The VLDB Journal
https://doi.org/10.1007/s00778-024-00863-y

SPECIAL ISSUE PAPER

Optimizing LSM-based indexes for disaggregated memory

Ruihong Wang[1] · Chuqing Gao[1] · Jianguo Wang[1] · Prishita Kadam[1] · M. TamerÖzsu[2] · Walid G. Aref[1]

Received: 15 March 2023 / Accepted: 4 June 2024
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2024

Abstract
The emerging trend of memory disaggregation where CPU and memory are physically separated from each other and are connected via ultra-fast networking, e.g., over Remote Direct Memory Access (RDMA), allows elastic and independent scaling of compute (CPU) and main memory. This paper investigates how indexing can be efficiently designed in the memory disaggregated architecture. Although existing research has optimized the B-tree for this new architecture, its performance is unsatisfactory. This paper focuses on LSM-based indexing and proposes `dLSM`,the first highly optimized LSM-tree for disaggregated memory. `dLSM` introduces a suite of optimizations including reducing software overhead, leveraging near-data computing, tuning for byte-addressability, and an instantiation over RDMA as a case study with RDMA-specific customizations to improve system performance. Experiments illustrate that `dLSM` achieves 2.3× to 11.6× higher write throughput than running the optimized B-tree and four adaptations of existing LSM-tree indexes over disaggregated memory. `dLSM` is written in C++ (with approximately 54,400 LOC), and is open-sourced.

Keywords Log-structured merge (LSM) tree · Disaggregated memory · RDMA

1 Introduction

Memory disaggregation is an emerging trend in modern data centers to allow independent and elastic scaling. Companies, e.g., Microsoft, Alibaba, and IBM, experiment with memory disaggregation [1, 27, 56]. Unlike traditional data centers that consist of a collection of traditional *converged* servers, where compute (CPU) and memory are tightly coupled into the same physical servers (Fig. 1a), with memory disaggregation, compute and memory are physically separated and are connected via fast networking (Fig. 1b). In the new architecture, there are two distinct types of servers in data centers to provide compute and memory: compute nodes and memory nodes.[1] Each compute node has powerful computing capability, e.g., 100 s of CPU cores but limited local memory, e.g., a few GBs, while each memory node has weak computing power, e.g., a few CPU cores, but abundant memory, e.g., 100 s of GBs [27, 66, 79, 81, 88, 93–96].

Memory disaggregation provides substantial benefits for data centers [1, 27, 56, 79, 81, 93–96]. It allows elastic and independent scaling of compute and memory, and improves memory utilization, which can be translated into lower total cost of ownership (TCO) because memory is an expensive resource. Also, it provides higher reliability as compute and memory can fail and be upgraded independently without affecting each other. Finally, it breaks the memory boundary within a single machine and facilitates easier sharing of main memory among multiple machines.

This paper focuses on indexing techniques for disaggregated memory, where the majority of data is stored in remote memory while caching hot data in local memory. Prior work, e.g., Sherman [80], studies how to optimize B-tree indexing for memory disaggregation. However, the write performance

✉ Jianguo Wang
csjgwang@purdue.edu

Ruihong Wang
wang4996@purdue.edu

Chuqing Gao
gao688@purdue.edu

Prishita Kadam
pkadam@purdue.edu

M. TamerÖzsu
tamer.ozsu@uwaterloo.ca

Walid G. Aref
aref@purdue.edu

1 Purdue University, West Lafayette, IN, USA

2 University of Waterloo, Waterloo, ON, USA

[1] With storage disaggregation, there are also dedicated storage nodes, but this paper focuses mainly on memory disaggregation.

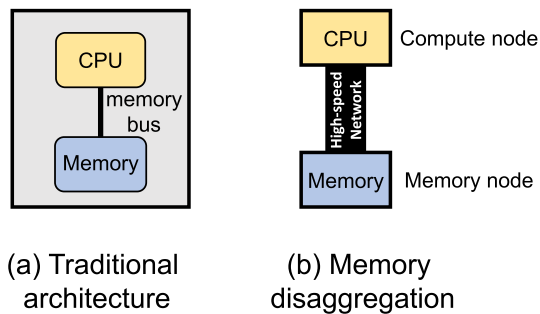


Fig. 1 Traditional architecture vs. memory disaggregation

is unsatisfactory (as shown in Sec. 11). To improve performance, this paper focuses on LSM-based (log-structured merge tree) indexing [65] in the presence of memory disaggregation.

Why Disaggregated LSM-tree? (1) The LSM-tree fits naturally into the two-level hierarchy of the disaggregated memory setting with local memory and remote memory, where the new updates are accumulated into local buffers and are regularly flushed to the remote memory in the background. This can move network accesses off the write path, because writes hit local buffers first. (2) The LSM-tree can achieve high write performance by converting small random writes to large sequential writes to best leverage network-bandwidth [63, 80]. As in our experiments on Remote Direct Memory Access (RDMA) Mellanox EDR ConnectX-4 NIC, there is a 100× performance difference between transferring the same amount of data in 64 byte units vs. 1MB units based on the RDMA benchmark [7]. (3) Our experiments (Sec. 11) show that the LSM-tree is feasible for disaggregated memory and, as expected, it outperforms the optimized B-tree on writes (i.e., Sherman [80]) in this new architecture.

Challenges. There are unique challenges in realizing an optimized LSM-tree over disaggregated memory. (1) The ultra-fast networking significantly narrows the performance gap between local and remote memories. Thus, software overhead that traditionally has not been a concern for slower devices, e.g., SSDs or HDDs, has now become a performance bottleneck for modern hardware with high-performance networks. Thus, it makes sense to minimize software overhead in this new setting. (2) The memory node has CPU cores to perform arbitrary computing that does not exist in other memory hierarchies. (3) In contrast to being block-addressable as is the case in conventional storage devices, remote memory is byte-addressable. Thus, index design needs to be aware of byte-addressability. (4) Another challenge is the effective use of the complex communication interfaces as they become crucial in the case of memory disaggregation, e.g., RDMA is non-trivial as it involves many alternative primitives, e.g., one- vs. two-sided RDMA (Sec. 2.2). Realizing efficient RDMA communication requires careful design.

The `dLSM` Approach. This paper presents `dLSM`, a purpose-built LSM-based index for disaggregated memory. `dLSM` investigates LSM-based indexing in this setting, and introduces a number of optimizations to address the aforementioned challenges. `dLSM` reduces the software overhead, e.g., the synchronization. `dLSM` offloads LSM-tree compaction selectively to the remote memory node to reduce data movement by exploiting the CPUs in memory nodes if possible. `dLSM` tunes the index layout to leverage byte-addressability in the remote memory. `dLSM` optimizes communication including customized remote procedure calls (RPC, for short) and asynchronous I/O.

The paper makes the following contributions:

- **Index design over disaggregated memory:** We present the design of `dLSM`, the first optimized LSM-based index for disaggregated memory. `dLSM` is implemented in C++ (with approximately 54,400 LOC), and is available as open-source at https://github.com/ruihong123/dLSM.
- **Reducing software overhead:** `dLSM` reduces the software overhead, e.g., the synchronization overhead.
- **Near-data computing for remote compaction:** `dLSM` leverages the idea of near-data computing in the context of the disaggregated memory architecture, and selectively pushes down the LSM-tree compaction to the remote memory to significantly reduce data transfer.
- **Customized optimizations for byte-address -ability:** `dLSM` is tuned to deprecate the concept of block structures to leverage the byte-addressability in disaggregated memory to improve performance.
- **Customized optimizations for RDMA:** We instantiate `dLSM` over RDMA-enabled disaggregated memory. `dLSM` applies RDMA-specific optimizations, e.g., asynchronous I/O and customized RPC for high performance.

We instantiate `dLSM` over RDMA as a case study. However, many of the ideas (e.g., reducing software overhead and customized optimizations for byte-addressability) can be applied to other technologies, e.g., CXL [4].

Overview of Experiments. We conduct experiments to evaluate `dLSM` using the standard RocksDB benchmark [9] with 100 million key-value pairs (over a 40GB dataset). We compare `dLSM` with five baseline solutions (Nova-LSM [49], disk- and memory-optimized RocksDB, and Sherman [80]) over the memory disaggregated architecture. Experiments show that `dLSM` achieves 2.3× to 11.6× higher write throughput while losing up to 12.5

This article is an extended version of the conference version presented in [82]. We have the following new contributions.

- We develop a new adaptive compaction strategy to automatically adjust the LSM compaction to the remote

memory node depending on varied computing power in the remote memory node (Sec. 5.3).
- We introduce a new technique to optimize the usage of buffer memory in the compute node, which makes the index perform well when the data volume is large and the local buffer size is limited (Sec. 6).
- We develop a non-blocking transaction-consistent checkpointing technique to make dLSM persistent (Sec. 8).
- We provide more technical details that are omitted in the conference version [82], e.g., thread local queue pair (Sec. 10.2.1), RDMA Memory Region allocator (Sec. 10.2.2), and RDMA-based file system implementation (Sec. 11.1).
- We add new experiments for more comprehensive evaluation (Fig. 14 and Fig. 16).

Paper Organization. The rest of this paper proceeds as follows. Sec. 2 presents background material. Sec. 3 demonstrates the overall system architecture. Sec. 4 presents the new optimizations to reduce software overhead. Sec. 5 presents optimizations that leverage near-data computing in the context of remote compaction. Sec. 6 introduces optimizations for byte-addressability. Sec. 7 introduces optimizations to deal with the mixed workload. Sec. 8 demonstrates the way to efficiently make dLSM persistent. Sec. 9 discusses the method to extend dLSM onto multiple compute and memory nodes. Sec. 10 discusses the instantiation of dLSM over RDMA-enabled disaggregated memory. Sec. 11 presents the experimental evaluation of dLSM. Sec. 12 reviews related work. Sec. 13 concludes the paper.

2 Background

2.1 Resource disaggregation

Resource disaggregation is an innovative technology in data centers [27, 28, 66, 80, 88, 92, 94, 96], in large part due to the recent breakthroughs in fast networking technologies, e.g., RDMA [51, 55]. Traditionally, data centers are composed of servers that physically contain predefined amounts of compute, memory, and storage connected by high-speed buses on the same server box. However, in a fully disaggregated data center, resources are separated into "disaggregated" components connected by a fast network fabric. This brings in many benefits, e.g., higher resource utilization, better elasticity, and lower cost [1, 27, 56, 93–96].

There are two popular types of resource disaggregation: (1) Storage disaggregation that decouples compute from storage; (2) Memory disaggregation that separates compute from memory. Industrial-strength systems, e.g., Amazon AWS, Alibaba Cloud, and Microsoft Azure, have deployed storage disaggregation into production. They have reinvented database systems, e.g., Aurora [76], PolarDB [26], and Socrates [14] to explicitly optimize for disaggregated storage.

Recently, memory disaggregation has gained significant attention in both industry and academia [1, 5, 27, 53, 93, 94, 96]. In contrast to storage disaggregation, it is more challenging to optimize DBMSs for memory disaggregation due to the increased severity of performance issues [53, 92–94]. Moreover, memory disaggregation usually relies on a high-speed network fabric, e.g., RDMA, while storage disaggregation can be built on conventional RPCs [76].

2.2 Interconnection for disaggregated memory

Ultra-fast networking technologies exist for interconnecting compute and memory nodes. For example, RDMA is a high-speed inter-memory communication mechanism with low latency. It allows direct access to memory in remote nodes [51]. It bypasses the host operating system when transferring data to avoid extra data copy. RDMA operates over Infiniband or lossless Ethernet. Its kernel-bypassing and low-latency features make it applicable to high-performance data centers [1, 5, 27, 96]. Another promising communication technology is Compute Express Link (CXL) [4], which is a high-speed interconnection between advanced CPU and peripheral devices, e.g., CXL-extended memory. CXL connection protocols guarantee cache coherence between CPU cache and connected memory. Thus, the CPU can directly access the remote CXL-based memory via load and store. This paper focuses on RDMA as it is more mature.

2.3 Log-structured merge (LSM) tree

The LSM-tree [65] is a widely used index in modern data systems. It is optimized for write-intensive workloads by trading random writes for sequential writes. It has a memory component and multiple disk components. Writes are first inserted into the memory component, and when it gets full, it is flushed to disk to form a new disk component. The disk components can be merged through a *compaction* phase to form multiple layers. The LSM-tree is an immutable index structure as all the disk components are immutable.

There are many implementations of the LSM-tree. Among these, RocksDB [10] – improved version of LevelDB [6] – is one of the most widely adopted implementation. We use the RocksDB implementation to introduce LSM-tree concepts and terminology. Entry inserts, updates, and deletes are all appended into a write buffer. The entries are first written into a write batch that are committed all at once. Then, the write batches are assigned with sequence numbers to reflect the time order of the entries. To guarantee durability, the write batch is written to a write-ahead log (WAL). Then, the key-value pairs are inserted into an in-memory

skip list [69], termed the *MemTable*. When the size of the MemTable reaches a certain threshold, it is switched to an immutable read-only table that waits for a scheduled flushing task. Flushing serializes the MemTable to files termed *sorted string tables* (SSTables, for short) that contain data blocks, index blocks, and Bloom filters.

ptThe SSTables are organized into different levels. The newly flushed SSTables are dumped into Level 0. Since the SSTables in Level 0 are not sorted to improve write performance, there is a limit (*level0_stop_writes_trigger*) for the total number of SSTables in Level 0; exceeding the limit results in a write stall. When the number of SSTables at one level reaches a preset threshold, a compaction process is triggered to merge data files into the next level. Compaction works as follows. The target SSTables are picked from two consecutive levels. All the SSTables within one level will be ordered, and they do not have overlaps (except Level 0). Multiple background threads handle flush and compaction tasks. When a task finishes, a background thread modifies the LSM-tree metadata to record this change in LSM-tree structure.

In order for a reader to fetch a key-value pair, a sequence number is assigned for this reader to ensure that the proper version is read, and any read-write conflicts are resolved through snapshot isolation [10]. In order to have a consistent view of the LSM-tree, the reader gets an immutable copy of the LSM-tree metadata that corresponds to a snapshot. During the read process, the reader thread traverses the MemTable and immutable tables, and then traverses the SSTables from Levels 0 to n. Whenever the reader finds the matched key-value pair, it returns directly and skips the remaining tables. It also leverages Bloom filters to improve read performance because if a key-value is not present in the Bloom filter of a SSTable then it is not necessary to check that SSTable.

3 Overview of `dLSM` architecture

Figure 2 gives the architecture of `dLSM` deployed on one compute node and one memory node following prior LSM-tree designs, e.g., as in RocksDB [10] and LevelDB [6] that are designed for a single-node setting. This is appropriate to describe `dLSM`'s design. Even in this configuration, there are non-trivial and interesting challenges as mentioned in Sec. 1. In Sec. 9, we discuss how to extend `dLSM` to multiple compute and memory node configurations.

In the disaggregated memory setup that we study, a compute node has strong computing resources and limited memory capacity, while a memory node has limited computing power and large memory size. This asymmetric architecture is exploited in `dLSM` so that the compute and memory nodes hold different components of an LSM index. The compute node keeps the MemTables while the memory node stores the SSTables. Moreover, the compute node keeps the LSM-tree metadata, index blocks, and Bloom filters for the SSTables to improve read performance.

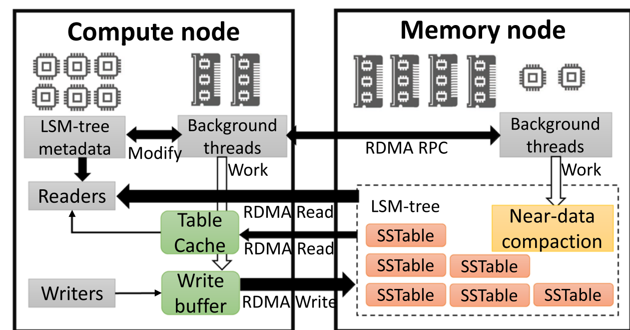


Fig. 2 Overall architecture of `dLSM`

Writes. `dLSM` supports concurrent writes and guarantees snapshot isolation with minimal software overhead to best unlock the potential of low-latency remote memory (Sec. 4). The MemTable is implemented as a lock-free skip list [43, 69]. When the MemTable is full, it is switched into an immutable MemTable ready to be flushed. Multiple background threads flush the immutable MemTable to remote memory using asynchronous I/O. When the flushing finishes, the background thread modifies the LSM-tree metadata in a copy-on-write manner to support snapshot isolation.

Compaction. To reduce data movement among compute and memory nodes, `dLSM` offloads the compaction process to the memory node. This is termed *near-data compaction* (Sec. 5). Basically, the compute node decides which SSTables to compact, and sends relevant metadata information to the memory node via an RPC. The memory node gets the input SSTables' metadata from the RPC to perform compaction. After compaction completes, the compute node receives a reply to modify the LSM-tree metadata accordingly. `dLSM` addresses a number of challenges related to near-data compaction to improve performance (Sec. 5).

Reads. The reader traverses the MemTable, immutable MemTable, and SSTables in the LSM-tree from the newest to the oldest according to the LSM-tree metadata. With snapshot isolation, a read refers to a proper version of the LSM-tree metadata before searching the tables. A read does not conflict with the background compaction or flushing processes. To accelerate reads, `dLSM` uses Bloom filters, and has a new index layout to directly locate a single key-value pair without fetching the whole block to take advantage of byte-addressability in the remote memory (Sec. 6).

4 Minimizing software overhead

We present one optimization to improve `dLSM`'s throughput by minimizing software overhead. For relatively slow

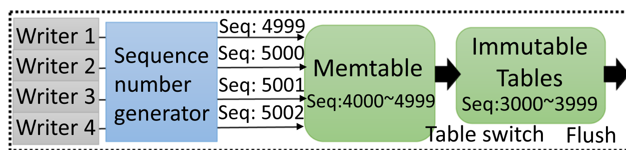


Fig. 3 Concurrent in-memory writes in dLSM

storage devices (e.g., SSDs or HDDs), software overhead is negligible. However, it can be a performance bottleneck for the ultra-fast networks such as RDMA. Significant software overhead is present in existing disk-based LSM-tree implementations [46]. This is due to maintaining concurrency control in the presence of concurrent writes to the MemTable. Snapshot Isolation [21] is a concurrency technique that provides consistent reads in the presence of writes and background compaction.

To support efficient concurrent writes to the MemTable, dLSM follows existing systems in using a lock-free skip list to minimize lock use. To support snapshot isolation, dLSM relies on an atomic sequence number generator implemented by `fetch_and_add` to assign a sequence number to each writer without locking (Fig. 3). However, correctly switching the MemTable to be immutable when multiple writers detect that it is full remains a challenge, especially if we want to minimize the lock synchronization to avoid its overhead.

A straightforward solution is to use double-checked locking to guarantee that only one writer switches the MemTable successfully. With double-checked locking, the lock is not acquired unless the writer finds that the current MemTable exceeds its size limit. However, this is problematic, because a newer MemTable cannot be guaranteed to contain the most updated version of a key. The reason is that assigning the sequence number and inserting the key-value pair to MemTable are not collectively atomic. To illustrate, suppose there are two writers w_1 and w_2 inserting two different values v_1 and v_2, respectively, for the same key k, and assume that w_1's sequence number is larger than w_2's (implying that v_1 is the newer version). Using the above approach, it is possible, for w_1 to insert v_1 into the old MemTable while w_2 inserts v_2 into the new MemTable. Then, it is problematic when a reader later searches for k. The result will be v_2 and not v_1 because the read process stops upon finding the first matched key.

The dLSM Approach. dLSM introduces a new approach to make MemTables immutable. To avoid the problem mentioned above, each MemTable in dLSM is assigned a predefined range of sequence numbers at its creation, e.g., 4000−4999 in Fig. 3. The new MemTable's range is a consecutive range after the current MemTable's range. When writers insert the key-value pair into the MemTable, they check the sequence number against the sequence number range of the current MemTable to decide if a MemTable switch is required. Figure 3 gives an example where the sequence number range of the current MemTable is 4000 − 5000. Four concurrent writers try to insert key-value pairs into the MemTable. Writers of 4999, 5000 insert directly into the current MemTable as the sequence number is within the range. The writers of 5001 and 5002 are outside the range of the current MemTable's sequence number range. Both try to switch the MemTable. Double-checked locking guarantees that only one writer can switch the MemTable. This solution predefines which MemTable each key-value pair belongs to so that a key-value pair with a new sequence number is guaranteed to be inserted into a new table. If the Memtable's sequence number range is sufficiently large, the writers will rarely have their sequence numbers outside the MemTable's range, so the lock is rarely used. Thus, synchronization overhead of in-memory writes is minimized.

Algorithm 1: In-memory write in dLSM.

```
1  Input   : Write operation w.
2  Atomically get a sequence number n from sequence number
   generator for w;
3  if n ≤ MemTable's largest seq. number then
4      if n ≥ MemTable's smallest seq. number then
5          InsertToMemTable(w);
6      else
7          Find the correct table in immutable table list;
8          InsertToMemTable(w);
9  else
10     Acquire the mutex lock for MemTable switching;
11     if n ≤ MemTable's largest seq. number then
12         Switch the MemTable to be immutable;
13         Create a new MemTable;
14         InsertToMemTable(w);
15     else
16         if n ≥ MemTable's smallest seq. number then
17             InsertToMemTable(w);
18         else
19             Find the correct table in the immutable table list;
20             InsertToMemTable(w);
```

Algorithm 1 lists the write process in dLSM. Before writing to a MemTable, the writer checks if this key-value's sequence number is within the range of the current MemTable (Lines 2 and 3). If so, the writer writes directly to the MemTable without a lock (Line 4). Due to CPU scheduling, the sequence number of a writer may also belong to an immutable table. Then, the writer iterates over the immutable table list to find which immutable MemTable to insert the key-value pair into (Lines 6 and 7). If the sequence number is larger than the upper-bound of the current MemTable, then it is made immutable. The writer obtains the lock and switches the MemTable to be immutable (Line 9). Multiple writers may detect that the MemTable is full, but only the first writer that successfully acquires the lock switches the MemTable, and creates a new MemTable (Lines 11-13). If any other writer's sequence number is within the range of

the newly created MemTable, it directly inserts the key-value pair (Line 16). Else, it finds the right immutable MemTable to insert the key-value pair (Lines 18 and 19).

5 Near-data compaction

Near-data compaction leverages the compute capability of the remote memory node. It can improve performance by reducing data movement.

Main Idea. Near-data computing [8, 22, 25, 49, 87] moves execution closer to data to reduce data movement. dLSM adopts this strategy by offloading some of the LSM-tree compaction process to the remote memory node. The compaction phase, if executed on the compute node, would read many SSTables from the memory node, and then would write back the merged SSTable to the memory node. This would lead to significant data transfer over the network. However, pushing down all the compaction process to the remote memory is sub-optimal because the remote computing power may be limited. Furthermore, there is currently no consensus regarding the amount of computing power available in the disaggregated memory. The pushdown strategy should be automatically adjusted according to the hardware configuration (e.g., the number of remote CPU cores) and real-time CPU usage in the memory node.

Near-data computing has been around for decades in several contexts, e.g., database machines [35, 36], object databases [44, 77], active disks [52, 71], Smart SSDs [38, 50, 67, 78], and storage disaggregation [8, 22, 25, 49]. We investigate its use in realizing an LSM-based index for disaggregated memory. The novelty in dLSM is in applying near-data computing to this new environment and in handling all implementation challenges. dLSM differs from other works on LSM-tree remote compaction for storage disaggregation, e.g., Rockset [8], Nova-LSM [49], and Hailstorm [22]. Rockset [8] offloads the compaction, but fetches the SSTables over the network as the data is stored separately in S3.

Challenges. Efficiently offloading compaction to remote memory in challenging. In dLSM, we address the following questions: (1) How to place LSM-tree's metadata to ensure efficient near-data compaction while facilitating query processing? (Sec. 5.1), and (2) How to perform garbage collection in the context of near-data compaction? (Sec. 5.2) (3) How to design an adaptive push-down strategy for compaction to achieve high performance regardless of hardware configuration (e.g., the number of cores in remote memory)? (Sec. 5.3)

5.1 Placing LSM-tree metadata

An important issue for near-data compaction is to decide where to store the LSM-tree metadata. Metadata is critical to LSM reads, writes, and compaction as it maintains SSTable, e.g., the position and structures of the table contents in remote memory. The metadata can be placed in remote memory to easily perform the compaction task and efficiently access the metadata for compaction. However, this significantly hurts query performance as readers need to access the metadata from remote memory (e.g., using RDMA read) to find the SSTables. Thus, query latency will be high.

dLSM maintains the LSM-tree metadata in the compute node.[2] The compute node decides which SSTables to compact, and triggers near-data compaction through a customized RPC (Sec. 10.4). When the memory node receives the RPC, it compacts these SSTables. dLSM applies the compaction strategy as in Sec. 2.3. To reduce write-stalls from Level 0, it uses multiple background compaction threads to divide a large compaction task into multiple parallel sub-compaction tasks. When compaction finishes, the memory node sends an acknowledgement to the compute node that in turn issues another RPC to copy the metadata of the compacted SSTables to the desired working space.

dLSM has several optimizations to avoid network round trips and memory copy by allowing the memory node to allocate memory locally. Memory in the memory node is divided into two disjoint memory regions where one region is controlled (and allocated) by the compute node for regular MemTable flushing while the other memory region is controlled by the memory node itself for near-data compaction. Then, the memory node can perform compaction in its private memory space. After compaction finishes, the memory node sends the metadata of the new SSTables to the compute node in an RPC reply. The compute node modifies the LSM-tree metadata according to the reply and makes the new SSTables visible to readers. From our experiments, on average, the SSTable metadata is modified every 0.02s. Thus, metadata updates are synchronized by a mutex lock.

5.2 Garbage collection

As dLSM supports multiple versions, it becomes necessary to perform garbage collection on obsolete data. Even though SSTable compaction logically clears out old data versions, the remote memory occupied by these obsolete SSTables cannot be immediately reclaimed. This is because there may still be readers referring to these old data versions. As a result, garbage collection of remote memory remains a challenging task. The following issues need to be addressed:

[2] Note that it is possible to store a copy of LSM-tree metadata in the memory node but the key point is that the compute node initiates and controls the timing of compaction and the memory node performs the task.

- How to garbage collect the obsoleted SSTables in the remote memory efficiently and correctly when both compute and memory nodes are involved in remote memory allocation (due to near-data compaction)?
- When to garbage collect the SSTables?

To address the first issue, in `dLSM`, memory allocated for near-data compaction is recycled by the memory node while memory allocated for flushing is recycled by the compute node. In `dLSM`, the SSTable metadata contains the node ID denoting its origin. During garbage collection, a compute node's garbage collector identifies from the node ID where the table is originally created. If it is local, the garbage collector recycles its remote memory by the local allocator. Otherwise, an RPC (See Sect. 10.4) is triggered to recycle the tables' memory remotely. To reduce communication, multiple garbage collection tasks are grouped locally first and are sent in batch to the remote memory.

To address the second issue, during reads, a `dLSM`'s reader pins a snapshot of LSM-tree metadata before searching the SSTables. The LSM-tree metadata snapshot further pins all the SSTables that it contains. When a MemTable flush or SSTable compaction finishes, the LSM-tree metadata is modified in a copy-on-write manner to create multiple LSM-tree metadata snapshots. When reading finishes, a reader unpins the LSM-tree metadata snapshot and unpins relevant SSTables that are garbage collected automatically.

5.3 Adaptive near-data compaction

Another significant design decision for near-data compaction is to determine which compaction tasks should be delegated to the remote memory node and which compaction tasks should be executed at the compute node. Blindly pushing down all compaction tasks may not achieve the best performance because the computing power in the remote memory node may be limited. Thus, an optimal compaction strategy would dynamically consider the hardware configuration (in the remote memory node) and the workload in order to adjust its compaction strategy accordingly.

Main Idea. The compute node regularly monitors CPU utilization on both sides and dynamically determines where the compaction should be performed (see Algorithm 2). The algorithm gives priority to Level-0 compaction (Line 2) due to its significant impact on performance. For Level-0 compaction, we dynamically estimate the execution time using the collected information and then allocate the compaction to the side with the smaller estimated time. For compactions at other levels, we assess the availability of cores in the remote memory. If there are available cores, `dLSM` delegates the compaction to the remote memory; otherwise, the compaction is performed at the compute node.

Challenges. There are a few challenges to realize the adaptive compaction strategy. (1) How to collect the essential information for decision-making? (Sec. 5.3.1) (2) How to prioritize Level-0 compaction and dynamically decide where the compaction tasks should be placed? (Sec. 5.3.2) (3) How to estimate the execution time with the given information for Level-0 compaction? (Sec. 5.3.3)

Algorithm 2: Adaptive near-data compaction strategy

Input : Compaction task c.
Output: Decision for compaction placement.

```
1  Boolean NeedNearDataCompaction;
2  if c is in Level 0 then
3      if isLongTask(c) then
4          Estimate the execution time on both sides via the number
           of all cores;
5      else
6          Estimate the execution time on both sides via the number
           of available cores;
7      if the estimated execution time on the memory node is less
       than that on the compute node then
           /* push down the compaction */
8          NeedNearDataCompaction = true;
9      else
           /* perform the compaction locally */
10         NeedNearDataCompaction = false;
11 else
12     repeat
13         Check the CPU utilization in both sides;
14     until memory_node.CPU_utilization < θ or
       compute_node.CPU_utilization < θ;
15     if memory_node.CPU_utilization < θ then
16         NeedNearDataCompaction = true;
17     else
18         NeedNearDataCompaction = false;
19 Return NeedNearDataCompaction;
```

5.3.1 Collecting essential information for compaction

The number of CPU cores as well as the dynamic CPU utilization for both sides are essential information for the adaptive compaction policy. When `dLSM` is started, the compute node gathers the CPU information from both sides. Then, the memory node sends heartbeat RPCs (Sec. 10.4) to the compute node every t milliseconds. The remote memory attaches its current CPU utilization within each heartbeat. Thus, the remote CPU utilization is refreshed every t milliseconds. t should be far smaller than the time elapsed of a compaction task,[3] so the compaction decision can respond to workload changes rapidly.

[3] By our observation, a compaction lasts for at least 100 milliseconds. Thus, we set t to 25ms by default.

5.3.2 Algorithm for adaptive compaction

A straightforward algorithm for adaptive compaction is to assign a compaction task to remote memory as long as there is available computing power on the remote side. If the compute node detects that the remote memory is about to saturate all its computing resources, then the compute node schedules compaction to local compute-side threads. However, this approach is sub-optimal for two reasons. First, it treats Level-0 compaction the same as those of the other levels. This is problematic due to the fact that Level-0 compaction takes longer time than the compaction of other levels because all the SSTables in Level-0 are compacted together. In addition, as noted earlier, Level-0 compaction has a more significant impact on performance since only the over-sizing in Level 0 causes write stall. Therefore, Level-0 compaction should have higher scheduling priority. Second, this approach ignores the fact that it is not ideal to always push down Level-0 compaction to the remote memory node. To accelerate the execution of Level-0 compaction, `dLSM` divides it into multiple parallel tasks following RocksDB. However, if compaction is always pushed down, a weak computing power of the remote side may impede parallelism. For instance, if the remote memory only has one core, then compacting on the compute node with multiple threads performs better. Thus, it is necessary for the compute node to estimate the execution time on both sides before scheduling Level-0 compaction.

`dLSM` schedules Level-0 compaction with higher priority, as in Algorithm 2, by distinguishing it from compaction at the other levels (Line 2). It uses different strategies to determine where the compaction should be performed. For Level-0 compaction, it estimates the execution time coarsely for both sides (Sect. 5.3.3). Based on the estimated time for both sides, it determines where to place Level-0 compaction.

For the other levels, it is beneficial to push down the compaction when there are available CPU cores on the remote side. The reason is that the compaction tasks at these levels only contain a few tables and are only handled by a single thread, so the execution time is always shorter in the remote memory if there are remote CPU cores available. As a result, the compute node checks for CPU utilization on the remote memory every time it schedules compaction outside Level-0. If CPU utilization is smaller than a defined threshold θ, then compaction is executed there.[4] Otherwise, it is performed on the compute node until the local compute power is exhausted.

5.3.3 Execution time estimation for compaction

We decide on the location of Level-0 compaction by roughly estimating the execution time on both the compute and memory nodes. Theoretically, the execution time of compaction is proportional to the total size of SSTables participating in the compaction divided by the parallelism the compaction can achieve during execution. The total size of the SSTables can be roughly determined by the number of SSTables to be compacted, because most of the SSTables are of similar sizes (64MB). Let T_i^K be the estimated execution time for compacting i SSTables, where K can be either compute side (C) or memory side (M). T_i^K can be calculated as follows:

$$T_i^K = \frac{i \times t_{avg}^K}{min((1 - \mu_K) \times c_K, \mathcal{P})} \tag{1}$$

where $\mathcal{P}$ is the maximum parallelism the compaction task can support regardless of the given number of cores,[5] μ_K represents the most-recent known CPU utilization and c_K represents the number of the cores on side K. We use $(1 - \mu_K) \times c_K$ to roughly quantify the number of unoccupied cores at the moment. It also leverages t_{avg}^K; the average compaction execution time per unit SSTable given a single thread on side K. We find that t_{avg}^M is about half of t_{avg}^C, due to the network I/O introduced by the execution on the compute node.

The above estimation function is accurate for short compaction tasks (smaller than 1 μs) but not for long ones. The reason is that $(1 - \mu_K) \times c_K$ can only represent the available core *temporarily*. As current compaction tasks get finished, the available number of CPU cores for Level-0 compaction increases gradually and finally includes the total number of CPU cores. Thus, Algorithm 2 prioritizes Level-0 compaction while the other compactions will be scheduled such that Level-0 compaction is not overloaded. Therefore, if compaction takes too long, we replace the currently available number of cores $(1 - \mu_K) \times c_K$ with the total number of cores c_K, which leads to the following formula:

$$T_i^K = \frac{i \times t_{avg}^K}{\min(c_K, \mathcal{P})} \tag{2}$$

Therefore, in Algorithm 2, the compute node determines whether the compaction task is a long-time compaction before deciding on which side the compaction should be placed. Note that whether a compaction is long or short is relative to the current compaction task being executed. Thus, we classify a Level-0 compaction as long when the estimated execution time is longer than the execution time of the other levels, which only contains few SSTables. Since we do not know in advance which side the compaction will be assigned, we select Formula 2 to the memory side to roughly estimate the execution time. Since the compactions at non-zero levels are executed by a single thread, the execution time is equal

[4] By default, θ equals to the CPU utilization when there is only one core available.

[5] $\mathcal{P}$ equals the number of files in the second level of the compaction input.

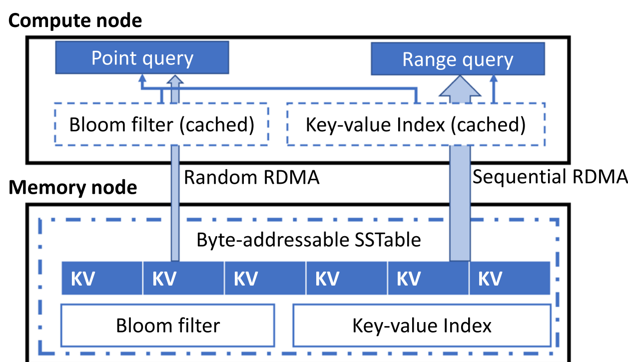


Fig. 4 Byte-addressable SSTable layout in dLSM

to $i \times t^K_{avg}$. Overall, we use the formula below to classify whether a compaction is long or not:

$$\frac{i \times t^K_{avg}}{\min(c_M, \mathcal{P})} < s \times t^K_{avg} \quad (3)$$

where the left-hand side is the time estimate following Formula 2 on the memory node, while the right-hand side is the time estimate for compacting s SSTables. Empirically, dLSM performs best when s is around 5.

6 Optimizing for byte-addressability

SSTables can be optimized for byte-addressability. LSM indexes, e.g., RocksDB, use block-based SSTables as the table format because they are optimized for block-based disk storage, e.g., HDDs and SSDs. However, for memory disaggregation, remote memory is byte-addressable. Existing designs become sub-optimal as fetching a single key-value pair still requires accessing a whole block causing read amplification. Although there are other data formats for SSTables, e.g., Cuckoo Hashing table format [16] or PlainTable [39], they fall short, e.g., Cuckoo Hashing does not support range queries efficiently; PlainTable relies on `mmap` that cannot be efficiently applied to remote memory settings.

Next, we introduce an optimized SSTable design for disaggregated memory to leverage byte-addressability in Sec. 6.1, and further improve it in Sec. 6.2.

6.1 Byte-addressable SSTable layout

Data layout. dLSM drops the notion of "blocks" to directly access a single key-value pair (Fig. 4). This improves read performance by reducing read amplification as a single key-value pair can be directly fetched without fetching a whole block. This design can improve write performance by eliminating extra memory copies as it is not necessary to wrap key-value pairs into blocks anymore. Thus, building an SSTable is accelerated as key-value pairs are directly serialized to the target buffer without waiting to form blocks. To efficiently support range queries, key-value pairs for an SSTable are sorted and stored in contiguous memory regions.

Index layout. To benefit from byte-addressability, the index needs to directly address every key-value pair. Note that the index mentioned in Section VI indexes the blocks inside the SSTables. Key-value pairs are variable-length. Thus, an index entry contains the key, offset, and length. dLSM uses binary search to answer point and range queries. To avoid network round trips during query processing, the compute node caches the index. Index size is expected to fit in a compute node's local memory as it only stores keys (not values). If a compute node has limited memory, dLSM utilizes the optimization in Sect. 6.2 to switch back some cold Byte-addressable SSTables to block-based SSTables.

Supporting point and range queries. Refer to Fig. 4. For a point query, the compute node uses a Bloom filter to check if the target key is in this table, and if so, the reader uses the index to locate the address of the target key-value pair. Then, the reader issues a network read to fetch the single key-value pair from remote memory. In contrast, for range queries, dLSM prefetches large chunks of key-value pairs by sequential I/O. Specifically, when handling a range query, the compute node creates an iterator with sub-iterators across all levels. The sub-iterators locate the first keys in the range by the LSM-tree meta-data and the SSTable's index. Then, the sub-iterators prefetch the data chunks in the SSTable. The outer iterator scans the next key-value pairs of all sub-iterators until the end of range.

6.2 Optimization for limited cache space

A limitation of the byte-addressable SSTables is the consumption of substantial memory in compute nodes due to caching of SSTable index blocks, especially when the size of keys and values are large. The growth of data volume, combined with the limited local cache size at compute nodes, can result in frequent cache misses of index blocks, leading to an increase in network accesses. Additionally, the byte-addressable index blocks are larger than the traditional block-based index blocks, exacerbating the performance degradation. Next, we describe how dLSM addresses this issue.

Main idea. When the local cache reaches its maximum size, it becomes necessary to convert some of the byte-addressable SSTables to block-based SSTables to achieve a balance between minimizing cache misses and maximizing performance gains from the byte-addressable SSTables. Moreover, when the compute node converts byte-addressable SSTables to block-based SSTables, it is beneficial to give

Algorithm 3: Decision module for compaction outputs' table type

Input : A compaction c, remaining cache space $\mathcal{S}$.
Output: The type T of all new-generated SSTables.

```
1  Let S be the remaining cache space.;
2  c.TableType is table type of c's output SSTables;
3  L_c is the level of compaction c;
4  f^c ← (L_max − L_c)/L_max;
5  if S × f^c < Δ then
       /* generate block-based SSTable with
          probability p */
6      p = (Δ − S × f^c)/Δ;
7      a ← A ~ U(0, 1) ;
8      if a ≤ p then
9          c.TableType = block_based;
10     else
11         c.TableType = byte_addressable;
12 else
13     c.TableType = byte_addressable;
14 Return c.TableType;
```

priority to the bottom-levels because those data are not frequently accessed.[6]

Challenges. There are two challenges to realize the solution described above. (1) How to dynamically adjust the proportion of byte-addressable SSTables? The goal is to minimize the impact of the SSTable conversion on performance. (Sec. 6.2.1) (2) How to prioritize the bottom-levels when converting the tables from byte-addressable SSTables to block-based SSTables? (Sec. 6.2.2)

6.2.1 SSTable type decision during compaction

For the first challenge, observe that the increase in index size in the cache always comes from the new data loaded into the system that may trigger frequent compactions. Thus, we piggyback SSTable type conversion to those triggered compactions to save on computing resources. `dLSM` employs a decision module on the compaction scheduler of compute nodes to determine dynamically the type of the newly generated SSTables.

During the startup phase, all generated SSTables are set to be byte-addressable by default as the local memory usage is low. We introduce a threshold Δ[7] for the available space that is predefined to reserve enough space to ensure that SSTable conversions take effect before the cache reaches its limit. As the local cache size of index blocks approaches its limit, the compaction scheduler prefers to generate block-based SSTables to decrease memory usage. Conversely, if the local cache size falls and the idle space exceeds the threshold, compactions continue to generate byte-addressable SSTables to enhance performance.

`dLSM` regularly removes the entries of outdated SSTables, i.e., those that have been compacted into the next level, and immediately caches all index blocks of the newly created SSTables in local memory. Therefore, the decision module can acquire more accurate real-time cache information.

6.2.2 Level priority in SSTables' type conversion

For the second challenge, we prefer the bottom-levels when converting the byte-addressable SSTables to block-based ones. To achieve that, we scale the remaining cache space $\mathcal{S}$ by a level prioritized factor f_c to get a weighted remaining cache space $\mathcal{S}_{weighted}$. f_c should be larger at the top-levels and smaller at the bottom-levels. Thus, the compactions at the bottom-levels are more likely to generate block-based tables due to the smaller weighted remaining cache space. We define f_c as:

$$f_c = \frac{L_{max} - L_c}{L_{max}} \tag{4}$$

where L_{max} represents the max level in the LSM-tree set by configuration, and L_c represents the upper level where the input of compaction c is located. Thus, the weighted remaining cache space is computed as:

$$\mathcal{S}_{weighted} = \mathcal{S} \times f_c. \tag{5}$$

We can compare the weighted size of remaining cache space $\mathcal{S}_{weighted}$ against the predefined threshold Δ to decide whether there is a need for SSTable conversion. Therefore, cases where byte-addressable SSTables are not preferred are those where

$$\mathcal{S}_{weighted} < \Delta. \tag{6}$$

If Formula 6 holds, then the available space is approaching its limit. A straightforward followup step can be to generate block-based SSTables only when the formula holds true, and generate byte-addressable SSTables otherwise. However, if the compaction workload is heavy, this approach can convert more tables than necessary, since there is a lag before the compaction result takes effect. To avoid converting many SSTables at one time, we introduce a probability factor p for generating the block-based SSTables. The probability approaches 1 as the remaining cache space approaches zero. The less the cache space that is left, the more eager we are to perform SSTable type conversions. On the contrary, if $\mathcal{S}_{weighted}$ is close to Δ, then there is still sufficient cache, so we can safely slow down the conversion. Therefore, the

[6] Note that we view Level 0 as the top level, and Level n as the bottom level.

[7] We set Δ to 256MB by default.

probability factor is devised as below:

$$p = \frac{(\Delta - \mathcal{S}_{weighted})}{\Delta} \tag{7}$$

The optimized algorithm is listed in Algorithm 3. When the weighted remaining cache space is larger than Δ, the compaction generates byte-addressable SSTables only. If the weighted remaining space is smaller than the threshold described in Formula 6, the compute node has only a probability p to generate the block-based SSTables.

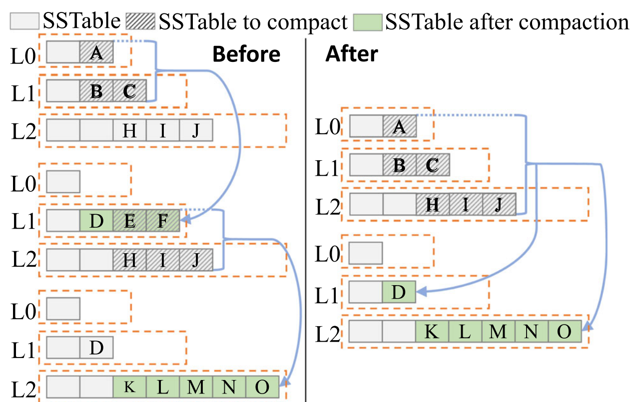


Fig. 5 Combinations of compaction results

7 Optimizing for mixed R/W workloads

Observe that dLSM with logs disabled achieves moderate performance on the mixed workloads with reads and writes when compared to the performance of the 100

To address this challenge, we follow Nova-LSM's approach [49] to divide the entire key range into λ ($\lambda \geq 1$) shards based on the range information and build a separate LSM-tree per shard. This adds more parallelism to Level 0's compaction and also reduces the number of SSTables that a reader needs to traverse. We evaluate the impact of λ in Fig. 11. The results show that the performance drop of mixed workload can be mitigated by sharding the data within the compute node. The higher the λ is, the smaller the performance drop will be.

8 Persistence

dLSM is an LSM index targeted for use in main-memory databases (e.g., VoltDB [11]) with memory disaggregation. Thus, we do not provide the whole procedure for data recovery in the index part (following the prior index works including Sherman [80], an optimized B-tree index for disaggregated memory, and other indexing works, e.g., [37, 85, 97, 98]) because data persistence is achieved at the database layer. To illustrate, many modern main-memory databases, e.g., VoltDB [11], BatchDB [61], and PACMAN [84], do not use traditional redo/undo logging for persistence in order to achieve fast performance. Instead, they use an alternative technique termed command log [62] that logs the high-level operations, e.g., SQL and stored procedures, in contrast to logging the physical updates into the index. The index is periodically flushed to disk. If the system crashes, the logged operations are re-executed from the last transaction-consistent checkpoint. Thus, as long as the index provides a transaction-consistent checkpoint, the overall database system can be recovered by the command log. How to implement such a transaction-consistent checkpoint is the research question in this section. To the best of our knowledge, there is no prior work on how to implement the transaction-consistent checkpoint for an in-memory LSM-tree.

Main Idea. dLSM offers an LSM-based index that natively provides a transaction-consistent checkpoint. The checkpointing in dLSM is to regularly copy the SSTables and their metadata to durable storage. Compared to checkpointing in in-memory database systems, checkpointing in-memory LSM-trees does not require a complete scan and flush of all SSTables every time; it only requires the changes since the last checkpoint to be synchronized to durable storage. Therefore, when an SSTable compaction or MemTable flushing is completed in dLSM, the background threads on the remote memory ensure that any additions, deletions, and metadata changes to the SSTables are replayed in durable storage.

Furthermore, to make the checkpoint transaction-consistent, a transaction writes all its updates into a single MemTable that becomes a single SSTable after flushing. As long as the checkpoint is prepared in the granularity of SSTables, it is impossible for the checkpoint to contain only part of a transaction's updates. Thus, the checkpoint is guaranteed to be transaction-consistent. Note that the new updates are stored in local memory while the checkpointing process performs in remote memory. This isolation enables dLSM to offer a non-blocking transaction-consistent checkpoint automatically, which is hard to achieve for other traditional index types.

Challenge. The design outlined above faces a challenge that the checkpoint in the durable storage may significantly lag behind the LSM-tree in the disaggregated memory under a write-intensive workload. Consequently, this issue results in unbounded usage of remote memory since the SSTables cannot be garbage collected until they are persisted in durable storage. The reasons for falling behind stem from two aspects. (1) MemTable flushing and SSTable compaction result in the addition and deletion of a very high number of SSTables for write-intensive workloads. (2) The I/O bandwidth between the compute node and the disaggregated

memory is much larger than that between the disaggregated memory and the durable storage, which further slows down checkpointing.

To deal with this challenge, we merge the SSTable compaction results since the last checkpoint to reduce the amount of I/O that the checkpointing requires. Specifically, we observe that when replaying the compaction results, some output SSTables in one compaction are the input SSTables in another compaction. These intermediate SSTables can be omitted in the checkpointing process. For example, in Fig. 5 left-hand side, there are two successive compactions $(A, B, C \rightarrow D, E, F)$, and then $(E, F, H, I, J) \rightarrow (K, L, M, N, O)$. The compaction results together are equivalent to the compaction result on the right-hand side $(A, B, C, H, I, J) \rightarrow (D, K, L, M, N, O)$. The combined result omits the intermediate results E and F, saving the bandwidth for checkpointing.

9 Multi-compute and multi-memory nodes

To serve massive amounts of data and improve scalability, dLSM can be distributed and deployed across multiple compute and memory nodes. dLSM's scale-out consists of two parts: Scaling out for compute nodes and for memory nodes. For compute nodes, the key question is how to guarantee cache coherence across multiple compute nodes. Existing solutions include single-writer-multiple-readers [27, 76], software-level cache coherence protocol [68, 81] or range sharding across the compute nodes [49]. For the first solution, the update throughput is bounded by the single writer node and other reader nodes cannot see the updates buffered in the MemTables immediately. The second solution can bring in huge overhead for the cache coherence protocol. In dLSM, we follow the sharding solution, which is popular among existing LSM-based systems [2, 3, 8, 13, 48, 74].

To scale out memory nodes, a key question is how to distribute data among memory nodes. A finer granularity (by uniformly distributing the data chunks for every SSTable) benefits load balancing, but it forces near-data compaction to have high network I/O. To benefit from near-data compaction, we distribute the data in the granularity of small shards so that all the data in the same range are stored in the same node.

Let c, m, and λ be the number of compute nodes, memory nodes, and shards within a compute node, respectively. From Fig. 6, dLSM initially assigns the $c \cdot \lambda$ shards evenly among the m memory nodes in round-robin fashion to achieve best load balancing. For each shard, dLSM builds an individual LSM-tree that is stored in a single memory node with MemTables being cached in a single compute node. In addition, the shards can move their counterparts among the compute nodes and memory nodes later to balance the workload. One advantage of this design is that there is no synchronization overhead for single-shard key-value accesses but at the expense of distributed transactions for cross-shard accesses. Sec. 11.3.10 evaluates this multi-node design for dLSM.

10 Instantiating dLSM over RDMA

RDMA-enabled disaggregated memory is well-studied [42, 80, 90, 98]. Thus, we use it to instantiate and test dLSM. However, several of the designs presented in this work can also be applied to CXL-based memory disaggregation, e.g., the designs discussed in Sects. 4, 6, 7, 8, 9.

10.1 dLSM Codebase

We build dLSM from scratch but reuse certain data structures and algorithms (e.g., concurrent skip list, Bloom filters, immutable MemTables, SSTable compaction) from LevelDB and RocksDB. dLSM contains approximately 54,400 lines of C++ code in which 4,500 lines of code are from RocksDB, 24,300 lines of code are from LevelDB.

10.2 RDMA manager

Efficiently utilizing RDMA primitives plays an important role in designing the LSM index over disaggregated memory. In dLSM, the RDMA manager is the intermediate implementation connecting dLSM's codebase to RDMA verbs.

10.2.1 Thread local queue pair for concurrent accesses

To support highly concurrent accesses, a unique challenge in RDMA programming lies in how to organize multiple queue pairs in the system. Since a single queue pair can have at most one completion queue, all the threads accessing the same queue pair will have their completion notifications mixed up. Existing completion notification mechanisms, such as using *wr_id*, introduce synchronization overhead when distinguishing their own completion from the queue. To remove the synchronization overhead, in dLSM, every thread that needs to perform one-sided RDMA has a local queue pair in the RDMA manager. When transmitting the data, the thread uses its thread-local queue pair if it exists, or creates one otherwise. This way, threads do not collide when performing one-sided RDMA, and no synchronization is needed to access the queue pairs.

10.2.2 Allocating RDMA memory region

For each RDMA operation, both the source and destination memory buffers need to be registered in the NIC through *ibv_reg_mr*. The registration pins the memory, and prevents

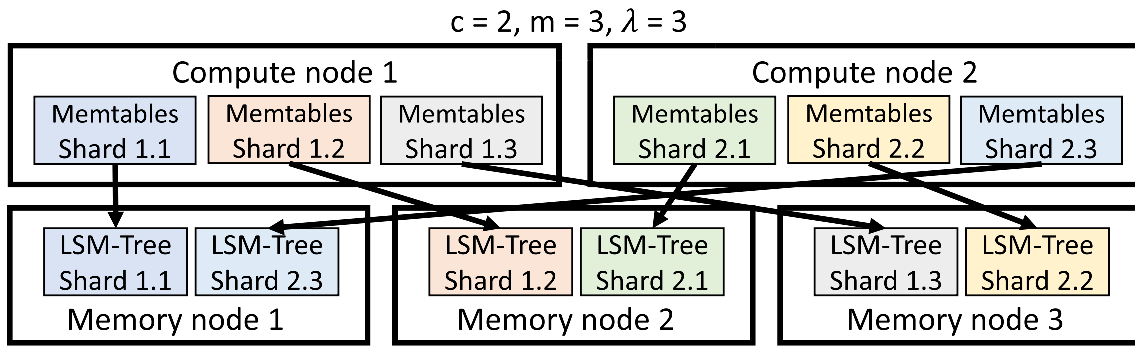


Fig. 6 Supporting multi-compute and multi-memory nodes

it from being swapped. Performing frequent RDMA registrations can introduce non-negligible overhead. It is important to minimize this overhead.

Many systems (e.g., [40, 63]) tend to pre-register large memory regions, and then reallocate memory in the user space. dLSM follows this approach. dLSM allocates the Memory Region in a memory pool method, because the memory pool allocator has less memory fragmentation and maintenance overhead. When initializing dLSM, the memory node preregisters multiple 1GB-sized memory regions. During execution, various worker threads request memory from different memory pools. If the memory pools are out of memory, they request a 1GB memory region through the RDMA manager, and divide it into chunks. When a thread allocates a memory chunk, the pool allocator returns *ibv_mr* that contains not only the pointer but also the local/remote access key for this chunk. In dLSM, to reduce RDMA round trips during MemTable flushing, the compute node prefetches 1GB-sized memory regions from remote memory and manages it locally by the memory pool allocator. Thus, the memory node preserves some of the 1GB-sized memory regions for the compute node. The compute node issues an RDMA RPC to get the registered remote memory regions from the memory node. Then, the compute node can allocate the remote memory by its own memory pool allocator.

10.3 Asynchronous I/O for MemTable flushing

Local memory in compute nodes is limited. Thus, MemTables are flushed periodically to remote memory. When flushing cannot catch up with in-memory writes, the writers slow down their write rate or wait until the background flush completes. Thus, improving MemTable flush speed is essential.

Main Idea. The RDMA primitives allow us to issue RDMA work request and check the work request's completion separately. We redesign the MemTable flushing process to take advantage of this asynchronous feature. In dLSM,

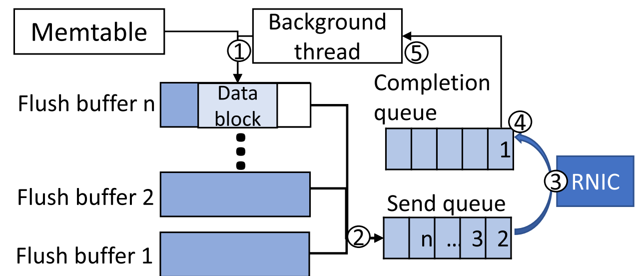


Fig. 7 Efficient flushing in dLSM

background workers do not wait for I/O completion and continue to serialize the data over new buffers instead.

Challenges. Utilizing asynchronous I/O does not simply replace the I/O interface. One issue is how to seamlessly integrate asynchronous I/O into the flushing process. Another issue is buffer recycling. When the data in a buffer is successfully transmitted to the remote node using RDMA write, we call the buffer a *finished buffer*. The flushing thread needs to recycle the finished buffers to reduce memory footprint. But asynchronous I/O for RDMA does not specify which buffer the finished RDMA operation refers to. Thus, it needs a new way to recycle the finished buffers.

The dLSM Approach. Fig. 7 illustrates dLSM's design. To address the first challenge, dLSM prepares multiple buffers for the MemTable flushing thread. Asynchronous flushing in dLSM proceeds as follows: (1) The thread directly serializes the data into the write buffer without any data copy. (2) When the buffer is full, the asynchronous write work request is submitted, and the thread continues to serialize the data into the next buffer without blocking. (3) The write request is processed on the RDMA Network Interface Card (NIC). Multiple work requests can be pending in the send queue. (4) The writer thread checks for work request completions every time it submits a new request. If it finds that a work request has been finished, it can reuse the old buffer. Otherwise, it allocates a new buffer for the next serialization and flushing task.

To handle the second challenge, dLSM leverages the FIFO feature of the RDMA work request queue. The pending flush buffers are organized as a linked-list-style queue that reflects the order of the issued work requests. The flushing thread maintains pointers to the linked list's head and tail. The head is the buffer that is about to finish data transmission, and the tail is the newest buffer that is still being serialized. When the background threads fill a buffer and issue an RDMA write, the thread may allocate a new buffer and append it to the tail of the linked list. When an I/O finishes in the completion queue, the linked list's head is popped and is recycled.

10.4 Customized RPC for near-data compaction

One way to implement RPC is to use two-sided RDMA send & receive. But this needs a centralized message dispatcher to forward the message to the target thread. This could create a potential bottleneck for RPC throughput with heavy traffic. dLSM utilizes one-sided RDMA write to issue a reply message so that the message can bypass the dispatcher. Below, we describe the general-purpose RPC in dLSM to handle simple operations such as queue pair establishing and remote memory allocation and the customized RPC for near-data compaction.

10.4.1 General-purpose RPC

The RPC for the general case. proceeds as below.

1. The requester allocates an RDMA-registered buffer to receive the reply message.
2. The address and the remote-access key (rkey) of the buffer are attached to the RPC request (realised by RDMA send & receive).
3. The responder processes the RPC, and returns the results by an RDMA write to the reply buffer.
4. The requester continuously polls a boolean flag at the end of the reply buffer. When the polling result is TRUE, the message is guaranteed to be ready. The polling thread can directly handle the reply message.

The reply message bypasses the dispatcher. Thus, dLSM can achieve higher RPC throughput. If necessary, dLSM can maintain multiple dispatchers and queue pairs.

10.4.2 Customized RPC for near-data compaction

The RPC of near-data compaction is more complex than that of the general case for the following reasons:

- Usually, near-data compaction takes longer time than the general case. Thus, the compute node needs a sleep and wake up mechanism through RDMA to avoid wasting the CPU resources on the compute node.
- Also, the size of an RPC argument (e.g., metadata of many SSTables to compact) for near-data compaction is usually bigger than the general case that requires specialized handling for high performance.

We introduce a customized RPC for near-data compaction.

Sleep & wake up through *RDMA write with immediate*. dLSM uses *RDMA write with immediate* to make the RPC dispatcher aware of the reply message and wake up the corresponding requester thread to handle the reply message. The requester attaches a 4-byte number as the unique ID in the near-data compaction RPC request, and goes to sleep. When the responder sends the reply, it sets the unique ID as the immediate in the RDMA write reply message. The unique ID helps the thread notifier identify which requester this reply message belongs to so the thread notifier can awaken the corresponding thread.

Large RPC argument through RDMA read. To support highly concurrent RPCs, the request message in a general-purpose RPC is usually small, e.g., 10 s of bytes, to reduce the overhead of message dispatching on the responder side. However, for near-data compaction, the argument size is larger, e.g., 100 s to 1000 s of bytes as the argument contains all necessary metadata for SSTables compaction.

dLSM does not attach the compaction metadata in the RPC request message. Instead, compaction metadata is serialized into an RDMA registered buffer. Then, the address, size, and remote key for the serialized buffer are attached to the RDMA request message. Upon having an RDMA request, the remote memory node gets the required compaction metadata from the compute node via an RDMA read. Upon metadata access, RPC workers in the thread pool can read the remote table content locally in the memory node. After compaction finishes, the memory node sends the metadata of the new SSTables to the compute node using an RDMA write.

11 Experiments

11.1 Baselines

Since there is no prior LSM index over disaggregated memory, we use the following baselines to evaluate dLSM:

Baseline #1: **RocksDB-RDMA (8KB).** This baseline is a port of an existing LSM-tree to the RDMA-extended remote memory. We choose RocksDB due to its wide adoption and its recognition as the prototypical LSM implementation. We refer to this baseline by "RocksDB-RDMA (8KB)". The block size is 8KB by default in RocksDB's benchmark. Write-ahead logging is disabled for fair comparison (see Sec. 8).

Baseline #2: **RocksDB-RDMA (2KB).** This is similar to Baseline #1 with one difference being a smaller block size to better leverage byte-addressability in the remote memory. We choose 2KB and term this baseline "RocksDB-RDMA (2KB)".

Baseline #3: **Memory-RocksDB-RDMA.** This baseline uses an even smaller block size that matches the size of a key-value pair. The SSTable index blocks are cached on the compute node for better performance. Prefetching is enabled to accelerate sequential reads during compaction. We term this baseline "memory-optimized RocksDB-RDMA" (or "Memory-RocksDB-RDMA", for short).

Baseline #4: **Nova-LSM** [49]. This baseline is an optimized LSM-tree for storage disaggregation (instead of memory disaggregation). We use Nova-LSM's available source code [49]. We configure the file system in Nova-LSM as `tmpfs`, a memory-oriented file system in Linux that stores all the files in main memory to avoid disk accesses. Besides that, write-ahead logging is also disabled.

Baseline #5: **Disaggregated B-tree (Sherman** [80]). The last baseline is a highly optimized B-tree termed Sherman [80] for the memory disaggregated architecture. We use Sherman's available source code [80].

11.2 Experimental setup

Platform. We conduct the experiments mostly on a platform consisting of two servers each having 8 NUMA nodes), but our experiments only use one NUMA node per server to eliminate the impact of NUMA remote memory access. Each NUMA node has a Xeon Platinum 8168 CPU (24 cores, 2.7GHz) and 384GB of DRAM. Two servers are connected by an RDMA-enabled Mellanox EDR Connectx-4 NIC with a bandwidth of 100Gb/s. Each node runs Ubuntu 18.04.5. For the scalability experiments that require multiple compute and memory nodes, we use CloudLab [41] (as in Sec. 11.3.10).

Datasets. We run the standard benchmark "db_bench" of RocksDB. We insert 100 million random key-value pairs in each system by default. The default key size is 20 bytes and value size is 400 bytes. The query set is 100 million key-value pairs.

Parameter Configurations. We set the same parameters of `dLSM` and other baseline solutions. The SSTable file size is 64MB and the Bloom filters' key size is 10 bits. For in-memory buffers, the MemTable size is 64MB. We set 12 and 4 background threads for compaction and flushing, respectively. The number of immutable tables is 16 to fully utilize the background flushing threads. To accelerate compaction further, subcompaction is enabled with 12 workers. These parameters are largely consistent with RocksDB's settings. By default, we disabled the WAL in all the systems. Unless otherwise stated, `dLSM` is configured to have 1 shard. For Nova-LSM [49], the subrange is 64 to maximize concurrency in background compaction. In Sherman [80], we follow the default block size (1KB) in the source code. To minimize RDMA remote accesses, we follow [80] to cache the internal nodes of the B-tree in local memory.

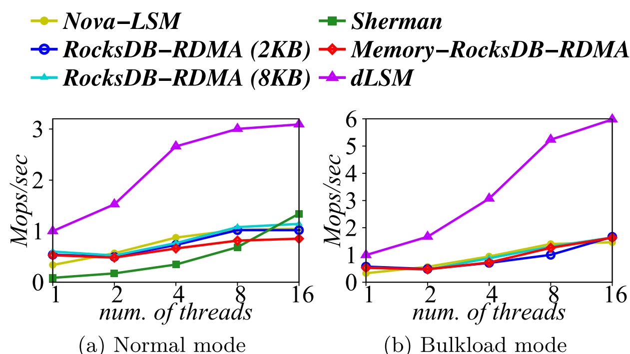


Fig. 8 Evaluating write performance

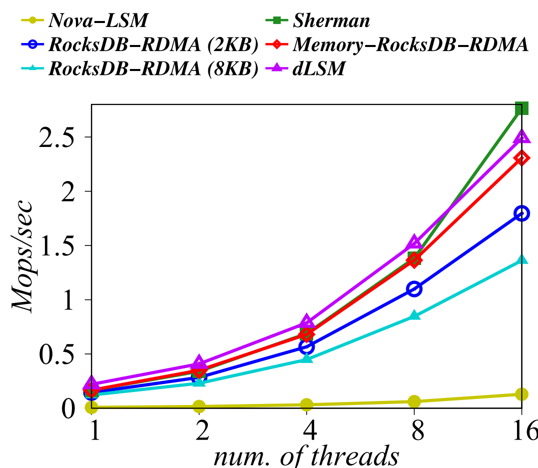


Fig. 9 Evaluating read performance

11.3 Results

11.3.1 Evaluating write performance

In this experiment, we evaluate the write performance of `dLSM` by comparing it with the five baseline solutions. We use the "randomfill" benchmark in RocksDB to generate 100 million random key-value pairs, and insert them into the different systems.

In this benchmark, an important parameter, termed *level0_stop_writes_trigger*, represents the maximum number of unsorted files (i.e., SSTables) in Level 0 in LSM-tree variants. When the number of files exceeds the predefined parameter, the writers stall to wait for the compaction of Level 0 to complete. Thus, the smaller the number of files, the more frequent the write stall becomes. In this exper-

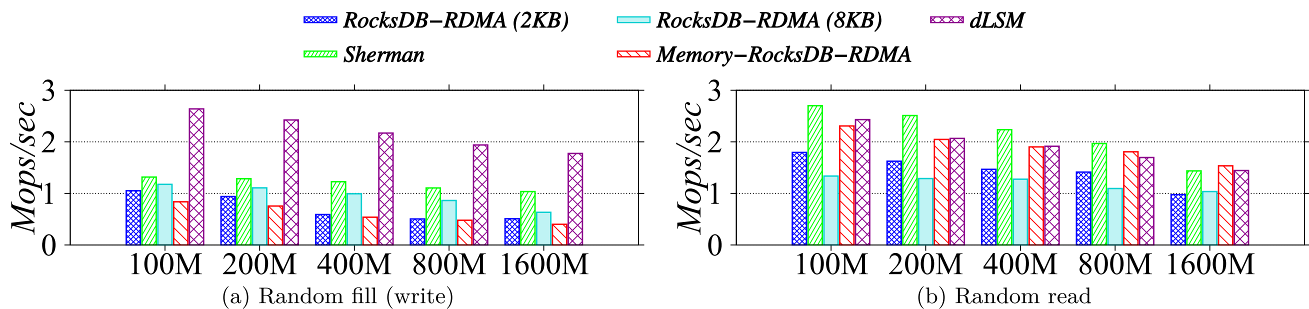


Fig. 10 Evaluating varied data sizes

iment, we evaluate dLSM in two modes with different *level0_stop_writes_trigger*:

- Normal mode: *level0_stop_writes_trigger* is 36 that is the default value in RocksDB.
- Bulkload mode: *level0_stop_writes_trigger* is infinity. In this case, there is no write stall triggered.

Figure 8(a) gives the write throughput with different numbers of threads under the "Normal mode." The random write throughput of dLSM can achieve as high as 2.6 million operations per second and dLSM outperforms the other baseline solutions significantly. Specifically, dLSM is 1.7× ∼ 3.5× faster than RocksDB-RDMA (8KB), 1.9× ∼ 3.6× faster than RocksDB-RDMA (2KB), 1.9× ∼ 4.0× faster than Memory-RocksDB-RDMA, 2.9× ∼ 3.0× faster than Nova-LSM, and 2.3× ∼ 11.6× faster than Sherman [80]. The performance advantage of dLSM demonstrates the effectiveness of dLSM's optimizations including reducing software overhead, near-data compaction, optimized RDMA communications, and byte-addressable index design. Observe that Sherman [80] only caches internal B-tree nodes in local memory and stores leaf nodes in remote memory. Thus, in Sherman every write operation needs to invoke an RDMA read operation to fetch the leaf page to local memory, modifies it, and writes back to the remote memory. This creates considerable performance overhead. dLSM improves write performance by buffering writes to local memory (MemTables) first and converts random writes to large sequential writes. We observe a bottleneck for LSM-based competitors when the number of threads increases. The reason is that background compaction at Level 0 cannot catch up with SSTable flushing from the compute node, making the front-end writers stall. The "Bulkload mode" removes this bottleneck by allowing infinite number of files in Level 0.

Figure 8(b) gives the write throughput for the "Bulkload mode" when varying the number of threads. In this mode, there are no write stalls resulting from background compaction. Every writer completes its task as soon as it inserts the key-value pair into the MemTable. Therefore, the system performance purely represents the in-memory write performance without write stalls. dLSM outperforms all competitor baselines and demonstrates the effectiveness of minimizing software overhead (as in Sec. 4). Specifically, dLSM is up to 5.2× faster than RocksDB-RDMA (8KB), 4.2× faster than RocksDB-RDMA (2KB), 4.0× faster than Memory-RocksDB-RDMA, and 4.0× faster than Nova-LSM. Note that Sherman [80] is not applicable to this mode.

11.3.2 Evaluating read performance

We evaluate the random read performance of dLSM against the baseline solutions. We run the "randomread" benchmark in RocksDB. We run 100 million random key-value queries and report the throughput. The generated keys have the same range as the keys in the "randomfill" benchmark. To remove the impact of overlapped SSTables, the benchmark starts after all the background compaction tasks finish. All the competitors utilize memory on the compute node to accelerate the read. dLSM, Memory-RocksDB-RDMA, RocksDB-RDMA (2KB), RocksDB-RDMA (8KB) take 2.62GB, 1.37GB, 0.33GB and 0.15GB, respectively, for SSTable index and Bloom filters. Sherman takes around 2.53 GB to cache the level-0 internal nodes. Fig. 9 gives the results for various numbers of threads. All the systems achieve good scalability as we increase the number of threads. dLSM outperforms all other LSM-tree solutions. The reason is that dLSM is optimized for byte-addressable remote memory (See Sec. 6). Memory-RocksDB-RDMA and RocksDB-RDMA (2KB) are faster than RocksDB-RDMA (8KB) due to the smaller block size that can reduce the amount of unnecessary data accessed. dLSM has higher read performance than Memory-RocksDB because it does not need to go through the block wrapper. Nova-LSM is slower due to the long read path and memory copy when fetching a key-value from the remote node's tmpfs.

When compared with Sherman [80], dLSM has slightly worse read performance (up to 8.5

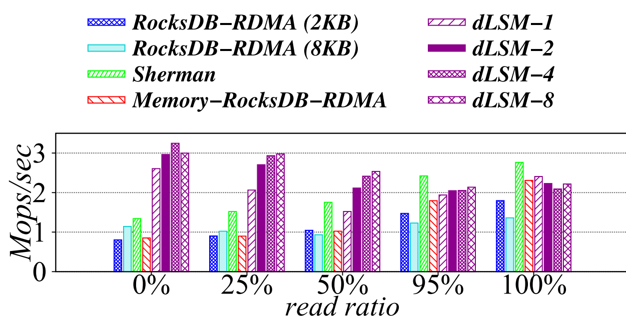


Fig. 11 Evaluating mixed read-write performance

11.3.3 Evaluating varied data sizes

In this experiment, we evaluate the performance of dLSM under various data sizes. We run "randomfill" and then "randomread" with increased number of key-value pairs and report the throughput. The inserted key range is also increased with the number of loaded key-value pairs. Figure 10 gives the performance results. There is decrease in performance for all the competitors when increasing data size. For LSM-based indexes, a larger data size increases the compaction workload, resulting in slow write performance. A similar trend has been observed in existing LSM-tree studies [17]. Besides that, read latency for LSM-trees increases because the data fills up more levels, resulting in more RDMA reads. Note that increasing the data within one memory node is not the ideal way to accommodate a large data set. A better way is to increase the data over multiple memory nodes (see Sect. 11.3.10). For Sherman, the performance decreases mainly due to the higher CPU cache misses and the bigger memory footprint. Observe that space usage in the remote memory is different across the competitors. With 100 million key-value pairs, RocksDB-RDMA (8KB) takes 39GB, RocksDB-RDMA (2KB) takes 44GB, Memory-RocksDB-RDMA takes 52GB, dLSM takes 59GB, and Sherman takes 68GB.

11.3.4 Evaluating mixed performance

We evaluate the performance of dLSM against the baselines on the mixed workloads with reads and writes by using the "randomreadrandomwrite" benchmark in RocksDB. This benchmark has the same number of keys and the same key range as in the previous experiments. Recall that in Sec. 7, dLSM can have different number of shards. We use dLSM-λ to indicate that dLSM uses λ shards.

Figure 11 gives the results for various read/write ratios. dLSM outperforms all LSM-tree variants in all cases although it loses to Sherman slightly when the read ratio is 95

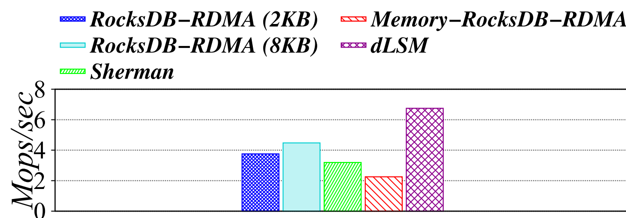


Fig. 12 Evaluating range query performance

11.3.5 Evaluating range query performance

In this experiment, we evaluate dLSM's performance during table scan by running "readseq" in RocksDB. The benchmark creates an iterator iterating through the whole database. All LSM systems enable table prefetching to improve performance by sequential I/O. Sherman [80] uses the cached internal nodes to accelerate the sequential read for table scan. We only test the benchmark with a single thread to avoid saturating the infiniband bandwidth (dLSM takes around 2.5 GB/s by 1 thread, while the hardware limit is 12.5GB/s). We omit the result of Nova-LSM in this experiment due to a bug on the range index for Nova-LSM. The results in Fig. 12 demonstrate that dLSM outperforms RocksDB-RDMA (8KB) by 1.5×, RocksDB-RDMA (2KB) by 1.8×, Memory-RocksDB-RDMA by 3.0× and Sherman by 2.1×. Compared to LSM-tree competitors, the huge performance advantage of dLSM comes from the removal of block unwrapping. Another reason is that the iterators in RocksDB baselines is still block-based that needs to access the SSTable index frequently, while dLSM can directly parse the key-value pairs from the prefetched buffer. Compared with Sherman [80], the performance advantage of dLSM comes from the larger chunk (several MBs) prefetching with one RDMA round trip, while Sherman fetches data in blocks (1KB). The reason RocksDB-RDMA (8K) is faster than RocksDB-RDMA (2K) and Memory-RocksDB-RDMA is that RocksDB-RDMA (8KB) unwraps the block less frequently due to the larger block size.

11.3.6 Evaluating compaction

In this experiment, we study the impact of near-data compaction to dLSM. We separate the experiments into two parts. In the first part, we disable the adaptive compaction strategy and explore the performance of the purely push-down compaction strategy. Then, in the second part, we enable the adaptive compaction strategy to see its benefits compared to the pure push-down strategy. We run the "randomfill" benchmark under normal mode with 16 threads.

Pure push-down strategy. In the first experiment, we test the pure push-down performance with different remote computing power and different pressures of front-end insertion.

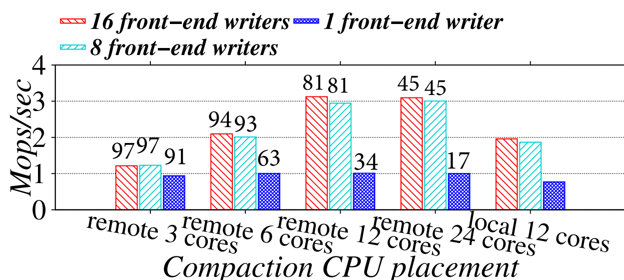


Fig. 13 The impact of remote CPU cores for pure push-down strategy

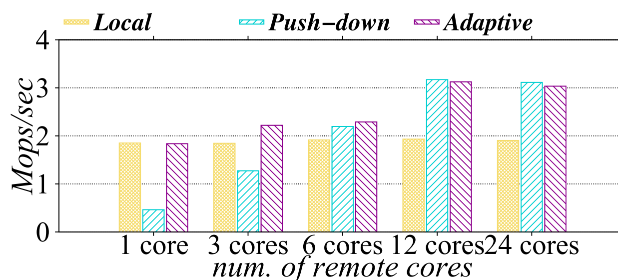


Fig. 14 The impact of adaptive compaction push-down strategy

Besides that, we compare the results of performing compaction purely in the compute vs. memory nodes to demonstrate the effect of purely near-data compaction (Sec. 5). Figure 13 presents the results. The percentage over the bar represents the CPU utilization over all cores during the benchmark. From left to right, the figure shows the impact of near-data compaction with different remote computing power. The last group of bars represents the system performance without near-data compaction. When there is little computing power, CPU utilization is very high and the performance is bounded by the background compaction, but there is an upper limit (for 12 cores). The reason is that Level 0's compactions are overlapped so they have to be done together. When there is a small number of front-end writers, e.g., 1 front-end writer, near-data compaction does not help much because performance is bounded by the front-end insertions. With sufficient front-end writes, near-data compaction can boost dLSM's performance by 60

Adaptive push-down strategy. In this experiment, to show the benefit of our adaptive approach, we compare the performance of our adaptive strategy against two other strategies. One pushes down all compactions (push-down only) and the other performs all the compactions locally only (local only). We vary the remote computing power from 1 to 24 cores to show the impact of remote computing power to different strategies. As in Fig. 14, the "push-down only" approach performs poorly when the remote computing power is insufficient while the "local only" compaction strategy performs moderately across all the cores. Our adaptive strategy shows better performance compared to the other strategies when the remote CPU core is limited. It is only slightly slower than pure push-down when the remote computing power is larger than 12 cores because of the overhead for dynamic decision-making.

11.3.7 Evaluating byte-addressable SSTable

In this experiment, we study the impact of byte-addressable SSTables for read and write. We enable and disable the byte-addressable index design of Sec. 6, termed dLSM and dLSM-Block, respectively. dLSM-Block uses 8KB as the block size for SSTables. We run the "randomfill" and "randomread" benchmarks to test the performance of random writes and reads. From Fig. 15, dLSM is faster than dLSM-Block for both writes and reads, especially for reads (up to 60 The reason is that byte-addressability enables directly fetching a single key-value pair without accessing a whole block. Write performance is improved due to eliminating unnecessary data copy once the notion of "block" is removed.

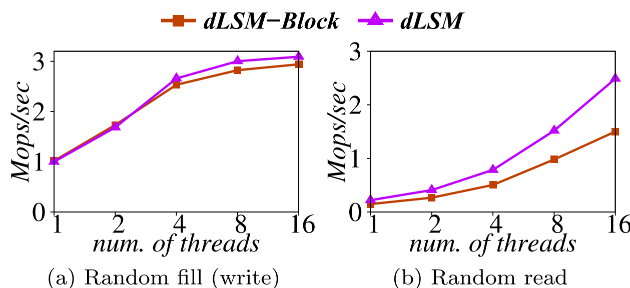


Fig. 15 Evaluating byte-addressable SSTable

11.3.8 Evaluating local cache space optimization

In this experiment, we study the performance of our optimizations for the limited cache space described in Sec. 6.2 by running Benchmark "readwhilewrite." "readwhilewrite" contains one thread loading the data and 16 other threads reading the data concurrently. The loading thread inserts 400 million key-value pairs, and the whole loading process is divided into 100 batches (4 million insertions in each batch). The benchmark reports the read throughput after each batch loading. The local cache size limit is set to 4 GB.

We compare our approach with two others: one that only contains block-based SSTables in the tree (Block-based only), and the other only contains byte-addressable SSTables in the tree (Byte-addressable only). As in Fig. 16, the "byte-addressable only" competitor experiences a significant performance drop at around the 30th batch. The reason behind this is that the local cache reaches its limit, resulting in cache misses for index blocks. The block-based-only competitor does not have an apparent performance degradation but it performs moderately throughout the experiment. Our

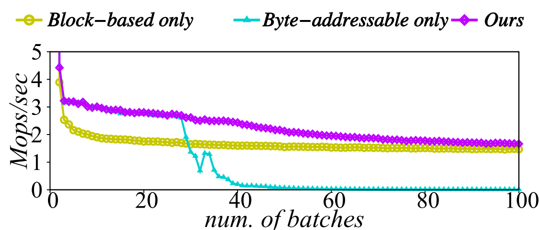


Fig. 16 The impact of limited cache space optimization

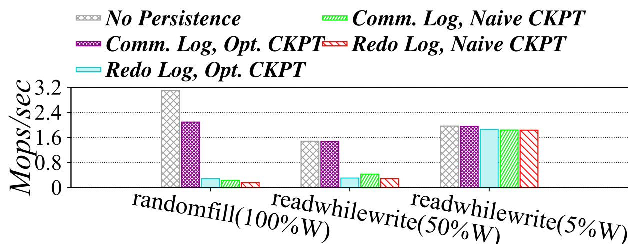


Fig. 17 Evaluating the impact of data persistence

optimization has the best of the two worlds; it shows similar performance compared to “byte-addressable only" initially and does not degrade in performance in the longer term.

11.3.9 Evaluating persistency design

In this experiment, we evaluate the optimizations for persistence mentioned in Sect. 8 by running benchmark “randomfill" and “readwhilewrite" (with 50 to represent the redo log. As for the checkpoint, the baseline implementation is to persist all the generated SSTables for compactions, while the optimized one merges the compaction results before we persist them to avoid unnecessary SSTable persistence.

As shown in Fig. 17, the command log and optimized checkpoint outperform the redo log and naive checkpoint implementations, especially in write-intensive workloads. The primary reason for this improvement is that the command log reduces log synchronization and log size. Additionally, the optimized checkpoint closely aligns with the pace of front-end writes, ensuring that memory usage always remains within the specified limit. As a result, the remote memory allocator never reaches its limit and does not wait for garbage collection. Compared to the system without persistence, enabling our optimized persistence mechanism barely incurs any performance penalty, except in the case of a pure write workload. The performance drop in pure write workloads is expected because command logging and checkpoint operations involve storage I/O.

11.3.10 Evaluating multi-node design

In this experiment, we evaluate the multi-node design of `dLSM` to support multiple compute nodes and multiple memory nodes as described in Sec. 9. We use *CloudLab*[8] [41] that provides multiple nodes. We choose the instance type of c6220, where each node contains two Xeon E5-2650v2 processors (8 cores each, 2.6GHz) and 64GB memory. The nodes are connected by an RDMA-enabled Mellanox FDR Connectx-3 NIC with a bandwidth of 56Gb/s. Each node runs Ubuntu 18.04.1.

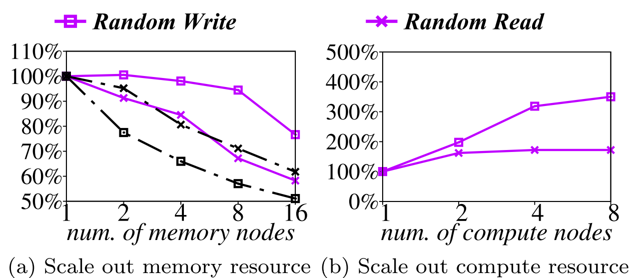
We show three experiments: scale out memory nodes only; scale out compute nodes only; scale out both compute and memory nodes. We run the benchmarks “randomfill" and “randomread" under normal mode, with minor modifications to support the multi-node setup. All the other LSM-tree parameters are set the same as in the previous experiments.

In the first experiment, we fix the number of compute nodes to 1 and scale out memory resources as well as the data volume from 1 node (50 million key-value pairs) to 16 nodes (800 million key-value pairs). This experiment shows a different way to increase the data size compared to Sect. 11.3.3, in which data is increased within a single server. From Fig. 18(a), increasing the data size over multiple memory nodes leads to performance degradation for both reads and writes. The reason is the same as the reason when increasing the data size within a single server (Sect. 11.3.3). In Fig. 18(a), we add a black dotted line to represent the result of holding the same amount of data within a single server. Notice that increasing the data size over multiple memory servers has better scalability than that in a single server, especially for the writes. The reason is that the remote computing power increases as we add the memory nodes, which accelerates the compaction.

In the second experiment, we fix the number of memory nodes to 1 and scale out the compute resources from 1 to 8 nodes. We set the data size to 50 million key-value pairs. From Fig. 18(b), writing has better scalability than reading. The reason is that the sequential I/Os for writes can utilize more RDMA bandwidth than random I/Os for reads. Besides that, we find that scaling up the computing node will increase the space consumption in the memory node, making the experiment out of memory at 8 nodes.

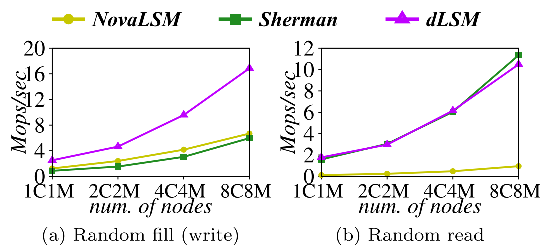
Finally, we vary the number of compute and memory nodes together from 1 to 8 and the data size has been increased from 50 Million to 400 Million. We use xCyM to indicate x compute nodes and y memory nodes in the system and set λ to 8. Sherman and NovaLSM are also tested in this setup. Figure 19 gives the results, indicating that `dLSM` scales well for multiple nodes and `dLSM` achieves better performance compared to the other competitors.

[8] https://www.cloudlab.us/



(a) Scale out memory resource (b) Scale out compute resource

Fig. 18 Evaluating scalability



(a) Random fill (write) (b) Random read

Fig. 19 Evaluating multi-node design

12 Related work

Resource Disaggregation. Resource disaggregation offers great benefits in data centers for cost efficiency, resource utilization, and elasticity. Achieving good performance in disaggregated architectures requires redesign of many aspects, e.g., operating systems [73], hardware [57], and networking [45, 47]. dLSM focuses on the data indexing aspect.

Databases for Disaggregated Architectures. Database systems require significant rethinking to leverage disaggregated architectures, see [79] for a recent survey. Cloud-native databases (OLTP and OLAP) are re-designed to follow this trend, e.g., Aurora [76], OpenAurora [66], PolarDB [26], Socrates [14], Taurus [34], Snowflake [28], and FlexPushdownDB [87] are built on top of a distributed shared storage pool. The innovation is in separating storage from compute to support independent scaling (of compute and storage) and elasticity. Many optimizations are adopted, e.g., caching, offloading, and shipping logs. These works still couple compute with memory in the same server, while dLSM focuses on memory disaggregation.

Recently, works that optimize databases for memory disaggregation, e.g., Zhang et al., [93, 94] study the impact of memory disaggregation on OLAP databases (both disk-based and memory-based) [93, 94] and report significant performance degradation that motivates further optimization as shown in [95]. Farview [53] is an analytical database system optimized for memory disaggregation using FPGA. It separates query processing from buffering, and uses near-data computing to offload operations, e.g., selection and aggregation, to reduce data transfer. dLSM offloads compaction of the LSM-tree. PolarDB is a customized cloud-native database that disaggregates memory [27, 96] with index prefetching, optimistic locking, and optimized recovery. Zuo et al. [98] develop a hash index for disaggregated memory but it cannot support range queries as in dLSM. Sherman [80] is a highly optimized B-tree index structure for the disaggregated memory architecture. dLSM focuses on LSM indexes with disaggregated memory and has not been studied in [27, 53, 80, 93, 94, 96]. Experiments show that dLSM achieves much faster write performance over Sherman [80] while offering comparable read performance (Fig. 8 and Fig. 9).

RDMA-optimized Databases. Many works optimize databases for RDMA networking, e.g., see [18, 23] for an overview. Proposals include using RDMA to extend memory [55] and remote cache [92] to improve query processing [19, 72], B-tree [97], hashing [63], transactions [89], and enhancing availability [90]. In contrast, dLSM targets LSM-tree indexing for RDMA-enabled remote memory.

Distributed Shared Memory. Proposals exist for building distributed shared memory from multiple servers connected by RDMA [12, 24, 40, 42, 54, 64, 68, 75]. The main idea is to implement a shared memory pool that can elastically provide any amount of memory resources as needed. dLSM's memory node can be replaced by a shared memory pool to mimic a near-infinite memory resource.

Log-structured Merge Tree There are prior studies optimizing LSM over traditional storage. For example, SlimDB [70] and Monkey [29] reduce the false positive rate for the Bloom filters. SuRF [91] and Rosetta [60] introduce range filters to avoid unnecessary I/Os for range queries. Chucky [32] shows that Cuckoo filters can achieve a lower false positive rate with a smaller memory footprint compared to Bloom filters. Dostoevsky [30] and LSM-bush [31] introduce LSM trees with an adaptive merging policy to achieve optimal performance given a workload. Spooky [33] proposes a novel compaction granulation method to reduce write amplification.

In a disaggregated memory architecture, local and remote memories form a hierarchy similar to that of local and non-volatile memories, e.g., Intel 3D Xpoint. Recent research optimizes the LSM-tree (or key-value stores, in general) for non-volatile memory, e.g., [15, 20, 58, 59, 83, 86]. However, there are at least two main differences in dLSM: (1) The remote memory node in dLSM supports offloading (i.e., near-data compaction) while non-volatile memory does not provide offloading. Even if there are Smart SSDs [38, 50, 78], they can perform very limited offloading. (2) dLSM has RDMA-specific optimizations that those works do not have.

13 Conclusion

In this paper, we investigate realizing an LSM-based index over disaggregated memory. dLSM utilizes several opti-

mizations to best leverage the communication layer's features, e.g., the byte-addressable low-latency of RDMA-based remote memory. The main ideas include reducing software overhead, near-data compaction, byte-addressability, and efficient RDMA communication. Experiments show that dLSM achieves higher performance than porting existing LSM-trees or running the optimized B-tree to the disaggregated memory architecture.

Acknowledgements Walid Aref acknowledges the support of the National Science Foundation under Grant Number IIS-1910216. M. Tamer Özsu's research is funded in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada. Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number IIS-2337806.

References

1. Advancing Cloud with Memory Disaggregation. https://www.ibm.com/blogs/research/2018/01/advancing-cloud-memory-disaggregation/
2. Apache Cassandra. https://cassandra.apache.org/
3. Apache HBase. https://hbase.apache.org/
4. Compute express link: the breakthrough CPU-to-device interconnect. https://www.computeexpresslink.org/about-cxl
5. Intel RSD. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html
6. LevelDB. https://github.com/google/leveldb
7. Open Fabrics Enterprise Distribution (OFED) Performance Tests. https://github.com/linux-rdma/perftest
8. Remote Compactions in RocksDB-Cloud. https://rockset.com/blog/remote-compactions-in-rocksdb-cloud/
9. RocksDB Benchmarking Tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools
10. RocksDB. http://rocksdb.org/
11. VoltDB. https://www.voltdb.com/
12. Aguilera, M. K., Amit, N., Calciu, I., Deguillard, X., Gandhi, J., Subrahmanyam, P., Suresh, L., Tati, K., Venkatasubramanian, R., Wei, M.: Remote memory in the age of fast networks. In: Proceedings of the Symposium on Cloud Computing (SoCC), pp. 121–127 (2017)
13. Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V.R., Bu, Y., Carey, M.J., Cetindil, I., Cheelangi, M., Faraaz, K., Gabrielova, E., Grover, R., Heilbron, Z., Kim, Y., Li, C., Li, G., Ok, J.M., Onose, N., Pirzadeh, P., Tsotras, V.J., Vernica, R., Wen, J., Westmann, T.: AsterixDB: a scalable, open source BDMS. Proceedings of the VLDB Endowment (PVLDB) **7**(14), 1905–1916 (2014)
14. Antonopoulos, P., Budovski, A., Diaconu, C., Saenz, A. H., Hu, J., Kodavalla, H., Kossmann, D., Lingam, S., Minhas, U. F., Prakash, N., Purohit, V., Qu, H., Ravella, C. S., Reisteter, K., Shrotri, S., Tang, D., Wakade, V.: Socrates: The New SQL Server in the Cloud. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1743–1756 (2019)
15. Arulraj, J., Levandoski, J.J., Minhas, U.F., Larson, P.: BzTree: a high-performance latch-free range index for non-volatile memory. Proceedings of the VLDB Endowment (PVLDB) **11**(5), 553–565 (2018)
16. Balasundaram, R.: Cuckoo hashing table format (http://rocksdb.org/blog/2014/09/12/cuckoo.html) (2014)
17. Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., Didona, D.: SILK: preventing latency spikes in log-structured merge key-value stores. In: USENIX Annual Technical Conference (ATC), pp. 753–766 (2019)
18. Barthels, C., Alonso, G., Hoefler, T.: Designing databases for future high-performance networks. IEEE Database Eng Bull **40**(1), 15–26 (2017)
19. Barthels, C., Loesing, S., Alonso, G., Kossmann, D.: Rack-scale in-memory join processing using RDMA. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp 1463–1475 (2015)
20. Benson, L., Makait, H., Rabl, T.: Viper: an efficient hybrid PMem-DRAM key-value store. Proceedings of the VLDB Endowment (PVLDB) **14**(9), 1544–1556 (2021)
21. Berenson, H., Bernstein, P. A., Gray, J., Melton, J., O'Neil, E. J., O'Neil, P. E.: A Critique of ANSI SQL isolation levels. In: Proceedings of the ACM international conference on management of data (SIGMOD), pp. 1–10 (1995)
22. Bindschaedler, L., Goel, A., Zwaenepoel, W.: Hailstorm: disaggregated compute and storage for distributed LSM-based databases. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 301–316 (2020)
23. Binnig, C., Crotty, A., Galakatos, A., Kraska, T., Zamanian, E.: The end of slow networks: it's time for a redesign. Proceedings of the VLDB Endowment (PVLDB) **9**(7), 528–539 (2016)
24. Cai, Q., Guo, W., Zhang, H., Agrawal, D., Chen, G., Ooi, B.C., Tan, K., Teo, Y.M., Wang, S.: Efficient distributed memory management with RDMA and caching. Proceedings of the VLDB Endowment (PVLDB) **11**(11), 1604–1617 (2018)
25. Cao, W., Liu, Y., Cheng, Z., Zheng, N., Li, W., Wu, W., Ouyang, L., Wang, P., Wang, Y., Kuan, R., Liu, Z., Zhu, F., Zhang, T.: POLARDB meets computational storage: efficiently support analytical workloads in cloud-native relational database. In: USENIX Conference on File and Storage Technologies (FAST), pp. 29–41 (2020)
26. Cao, W., Liu, Z., Wang, P., Chen, S., Zhu, C., Zheng, S., Wang, Y., Ma, G.: PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. Proceedings of the VLDB Endowment (PVLDB) **11**(12), 1849–1862 (2018)
27. Cao, W., Zhang, Y., Yang, X., Li, F., Wang, S., Hu, Q., Cheng, X., Chen, Z., Liu, Z.,Fang, J., Wang, B., Wang, Y., Sun, H., Yang, Z., Cheng, Z., Chen, S., Wu, J., Hu, W., Zhao, J., Gao, Y., Cai, S., Zhang, Y., Tong, J.: PolarDB Serverless: a cloud native database for disaggregated data centers. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 2477–2489 (2021)
28. Dageville, B., Cruanes, T., Zukowski, M., Antonov, V., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., Lee, A.W., Motivala, A., Munir, A.Q.,Pelley, S., Povinec, P., Rahn, G., Triantafyllis, S., Unterbrunner, P.: The snowflake elastic data warehouse. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 215–226 (2016)
29. Dayan, N., Athanassoulis, M., Idreos, S.: Monkey: optimal navigable key-value store. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 79–94 (2017)
30. Dayan, N., Idreos, S.: Dostoevsky: better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp 505–520 (2018)
31. Dayan, N., Idreos, S.: The log-structured merge-bush and the Wacky Continuum. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 449–466 (2019)
32. Dayan, N., Twitto, M.: Chucky: a succinct cuckoo filter for LSM-tree. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 365–378 (2021)
33. Dayan, N., Weiss, T., Dashevsky, S., Pan, M., Bortnikov, E., Twitto, M.: Spooky: granulating LSM-tree compactions correctly. In: Pro-

ceedings of the VLDB endowment (PVLDB), pp. 3071–3084 (2022)
34. Depoutovitch, A., Chen, C., Chen, J., Larson, P., Lin, S., Ng, J., Cui, W., Liu, Q., Huang, W., Xiao, Y., He, Y.: Taurus database: how to be fast, available, and frugal in the cloud. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1463–1478 (2020)
35. DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A.: A performance analysis of the gamma database machine. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 350–360 (1988)
36. DeWitt, D.J., Hawthorn, P.B.: A performance evaluation of data base machine architectures (invited paper). In: International Conference on Very Large Data Bases (VLDB), pp. 199–214 (1981)
37. Diaconu, C., Freedman, C., Ismert, E., Larson, P., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL server's memory-optimized OLTP engine. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1243–1254 (2013)
38. Do, J., Kee, Y., Patel, J.M., Park, C., Park, K., DeWitt, D.J.: Query processing on smart SSDs: opportunities and challenges. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1221–1230 (2013)
39. Dong, S.: PlainTable—a new file format. (http://rocksdb.org/blog/2014/06/23/plaintable-a-new-file-format.html) (2014)
40. Dragojevic, A., Narayanan, D., Castro, M., Hodson, O.: FaRM: Fast remote memory. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 401–414 (2014)
41. Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., Mishra, P.: The design and operation of CloudLab. In: USENIX Annual Technical Conference (ATC), pp. 1–14 (2019)
42. Fent, P., van Renen, A., Kipf, A., Leis, V., Neumann, T., Kemper, A.: low-latency communication for fast DBMS using RDMA and shared memory. In: International Conference on Data Engineering (ICDE), pp. 1477–1488 (2020)
43. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 50–59 (2004)
44. Franklin, M. J., Jónsson, B. T., Kossmann, D.: Performance tradeoffs for client-server query processing. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 149–160 (1996)
45. Gao, P.X., Narayan, A., Karandikar, S., Carreira, J., Han, S., Agarwal, R., Ratnasamy, S., Shenker, S.: Network requirements for resource disaggregation. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 249–264 (2016)
46. Golan-Gueta, G., Bortnikov, E., Hillel, E., Keidar, I.: Scaling concurrent log-structured data stores. In: Proceedings of the tenth european conference on computer systems (EuroSys), pp. 32:1–32:14 (2015)
47. Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., Shin, K.G.: Efficient memory disaggregation with Infiniswap. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 649–667 (2017)
48. Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., Wei, W., Liu, C., Zhang, J., Li, J., Wu, X., Song, L., Sun, R., Yu, S., Zhao, L., Cameron, N., Pei, L., Tang, X.: TiDB: a raft-based HTAP database. Proceedings of the VLDB Endowment (PVLDB) **13**(12), 3072–3084 (2020)
49. Huang, H., Ghandeharizadeh, S.: Nova-LSM: a distributed, component-based LSM-tree key-value store. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 749–763D (2021)
50. István, Z., Sidler, D., Alonso, G.: Caribou: intelligent distributed storage. Proceedings of the VLDB Endowment (PVLDB) **10**(11), 1202–1213 (2017)
51. Kalia, A., Kaminsky, M., Andersen, D.G.: Design guidelines for high performance RDMA systems. In: USENIX Annual Technical Conference (ATC), pp. 437–450 (2016)
52. Keeton, K., Patterson, D.A., Hellerstein, J.M.: A case for intelligent disks (IDISKs). SIGMOD record **27**(3), 42–52 (1998)
53. Korolija, D., Koutsoukos, D., Keeton, K., Taranov, K., Milojicic, D.S., Alonso, G.: Farview: disaggregated memory with operator off-loading for database engines. In: Conference on Innovative Data Systems Research (CIDR) (2022)
54. Lagar-Cavilla, H. A., Ahn, J., Souhlal, S., Agarwal, N., Burny, R., Butt, S., Chang, J., Chaugule, A., Deng, N., Shahid, J., Thelen, G., Yurtsever, K.A., Zhao, Y., Ranganathan, P.: Software-defined far memory in warehouse-scale computers. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 317–330 (2019)
55. Li, F., Das, S., Syamala, M., Narasayya, V.R.: Accelerating relational databases by leveraging remote memory and RDMA. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 355–370 (2016)
56. Li, H., Berger, D.S., Hsu, L., Ernst, D., Zardoshti, P., Novakovic, S., Shah, M., Rajadnya, S., Lee, S., Agarwal, I., Hill, M.D., Fontoura, M., Bianchini, R.: Pond: CXL-based memory pooling systems for cloud platforms. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 574–587 (2023)
57. Lim KT., Chang, J., Mudge, T.N., Ranganathan, P., Reinhardt, S. K., Wenisch, T. F.: Disaggregated memory for expansion and sharing in blade servers. In: International Symposium on Computer Architecture (ISCA), pp. 267–278 (2009)
58. Liu, J., Chen, S., Wang, L.: LB+-trees: optimizing persistent index performance on 3DXPoint memory. Proceedings of the VLDB Endowment (PVLDB) **13**(7), 1078–1090 (2020)
59. Lu, B., Hao, X., Wang, T., Lo, E.: Dash: scalable hashing on persistent memory. Proceedings of the VLDB Endowment (PVLDB) **13**(8), 1147–1161 (2020)
60. Luo, S., Chatterjee, S., Ketsetsidis, R., Dayan, N., Qin, W., Idreos, S.: Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 2071–2086 (2020)
61. Makreshanski, D., Giceva, J., Barthels, C., Alonso, G.: BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 37–50 (2017)
62. Malviya, N., Weisberg, A., Madden, S., Stonebraker, M.: Rethinking main memory OLTP recovery. In: International Conference on Data Engineering (ICDE), pp. 604–615 (2014)
63. Mitchell, C., Geng, Y., Li, J.: Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In: USENIX Annual Technical Conference (ATC), pp. 103–114 (2013)
64. Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., Oskin, M.: Latency-tolerant software distributed shared memory. In: USENIX Annual Technical Conference (ATC), pp. 291–305 (2015)
65. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The log-structured merge-tree (LSM-Tree). Acta Informat. **33**(4), 351–385 (1996)

66. Pang, X., Wangm J.: Understanding the performance implications of the design principles in storage-disaggregated databases. In: Proceedings of ACM Conference on Management of Data (SIGMOD) (2024)
67. Picoli, I.L., Bonnet, P., Tözün, P.: LSM management on computational storage. In: Proceedings of the International Workshop on Data Management on New Hardware (DaMoN), pp. 17:1–17:3 (2019)
68. Pröbstl, M., Fent, P., Schüle, M.E., Sichert, M., Neumann, T., Kemper, A.: One buffer manager to rule them all: using distributed memory with cache coherence over RDMA. In: International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB), pp. 17–26 (2021)
69. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM (CACM) **33**(6), 668–676 (1990)
70. Ren, K., Zheng, Q., Arulraj, J., Gibson, G.: SlimDB: a space-efficient key-value storage engine for semi-sorted data. Proceedings of the VLDB Endowment (PVLDB) **10**(13), 2037–2048 (2017)
71. Riedel, E., Gibson, G.A., Faloutsos, C.: Active storage for large-scale data mining and multimedia. In: International Conference on Very Large Data Bases (VLDB), pp. 62–73 (1998)
72. Salama, A., Binnig, C., Kraska, T., Scherp, A., Ziegler, T.: Rethinking distributed query execution on high-speed networks. IEEE Data Engineering Bulletin **40**(1), 27–37 (2017)
73. Shan, Y., Huang, Y., Chen, Y., Zhang, Y.: LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 69–87 (2018)
74. Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., Bardea, P., Ranade, A., Darnell, B., Gruneir, B., Jaffray, J., Zhang, L., Mattis, P.: CockroachDB: The resilient geo-distributed SQL database. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1493–1509 (2020)
75. Taranov, K., Girolamo, S. D., Hoefler, T.: CoRM: Compactable remote memory over RDMA. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1811–1824 (2021)
76. Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., Bao, X.: Amazon aurora: design considerations for high throughput cloud-native relational databases. In: ACM Conference on Management of Data (SIGMOD), pp. 1041–1052 (2017)
77. Voruganti, K., Özsu, M. T., Unrau, R.C.: An adaptive hybrid server architecture for client caching ODBMSs. In: International Conference on Very Large Data Bases (VLDB), pp. 150–161 (1999)
78. Wang, J., Park, D., Kee, Y.-S., Papakonstantinou, Y., Swanson, S.: SSD In-storage computing for list intersection. In: Proceedings of the International Workshop on Data Management on New Hardware (DaMoN), pp 4:1–4:7 (2016)
79. Wang, J., Zhang, Q.: Disaggregated Database Systems. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 37–44 (2023)
80. Wang, Q., Lu, Y., Shu, J.: Sherman: a write-optimized distributed B+Tree index on disaggregated memory. In: ACM International Conference on Management of Data (SIGMOD), pp. 1033–1048 (2022)
81. Wang, R., Wang, J., Idreos, S., Özsu, M.T., Aref, W.G.: The case for distributed shared-memory databases with RDMA-enabled memory disaggregation. Proceedings of the VLDB Endowment (PVLDB) **16**(1), 15–22 (2022)
82. Wang, R., Wang, J., Kadam, P., Özsu, M.T., Aref, W.G.: dLSM: An LSM-based index for memory disaggregation. In: International Conference on Data Engineering (ICDE), pp. 2835–2849 (2023)
83. Wang, T., Levandoski, J.J., Larson, P.: Easy lock-free indexing in non-volatile memory. In: International Conference on Data Engineering (ICDE), pp. 461–472 (2018)
84. Wu, Y., Guo, W., Chan, C., Tan, K.: Fast failure recovery for main-memory DBMSs on multicores. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 267–281 (2017)
85. Xiao, M., Wang, H., Geng, L., Lee, R., Zhang, X.: Catfish: adaptive RDMA-enabled R-tree for low latency and high throughput. In: International Conference on Distributed Computing Systems (ICDCS), pp. 164–175 (2019)
86. Yan, B., Cheng, X., Jiang, B., Chen, S., Shang, C., Wang, J., Huang, K., Yang, X., Cao, W., Li, F.: Revisiting the design of LSM-tree based OLTP storage engine with persistent memory. Proceedings of the VLDB Endowment (PVLDB) **14**(10), 1872–1885 (2021)
87. Yang, Y., Youill, M., Woicik, M.E., Liu, Y., Yu, X., Serafini, M., Aboulnaga, A., Stonebraker, M.: FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS. Proceedings of the VLDB Endowment (PVLDB) **14**(11), 2101–2113 (2021)
88. Yu, Q., Guo, C., Zhuang, J., Thakkar, V., Wang, J., Cao, Z.: CaaS-LSM: compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure. In: SIGMOD (2024)
89. Zamanian, E., Shun, J., Binnig, C., Kraska, T.P.: Chiller: contention-centric transaction execution and data partitioning for modern networks. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 511–526 (2020)
90. Zamanian, E., Yu, X., Stonebraker, M., Kraska, T.: Rethinking database high availability with RDMA networks. Proceedings of the VLDB Endowment (PVLDB) **12**(11), 1637–1650 (2019)
91. Zhang, H., Lim, H., Leis, V., Andersen, D. G., Kaminsky, M., Keeton, K. Pavlo, A.: SuRF: practical range query filtering with fast succinct tries. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 323–336. ACM (2018)
92. Zhang, Q., Bernstein, P.A., Berger, D.S., Chandramouli, B.: Redy: remote dynamic memory cache. Proceedings of the VLDB Endowment (PVLDB) **15**(4), 766–779 (2022)
93. Zhang Q, Cai, Y., Angel, S., Liu, V., Chen, A., Loo, B. T.: Rethinking data management systems for disaggregated data centers. In: Conference on Innovative Data Systems Research (CIDR) (2020)
94. Zhang, Q., Cai, Y., Chen, X., Angel, S., Chen, A., Liu, V., Loo, B.T.: Understanding the effect of data center resource disaggregation on production DBMSs. Proceedings of the VLDB Endowment (PVLDB) **13**(9), 1568–1581 (2020)
95. Zhang, Q., Chen, X., Sankhe, S., Zheng, Z., Zhong, K., Angel, S., Chen, A., Liu, V., Loo, BT.: Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In: ACM International Conference on Management of Data (SIGMOD), pp. 1345–1359 (2022)
96. Zhang, Y., Ruan, C., Li, C., Yang, J., Cao, W., Li, F., Wang, B., Fang, J., Wang, Y., Huo, J., Bi, C.: Towards cost-effective and elastic cloud database deployment via memory disaggregation. Proceedings of the VLDB Endowment (PVLDB) **14**(10), 1900–1912 (2021)

97. Ziegler, T., Vani, S. T., Binnig, C., Fonseca, R., Kraska, T.: Designing distributed tree-based index structures for fast RDMA-capable networks. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 741–758 (2019)
98. Zuo, P., Sun, J., Yang, L., Zhang, S., Hua, Y.:One-sided RDMA-conscious extendible hashing for disaggregated memory. In: USENIX Annual Technical Conference (ATC), pp. 15–29 (2021)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.