential value reads, significantly increasing the network round-trips of GC-Read under disaggregated environments. We address this limitation by enabling *batched and localized GC-Lookup*, allowing batch value retrievals that reduce round-trips and mitigate **Challenge 2**.

4.4.2 Batched and Localized GC-Lookup. To minimize network overhead in GC-Lookup, we propose **Batched and Localized GC-Lookup**, which executes index queries on compute nodes without maintaining persistent state. As index data is typically small, it can be fully cached in memory, enabling GC-Lookup to execute locally without network round-trips and also accelerating large-KV foreground queries. To enable efficient batched execution of GC-Lookup, we introduce the *Flat Index Cache* to reduce cache contention and improve CPU cache locality under heavy load. It organizes keys and index entries for large KVs into separate contiguous memory regions, allowing GC-Lookup to first scan the key region to collect all offsets and then fetch index entries in bulk, improving spatial locality and cache efficiency. The cache shares allocated memory with the native block cache and remains read-only; thus, node failures only cause warm-up delays commensurate with existing cache capacity, consistent with general scenarios.

For GC executed by dedicated workers, Terark-DS maintains an auxiliary LSM-tree, the *Invalid Tree*, which records keys deemed obsolete during compaction. Unlike the full index LSM-tree that serves live queries, the *Invalid Tree* contains only a small subset of keys with no associated values and is updated exclusively by background compaction. GC-Lookup uses this structure to verify validity via a single RPC request, substantially reducing network overhead. By restricting the query scope to a compact metadata index, the *Invalid Tree* is well-suited for offloaded GC operations.

4.4.3 Adaptive Readahead. Building on batched and localized GC-Lookup, we propose **Adaptive Readahead** to further reduce network round-trips during the value retrieval phase of GC-Read. Terark-DS loads candidate keys in batch and determines their validity via GC-Lookup, producing a global validity bitmap. This bitmap enables efficient identification of contiguous valid data regions. However, fragmentation may still increase the IOPS during value fetching, which results in numerous network round-trips.

Adaptive Readahead incrementally merges adjacent valid data intervals, coalescing those separated by less than 16KB when the resulting window maintains at least 80% valid entries. These merges generate aggregated request windows that expand until reaching a 2MB upper bound, chosen both to provide a bandwidth-efficient access granularity for remote storage systems and to align with the default compaction readahead size [44]. Although most windows are smaller in practice, this cap balances network efficiency and retransmission overhead. By consolidating fragmented regions into larger contiguous blocks, Adaptive Readahead decreases the number of network round-trips required during GC-Read.

Figure 10 illustrates an example. The client or worker initiates GC-Read by retrieving candidate keys (K_1–K_8), transmitting only keys to conserve bandwidth. Validity is resolved via localized GC-Lookup using compute-side *Flat Index Cache* or the compact *Invalid Tree*, with remote accesses incurred only on cache misses or initial loads. Batched and localized GC-Lookup yields a validity bitmap, which guides the merging of adjacent data regions. In Figure 10, K_1–K_5 is merged due to a high post-merge validity ratio, avoiding a separate request for fragmented K_1V_1. Consequently, two value-fetch requests are issued, V_1–V_5 and V_7–V_8, eliminating unnecessary transfers (e.g., V_6) and substantially reducing network overhead.

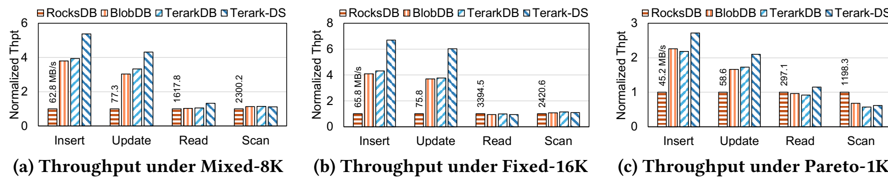
Summary. As Figure 9(c) shows, network-efficient GC minimizes per-GC overhead by transmitting only necessary data and reducing round-trips during GC-Read and GC-Lookup, collectively accelerating execution of GC and enhancing storage efficiency.

5 EVALUATION

5.1 Setup

Testbed. We deploy ByteDance's disaggregated storage system across six storage nodes, each with dual Intel Xeon Platinum 8336C CPUs, three Intel SSDs (SSDPF2KX019T1M), and one 25 Gbps Mellanox ConnectX-5 IB RNIC. All nodes run Debian 10 with Linux kernel 5.4. Among them, three nodes act as *MetaServers* responsible for metadata management, while all six nodes serve as *ChunkServers* handling data reads and writes. The workload generator is deployed on a separate node with identical hardware specifications.

Workloads. We evaluate three representative workloads: **(1) Mixed-8K**: a 1:1 mix of small and large key-value pairs, where small values (100-512B, uniformly distributed) stem from incremental updates and large values (16 KB) correspond to data pages. It aligns with practical cloud-native database scenarios at ByteDance [29]. **(2) Fixed-16K**: uniformly large KV pairs (e.g, fixed 16KB), typical of feature store and state storage workloads [6]. **(3) Pareto-1K**: variable-length values following a generalized Pareto distribution [45, 46], dominated by small and medium sizes, representative of filesystem metadata workloads [47]. Unless otherwise stated, each experiment initializes a 100 GB dataset, followed by 300 GB of UPDATEs, 100 GB of READs, and 40 million SCANs with scan lengths uniformly distributed between 2 and 1000.



(a) Throughput under Mixed-8K (b) Throughput under Fixed-16K (c) Throughput under Pareto-1K

Figure 11: Microbenchmarks under Various Workloads.

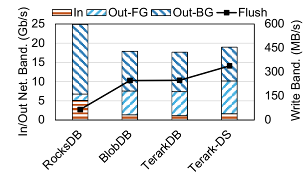
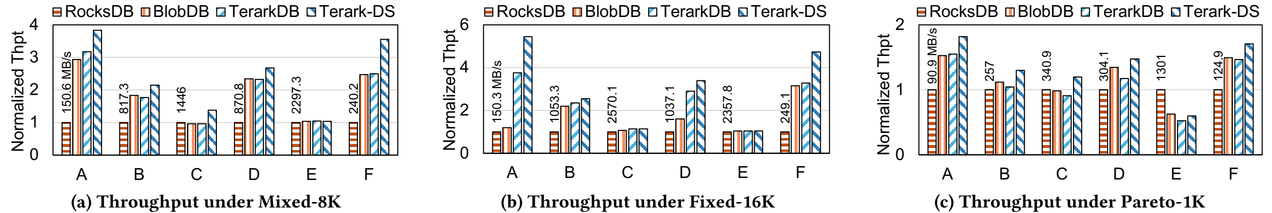


Figure 12: Network Usage of Client



(a) Throughput under Mixed-8K (b) Throughput under Fixed-16K (c) Throughput under Pareto-1K

Figure 13: YCSB under Various Workloads.

Parameters. Following the tuning guide [48], we standardize configurations across all evaluated systems. The key-value separation threshold is 512 B. Each instance maintains up to four Memtables (128 MB each). Value SSTs are capped at 256 MB, and Key SSTs at 128 MB. Bloom filters use 10 bits per key, with a 1 GB block cache (1% of dataset size). Systems are provisioned with 32 foreground threads for user operations and 16 background threads for internal tasks. All other LSM-tree parameters remain default.

Baselines. For fairness, **all evaluated systems are deployed atop ByteDance's disaggregated storage**. We modify RocksDB to support the disaggregated environment and integrate BlobDB [15] to evaluate KV separation. Additionally, we compare against the unoptimized version of TerarkDB. Unless otherwise stated, all baselines adopt 3-way replication (**R3**). Configurations suffixed with **-EC** employ Reed-Solomon (RS=4:2) erasure coding.

5.2 Microbenchmark

We first evaluate the throughput of fundamental operations, including **Insert**, **Update**, **Read** (point query), and **Scan** (range query), across various workloads, as shown in Figure 11.

Insert. Terark-DS exhibits the most significant improvement in insert operations, achieving 2.72×-6.70× the throughput of RocksDB. Its performance benefits stem from two factors: (1) KV separation reduces write amplification, also boosting the write throughput of BlobDB and TerarkDB by 2.19×–4.31×. (2) Optimized redundancy and adaptive WAL writing further improve Terark-DS by 20.4%–63.9% over other KV-separated LSM-trees.

To further investigate the performance gains of Terark-DS, we profile network bandwidth usage for insert operations under the Mixed-8K workload, as shown in Figure 12. KV-separated BlobDB reduces average network bandwidth by 28.3% compared to vanilla RocksDB. By breaking down outbound traffic into foreground and background operations, we observe that KV separation shifts bandwidth usage from compaction to user requests. Terark-DS exhibits 37.6% higher flush bandwidth than TerarkDB under similar overall network usage, contributing to its higher throughput.

Update. Terark-DS achieves significant improvements for update operations, delivering 2.10×–6.02× the throughput of RocksDB and outperforming other KV-separated LSM-trees by 21.3%–62.6%. Compared to **Insert** operations, the benefits are partially offset by increased GC activity in update-intensive workloads, which compete for network bandwidth with foreground writes. Despite this contention, Terark-DS maintains a stable performance advantage.

Across workloads, Terark-DS achieves the largest performance gains under Fixed-16K, while improvements under Pareto-1K are less pronounced. In Fixed-16K, large KV pairs fully exploit the benefits of KV separation, redundancy optimization, and adaptive WAL writing, maximizing disaggregated bandwidth utilization and accelerating writes. In contrast, Pareto-1K, characterized by small I/O sizes and higher write amplification, sees diminished advantages.

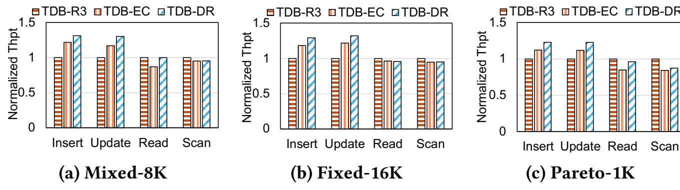
Read. For point queries, RocksDB, BlobDB, and TerarkDB show comparable performance. Though Terark-DS stores Value SSTs with erasure coding, it offsets potential network and I/O overhead through differentiated redundancy and index caching, eliminating extra RPCs for index lookup. As a result, Terark-DS improves read throughput by 32.1% under Mixed-8K and 25.7% under Pareto-1K.

Scan. For range queries, KV-separated LSM-trees perform similarly to RocksDB in large-value workloads (Mixed-8K, Fixed-16K). However, in small-value Pareto-1K, weakened value locality from KV separation results in degraded scan performance, as reported [25]. Nevertheless, Terark-DS introduces no additional regressions and performs comparably to other KV-separated LSM-trees.

5.3 YCSB

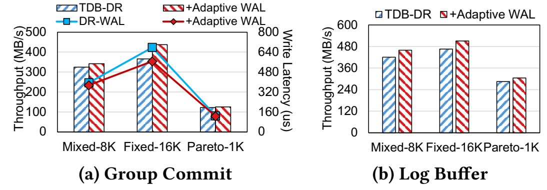
To evaluate performance across diverse workload patterns, we employ YCSB [49], a widely adopted framework for benchmarking key-value stores, covering a range of read-write distributions and access patterns. The results are shown in Figure 13.

Large-Value Workloads (Mixed-8K and Fixed-16K). Terark-DS delivers the highest throughput under large-value workloads, especially in write-intensive cases like YCSB-A and YCSB-F. Under Fixed-16K, Terark-DS outperforms unoptimized TerarkDB by 45.2% and shows far greater advantages over BlobDB under YCSB-A. Notably, BlobDB experiences severe foreground stalls in YCSB-A due to blob rewrites during compaction, reducing its throughput to the level of RocksDB. In read-intensive workloads, Terark-DS enhances point query performance via index caching while maintaining similar range query performance to other baselines.



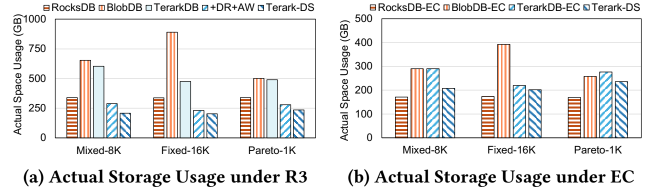
(a) Mixed-8K (b) Fixed-16K (c) Pareto-1K

Figure 14: Effect of Differentiated Redundancy Strategy



(a) Group Commit (b) Log Buffer

Figure 15: Effect of Adaptive WAL Write



(a) Actual Storage Usage under R3 (b) Actual Storage Usage under EC

Figure 16: Effect of Network-Efficient GC

Small-Value Workloads (Pareto-1K). For small-value workloads, Terark-DS achieves 14.4%–44.1% higher throughput than other KV-separated LSM-trees on most workloads, except for the scan-intensive YCSB-E. AWhile the gains are smaller than in large-value workloads, Terark-DS consistently leads KV-separated baselines. In YCSB-E, its throughput falls behind non-separating RocksDB but remains competitive with other KV-separated designs, consistent with our earlier microbenchmark analysis of scan performance.

5.4 Features

To validate the effectiveness of each optimization proposed in Terark-DS, we evaluate their individual contributions.

5.4.1 Differentiated Redundant Strategy. To validate the effectiveness of the differentiated redundancy strategy, we evaluate TerarkDB under three redundancy configurations: All-R3 (**TDB-R3**), All-EC (**TDB-EC**), and the proposed differentiated redundancy strategy (**TDB-DR**). The results are presented in Figure 14.

Write (Insert and Update). TDB-DR improves throughput by 22.7%–32.0% over TDB-R due to two key factors: (1) using EC for Value SSTs reduces network overhead (as in TDB-EC), contributing 12.0%–21.7% improvement; and (2) quorum-based WAL replication accelerates user writes. These benefits are more evident in large-value workloads where EC reduces more network traffic. The corresponding bandwidth trends are illustrated in Figure 17(a).

Read. TDB-EC suffers 12.9% and 15.2% throughput degradation under Mixed-8K and Pareto-1K due to high small-KV ratios, where EC introduces higher latency for small I/Os. In contrast, TDB-DR preserves read performance by applying R3 to Key SSTs, which hold latency-sensitive small KVs, maintaining throughput close to TDB-R. These findings align with our analysis in Section 4.2.3. As further shown in Figure 17(b), EC also incurs additional CPU overhead and increases network bandwidth usage due to read amplification.

Scan. While TDB-DR maintains stable scan performance under large-value workloads (Mixed-8K and Fixed-16K), it degrades under Pareto-1K due to EC overhead on small-value scans. Fortunately, Terark-DS mitigates this limitation with the dedicated *Flat Index Cache*, yielding moderate improvements in both microbenchmarks and YCSB-E, as shown in Figure 11 and Figure 13.

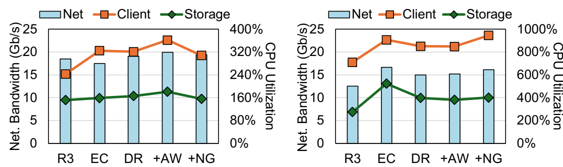
5.4.2 Adaptive WAL Writing. To evaluate the impact of adaptive WAL writing, we conduct experiments focusing on insert operations, as WAL write latency directly influences write throughput, with update operations exhibiting similar behavior. All experiments build upon the TDB-DR configuration discussed previously.

As shown in Figure 15(a), adaptive WAL writing improves write throughput by 6.1%–19.6% under large-value workloads, with the most notable gains observed in Fixed-16K. For small-value workloads (Pareto-1K), throughput shows negligible changes as WAL writing automatically falls back to serial mode. Meanwhile, WAL latency is reduced by 5.1%–17.4% under large-value workloads, and by 3.1% for small ones. A breakdown of batch sizes reveals that larger average batch sizes (263 KB in Fixed-16K vs. 135 KB in Mixed-8K) correlate with higher performance benefits, whereas 16.9KB batch size under Pareto-1K explains its limited improvements.

We further evaluate performance under log buffering, as shown in Figure 15(b). Compared to group commit, it improves write throughput by 29.8% for large-value workloads and up to 134% for small-value workloads. Moreover, enabling adaptive WAL writing further yields an additional 9.5% improvement, even under Pareto-1K. However, its relative benefit is smaller than group commit, as WAL latency becomes a less dominant factor.

5.4.3 Network-Efficient Garbage Collection. To evaluate the effectiveness of network-efficient GC, beyond the read performance optimizations discussed in the microbenchmarks, we primarily focus on its impact on storage efficiency. To ensure a fair comparison, we first load a 100 GB base dataset and then perform continuous throttled update operations, limiting 200 MB/s for large-value workloads (Mixed-8K and Fixed-16K) and 100 MB/s for small-value workloads (Pareto-1K) to maintain stable pressure. Each experiment runs for 1800 seconds, after which we measure the actual storage consumption. We evaluate these metrics across various KV sizes under both R3 and EC redundancy strategies. Additionally, we introduce **DR+AW**, which represents TerarkDB with differentiated redundancy and adaptive WAL writing enabled, but without network-efficient GC. By comparing **DR+AW** with Terark-DS, we isolate and quantify the contribution of network-efficient GC.

Figure 16 illustrates the actual storage usage of three baselines (RocksDB, BlobDB, and TerarkDB) under R3 and EC redundancy strategies. Under the R3 strategy, Terark-DS achieves the lowest storage usage across all workloads, reducing storage by 30.4%–40.1% compared to non-separated RocksDB and 51.9%–77.3% compared to KV-separated BlobDB and TerarkDB, benefiting from EC for Value SSTs and network-efficient garbage collection, which also reduces storage by 12.0%–28.1% compared to Terark-DS without GC optimization **(+DA+AW)**. Under the EC strategy, Terark-DS reduces storage overhead by 7.9%–28.4% compared to BlobDB-EC



(a) Update under Mixed-8K **(b) Read under Mixed-8K**

Figure 17: CPU and Network Overhead

and TerarkDB-EC, while delivering superior performance. Compared to RocksDB-EC, Terark-DS incurs higher storage usage due to delayed value data reclamation, but this tradeoff yields significant performance gains, such as 3.43× Insert and 1.43× Read throughput under Mixed-8K, enhancing overall cost-effectiveness.

5.5 Cost Analysis

CPU and Network Overhead. We evaluate the network and CPU overhead of key features using TerarkDB with different redundancy strategies (**R3**, **EC**, and **DR**), TDB-DR-AW (**+AW**, with adaptive WAL writing), and Terark-DS (**+NG**, with Network-efficient GC), under the Mixed-8K workload. CPU usage encompasses both compute and storage nodes, where storage node utilization ruled out I/O polling threads per disk. As shown in Figure 17, **DR** incurs lower CPU overhead than EC while maintaining similar network usage. Combined with the Figure 14(a) results, it provides a better cost-performance trade-off as a compromise between EC and R3. Adaptive WAL writing leverages abundant client-side CPU to accelerate foreground writes, adding 0.41 cores of overhead on the client with minimal impact on storage nodes. Network-efficient GC reduces overall CPU overhead by 0.54 cores compared to **AW** under update workloads owing to diminished GC costs. Conversely, it incurs an extra 0.97 cores under read workloads, trading off for improved read performance via enhanced index caching.

Cost Summary. As shown in Table 4, we summarize storage and computational costs across all evaluated systems. Storage cost is calculated based on the pricing of local SSDs (as actually used in ByteDance's disaggregated storage) [50], assuming data is retained for one month and charged per GB. Computational cost is measured by profiling CPU usage during 300GB of update (Put) and read (Get) operations via our microbenchmark setup. Terark-DS incurs only slightly higher storage cost than EC-based RocksDB, outperforming all other systems thanks to its differentiated redundancy and network-efficient GC. It also achieves the lowest compute cost for both Put and Get due to KV separation, adaptive WAL writing, and differentiated redundancy strategy. Overall, Terark-DS demonstrates the lowest total cost, effectively reducing the combined cost by 22.7% to 58.6% compared to other LSM-tree designs.

Table 4: Cost Summary (USD)

Redundancy	KVS	Storage	Put	Get	Total
R3	RocksDB	0.3788	0.5760	0.0364	0.9912
	BlobDB	0.7323	0.1933	0.0368	0.9625
	TerarkDB	0.6749	0.1830	0.0361	0.8940
EC (RS=4:2)	RocksDB	0.1920	0.3822	0.0460	0.6202
	BlobDB	0.3253	0.1582	0.0479	0.5313
	TerarkDB	0.3250	0.1627	0.0480	0.5358
DR	Terark-DS	0.2329	0.1453	0.0323	0.4106

6 RELATED WORK

Disaggregated databases. Disaggregation addresses limitations of monolithic databases by decoupling storage and computation resources. Systems such as Snowflake [51], Aurora [18], and PolarDB [19] utilize disaggregation to achieve improved scalability, elasticity, and resource efficiency. These systems primarily focus on architectural optimizations, employing techniques such as log-structured storage, multi-version concurrency control, and fine-grained data access to improve compute-storage interactions, while leaving storage engine-specific challenges less explored. Nevertheless, the architectural principles and lessons learned from these disaggregated database systems provide valuable insights for building efficient disaggregated LSM-tree storage engines.

Disaggregated LSM-trees. Recent efforts extend disaggregation to storage engines by decoupling computation from storage while retaining the core benefits of LSM-trees. RocksDB-Cloud [52] and Disaggregated RocksDB [20] adapt RocksDB for multi-tiered cloud storage through parameter tuning and policy refinement but retain the monolithic LSM-tree design without KV separation, leading to severe write amplification and network overhead. NovaLSM [34] and HailStorm [53] target multi-instance scalability and load balancing via partitioning and remote compaction. However, their client-side partitioning and reliance on local file systems may complicate deployment and limit optimization in fully disaggregated storage. Terark-DS, by comparison, optimizes a single LSM-tree instance for performance and cost efficiency on disaggregated storage. MirrorKV [22] incorporates KV separation and partitioning but fails to adequately address write amplification and storage cost. In contrast, Terark-DS employs KV separation to reduce write amplification, differentiated redundancy strategy to lower network cost, adaptive WAL writing to improve write performance under diverse workloads, and network-efficient GC to control storage overhead. CaaS [21] and SAS-Cache [54] address cache persistence and remote compaction scheduling, which are orthogonal to our objectives of improving write efficiency and reducing storage cost.

7 CONCLUSION

We present Terark-DS, a high-performance and storage-efficient KV-separated storage engine tailored for disaggregated architectures. Terark-DS mitigates write amplification through KV separation, reduces network overhead and enhances write performance via differentiated redundancy strategies and adaptive WAL writing, and lowers storage costs with network-efficient garbage collection. By balancing performance and storage efficiency in disaggregated LSM-trees, Terark-DS consistently outperforms existing solutions. Experimental evaluations demonstrate its effectiveness as a robust and cost-efficient solution for disaggregated storage engines.

ACKNOWLEDGMENTS

This work was funded by the National Key Research and Development Program (No.2022YFB4501300), the National Natural Science Foundation of China (No.U22A2027 and 62402187), the China Postdoctoral Science Foundation (No.GZB20240243 and 2024M751009), and the Postdoctoral Project of Hubei Province (No.2024HBBHCXA024).

REFERENCES

[1] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[2] 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. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[3] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. 2022. ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[4] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 353–369.

[5] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC)*. 145–156.

[6] Yuan Mei, Rui Xia, Zhaoqian Lan, Kaitian Hu, Lei Huang, Paris Carbone, Yanfei Lei, Vasiliki Kalavri, Han Yin, and Feng Wang. 2025. Disaggregated State Management in Apache Flink 2.0. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4846–4859.

[7] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[8] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 497–514.

[9] Dingheng Mo, Siqiang Luo, and Stratos Idreos. 2025. How to Grow an LSM-tree? Towards Bridging the Gap Between Theory and Practice. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–25.

[10] Google. 2023. LevelDB. Available online. Retrieved Jan 2026 from https://github.com/google/leveldb

[11] Facebook. 2012. RocksDB. Available online. Retrieved Jan 2026 from https://github.com/facebook/rocksdb

[12] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage* 13, 1 (2017), 1–28.

[13] Junfeng Liu, Haoxuan Xie, and Siqiang Luo. 2025. [Technical Report] ArceKV: Towards Workload-driven LSM-compactions for Key-Value Store Under Dynamic Workloads. *arXiv preprint arXiv:2508.03565* (2025).

[14] Pingcap. 2025. Titan. Available online. Retrieved Jan 2026 from https://github.com/tikv/titan

[15] Facebook. 2021. BlobDB. Available online. Retrieved Jan 2026 from https://github.com/facebook/rocksdb/wiki/BlobDB

[16] Bytedance. 2025. TerarkDB. Available online. Retrieved Jan 2026 from https://github.com/bytedance/terarkdb

[17] Jianshun Zhang, Fang Wang, Sheng Qiu, Yi Wang, Jiaxin Ou, Junxun Huang, Baoquan Li, Peng Fang, and Dan Feng. 2024. Scavenger: Better Space-Time Trade-Offs for Key-Value Separated LSM-trees. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4072–4085.

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

[19] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[20] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, et al. 2023. Disaggregating RocksDB: A production experience. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[21] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[22] Zhiqi Wang and Zili Shao. 2023. MirrorKV: An Efficient Key-Value Store on Hybrid Cloud Storage with Balanced Performance of Compaction and Querying. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[23] Weiping Yu, Fan Wang, Xuwei Zhang, and Siqiang Luo. 2024. Are Joins over LSM-Trees Ready? Take RocksDB as an Example. *Proceedings of the VLDB Endowment* 18, 4 (2024), 1077–1090.

[24] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC)*. 1007–1019.

[25] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC)*. 673–687.

[26] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. FenceKV: Enabling Efficient Range Query for Key-Value Separation. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3375–3386.

[27] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. 2020. NovKV: Efficient Garbage Collection for Key-Value Separated LSM-Stores. In *36th International Conference on Massive Storage Systems and Technology (MSST)*. 8.

[28] Jianshun Zhang, Fang Wang, Jiaxin Ou, Yi Wang, Ming Zhao, Sheng Qiu, Junxun Huang, Baoquan Li, Peng Fang, and Dan Feng. 2025. Scavenger+: Revisiting Space-Time Tradeoffs in Key-Value Separated LSM-trees. *IEEE Trans. Comput.* 74, 10 (2025), 3332–3346.

[29] Hao Wang, Jiaxin Ou, Ming Zhao, Sheng Qiu, Yizheng Jiao, Yi Wang, Qizhong Mao, Zhengyu Yang, Yang Liu, Jianshun Zhang, et al. 2024. LavaStore: ByteDance's Purpose-Built, High-Performance, Cost-Effective Local Storage Engine for Cloud Services. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3799–3812.

[30] Rong Kang, Yanbin Chen, Ye Liu, Fuxin Jiang, Qingshuo Li, Miao Ma, Jian Liu, Guangliang Zhao, Tieying Zhang, Jianjun Chen, and Lei Zhang. 2025. ABase: the Multi-Tenant NoSQL Serverless Database for Diverse and Dynamic Workloads in Large-scale Cloud Environments. In *Companion of the 2025 International Conference on Management of Data*. ACM, 471–484.

[31] Jianchuan Li, Peiquan Jin, Yuanjin Lin, Ming Zhao, Yi Wang, and Kuankuan Guo. 2021. Elastic and Stable Compaction for LSM-tree: A FaaS-Based Approach on TerarkDB. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3906–3915.

[32] Zhuohui Duan, Hao Feng, Haikun Liu, Xiaofei Liao, Hai Jin, and Bangyu Li. 2025. AegonKV: A High Bandwidth, Low Tail Latency, and Low Storage Cost KV-Separated LSM Store with SmartSSD-based GC Offloading. In *23rd USENIX Conference on File and Storage Technologies (FAST)*. 321–335.

[33] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 449–462.

[34] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: a distributed, component-based LSM-tree key-value store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.

[35] Peng Fang, Arijit Khan, Siqiang Luo, Fang Wang, Dan Feng, Zhenli Li, Wei Yin, and Yuchao Cao. 2023. Distributed Graph Embedding with Information-Oriented Random Walks. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1643–1656.

[36] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, 2477–2489.

[37] Marcin Copik, Marcin Chrapek, Larissa Schmid, Alexandru Calotoiu, and Torsten Hoefler. 2024. Software Resource Disaggregation for HPC with Serverless Computing. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 139–156.

[38] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 17–32.

[39] Intel. 2025. SPDK: Storage Performance Development Kit. Available online. Retrieved Jan 2026 from https://spdk.io/

[40] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.

[41] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 69–87.

[42] Facebook. 2020. WAL-Performance. Available online. Retrieved Jan 2026 from https://github.com/facebook/rocksdb/wiki/WAL-Performance

[43] Facebook. 2017. FlushWAL; less fwrite, faster writes. Available online. Retrieved Jan 2026 from https://rocksdb.org/blog/2017/08/25/flushwal.html

[44] Facebook. 2023. Compaction. Available online. Retrieved Jan 2026 from https://github.com/facebook/rocksdb/wiki/Compaction

[45] Jonathan RM Hosking and James R Wallis. 1987. Parameter and quantile estimation for the generalized Pareto distribution. *Technometrics* 29, 3 (1987), 339–349.

[46] Facebook. 2025. RocksDB Trace, Replay, Analyzer, and Workload Generation. Available online. Retrieved Jan 2026 from https://github.com/facebook/rocksdb/wiki/RocksDB-Trace-Replay-Analyzer-and-Workload-Generation

[47] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S Kim, and Sungjin Lee. 2021. Modernizing file system through in-storage indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 75–92.

[48] Facebook. 2025. RocksDB Tuning Guide. Available online. Retrieved Jan 2026 from https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[49] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[50] ByteDance. 2025. Elastic Compute Service with local SSDs. Available online. Retrieved Jan 2026 from https://www.volcengine.com/docs/6396/68530

[51] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 215–226.

[52] Rockset. 2025. RocksDB-Cloud. Available online. Retrieved Jan 2026 from https://github.com/rockset/rocksdb-cloud

[53] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated compute and storage for distributed LSM-based databases. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 301–316.

[54] Chang Guo Zhang Cao, Ziyuan Lv, Anand Ananthabhotla, and Zhichao Cao. 2024. SAS-Cache: A Semantic-Aware Secondary Cache for LSM-based Key-Value Stores. In *Proceedings of The 38th International Conference on Massive Storage Systems and Technology (MSST)*. 16.