O^3-LSM: Maximizing Disaggregated LSM Write Performance via Three-Layer Offloading

QI LIN, Arizona State University, USA
GANGQI HUANG, Arizona State University, USA
TE GUO, Purdue University, USA
CHANG GUO, Arizona State University, USA
VIRAJ THAKKAR, Arizona State University, USA
ZHICHEN ZHU, Boston University, USA
JIANGUO WANG, Purdue University, USA
ZHICHAO CAO, Arizona State University, USA

Log-Structured Merge-tree-based Key-Value Stores (LSM-KVS) have been optimized and redesigned for disaggregated storage via techniques such as compaction offloading to reduce the network I/Os between compute and storage. However, the constrained memory space and slow flush at the compute node severely limit the overall write throughput of existing optimizations. In this paper, we propose **O^3-LSM**, a fundamental new LSM-KVS architecture, that leverages the shared Disaggregated Memory (DM) to support a *three-layer* offloading, i.e., memtable **O**ffloading, flush **O**ffloading, and the existing compaction **O**ffloading. Compared to the existing disaggregated LSM-KVS with compaction offloading only, O^3-LSM maximizes the write performance by addressing the above issues.

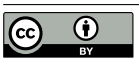
O^3-LSM first leverages a novel *DM-Optimized Memtable* to achieve *dynamic memtable offloading*, which extends the write buffer while enabling fast, asynchronous, and parallel memtable transmission. Second, we propose *Collaborative Flush Offloading* that decouples the flush control plane from execution and supports memtable flush offloading at any node with dedicated scheduling and global optimizations. Third, O^3-LSM is further improved with the *Shard-Level Optimization*, which partitions the memtable into shards based on disjoint key-ranges that can be transferred and flushed independently, unlocking parallelism across shards. Besides, to mitigate slow lookups in the disaggregated setting, O^3-LSM also employs an adaptive *Cache-Enhanced Read Delegation* mechanism to combine a compact local cache with DM-assisted memtable delegated read. Our evaluation shows that O^3-LSM achieves up to **4.5**X write, **5.2**X range query, and **1.8**X point lookup throughput improvement, and up to **76%** P99 latency reduction compared with Disaggregated-RocksDB, CaaS-LSM, and Nova-LSM.

CCS Concepts: • **Information systems → Key-value stores**.

Additional Key Words and Phrases: LSM-Tree, Key-value stores, Disaggregated Memory

ACM Reference Format:
Qi Lin, Gangqi Huang, Te Guo, Chang Guo, Viraj Thakkar, Zhichen Zhu, Jianguo Wang, and Zhichao Cao. 2026. O^3-LSM: Maximizing Disaggregated LSM Write Performance via Three-Layer Offloading. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 216 (June 2026), 27 pages. https://doi.org/10.1145/3802093

Authors' Contact Information: Qi Lin, Arizona State University, Tempe, Arizona, USA, qlin36@asu.edu; Gangqi Huang, Arizona State University, Tempe, Arizona, USA, alexhuang1403@gmail.com; Te Guo, Purdue University, USA, guo777@purdue.edu; Chang Guo, Arizona State University, Tempe, Arizona, USA, cguo51@asu.edu; Viraj Thakkar, Arizona State University, Tempe, Arizona, USA, viraj.dt@asu.edu; Zhichen Zhu, Boston University, USA, zczhu@bu.edu; Jianguo Wang, Purdue University, USA, csjgwang@purdue.edu; Zhichao Cao, Arizona State University, Tempe, Arizona, USA, zhichao.cao@asu.edu.



ACM 2836-6573/2026/6-ART216
https://doi.org/10.1145/3802093

1 Introduction

With the fast development of cloud computing and high-speed networks, Disaggregated Data Centers (DDCs) have been widely adopted in the industry to address the challenges of resource utilization, load-balancing, and scalability in traditional monolithic setups [33, 40, 48, 51, 52, 59, 60]. DDCs achieves high flexibility by decoupling compute, memory, and storage into independently scalable resource pools, such as compute clusters, Disaggregated Storage (DS), and Disaggregated Memory (DM). Driven by this architectural evolution, Log-Structured Merge-tree-based Key–Value Stores (LSM-KVS) are being redesigned for DDCs, resulting in disaggregated LSM-KVS to fully exploit the DDC model and maximize both resource utilization and performance.

In a typical disaggregated LSM-KVS, ingested Key-Value pairs (KV-pairs) are first batched into a write buffer as memtables within one or more compute nodes (CNs). When a memtable becomes full, it is marked as immutable and flushed from the CN to DS as a Sorted String Table (SST) file over the network. To improve space efficiency and read performance, SST files in DS will be periodically read back to CN, merged as new SST files, and written back to DS. The periodic merging, called *compaction*, runs as a background process and is designed not to interfere with the foreground user ingestion process. The decoupling provided by DDCs is a natural fit for this task, as it allows **compaction processes to be separated from CNs and offloaded into DS**, significantly reducing resource contention and network traffic on CNs. As such, many state-of-the-art disaggregated LSM-KVS – including Disaggregated-RocksDB at Meta [19], TerarkDB at Bytedance [10], IS-HBase [12], Hailstorm [9], Nova-LSM [27], and CaaS-LSM [58] – adopt this design by offloading compaction to DS nodes or redistributing it across CNs to mitigate write performance penalties caused by data movement between CN and DS during compaction.

However, compaction offloading does not effectively mitigate *write slowdowns and stalls* caused by **limited write-buffer memory and slow flush operations**. When the write-buffer limit is reached during data ingestion, new writes are typically throttled or blocked until the memory space is freed via flushing, which directly causes slowdowns and stalls. Given that an LSM-KVS instance usually requires substantial memory on both the read path (i.e., block cache) and the write path (i.e., memtables) [13], the memory of CNs is often limited, particularly in DDC setups where a single CN can simultaneously host tens of LSM-KVS instances and other applications. In addition to limited memory, memtable flush also incurs heavy network I/O to DS with high latency [12, 19, 58]. This network traffic competes with other background and foreground I/Os (e.g., WAL appends/syncs, compaction, and SST file reads) on the CN, prolonging the flush process and leading to further write slowdowns and stalls [45, 46, 57]. Moreover, flush operations can also be *suspended* if the number of SST files at level 0 (L_0) reaches the predefined limit due to serial and slow L_0 compactions [8, 57].

With widely deployed remote-memory technologies such as RDMA [38], DDCs can provide shared Disaggregated Memory (DM) pools for CN as a secondary memory tier. Given this capability, a straightforward solution to a constrained write buffer is to extend the disaggregated LSM-KVS write buffer by offloading memtables to DM as a short-term staging area, rather than flushing them to DS immediately. However, **simply offloading memtables to DM instead leads to lower ingestion throughput and higher read latency**.

Problem 1: Costly remote memtable shipping and rebuild. Even with fast RDMA, accessing remote memory is slower than local DRAM, so moving a memtable to DM is not a cheap memcpy. The memtable traverses the RNIC, PCIe, and fabric, adding non-negligible transfer latency and bandwidth consumption. Moreover, a memtable is pointer-intensive (e.g., skiplist nodes allocated from arenas), so a raw byte copy invalidates addresses on the DM. Making it searchable on DM requires extra updating of pointers and reconstruction of the index. Therefore, both memtable transfer and memtable rebuild at DM will have explicit performance penalties.

Problem 2: Inefficient memtable flush. Offloading immutable memtables to DM is only the first step. For disaggregated LSM-KVS, memtables still need to be flushed to DS for persistence. However, DM cannot directly flush these offloaded memtables to DS, since flush requires KV-pair sorting and merging, compression, SST file construction, and Manifest updates, which are missing in DM. Therefore, the memtables need to be read back to the CN and perform the flush locally by the owning LSM-KVS. This process adds an extra network hop, reintroduces serialization and deserialization overhead, and may be further hindered by I/O contention with other foreground and background operations on the same CN. Meanwhile, as many DM-resident memtables are flushed to DS, L_0 SST files accumulate and exacerbate the issue of slow L_0 compaction. All of these effects degrade overall performance.

Problem 3: Slower memtable search at DM. After offloading memtables to DM, searching them can incur significant read performance regression. On one hand, a key lookup within a single memtable requires multiple memory accesses due to its semi-sorted, pointer-intensive structure (e.g., skiplist or tree). When the memtable is at DM, this process will involve multiple remote memory access round-trips and thus lead to higher lookup latency. On the other hand, a read query may need to search all the accumulated memtables belonging to this LSM-KVS at DM. With more memtables accumulated at DM, the read performance can be worse.

To address the aforementioned challenges of memtable offloading to DM, we propose **O^3-LSM**, a fundamental new LSM-KVS architecture that leverages the shared Disaggregated Memory (RDMA-Based) to support a *three-layer* offloading, encompassing memtable **O**ffloading, flush **O**ffloading, and the existing compaction **O**ffloading. Compared to the state-of-the-art disaggregated LSM-KVS with compaction offloading only, O^3-LSM achieves superior performance by introducing the following four key innovations:

- **DM-Optimized Memtable**: O^3-LSM introduces a DM-optimized memtable that is designed to be directly searchable on DM without heavy reconstruction. Instead of transferring and rebuilding pointer-intensive memtables remotely at DM, O^3-LSM writes memtables in a DM-friendly, memory-contiguous layout with index-data separation so that offloading becomes a cheap data transfer and the memtable can be queried on DM as is.
- **Collaborative Flush Offloading**: O^3-LSM develops a lightweight, judiciously designed multi-phase offloading protocol, which allows a memtable flush to be executed on *any* node with DM access. This protocol supports multiple execution modes (**Local**, **In-DM**, **Remote-CN**) and various controlling messages, providing the foundation for a **contention-aware flush scheduler**. This scheduler coordinates all flush operations by dynamically adapting to workload, data locality, and resource utilization and also by handling potential failures to maximize overall write performance with ensuring atomicity and isolation properties.
- **Asynchronous Sharding**: O^3-LSM also partitions each memtable into a set of non-overlapping key-range shards by dividing the KV-block into multiple ones, enabling asynchronous, highly parallel transfer to DM. When flushing multiple memtables, O^3-LSM aggregates KV-shard blocks of the same key range across different memtables. This approach produces non-overlapping, shard-aligned L_0 SST files, effectively breaking down a serial L_0 compaction into multiple parallel tasks, and thereby significantly reducing overall write stalls.
- **Cache-Enhanced Read Delegation**: O^3-LSM accelerates reads by the adaptive read delegation. Leveraging the DM-optimized memtable structure, O^3-LSM maintains a small key-offset cache at CN to decide the read path on every read. This adaptive mechanism determines whether to retrieve data directly from DM using one-sided `RDMA_READ` or delegate the read to DM with two-sided `RDMA_SEND`, reducing unnecessary remote network traversals and mitigating extra CN memory usage.

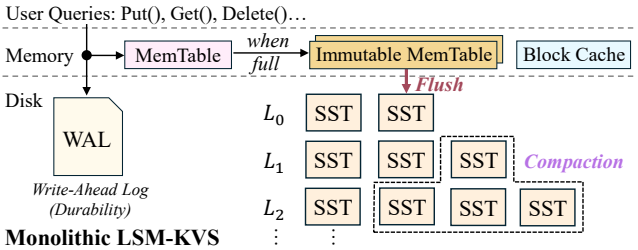


Fig. 1. Monolithic LSM-KVS Architecture.

Implementation and Overall Evaluation Results. We implemented the prototype of O^3-LSM based on RocksDB (version 8.2.0) with about 30,000 lines of code change, which is open-sourced at Github[1]. Comparative and comprehensive experiments were conducted at CloudLab [21] against state-of-the-art LSM-KVS schemes optimized for DDCs, including Disagg-RocksDB [19], CaaS-LSM [58], and Nova-LSM [27] (requires specific RDMA-based DS) under the same maximum memtable limit and identical total write buffer usage. Based on our evaluation, O^3-LSM demonstrates performance gains in both throughput and P99 latency across a variety of workloads. For random write workload, O^3-LSM achieves **4.5**X, **3.4**X, and **4.4**X throughput improvements while slashing P99 latency by up to **60%**, outperforming Disagg-RocksDB, CaaS-LSM, and Nova-LSM. For random read workload, O^3-LSM continues to excel, delivering **1.8**X, **0.6**X, and **0.3**X higher throughput with a P99 latency reduction of up to **69%** compared to these baselines. For range query workload, O^3-LSM achieves up to **5.2**X throughput improvements while reducing P99 latency by up to **22%**. Even in mixed workloads (50% read and 50% write operations), O^3-LSM also exhibits up to **3**X, **2.3**X, and **1.9**X throughput improvements while reducing P99 latency by as much as **76%**. As for the real-world application, Kvrocks [7], O^3-LSM achieves up to **3.4**X throughput improvements while reducing P99 latency by up to **54%**.

2 Background

2.1 LSM-based Key-Value Stores

LSM-KVS like RocksDB [18] and LevelDB [23] are optimized for high-performance writes via log-structured storage. As shown in Figure 1, LSM-KVS appends KV-pairs to a Write-Ahead-Log (WAL) and an in-memory active memtable. Full active memtables become immutable and are serialized as Sorted String Table (SST) files through a flush process into L_0. Consequently, each L_0 SST file may contain KV-pairs spanning the entire key space. SST files are organized across multiple levels. L_0 files are compacted sequentially into L_1 because their key ranges often overlap. In contrast, levels L_n ($n \geqslant 1$) have disjoint ranges, allowing parallel compaction with overlapping files in L_{n+1}. To manage backpressure, LSM-KVS triggers write slowdowns or stalls if immutable memtables or L_0 files become excessive. For read queries, LSM-KVS maintains a block cache in memory to cache the data/metadata blocks from SST files.

2.2 Disaggregated LSM-based Key-Value Stores

Disaggregated Data Centers (DDCs) have gained traction as a way to improve resource utilization and elasticity by decoupling compute, memory, and storage into independent resource pools interconnected by high-speed fabrics [6, 19, 40, 51, 52, 60]. Optimizing LSM-KVS for DDCs (called disaggregated LSM-KVS), especially ones storing persistent data (e.g., SST files) at Disaggregated

[1]https://github.com/asu-idi/O3-LSM

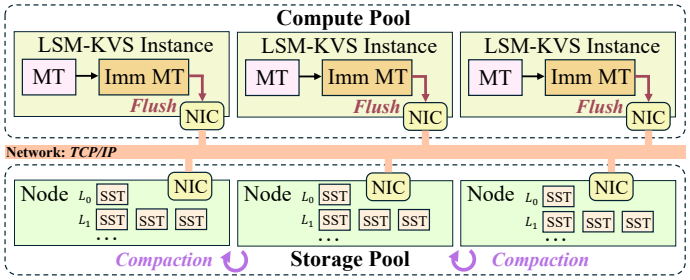


Fig. 2. Disaggregated LSM-KVS Architecture.

Storage (e.g., Tectonic [39], GFS [24], and HDFS [44]) have become popular [19, 58]. As shown in Figure 2, a typical disaggregated LSM-KVS design keeps memtables on compute nodes (CNs) and stores SSTables on DS, with CNs accessing DS via file or block interfaces. This separation enables better load balancing while retaining comparable write performance to local deployments [19].

Recent work has optimized disaggregated LSM-KVS over DS, primarily focusing on remote compaction (or compaction offloading). Disaggregated RocksDB [19], TerarkDB [10], Hailstorm [9], and CaaS-LSM [58] propose offloading compaction tasks to DS or dynamically redistributing them across CNs to improve throughput. Nova-LSM [27] leverages RDMA's high bandwidth through dynamic key-space partitioning to accelerate CN–DSN data transfers. These approaches demonstrate that moving SSTables to DS and optimizing compaction can be highly effective.

However, several bottlenecks remain unresolved, particularly those related to CN memory. LSM-KVS requires substantial memory for both the read path (block cache) and the write path (memtable buffers) [13]. In practice, CNs often host tens of LSM-KVS instances, making it infeasible to allocate large memory to each instance [3]. Insufficient memory at LSM-KVS instances triggers more frequent flush operations to persist memtables quickly. These flushes amplify network I/O to DS, where higher latency and shared bandwidth prolong the process and can cause write slowdowns or stalls. The problem is further exacerbated by L_0 compaction, where L_0 files span the full key space. Therefore, L_0 files can only be compacted sequentially, and high writing pressure can easily hit the L_0 file cap, leading to throttle or block flushes [8, 57]. Consequently, even with compaction offloading, memory pressure and slow flush remain core bottlenecks in current disaggregated LSM-KVS designs.

2.3 Opportunities with Disaggregated Memory: A New Tier for Memtables

The development of ultra-high-speed remote memory access technologies such as RDMA [36] and CXL [1] has enabled the emergence of disaggregated memory (DM) in DDCs, which provides memory pooling and sharing capabilities. Unlike DS, DM is usually not persistent to achieve ultra-high performance. Recently, researchers have begun to explore fully in-memory LSM-KVS on DM. dLSM [50, 53] stores SSTables on DM nodes while caching memtables in CN memory, reporting substantial speedups over persistent LSM-KVS such as RocksDB. However, dLSM does not provide data persistence guarantees, and its design objective is fully different from other disaggregated LSM-KVS like Disaggregated-RocksDB, Nova-LSM, and CaaS-LSM. In general, DM is a promising substrate for alleviating write-path memory pressure and improving performance in persistent, DS-based disaggregated LSM-KVS.

DM also differs from local memory on the read path. In RDMA-based DM, reads are typically issued in two ways: 1) one-sided `RDMA_READ`, the compute node pulls bytes directly from DM using a remote address and rkey, which avoids using the remote CPU but may incur a network round trip

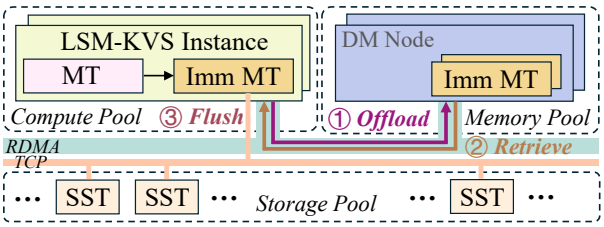


Fig. 3. A naive solution that offloads and retrieves immutable memtables via the Memory Pool.

Table 1. Impact of DM-offloaded memtables on write stalls

Metric	K=2	K=4	K=8	K=16	K=32
Memtable-induced write stalls (count)	737	614	268	115	34
Share of total stalls (%)	96	85	60	30	11
Write-stall latency P99 (ms)	2	6	10	14	18

per read. 2) two-sided `RDMA_SEND`/`RDMA_RECV`, where the compute node sends a request to a thread at DM that reads local DRAM and replies in a single request/response exchange, which reduces round-trips but consumes remote CPU and is typically slower than one-sided `RDMA_READ` due to RPC overheads. In practice, one-sided reads trade more network round-trips for zero remote CPU, while two-sided reads trade remote CPU for fewer round-trips.

3 Motivations and Challenges

Disaggregated storage (DS) offers elastic storage capacity for persistent LSM-KVS, but compute-node memory remains a bottleneck, especially with many co-located instances. Disaggregated memory (DM) can act as a low-latency, high-bandwidth tier between compute-node memory and DS. Therefore, leveraging a DM tier to cache additional memtables while retaining the durability of SST files on DS is a promising way to relieve per-instance memory pressure for high performance without sacrificing persistence.

3.1 A Naive Solution

A straightforward approach to relieve local memory pressure at compute nodes (CNs) is to offload immutable memtables to DM, making DM a fast, scalable extension of CN memory. As shown in Figure 3, this design allows more memtables to reside in memory across local memory and DM tiers, making local memory the fast tier and remote memory the slow tier. To evaluate this approach, we modified Disaggregated RocksDB [19] to transfer immutable memtables from CN to a DM node over RDMA. At the same time, to ensure data persistence, we maintain WAL at CN's local file system and store all the SST files at HDFS [44], a well-known distributed file system. We use the default db_bench "randomwrite" to write KV-pairs (16B key size and 100B value size).

As shown in Table 1, increasing the maximum number (K) of transferred memtables (64MB each) of remote memory from 2 to 32 resulted in a substantial reduction in memtable-induced write stalls by up to 95% (from 737 to 34). At first glance, this decrease in write stalls appears to suggest improved overall write performance. However, we observe a throughput drop (from 119 kop/s to 87 kop/s) and a rise in write-stall p99 latency (from 2 ms to 18 ms) when moving memtables to DM, indicating that the naive approach introduces new performance issues. To explore the root cause, we analyzed the write process and broke down the time spent in each stage. We find that although the number of memtable-induced write stalls is reduced, transferring and rebuilding memtables

on DM consumes a large portion of total time (roughly 18.7%). Moreover, the memtable retrieval process (i.e., flush operations requiring retrieval of memtables from DM back to the CN, and then flushing the memtable to DS) adds extra time (7.8%) compared to local flushes to DS (detailed in Section 5.3). As a result, write stalls last longer.

We also observe that read queries complete in 4.7 us on average at CN local memory, while reads served from DM incur about 29 us. Most of this gap comes from RDMA control and data exchange: approximately 12 us to send the request (`RDMA_SEND`) and 11 us to receive the reply (`RDMA_RECV`), plus about 6 us for the remote search on DM, which is slightly slower than a local search due to extra copying. These observations indicate that while moving memtables to DM reduces memtable-induced write stalls, the additional overhead in transfer, rebuild, and flush operations can outweigh the benefits, and read performance suffers as well.

3.2 Issues and Challenges

As discussed in Section 3.1, offloading memtables from CN local memory to DM may improve the performance. However, addressing performance issues, including 1) slower write from memtable transfers and rebuilding latency, 2) delayed flush due to memtable retrieval from DM, and 3) read performance regression from searching memtables at DM, is challenging.

How to efficiently transfer and rebuild memtables from CN to DM? Since the remote memory access latency can potentially be multiple times higher than local DRAM access [5, 22, 43], transferring memtables from CN to DM can lead to high latency and may block subsequent memtable writes. The situation can be even worse under highly write-intensive workloads. Additionally, due to the memtable data structure design (e.g., widely used skiplist-based memtable), the memtable consists of a large number of pointers. When the memtable is transferred to DM, all the pointers become invalid and require re-allocation at DM, which can be slow.

How to speed up the memtable flush from DM to DS? When immutable memtables are offloaded to DM, flushing them can become even slower when using the original flush logic. Directly creating SST files from memtable at DM is difficult, which involves flushing in-memory data to persistent storage while performing key sorting, value serialization, block and index building, compression, metadata management, WAL handling, and snapshot control. Since DM itself does not have flush execution logic, LSM-KVS must read the memtables back from DM to CN to flush the KV-pairs. This read-back process can introduce additional latency and may be further hindered by I/O contentions with other foreground and background I/O operations on the same CN. Furthermore, when a large number of accumulated memtables at DM are flushed to DS, the issue of slow L_0 compaction (a well-known problem in LSM-KVS due to the need to handle the entire key-space coverage of each L_0 SST file [57]) is exacerbated due to the L_0 SST file accumulation. This can result in increased write slowdown or write stall and can significantly impact overall performance.

How to mitigate read performance regression caused by slow memtable search at DM? All the memtables are accessed during each read query due to the key-range overlaps. When one key is searched in a memtable (e.g., widely used skiplist-based memtable), multiple memory accesses are needed to finally pinpoint the KV-pair (or confirm NotFound). When memtables are maintained in DM, due to the relatively slower memory access via RDMA, searching one memtable can cause explicitly high latency. Even worse, one read query needs to search multiple memtables until the KV-pair is found (or NotFound in all memtables), which further amplifies the high read latency.

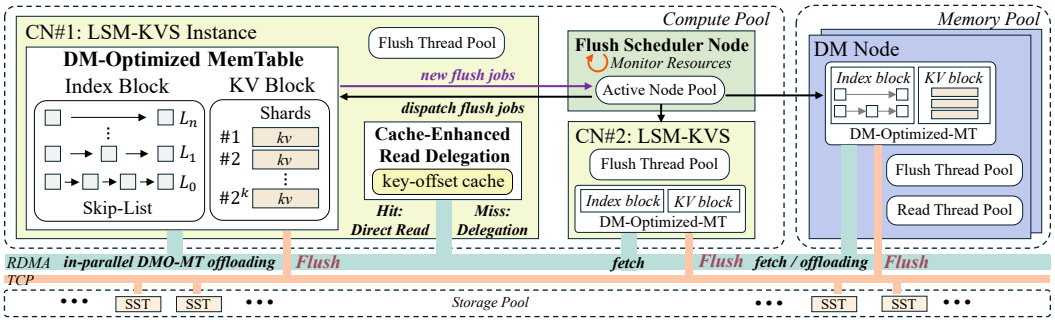


Fig. 4. The overall architecture of O^3-LSM.

4 O^3-LSM

4.1 Architecture Overview

The overall architecture of O^3-LSM is shown in Figure 4. Similar to existing designs (e.g., Disaggregated RocksDB [19], and CaaS-LSM [58]), O^3-LSM instances are deployed on CNs and store all SST files on DS. O^3-LSM employs a two-layered write buffer architecture, integrating limited local DRAM at CNs with large-capacity DM. Throughout the memtable lifecycle, new active memtables are first cached in CN's local memory to support fast KV-pair insertions. Then, the active memtables are offloaded to DM as immutable memtables, which are eventually flushed to DS.

First, to tackle the performance challenge of transferring and rebuilding memtables in DM, O^3-LSM introduces a novel **DM-Optimized MemTable** (detailed in Section 4.2) at both CNs and DM nodes. The DM-Optimized MemTable redesigns the skiplist structure with a pointer-based index-block (containing the linked list nodes of the skiplist) and the one KV-block (KV-block will be further divided into multiple KV-shard blocks if shard-level optimization is applied, as in Section 4.4 for more details). The KV-block stores KV-pairs sequentially, and KV-pairs are referenced by their offsets within the KV-block stores by the skiplist index nodes in the index-block. Therefore, KV-block can be directly transferred to DM without pointer reconstruction.

Second, to speed up memtable flushes, we introduce **Collaborative Flush Offloading** (Section 4.3), which decouples the flush path from the LSM-KVS instance and makes the flush job remotely executable. We propose a flush offloading protocol to transfer the metadata between the memtable owner and the flush executor. Since DM is shared by all CNs, rather than reading immutable memtables from DM back to the owning CN for flush, memtables at DM will be collaboratively flushed by any selected CN or DM nodes. We introduce a collaborative flush scheduler, which monitors resource information (i.e., I/O bandwidth, CPU utilization) and collects flush job metadata (i.e., memtable offset at DM nodes, original CN). The scheduler will select one CN or DM node with spare CPU and I/O to execute the memtable flush by using the flush offloading protocol, improving resource utilization and speeding up flush.

Third, to further improve scalability and parallelism, as well as L_0 compaction, we propose the **Shard-Level Optimizations**, which partition one memtable table into multiple shards based on the pre-defined key-ranges (the same key-range partition will be applied to all memtables). Thus, one KV-block is partitioned into multiple KV-shard blocks. In this way, multiple KV-shard blocks of one memtable can be transferred asynchronously with high parallelism, which further improves the memtable transferring efficiency. At the same time, instead of flushing each memtable separately, we propose to flush the same key-range from all memtables (i.e., merging all the KV-shard blocks of the same key-range) belonging to the same LSM-KVS at the DM together. The shard-based flush

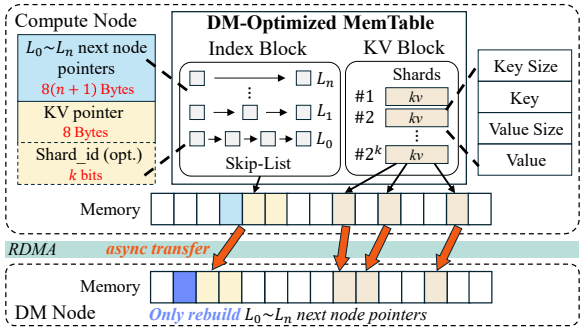


Fig. 5. DM-Optimized SkipList-Based MemTable

actually combines flush and L_0 compaction in a fine-grained way, which significantly reduces the number of SST files in L_0 with key-range overlap. O^3-LSM thus breaks down flush offloading from coarse-grained, memtable-level tasks to fine-grained, shard-level tasks, enabling greater scalability and parallelism, avoiding the L_0 penalties, and significantly improving overall performance.

Finally, to mitigate the high overhead of searching immutable memtables in DM through multiple RDMA accesses, O^3-LSM employs a **Cache-Enhanced Read Delegation** (detailed in Section 4.5). Hot KV-pairs in memtables at DM are cached in a small key-offset cache at CN (caching the key and its DM offset), and one-sided `RDMA_READ` fetches the corresponding KV-pair. On cache miss, O^3-LSM initiates a read delegation request to DM via two-sided `RDMA_SEND`, where DM-side CPU (read thread) searches the memtables to significantly reduce round trips. O^3-LSM also incorporates well-known RDMA optimizations (e.g., doorbell batching and inlining) to improve efficiency.

While O^3-LSM is evaluated on RDMA due to its maturity, availability, and interconnect-agnostic design [11, 28, 31, 50, 53]. The core principles of O^3-LSM remain essential even with the cache-coherent, low-latency access provided by emerging technologies like CXL [14, 37]. Specifically, a DM-friendly layout is still essential to mitigate costly remote pointer chasing, and our shard-level offloading and read delegation provide critical orchestration layers to balance flush-intensive workloads and maximize throughput. Thus, O^3-LSM remains both applicable and beneficial in a CXL-based disaggregated infrastructure.

4.2 DM-Optimized MemTable

Similar to most LSM-KVS implementations [10, 18, 19, 23, 50, 53, 58], O^3-LSM uses the skiplist data structure to implement memtables due to its efficient concurrency, fast point lookup, and range queries. The most widely used skiplist in LSM-KVS (e.g., in RocksDB [18]) is composed of a linked list of n nodes, where each node occupies a contiguous memory segment, with k pointers stored at lower memory addresses, followed by the encapsulated KV-pair. The first $k-1$ pointers form k levels of indexes, with the pointer 0 pointing to the next node. Pointer 1 skips a fraction of nodes, creating a sparse index, and so forth. The query process typically has a complexity of $O(\log n)$.

However, transferring and rebuilding the current skip-list-based memtable to DM are slow and inefficient. First, transferring the entire memtable to DM requires traversing all skiplist nodes at CN and reconstructing them with pointers in DM, which can slow down the memtable offloading process. Second, during the rebuilding process, we need to dynamically allocate the DM memory space for skiplist nodes. Thus, those nodes are scattered in the whole DM memory space, which incurs both space and performance overhead. A straightforward solution is to preallocate a sufficiently large contiguous memory block in DM (e.g., 64MB) for dynamic node allocation in one memtable. However, this strategy can lead to memory fragmentation and space wasting.

To address these, we proposed a **DM-Optimized MemTable**, as shown in Figure 5. We separate the memtable into a pointer-based index-block containing the linked list nodes and one KV-block that stores all the KV-pairs without pointers. Each node in the index block holds: 1) $8(n+1)$ bytes for the next node pointers, 2) a `KV pointer` with 8 bytes pointing to the corresponding KV-pair offset in the KV-block, and 3) a `shard_id` with k bits (if shard-level optimization is applied). In KV-block, KV-pairs are appended sequentially in a contiguous memory block and organized as tuples (<key, value, offset>). This separation allows the memtable to be stored in different, smaller contiguous memory blocks. The KV-block can be directly transferred and stored at a pre-allocated contiguous memory block in DM without further processing. Reconstruction is only needed for the index-block to correct the pointers with the new memory addresses of KV-pairs.

When a memtable becomes immutable, we enqueue it into an asynchronous transfer queue. A background worker first serializes the index-block into a compact, contiguous buffer, registers this buffer as a single RDMA memory region, and issues a one-sided `RDMA_WRITE` to a pre-allocated region on DM. KV-pairs do not need to be serialized, since they are length-prefixed in the KV-block as strings. The worker registers the KV-block and transfers it to its designated DM region with one-sided `RDMA_WRITE`. During the transfer, we maintain a small metadata record (114 bytes) containing the start addresses of the index-block and KV-block on both the CN and the DM. This compact record ensures the metadata overhead remains below 0.0002% for a 64 MB memtable. After writes are complete, the DM side reconstructs the index-block by refreshing the pointers of the skiplist nodes. The memtable is then marked searchable on DM, and the transfer job completes.

Note that the offset information of each KV-pair stored in the index-block does not need to be updated at DM. O^3-LSM will correct these fields in the index by adding the appropriate memory offset of the KV-block at query time. Also, KV-block does not need to be updated/reconstructed at DM since all KV-pairs are appended sequentially with length-prefix encoded. For example, suppose the starting memory address of one KV-block on the CN is A. After the KV-block is transferred to DM, the starting memory address is B. The offset of one KV-pair in the KV-block is C, which is stored in one index node at the index-block. Therefore, the correct DM address of the KV-pair can be determined by adding the offset within the KV-block to the offset of the KV-block at DM address (i.e., $C + B - A$), which can be corrected during memtable search in time complexity of $O(1)$.

4.3 Collaborative Flush offloading

In traditional LSM-KVS designs, the flush process is coupled to the LSM-KVS instance that created the flush job. This coupling causes two drawbacks when DM is used as an external write buffer: (1) since DM lacks flush execution logic (e.g., key sorting and flush-metadata management), the initiating LSM-KVS must fetch immutable memtables back from DM and flush them to DS, incurring transfer and (de)serialization overheads; and (2) memtable flush might still contention for network I/O with foreground reads and background compactions. As DM accumulates memtables, these costs grow and may throttle foreground reads/writes.

To address these challenges, we propose **Collaborative Flush Offloading** (Figure 6). The flush control plane is decoupled from the LSM-KVS and can be executed remotely on any node with DM access. The design has two components: 1) a lightweight flush-offloading protocol that prepares, coordinates, and conveys all required metadata, orchestrates workflow and failure handling, and finalizes job state, and 2) a logically centralized flush scheduler that assigns flush jobs to CNs or DM nodes based on load and locality. As shown in Figure 6, O^3-LSM offloads flush in four phases, supports three execution modes (local, in-DM, and remote-CN), and uses six control messages.

Phase 1: Preparing. When a memtable flush is invoked, the owning LSM-KVS snapshots its DB state under fine-grained locks, records the sequence-number bounds of the memtable to be flushed,

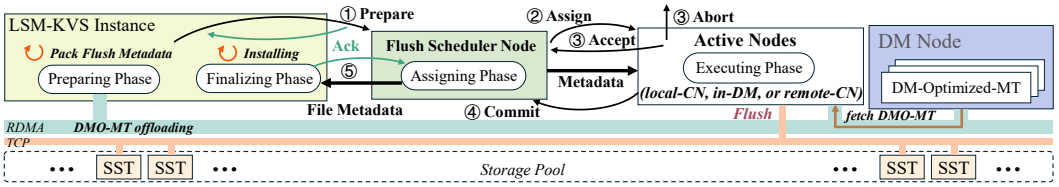


Fig. 6. Collaborative Flush offloading

and ensures the WAL is persisted up to that sequence before offloading the flush. Then, the owning LSM-KVS builds a compact flush-metadata package (572 bytes in total), which includes options (e.g., compression and block settings), file metadata (e.g., target level, file-number), KVS status metadata (e.g., column-family ID and sequence watermark), and WAL status (e.g., log file ID and persisted offset). The package also encodes DM locations for the index-block and KV-shard blocks. Since many metadata objects are shared in LSM-KVS, we precompute a small dependency graph and traverse it in a fixed order during packing to eliminate duplicates. Finally, the owning LSM-KVS sends a `PREPARE` message with the flush-metadata package to the scheduler.

Phase 2: Assigning. Upon receiving `PREPARE` message, the scheduler pushes the flush job into a FIFO queue and dispatches jobs in arrival order to ensure fairness. For each job, it consults its active executor pool and chooses an execution mode: *local* (the owning LSM-KVS executes the flush), *in-DM* (a DM node executes in place without reading back to a CN), or *remote-CN* (a different CN executes the flush job by reading the memtable from DM). After selecting a target node, the scheduler sends an `ASSIGN` message to the selected executor and starts a timeout. The executor responds with `ACCEPT` or the scheduler reassigns the job to another eligible executor at a different node when the timeout happens. Upon `ACCEPT`, the scheduler transmits the flush-metadata package and waits for `COMMIT` or `ABORT` from the executor.

Phase 3: Executing. After the executor receives the flush-metadata package, it fetches the referenced memtable data from DM using one-sided `RDMA_READ`. The executor directly reads the KV-block without pointer reconstruction, and the minimal in-memory index is re-anchored via base-address relocation. The executor (flush thread) merges records in sequence-number order, builds the required index and filter blocks, applies compression if configured, and writes L_0 SST files to DS as specified by the flush-metadata. If a failure occurs before completion, the executor aborts the job and sends an `ABORT` message to the scheduler so it can be safely reassigned (no state has been published). When the flush outputs are durable in DS, the executor sends a `COMMIT` message to the scheduler with the produced file metadata for the owning LSM-KVS to update flush state.

Phase 4: Finalizing. After receiving `COMMIT`, the scheduler forwards the produced file metadata to the owning LSM-KVS. The owning LSM-KVS validates the metadata, installs the new DB state into the Manifest, and removes the memtable from its metadata under the DB mutex before notifying the DM to reclaim space, ensuring in-flight reads either find the data or fall back to the DS reads. The owning LSM-KVS then sends `ACK` to the scheduler. The scheduler marks the job complete, removes it from the queue, and releases its tracking state and timers. This phase publishes results safely and frees resources promptly.

Flush Scheduler. The scheduler maintains a dynamic pool of flush executors across CNs and DM modes to balance flush loads and mitigate write bursts in certain CNs. The scheduler uses heartbeats for node registration and liveness monitoring. To balance the offloaded flush tasks, a deterministic cost model where each node i is assigned a load factor $\text{load}_i = w_{\text{cpu}} u_i^{\text{cpu}} + w_{\text{io}} u_i^{\text{io}} + w_{\text{queue}} u_i^{\text{queue}}$. Here, the weights ($w$) are pre-configured scaling factors that prioritize resources based on the hardware

bottleneck (e.g., higher w_{io} for I/O-bound environments). The utilization metrics (u) are normalized values: u_i^{cpu} is the ratio of active flush threads to the total thread pool size, u_i^{io} represents the current storage write throughput relative to the node's peak calibrated bandwidth, and u_i^{queue} measures the current number of pending shards against the maximum allowable queue depth.

Based on these signals, the scheduler calculates the estimated cost of assigning a flush job of size *bytes* to node i using the formula $\text{cost}_i = bytes \cdot (1 + \text{load}_i)$. We apply a greedy policy to select the node with the lowest cost to execute the flush job. For example, if Node A has a low load of 0.1 and Node B is heavily congested at 0.9, a 64 MB shard would result in costs of 70.4 and 121.6, respectively, leading the scheduler to select Node A. To ensure robustness, the scheduler recomputes load_i from fresh telemetry signals upon the completion of each shard, preventing transient mispredictions from accumulating. Multiple scheduler instances can be deployed for scalability, while more complex scheduling strategies are left for future work.

4.4 Shard-Level Optimizations

Sending 64MB blocks in single `RDMA_WRITE`s can cause bursty bandwidth contention, especially with other RDMA traffic [34]. Transferring such large blocks also limits parallelism and does not resolve the slow L_0 compaction problem. To address this, we propose sharding the memtable. Each memtable is partitioned into non-overlapping key ranges (called *shards*). The single KV-block is thus divided into multiple contiguous KV-shard blocks. Further, we propose flushing multiple shard blocks in the same key-range from different memtables together, effectively combining L_0 compaction with flush. This can improve flush parallelism and address L_0 compaction penalty.

Memtable Sharding. We partition the memtable key space into 2^k non-overlapping key ranges (shards) using the first k bits of each key. Within each shard, KV pairs are appended into a contiguous region, forming a *KV-shard block*. Each index-block node stores a k-bit `shard_id` alongside the KV pointer, and KV pairs are addressed by their offsets within the KV-shard block. Once a shard block becomes immutable, it is transferred to DM independently, so we do not wait for the entire memtable to seal, which shortens wait time and increases concurrency. With this design, smaller KV-shard blocks can be transferred asynchronously and in parallel without pointer reconstruction, fully utilizing RDMA bandwidth. During pointer correction on DM, we use the starting address of the KV-shard block rather than the whole KV-block for base-address relocation.

Shard-Level Flush Offloading. To further improve scalability and parallelism, as well as L_0 compaction efficiency, O^3-LSM uses shard-level flush. O^3-LSM materializes shard-level flush metadata (e.g., shard ID and KV-shard blocks offsets on DM) and redesigns flush to aggregate multiple KV-shard blocks from the same key range of different memtables at DM into a single flush job. This shifts from coarse memtable-level flushing to fine-grained shard-level operations, enabling flush and L_0 compaction to be executed together. Unlike traditional designs that trigger a flush when accumulated memtables reach a size limit, the shard-level flush is triggered only when the total size of its KV-shard blocks across immutable memtables exceeds one memtable size limit (e.g., 64 MB). This keeps the generated L_0 SST files approximately the same size as the memtables, with minimal key-range overlap, thereby mitigating the L_0 compaction penalty. The shard-level optimization also handles workload skew. The scheduler provides fine-grained control by assigning hot shards to the most suitable nodes, balancing CPU and network load across the cluster.

4.5 Cache-Enhanced Read Delegation

In the original LSM-KVS design, read queries (e.g., *Get* and *Scan*) can require traversing one or more memtables. Performing multiple one-sided `RDMA_READ`s to traverse the memtable at DM involves: 1) dereferencing to obtain the key, 2) dereferencing to obtain the next node address based

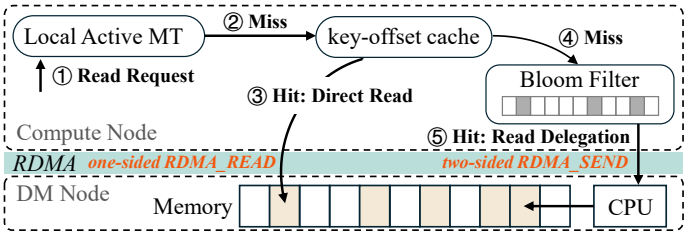


Fig. 7. The workflow of *Cache-Enhanced Read Delegation*.

on key comparison results, and 3) repeating this process iteratively until the KV-pair is found (or NotFound), which can slow down read performance by an order of magnitude or more [56].

To address this multi-round traversal problem, two main optimization approaches can be considered: 1) reading data back from DM to CN and conducting searching locally on the CNs (faster local processing, but searching the remote index-block can be slow), or 2) offloading searching tasks to the remote DM (asynchronized DM local search, but the computing resource might be limited). We combine those two approaches and propose **Cache-Enhanced Read Delegation** across local DRAM and remote DM. As shown in Figure 7, for frequently accessed KV-pairs, we implement a local key-offset cache in O^3-LSM at CN, reducing the round-trip needed for memtable lookups. For less frequently accessed or newly accessed KV-pairs, we leverage the read delegation approach to utilize DM's computational capabilities for DM-local lookup, thereby reducing the multiple RDMA round-trip for searching the index-block. This method effectively combines the benefits of both one-sided and two-sided RDMA operations within a multi-tier architecture.

Local key-offset cache. We designed a small, local key-offset cache for *O*^{3}-LSM. It uses an LRU-based policy to cache hot entries, mapping keys to their specific DM memory offsets. We cache offsets rather than full KV-pair values to minimize CN memory usage, and the value will be fetched at cache hit with a single one-sided `RDMA_READ`. The key-offset entries are inserted via the following two scenarios: 1) When an active memtable is cached in local DRAM, O^3-LSM tracks the read access frequency of each KV-pair, represented by `Freq` using 3 bits, which allows for 8 different frequency levels. When iterating the memtable during the transfer, key-offset entries are created for hot KV-pairs that meet a predefined frequency threshold (e.g., 4 or higher) and inserted into the cache. This approach enables one-sided `RDMA_READ` to fetch the target KV-pair using the cached DM offset directly without searching the index-block. 2) When there is a lookup miss in the local key-offset cache, the KV-pair will be retrieved from DM via read delegation. The corresponding key-offset entry will be created and cached to expedite future requests for the same KV-pair. Additionally, O^3-LSM creates and caches a Bloom filter for each KV-shard block of the memtables stored in DM. This filter is used to validate whether a read delegation is needed or not, effectively reducing the average number of remote accesses.

Read delegation at DM. If a miss occurs in the local key-offset cache, we leverage DM's computational resources to delegate the search task. O^3-LSM first traverses the Bloom filters of the corresponding KV-shard blocks based on the `lookup_key`'s `shard_id`. If any Bloom filter returns positive, O^3-LSM packages the `lookup_key` along with the set of memtable IDs into a read delegation request. Then, O^3-LSM executes an RPC call to DM using two-sided `RDMA_SEND`. On the DM side, we maintain a worker pool with multiple polling threads that receive and process these read delegation requests. Upon receiving a request from CNs, a worker thread polls and performs the local lookup on the targeted memtables based on the request (delegated *Get* operations) and transmits the result until the KV-pair is found (or NotFound if absent in all memtables). Once

O^3-LSM receives the results from DM, we cache the key and its remote offset as a key-offset entry in the local key-offset cache.

Other optimizations. We proposed several RDMA optimizations: 1) **Doorbell batching.** We post 8–16 one-sided `RDMA_READ`s at a time and ring the NIC doorbell once, which amortizes setup costs. 2) **Adaptive inlining for delegation replies.** For delegated lookups, the DM node returns small values directly within the RDMA reply. For larger values, it returns the metadata to perform a single one-sided follow-up read, effectively minimizing round-trips.

4.6 Fault Tolerance

O^3-LSM ensures consistency by treating the CN's WAL as the authoritative state and the Manifest as the sole visibility boundary.

Job-Level Failures. These are transient errors affecting individual operations: 1) Memtable Offloading: If an offload to DM fails, the CN releases the allocated DM region and rebuilds the memtable locally based on the WAL. No incomplete remote memtable is ever registered. 2) Shard-Level Flush: If a task fails during Flush execution, the scheduler discards partial SSTables and reschedules the shard using the immutable memtable on DM. Data only becomes visible after a successful Manifest update. 3) Read Delegation Failures: If a delegated read to DM fails or times out, the CN retries the request or, if the data has already been persisted, reads directly from the DS layer. This ensures that transient DM or network issues do not result in request failure.

Node-Level Failures. These involve the crash of entire system components: 1) CN Failure: On restart, the CN replays its local WAL to reconstruct memtables and re-offloads them to DM. The scheduler purges stale DM state associated with the failed CN. 2) DM Failure: The scheduler aborts all active jobs involving the failed DM. CNs rebuild the offloaded memtables from WAL locally. 3) Scheduler Failure: The scheduler recovers its state from a persistent log. It either resumes a recorded COMMIT or discards incomplete jobs, forcing CNs to retry. O^3-LSM maintains an all-or-nothing guarantee at the Manifest boundary. Partial outputs from failed jobs or nodes may necessitate retries, but they never result in partially installed SSTables or inconsistent metadata states.

5 Evaluation

We implement O^3-LSM based on RocksDB v8.2.0 and Disaggregated RocksDB. The source code is available on GitHub [2]. We conduct comprehensive evaluations to answer the following questions: 1) Does O^3-LSM achieve explicit higher performance than state-of-the-art disaggregated LSM-KVS? 2) What is the performance breakdown of each major design? And 3) How about the performance with real-world applications, and scalability?

5.1 Experimental Setup

Hardware Platform. We conduct our evaluation on CloudLab's [21] `c6220` instances, each equipped with 2 × Xeon E5-2650v2 CPUs (8 cores each), 64 GB of memory, and a 1 TB hard disk. To emulate the disaggregated setup, CNs are configured to use all available cores but only 2 GB of memory, whereas the DM node are configured to use 4 cores while utilizing 64 GB of memory. All CNs and the DM node are connected via Mellanox FDR ConnectX-3 NICs (40 Gbps connections). We deploy HDFS on the storage nodes as TCP-based DS (the same DS configuration as that in CaaS-LSM).

Baselines. To demonstrate the effectiveness of our designs, we use RocksDB optimized for DS [19] as our baseline (**Disagg-RocksDB**). We also include **CaaS-LSM** [58] (a state-of-the-art LSM with remote compaction on DS) and **Nova-LSM** [27] (a disaggregated LSM-KVS with RDMA for DS accesses). All three baselines configure up to 8 local memtables at CN (totaling 512 MB). In contrast,

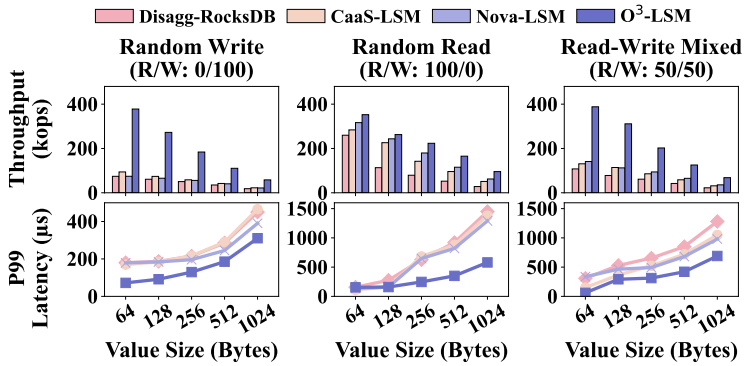


Fig. 8. Microbenchmark performance under a 1 Gbps DS connection

our design, **O^3-LSM**, uses only 2 local memtables (128 MB) and up to 6 memtables at the remote DM node. In addition, we also evaluate **dLSM** [53] (a fully in-memory LSM-KVS on RDMA-based DM) and use it as a performance upper-bound baseline (i.e., it is non-persistent and expected to achieve the best performance with RDMA-based DM).

Workloads. We use db_bench and **YCSB** [15] to evaluate O^3-LSM. For db_bench, we use *fillrandom*, *readrandom*, and *readrandomwriterandom* benchmarks. For YCSB, we use workloads with 4 types of read-write ratios and 2 types of key-value distributions (`Uniform` and `Zipfian`). In Zipfian, keys are selected according to a Zipfian distribution, with skewness set to 0.99. Unless otherwise stated, all experiments are conducted with 1 DM node, 1 CN, and 3 DS nodes. Scalability and breakdown evaluations will use more CNs. We use the same configuration for O^3-LSM and other baselines. For memory components, the Memtable size is set to 64 MB, the block cache size is 512 MB, and the local Key-offset cache size is 64 MB. *max_background_jobs* for compaction and flush is set to 4. The *slowdown_writes_trigger* for L_0 is set to 32, and the *stop_writes_trigger* is set to 48. The scheduling weights w_{cpu}, w_{io}, and w_{queue} are all set to 1.0. For NovaLSM, we configure *num_memtable_partitions* to 12 to maximize parallelism and fully utilize RDMA bandwidth. We perform 50 million operations for each benchmark, with key/value sizes set to 16 bytes and 64 bytes, respectively. All evaluations are executed 3 times, and we present the average number.

5.2 Overall Performance Evaluation

We use db_bench micro-benchmark to compare the overall performance of O^3-LSM with three baselines under limited network I/O (1 Gbps) and abundant I/O (2.5 Gbps) for accessing DS, as shown in Figure 8 and Figure 9 respectively. Also, we assess the performance of O^3-LSM with varying read-write ratios, key-value distributions via YCSB micro-benchmarks, and range queries.

Write Performance. We first focus on intensive random write benchmarking with varying value sizes. We use Operations Per Second (Ops/s) to measure the throughput. As shown in Figure 8, under 1 Gbps network bandwidth, O^3-LSM achieves up to 4.1X throughput improvement over *Disagg-RocksDB*, 3.2X improvement over *CaaS-LSM*, and 3.9X improvement over *Nova-LSM* (up to 6.4X, 4.6X, and 5.6X, respectively, for using 96 memtables in total). The P99 latency is also reduced by approximately 32% to 59%, demonstrating the significant write performance improvement of O^3-LSM. *Disagg-RocksDB* suffers throughput drops with larger value sizes due to increased write pressure, straining network I/Os to DS, and L_0 compaction. Similarly, O^3-LSM shows greater throughput improvement over *CaaS-LSM* as value sizes grow. This is because as value sizes increase, O^3-LSM's *Shard-Level Flush Offloading* results in lower write amplification

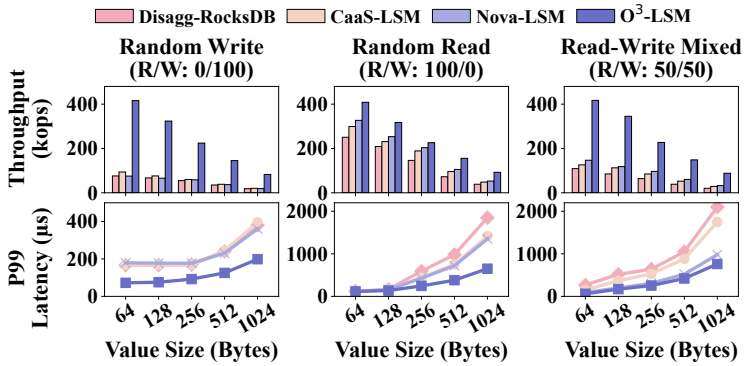


Fig. 9. Microbenchmark performance under a 2.5 Gbps DS connection

and finer-grained scheduling, enhancing overall performance. For *Nova-LSM*, throughput drops as value sizes increase, while P99 latency escalates sharply. Despite using RDMA to interconnect compute and storage clusters, the overhead of managing larger data blocks across the network eventually outweighs the bandwidth benefits provided by its sharded architecture. Notably, O^3-LSM outperforms *Nova-LSM* in both throughput and P99 latency with much slower TCP-based DS.

Figure 9 shows the random write performance evaluation under abundant bandwidth (2.5 Gbps DS connections). All four disaggregated LSM-KVS achieve better performance compared to the limited 1 Gbps network bandwidth. As the value size increases, O^3-LSM achieves up to 4.5X, 3.4X, and 4.4X throughput improvements over *Disagg-RocksDB*, *CaaS-LSM*, and *Nova-LSM*, respectively (up to 6.9X, 5.7X, 6.2X for using 96 memtables in total). Additionally, by increasing the CN to DS network bandwidth from 1 Gbps to 2.5 Gbps, the P99 latency of O^3-LSM decreases by approximately 37% to 60%, indicating that O^3-LSM can deliver better performance under abundant bandwidth. As a high-throughput, non-persistent LSM reference, dLSM reaches 1.23 million ops (Mops) at 64B value size and is omitted from the figure for clarity. As value sizes increase from 64B to 1024B, OPS across all baselines drops because the fixed network bandwidth creates a direct trade-off with payload size; similarly, dLSM's throughput scales from 1.23 to 0.58 Mops. Against this reference at 64B, O^3-LSM attains 0.415 Mops (34% of dLSM), whereas other persistent LSM-KVS baselines only achieve up to 0.09 Mops (7% of dLSM).

Read Performance. We evaluated the point lookup performance using the *readrandom* benchmark from db_bench. First, we loaded 50 million records and waited for all compactions to complete before conducting the *readrandom* test. As shown in Figure 8 and Figure 9, all systems achieved slightly better read throughput under abundant bandwidth compared to limited bandwidth. For P99 latency, both *CaaS-LSM* and *Nova-LSM* showed significant reductions of 35% and 33%, respectively, under abundant bandwidth with large value sizes. O^3-LSM reduced P99 latency by up to 69% compared to the other baselines under limited bandwidth. O^3-LSM demonstrated up to 1.8X, 0.6X, and 0.3X read throughput improvements over *Disagg-RocksDB*, *CaaS-LSM*, and *Nova-LSM*, respectively (up to 2.2X, 1.8X, 1.5X for using 96 memtables in total). However, as the value size increases, the performance of all systems declines, particularly for O^3-LSM. This is because, although Cache-Enhanced Read Delegation accelerates read performance when accessing memtables stored on DM, larger value sizes consume more RDMA bandwidth, leading to a performance drop. Compared with dLSM, O^3-LSM achieves up to 11% of the upper bound (2.24 Mops at 64B and 1.25 Mops at 1024B) . Other baselines can achieve only 0.2 Mops (8.9% of dLSM). In general, Cache-Enhanced Read Delegation effectively mitigates remote-memory access overhead for memtable hits at DM in O^3-LSM. More importantly, Shard-Level Flush Offloading produces much fewer L_0 SSTables

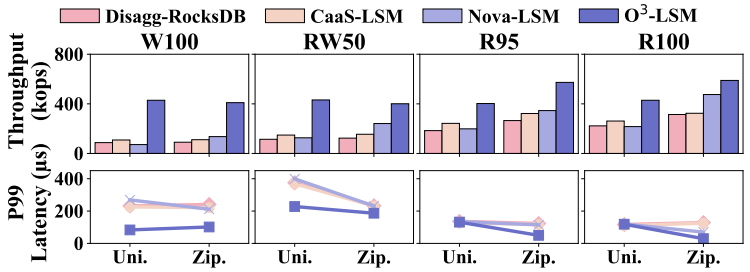


Fig. 10. YCSB micro-benchmark performance.

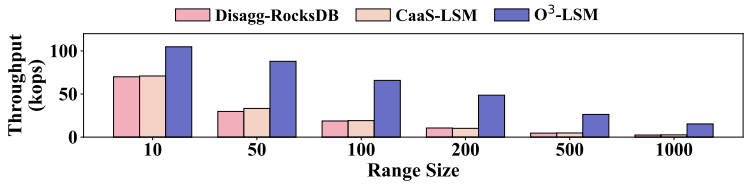


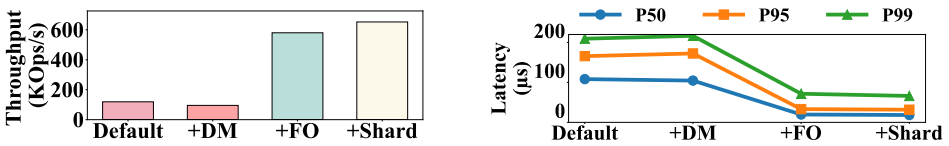
Fig. 11. Range query performance.

with key-range overlaps. Since most of the data has already been flushed to DS, most of the read queries are responded to by DS. Therefore, the current L_0 of O^3-LSM turns DS lookups from probing multiple overlapping files into a much fewer L_0 probes for point queries and significantly reduces iterator fan-in for scans. This structural reduction in read amplification eliminates redundant Bloom/metadata checks, lowers CPU and I/O overhead. Therefore, this design effectively achieves better read performance than other baselines due to fewer DS reads.

Read-Write Mixed Workload. We use the *readrandomwriterandom* benchmark. We evaluate a 50:50 read–write ratio under varying bandwidth pressure. As shown in Figure 8 and Figure 9, O^3-LSM achieves the best performance in both cases. Compared to *Disagg-RocksDB*, *CaaS-LSM*, and *Nova-LSM*, O^3-LSM achieves up to 3X, 2.3X, and 1.9X overall throughput improvements, respectively (up to 3.9X, 3.6X, 3X for using 96 memtables in total), and reduces the P99 latency up to 76%. O^3-LSM achieves up to 16.5% of the dLSM throughput (1.53 Mops at 64B and 0.82 Mops at 1024B) while other baselines only achieve lower than 10% of dLSM.

Micro-benchmark. We used YCSB, a widely adopted benchmark, in this evaluation. We employed *Uniform* and *Zipfian* key distributions and evaluated 4 different read-write ratios. We focused on four workloads: 1) **RW50** (50% read and 50% write), 2) **R95** (95% read and 5% write), 3) **R100** (100% read), 4) **W100** (100% write) with a value size of 64 bytes and 2.5Gbps sufficient bandwidth. Figure 10 shows that O^3-LSM achieves the best performance across all evaluations. We draw two main conclusions: First, the optimizations in O^3-LSM are more effective under the *Zipfian* distribution compared to the *Uniform* distribution, particularly for read-intensive workloads (R95 and R100). *Zipfian* generates hot KV-pairs, and the Key-offset Cache can effectively cache them in CN local memory, which significantly reduces the latency of accessing DS compared to the other three baselines. Second, O^3-LSM shows a slight performance drop in write-intensive workloads (W100) under the *Zipfian* distribution. The skew concentrates most inserts into a small set of shards, which impacts the shard-level parallelism and causes a lot of key-range overlapped SST files in L_0.

Range Query. We evaluate the performance of range queries using different range sizes. As shown in Figure 11, O^3-LSM outperforms both *Disagg-RocksDB* and *CaaS-LSM*. As the range size increases (e.g., from 10 to 1000), the throughput of all three systems decreases. Differently, O^3-LSM



(a) Throughput (b) Latency

Fig. 12. Ablation study on the *fillrandom* workload.

Table 2. Write Amplification Analysis (Data written in GB).

Workload	System	L0 Comp.	L1 Comp.	L2 Comp.	Total
Random Write (50M, 256B)	O^3-LSM	12.6	18.4	19.1	50.1
	Disagg-RocksDB	12.7	22.4	24.2	59.3
Mixed (50/50) (50M, 256B)	O^3-LSM	6.3	9.2	16.4	31.9
	Disagg-RocksDB	6.4	12.1	21.3	39.8

achieves significantly better throughput improvements with up to 5.2X and 4.6X compared to *Disagg-RocksDB* and *CaaS-LSM*, respectively. This improvement is due to two key factors: 1) O^3-LSM performs shard-level flush, which benefits range queries by assigning many consecutive keys to the same shard, significantly reducing prefetch time when reading SST files from DS. And 2) O^3-LSM saves the local memory for a larger block cache, which further reduces the DS I/Os.

5.3 Performance Breakdown Analysis

Write Performance. As shown in Figure 12a, simply enabling disaggregated memory (**+DM**) to extend the write buffer does not alleviate the write slowdown problem caused by the limited write buffer. In fact, it has about 18% throughput decrease and slight latency increase compared to *Disagg-RocksDB* (**Default**), due to its slow memtable transfer/rebuild and explicit flush operation delay. The performance can be even worse when flushing more memtables concurrently in DM. When the number of memtables to be flushed exceeds 10, flushing memtables at DM is slower than *Disagg-RocksDB*. However, when the proposed *Flush Offloading* (**+FO**) is integrated, flush throughput increases 34.6X, from 5MB/s to 178MB/s (100 memtable limit). More importantly, collaborative flush offloading can achieve high scalability (i.e., as the number of memtables increases, the aggregated flush throughput also increases almost linearly). Furthermore, by breaking down *Flush Offloading* into shard-level operations (**+Shard**), we achieve even better overall write throughput (652K Ops/s) and higher flush throughput (15% improvement compared with **+FO**).

Write Amplification Analysis. Table 2 provides a breakdown of data written during compaction across different LSM levels. O^3-LSM consistently achieves lower total write amplification (WA) compared to the Disagg-RocksDB. Specifically, for the 50M random write workload, O^3-LSM reduces the total data written by approximately 15.5%. This reduction is primarily observed in L_1 and L_2 compaction. By utilizing shard-level flush offloading and memtable offloading, O^3-LSM generates fewer overlapping L_0 SST files, which significantly reduces the overlapping key ranges during subsequent compaction levels, thereby mitigating the overall WA.

Flush Performance Breakdown. As shown in Table 3, compared with directly flushing the memtable from local memory to DS (Local Baseline), O^3-LSM's *Shard-Level Flush Offloading* adds only 7% average end-to-end latency. In the overall timeline, transferring a memtable to DM contributes 18.7% of total time, but our asynchronous memtable transfer design decouples this cost from the flush critical path by overlapping transfer with execution. Within the *Flush Offloading* critical

Table 3. Flush latency breakdown analysis.

Method	MP	Flush	IF	FM/Other	Total (ms)
Disagg-RocksDB	3.2	877	2.5	N/A	882.7
CaaS-LSM	3.1	874	2.6	N/A	879.7
Nova-LSM	1.9	852	2.1	N/A	856.0
Naive DM Solution	3.2	877	2.5	286.0†	1168.7
O^3-LSM	**4.7**	**875**	**3.2**	**54.0**	**936.9**

† Includes Transfer and rebuild (218 ms), and Retrieval (68 ms) from the naive implementation.

Table 4. Read performance breakdown (Uniform & Zipfian).

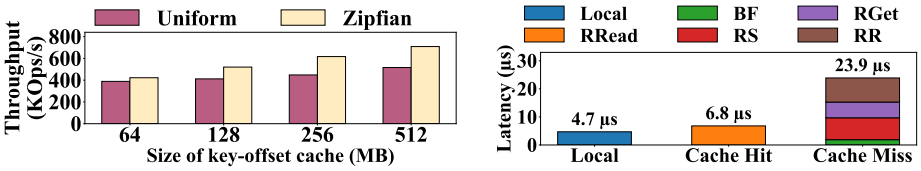
	P50 Latency (µs)		Throughput (KOps/s)	
Baselines	**Uniform**	**Zipfian**	**Uniform**	**Zipfian**
One-side Read	547.369	501.327	8.426	11.265
Disagg-RocksDB	37.465	31.129	296.975	325.315
RD	5.425	6.941	366.975	414.791
RD+LC	5.370	5.597	424.460	559.711

path, flush metadata packaging/parsing (**MP**) averages 4.7ms, install/finalization (**IF**) averages 3.2ms, and the executor's fetch of metadata and memtable blocks (**FM**) accounts for 5.7% of total execution time (54ms), the remainder is spent on merge/serialize/write to DS (**Flush**). These results indicate that the proposed collaborative flush offloading protocol overheads are modest and largely hidden by overlap, while the dominant costs lie in data movement and DS I/O.

Read Performance. As shown in Table 4, we compare O^3-LSM with several baselines to highlight the effectiveness of its read performance optimizations. First, we introduce a straightforward approach that O^3-LSM uses multiple one-sided `RDMA_READ` operations to search memtables at DM (**One-side Read**). One-sided read can only achieve 8,426 Ops/s and cause extremely high latency (547 us), which is even significantly worse than *Disagg-RocksDB*. It shows that directly using one-sided `RDMA_READ` operations to search memtables at DM can lead to extremely high overhead. Second, when we enable *Read Delegation* (**RD**) in O^3-LSM, it outperforms *Disagg-RocksDB*, delivering 0.23X and 0.27X higher throughput and 80% and 85% lower P50 latency under Uniform and Zipfian distributions, respectively. Finally, with the addition of a local key-offset cache (**RD+LC**) as the complete cache-enhanced read delegation design, compared with RD only scheme, **RD+LC** achieves a 15% and 35% improvement in throughput, along with a 2% and 19% reduction in latency, under Uniform and Zipfian distributions, respectively.

Key-Offset Cache Size. To analyze the impact of the key-offset cache size on the read performance, we conducted experiments using Uniform and Zipfian *readrandom* workloads. As shown in Figure 13a, the performance of O^3-LSM improves as the key-offset cache capacity increases. Specifically, for the Zipfian distribution workload, increasing the cache size from 64 MB to 512 MB resulted in a 67% throughput improvement. In contrast, for the uniform distribution workload, throughput increased by 32%. This shows that our design can significantly improve read performance with a larger key–offset cache and is more effective for skewed workloads.

Read Time Breakdown. As shown in Figure 13b, we present three read cases. The first scheme looks up memtables at local memory (**Local**) and costs 4.7us. The second and third correspond to our Cache-Enhanced Read Delegation. On a cache hit (**Cache Hit**), a single one-sided `RDMA_READ` completes in 6.8us (**RRead**). On a cache miss (**Cache Miss**), the pipeline performs a Bloom-filter check (**BF**, 1.89us), then `RDMA_SEND` to the DM node (**RS**, 7.8us), followed by the remote search on



(a) Varying key-offset cache (b) Time breakdown

Fig. 13. Read performance analysis.

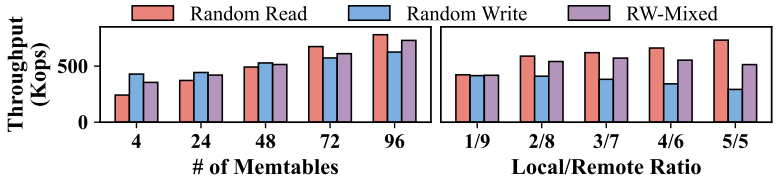


Fig. 14. Varying memtable count and local/remote ratio.

Table 5. Network I/O Breakdown (Data volume in GB).

Workload / System	CN→DM	DM→CN	CN→DS	DM→DS
Random Read (O^3-LSM)	1.1	6.7	4.2	0.0
Random Read (Disagg-RocksDB)	N/A	N/A	10.7	N/A
Random Write (O^3-LSM)	12.6	8.4	46.1‡	4.3
Random Write (Disagg-RocksDB)	N/A	N/A	59.3	N/A
Mixed (O^3-LSM)	6.3	5.2	29.6‡	2.4
Mixed (Disagg-RocksDB)	N/A	N/A	39.8	N/A

‡ Traffic is split between two CNs: **Random Write** (25.36/20.74 GB), **Mixed** (19.5.2/10.1 GB).

DM (**RGet**, 5.6us, higher than local due to extra copies), and finally `RDMA_RECV` of the result (**RR**, 8.6us), for a total of 23.9us. Both two-sided RDMA operations are amortized via doorbell batching, which lowers effective cost. Relative to the naive solution at 29us, the miss path reduces latency by 17.6% (5.1us), and the cache-hit path reduces latency by 76.6% (22.2us).

Performance Influence of Different Maximum Memtables. Figure 14 shows the performance of various maximum numbers of memtables allowed in O^3-LSM (2 at most in CN and the rest will be transferred to DM). As more memtables can be stored in DM, O^3-LSM demonstrates improved performance across different workloads. This outcome aligns with our motivation: caching more memtables in the write buffer enhances overall performance. This result highlights the success of our design in overcoming the new challenges by extending the write buffer with DM. For a fixed maximum number of memtables in the write buffer (in our tests, this was set to 10 memtables), the ratio of local to remote memtables also impacts overall performance. As shown in Figure 14, increasing the percentage of local memtables improves read performance since more KV-pairs can be searched in the local memtables. However, write performance declines due to inefficiencies in leveraging other nodes (CNs or DM nodes) to flush local memtables to the DS, leading to lower performance. Exploring automatic memtable ratios between local and DM will be our future work.

Network I/O Breakdown. As shown in Table 5, we analyze how O^3-LSM redistributes network traffic across different nodes. In *Disagg-RocksDB*, the CN (owning LSM-KVS) handles all I/O for flushing data to DS. In contrast, O^3-LSM can offload a significant portion of this traffic to other

Table 6. Shard-Level Optimization on Network Traffic (MB/s).

System	Metric	CN→DM	DM→CN	CN→DS	DM→DS
O^3-LSM w/ Shard	Peak	146.2	86.7	265.0	42.3
	Average	58.4	34.2	234.0	40.2
O^3-LSM w/o Shard	Peak	963.2	216.5	297.0	259.0
	Average	39.2	24.6	214.0	36.8

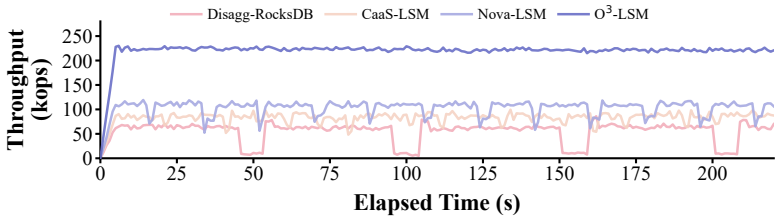


Fig. 15. Throughput Stability over Time.

available CNs or DM nodes. In our 2-CN evaluation of a 50M random write workload, although we introduce 12.6 GB of traffic for memtable offloading (CN→DM), we successfully reduce the direct CN→DS pressure from 59.3 GB to 46.1 GB. This traffic is effectively rebalanced across the two CNs and the DM node, with the two CNs contributing 25.36 GB and 20.74 GB respectively, while the DM node handles the remaining 4.3 GB. These results validate the effectiveness of our proposed scheduling algorithm in redistributing network I/O across components. This shift is critical during write-intensive bursts, as it prevents the network interface of a single node from becoming a bottleneck by distributing the network I/Os across all available resources.

Shard-level optimization. Table 6 demonstrates the critical role of shard-level partitioning in smoothing network traffic. Without shard-level optimization, the system experiences massive traffic spikes, with peak CN→DM throughput reaching 963.2 MB/s against an average of only 39.2 MB/s. This bursty behavior leads to severe network congestion. By partitioning memtables into shards and scheduling their transfer and flush in parallel, O^3-LSM reduces peak CN→DM traffic by 6.5× (from 963.2 to 146.2 MB/s), while increasing the average throughput. These results illustrate how sharded flush mitigates bandwidth bursts and smooths network traffic across nodes. Consequently, sharding boosts W100-Uniform throughput from 268.1 to 428.9 kops (1.6×). Even under W100-Zipfian skew, O^3-LSM achieves 409.5 kops, which is a 1.7× gain over the non-sharded baseline (241.2 kops). This demonstrates that parallelizing shard flushes mitigates hotspots and maintains high performance by allowing independent flush tasks to proceed concurrently.

Performance Stability. As illustrated in the time-series analysis (Figure 15), O^3-LSM exhibits high stability compared to other baselines. Disagg-RocksDB and Nova-LSM suffer from frequent throughput dips and wide variance. In contrast, O^3-LSM maintains a consistently high and smooth throughput profile. This robustness is further confirmed by the box plots in Figure 16, where O^3-LSM displays the most compact distributions and minimal outliers across all workloads and value sizes. This stability stems from replacing coarse-grained flushes with fine-grained shard-level offloading. By partitioning data into small shards and scheduling parallel transfers, O^3-LSM avoids the queuing delays and write stalls typical of traditional architectures. This resilience to latency fluctuations makes O^3-LSM ideal for latency-sensitive cloud applications.

Portability of O^3-LSM Optimizations. To evaluate the generalizability of our design, we integrate O^3-LSM memtable and flush offloading mechanisms into CaaS-LSM, a representative compaction

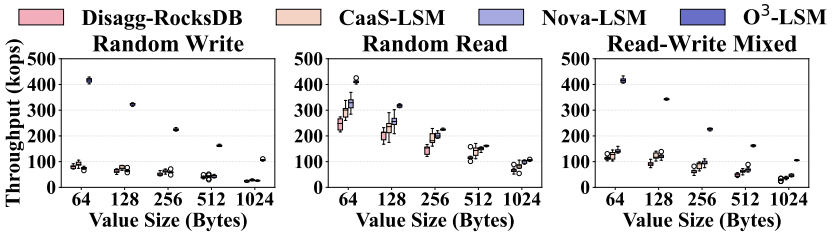


Fig. 16. Performance Variance and Outlier Analysis.

Table 7. CaaS-LSM w/ and (w/o) O^3-LSM Optimizations.

Workload	Throughput (kops/s)	P99 Latency (μs)
Random Write	265.8 (60.1)	82.4 (169.1)
Random Read	255.4 (189.1)	210.1 (440.8)
Read-Write Mixed	272.5 (84.8)	215.8 (528.0)

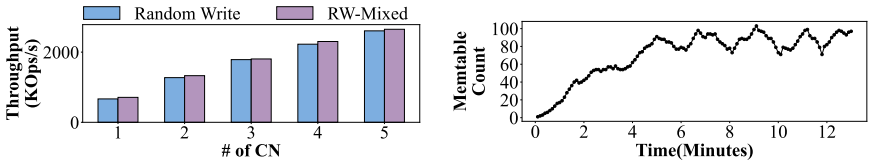
Table 8. Performance Comparison on KVrocks. Metrics are Throughput (kops/s) / P99 Latency (ms).

Workload	Disagg-RocksDB	CaaS-LSM	O^3-LSM
SET-only	86.3 / 391	105.9 / 287	**381.2 / 177**
GET-only	125.9 / 823	159.4 / 751	**272.9 / 476**
Mixed 10/90 (Get/Set)	92.5 / 450	115.2 / 320	**355.4 / 195**
Mixed 50/50 (Co-Located)	102.1 / 610	132.8 / 510	**315.6 / 310**
Mixed 50/50 (Remote)	72.4 / 1250	98.6 / 980	**285.4 / 480**
Mixed 90/10 (Get/Set)	118.4 / 780	148.6 / 680	**285.2 / 410**
Pipelined SET (16 Threads)	185.2 / 1250	235.4 / 980	**720.5 / 420**

offloading LSM-KVS [58]. As shown in Table 7, the integrated system achieves a Random Write throughput of 265.8 kops/s (a 4.4× improvement over CaaS-LSM's 60.1 kops/s) with a P99 latency of 82.4 μs (a 51% reduction from 169.1 μs). This demonstrates that memtable and flush offloading provide portable benefits. O^3-LSM effectively mitigates write stalls and I/O contention within compaction-offloading frameworks. These results underscore O^3-LSM's potential as a portable component that maximizes disaggregated LSM-tree efficiency through three-layer offloading.

5.4 Analysis of Real-World Application

We use Kvrocks [7] (a Redis-compatible distributed KVS that uses RocksDB as its storage engine) to evaluate the end-to-end performance. Kvrocks integrates three different disaggregated LSM-KVS solutions as its storage engine: Disagg-RocksDB, CaaS-LSM, and O^3-LSM. We use Redis benchmark and measure throughput and P99 latency for GET-only, SET-only, and mixed GET/SET with 10/90, 50/50, and 90/10 ratios, plus pipelined variants, and we also run both co-located and remote clients. The benchmark issues 40 million queries by 16 threads. As shown in Table 8, O^3-LSM consistently outperforms Disagg-RocksDB and CaaS-LSM across all Redis-benchmark workloads. In SET-only and Mixed 10/90 scenarios, it achieves up to 4.4× higher throughput and 54% lower P99 latency. Even under remote client configurations and pipelined workloads, O^3-LSM leads CaaS-LSM by 2.9× and 3.0× respectively. These gains stem from offloading memtables and parallelizing shard-level flushes, which prevents write stalls and I/O imbalances. These results demonstrate O^3-LSM's efficiency in real-world applications.



(a) Scale out compute node (b) Memtable Counts on DM node

Fig. 17. Scalability and memory elasticity of O^3-LSM.

Table 9. Performance with varying numbers of DM nodes.

Configuration	Random Write (kops/s)	RW-Mixed (kops/s)
3 CN + 1 DM	1,786	1,805
3 CN + 2 DM	1,920	1,950
3 CN + 3 DM	1,980	2,020

Table 10. Write stall mitigation: Fixed vs. Shared DM Pool.

Setting (5 CNs)	Tput (Mops)	Write Stalls	P99 Latency (ms)
Fixed Pool (0.5GB×5)	1.85	840	240
Shared Pool (2.5GB)	**2.62**	**65**	**45**

5.5 Scalability

Scalability with Multiple CNs. We incrementally add CNs under Random Write and Read-Write Mixed workloads to evaluate O^3-LSM scalability. Benchmarks were initiated at different time intervals, and we measured the total aggregate throughput across all active CNs. Figure 17a shows the throughput comparison as the number of CNs increases. For both workloads, we observed a consistent increase in throughput, demonstrating good scalability. Adding 4 more CNs improved total throughput by up to 2.9X compared to a single CN. As shown in Figure 17b, with 5 CNs sharing the DM, the total memtable counts demonstrate small fluctuations over an extended period. This stability reflects improved memory utilization and high efficiency, aligning with modern DDC objectives. By balancing flush operations across CNs, O^3-LSM prevents simultaneous I/O pressure on any single node, ensuring stable flushes.

Scalability with Multiple DM Nodes. We evaluated O^3-LSM with multiple DM nodes serving a CN cluster (3 CNs). As shown in Table 9, performance gains scale with the number of DM nodes. Upgrading from 1 to 3 DMs boosts the total aggregate throughput by 10.8% for Random Writes and 11.9% for RW-Mixed workloads. Each CN independently selects a target DM for offloading and stores the location (Node ID and memory address) within its local memtable metadata. By maintaining this tracking information locally, CNs can concurrently perform tasks (such as memtable offloading and read delegation) across multiple DM nodes. This design parallelizes the offloading overhead and prevents single-node bottlenecks for network I/O or CPU-intensive tasks. Such architectural flexibility allows O^3-LSM to linearly expand its memory-tier resources to support increasing concurrent requests and larger working sets in disaggregated environments.

Write Stall Mitigation via Shared DM Pool. In the fixed partitioning configuration, each of 5 CNs is assigned a 0.5 GB memory quota at DM (2.5 GB in total). Under this setup, individual CNs frequently exhaust their quotas during write bursts, triggering 840 total stalls and a high P99 latency of 240 ms. Conversely, the shared DM pool configuration utilizes the aggregate 2.5 GB (the same memory budget) capacity to elastically absorb skewed write traffic across the 5 CNs. This collective

buffering reduces the total stall count by 92.2% (from 840 to 65) and slashes P99 latency by over 5×, as shown in Table 10. By decoupling memory resources from specific compute instances and organizing them as a shared memory pool, O^3-LSM allows instances with high write intensity to borrow idle capacity from the pool. This elasticity, paired with parallelized shard flushes, maintains a high aggregate total throughput of 2.62 Mops/s (524 Kops/s per CN), a 41.6% improvement over the 1.85 Mops/s (370 Kops/s per CN) achieved by the fixed configuration.

6 Related Work

Persistent Disaggregated LSM-KVS. Currently, there are a number of studies focusing on optimizing persistent LSM-KVS on disaggregated storage, including Kemme et al [4], Hailstorm [9], Nova-LSM [27], IS-HBase [12], RocksDB-Cloud [41], TerarkDB [10], and DisaggreRocksDB [19]. Compaction offloading or remote compaction (first proposed in [4]) is mainly used in those studies to reduce the network I/Os between storage and compute nodes. Compared with existing studies of optimizing LSM-KVS for DS, O^3-LSM leverages the DM to achieve memory extension and explicit performance improvement.

In-memory Key-Value Store for Disaggregated Memory. Several in-memory KV store designs have been optimized for DM. Sherman [49] is a B+Tree-based KVS that places all tree nodes in DM while maintaining metadata and index caches locally at CNs, achieving high-performance lookups through designed RDMA access patterns. PolarDB Serverless [11] uses a multi-tenant memory pool over DM to cache data pages, while hash-based systems such as Clover [47], FUSEE [42], and Dinomo [30] store all KV-pairs and indexing structures entirely in DM. Some studies have also explored selective RDMA offloading to balance between one-sided and two-sided RDMA operations. Distributed tree-based indexes [25, 61] use this to accelerate traversal, and DEX [35] adopts a similar approach for range indexing on disaggregated memory. O^3-LSM uniquely integrates this mechanism within the LSM-tree lifecycle, read delegation is co-designed with WAL-based durability and shard-level flush semantics to ensure consistency across the three-layer offloading architecture.

LSM-KVS Write Bottleneck. Several studies aim to address the write performance bottleneck of LSM-KVS. [20, 29] add an NVM layer to the storage end of LSM-KVS, avoiding serialization overhead and mitigating write amplification by supporting faster flushing. O^3-LSM proposes remote flush to address the bottleneck caused by the slow DM. Some works [27, 55] have different designs for parallelizing L_0-L1 compaction. MatrixKV [55] maintains a specially partitioned L_0 layer on NVM, aiming to fully utilize NVM's high throughput and byte-addressable features. O^3-LSM actively merges the KV range of a single node in the cluster, offloads the memtables on DM and fully utilizes the compute resources of compute cluster. Other optimizations propose offloading compaction to other compute resources like DS node CPUs [4, 12, 19, 27, 32, 41], FPGAs [26] or other compute unit [17, 54] to mitigate compaction overhead, or maintaining stability [8, 16, 57].

7 Conclusion

In this paper, we propose O^3-LSM, the first LSM-KVS leveraging DM as a write buffer extension for DS. We address DM performance challenges via DM-optimized memtables while introducing cache-enhanced read delegation and shard-level flush offloading. Future work includes fine-grained DM memtable management, automated memory ratios, and advanced flush scheduling.

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. We thank all members of the ASU-IDI Lab for their help and useful comments. This work was partially funded by the National Science Foundation under Grant Numbers #2412436, #2443219, and #2337806.

References

[1] 2019. *CXL Consortium.* https://www.computeexpresslink.org

[2] 2024. *https://github.com/asu-idi/O3-LSM.*

[3] Accessed: June, 2023. *ZippyDB: A Modern, Distributed Key-Value Data Store.* https://www.youtube.com/watch?v=DfiN7pG0D0k

[4] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction Management in Distributed Key-Value Datastores. (2015), 850–861. doi:10.14778/2757807.2757810

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the 15th european conference on computer systems, EuroSys 2020* (2020-04-15) *(Proceedings of the 15th european conference on computer systems, EuroSys 2020)*. Association for Computing Machinery, Inc. doi:10.1145/3342195.3387522

[6] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing* (USA, 2020) *(HotCloud'20)*. USENIX Association, Article 15.

[7] Apache. [n. d.]. Kvrocks. https://github.com/apache/incubator-kvrocks

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019). 753–766.

[9] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed Lsm-Based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020). 301–316.

[10] bytedance. 2019. *Terark.* https://github.com/bytedance/terarkdb

[11] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. Polardb Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data* (2021). 2477–2489.

[12] Zhichao Cao, Huibing Dong, Yixun Wei, Shiyong Liu, and David H. C. Du. 2022. IS-hbase: An in-Storage Computing Optimized Hbase with I/O Offloading and Self-Adaptive Caching in Compute-Storage Disaggregated Infrastructure. 18, 2, Article 15 (2022). doi:10.1145/3488368

[13] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (2020). 209–223.

[14] Compute Express Link (CXL) Consortium. Accessed: November, 2022. CXL 3.0 White Paper. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf

[15] 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* (2010). ACM, 143–154.

[16] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-tree Compactions Correctly. 15, 11 (2022), 3071–3084. doi:10.14778/3551793.3551853

[17] Chen Ding, Jian Zhou, Jiguang Wan, Yiqin Xiong, Sicen Li, Shuning Chen, Hanyang Liu, Liu Tang, Ling Zhan, Kai Lu, and Peng Xu. 2023. DComp: Efficient Offload of LSM-tree Compaction with Data Processing Units. In *Proceedings of the 52nd International Conference on Parallel Processing* (¡conf-loc¿, ¡city¿Salt Lake City¡/city¿, ¡state¿UT¡/state¿, ¡country¿USA¡/country¿, ¡/conf-loc¿, 2023) *(Icpp '23)*. Association for Computing Machinery, New York, NY, USA, 233–243. doi:10.1145/3605573.3605633

[18] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR* (2017), Vol. 3. 3.

[19] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. 1, 2, Article 192 (2023). doi:10.1145/3589772

[20] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2023. Revisiting Log-Structured Merging for KV Stores in Hybrid Memory Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada, 2023) *(Asplos 2023)*. Association for Computing Machinery, New York, NY, USA, 674–687. doi:10.1145/3575693.3575715

[21] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, 2019-07). USENIX Association, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[22] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016). 249–264.

[23] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. (2011).

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003). 29–43.

[25] Chang Guo, Ning Yan, Lipeng Wan, and Zhichao Cao. 2025. LegoIndex: A Scalable and Modular Indexing Framework for Efficient Analysis of Extreme-Scale Particle Data. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*. 1–14.

[26] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale e-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands, 2019) *(Sigmod '19)*. Association for Computing Machinery, New York, NY, USA, 651–665. doi:10.1145/3299869.3314041

[27] 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* (2021-06). ACM. doi:10.1145/3448016.3457297

[28] IBM. 2018. *Disaggregated Memory.* https://www.ibm.com/blogs/research/2018/01/advancing-cloud-memory-disaggregation/

[29] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, 2022-07). USENIX Association, 161–177. https://www.usenix.org/conference/osdi22/presentation/kim

[30] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. 15, 13 (2022), 4023–4037. doi:10.14778/3565838.3565854

[31] Huaicheng Li, Daniel Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, and Ricardo Bianchini. 2022. First-Generation Memory Disaggregation for Cloud Platforms. (2022).

[32] 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* (Virtual Event, Queensland, Australia, 2021) *(Cikm '21)*. Association for Computing Machinery, New York, NY, USA, 3906–3915. doi:10.1145/3459637.3481913

[33] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. 2020. Disaggregated Data Centers: Challenges and Trade-Offs. 58, 2 (2020), 20–26. doi:10.1109/MCOM.001.1900612

[34] Xinxin Liu, Yu Hua, and Rong Bai. 2021. Consistent Rdma-Friendly Hashing on Remote Persistent Memory. In *2021 IEEE 39th International Conference on Computer Design (ICCD)* (2021). IEEE, 174–177. doi:10.1109/ICCD53106.2021.00037

[35] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2603–2616.

[36] Patrick MacArthur and Robert D. Russell. 2012. A Performance Study to Guide RDMA Programming Decisions. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems* (USA, 2012) *(Hpcc '12)*. IEEE Computer Society, 778–785. doi:10.1109/HPCC.2012.110

[37] Santhosh Nagaraj Nag. 2023. CXL (Compute Express Link) Technology. 11, 6 (2023), 94–102.

[38] Oracle. 2012. *Infiniband.* https://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf

[39] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, given-i=JR family=Tipton, given=JR, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021-02). USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[40] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. *Proc. ACM Manag. Data* 2, 3 (2024), 180:1–26.

[41] rockset. 2020. *Rockscloud.* https://github.com/rockset/rocksdb-cloud

[42] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (2023). 81–98.

[43] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019-02). USENIX Association, 255–270. https://www.usenix.org/conference/nsdi19/presentation/shrivastav

[44] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 10)* (2010). IEEE, 1–10.

[45] Viraj Thakkar, Dongha Kim, Yingchun Lai, Hokeun Kim, and Zhichao Cao. 2025. SHIELD: Encrypting Persistent Data of LSM-KVS from Monolithic to Disaggregated Storage. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[46] Viraj Thakkar, Madhumitha Sukumar, Jiaxin Dai, Kaushiki Singh, and Zhichao Cao. 2024. Can Modern Llms Tune and Configure LSM-based Key-Value Stores?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems* (2024). 116–123. doi:10.1145/3655038.3665954

[47] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference* (USA, 2020) *(Usenix Atc'20)*. USENIX Association, Article 3.

[48] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the 2023 International Conference on Management of Data* (2023). 37–44.

[49] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed b+ Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data* (2022). 1033–1048.

[50] Ruihong Wang, Chuqing Gao, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2024. Optimizing LSM-based Indexes for Disaggregated Memory. *VLDB Journal* 33, 6 (2024), 1813–1836.

[51] Ruihong Wang, Jianguo Wang, and Walid G. Aref. 2025. Cache Coherence Over Disaggregated Memory. *PVLDB* 18, 9 (2025), 2978–2991.

[52] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *PVLDB* 16, 1 (2022), 15–22.

[53] Ruihong Wang, Jianguo Wang, Prishita Kadam, M Tamer Özsu, and Walid G Aref. 2023. dLSM: An LSM-based Index for Memory Disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)* (2023). IEEE, 2835–2849.

[54] Peng Xu, Jiguang Wan, Ping Huang, Xiaogang Yang, Chenlei Tang, Fei Wu, and Changsheng Xie. 2020. LUDA: Boost LSM Key Value Store Compactions with Gpus. arXiv:2004.03054 [cs.DC]

[55] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. {MatrixKV}: Reducing Write Stalls and Write Amplification in {LSM-tree} Based {KV} Stores with Matrix Container in {NVM}. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020). 17–31.

[56] Seehwan Yoo, Hojin Shin, Sunghyun Lee, and Jongmoo Choi. 2022. A Read Performance Analysis with Storage Hierarchy in Modern KVS: A RocksDB Case. In *2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA)* (2022). 45–50. doi:10.1109/NVMSA56066.2022.00017

[57] Jinghuan Yu, Sam H. Noh, Young-ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, 2023-02). USENIX Association, 65–80. https://www.usenix.org/conference/fast23/presentation/yu

[58] 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. 2, 3 (2024), 1–26.

[59] Qizhen Zhang, Yifan Cai, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *Conference on Innovative Data Systems Research* (2020).

[60] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. 13, 9 (2020), 1568–1581. doi:10.14778/3397230.3397249

[61] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 international conference on management of data*. 741–758.

Received October 2025; revised January 2026; accepted February 2026