2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00217

dLSM: An LSM-Based Index for Memory Disaggregation

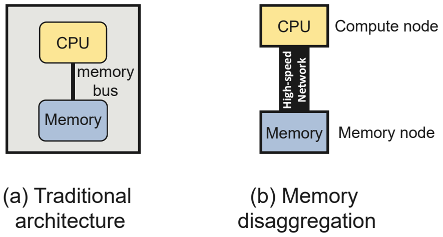
Ruihong Wang Jianguo Wang Prishita Kadam †M. Tamer Özsu Walid G. Aref
Purdue University †University of Waterloo
{*wang4996; csjgwang; pkadam; aref*}@*purdue.edu* †*tamer.ozsu@uwaterloo.ca*

***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 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 moderate. 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 1.6× to 11.7× 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 41,000 LOC), and is open-sourced.**

I. 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], [51]. 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 (Figure 1a), with memory disaggregation, compute and memory are physically separated and are connected via fast networking (Figure 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, respectively.[1] Each compute node has powerful computing capability, e.g., 100s 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., 100s of GBs [27], [72], [81]–[84].

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 [71], studies how to optimize B-tree indexing for memory disaggregation. However, the write performance is moderate (as shown in Sec. XI). To improve performance, this paper focuses on LSM-based (log-structured merge tree) indexing [59] in the presence of memory disaggregation.



(a) Traditional architecture (b) Memory disaggregation

Fig. 1: Traditional architecture vs. 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 [57], [71]. As in our experiments on RDMA Mellanox EDR ConnectX-4 NIC, there is a 100× performance gap between transferring the same amount of data in 64 byte units vs. 1MB units based on the RDMA benchmark [7]. (3) Our experiments (Sec. XI) show that the LSM-tree is feasible for disaggregated memory and, as expected, it outperforms the optimized B-tree on writes (i.e., Sherman [71]) 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. This provides an opportunity to improve performance, e.g., near-data processing. (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.,

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

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00217

RDMA is non-trivial as it involves many alternative primitive, e.g., one- vs. two-sided RDMA (Sec. II-B). 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 and flushing overheads, to unlock the full potential of fast networking. dLSM offloads LSM-tree compaction to the remote memory node to reduce data movement by exploiting the CPUs in memory nodes. dLSM tunes the index layout to leverage byte-addressability in the remote memory. dLSM optimizes communication including customized RPC and asynchronous I/O.

The paper makes the following contributions:

- **Index design over disaggreagted memory:** We present the design of dLSM, the first optimized LSM-based index for disaggregated memory. dLSM is implemented in C++ (with approximately 41,000 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 applies the idea of near-data computing in the context of the disaggregated memory architecture, and pushes down the LSM-tree compaction to the remote memory to significantly reduce data transfer.
- **Customized optimizations for byte-addressability:** 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].

II. Background

A. Resource Disaggregation

Resource disaggregation is an innovative technology in data centers [8], [27], [28], [71], [80], [82], [84], in large part due to the recent breakthroughs in fast networking technologies, e.g., RDMA [45], [50]. 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], [51], [81]–[84].

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 [68], 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], [48], [81], [82], [84]. In contrast to storage disaggregation, it is more challenging to optimize DBMSs for memory disaggregation because the performance issues become more severe for memory disaggregation [48], [80]–[82]. Moreover, memory disaggregation usually relies on a high-speed network fabric, e.g., RDMA, while storage disaggregation can be built based on conventional RPCs [68].

B. Interconnection for Disaggregated Memory

Ultra-fast networking technologies exist for interconnecting compute and memory nodes. For example, Remote Direct Memory Access (RDMA, for short) is a high-speed inter-memory communication mechanism with low latency. It allows direct access to memory in remote nodes [45]. RDMA bypasses the host operating system when transferring data to avoid extra data copy. RDMA is conducted over Infiniband or lossless Ethernet. RDMA's kernel-bypassing and low-latency features make it applicable to high-performance data centers [1], [5], [27], [84]. Another promising communication technology is Compute Express Link (CXL) [4], which is a high-speed interconnection between advanced CPU and periperal 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.

C. Log-structured Merge (LSM) Tree

The LSM-tree [59] 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 probably the most widely adopted implementation. We use the RocksDB implementation to introduce LSM-tree concepts and terminology. Inserts, updates, and deletes are all appended entries 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 [62], 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.

The 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.

III. 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. I. We discuss in Sec. IX 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 MemTable and immutable tables 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.

Writes. dLSM supports concurrent writes and guarantees snapshot isolation with minimal software overhead to best unlock the potential of low-latency remote memory (Sec. IV).

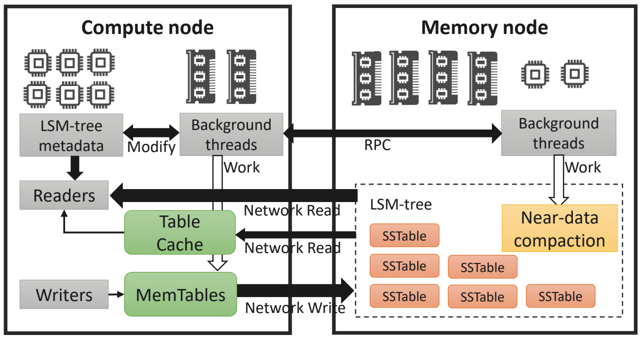


Fig. 2: Overall architecture of dLSM

The MemTable is implemented as a lock-free skip list. 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 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 between the compute and memory nodes, dLSM offloads the compaction process to the memory node. This is termed *near-data compaction* (Sec. V). 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 receive the reply to modify the LSM-tree metadata accordingly. dLSM addresses a number of challenges related to near-data compaction to improve performance (Sec. V).

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. VI).

IV. Minimizing Software Overhead

We present an optimization to improve dLSM's throughput by minimizing software overhead. For relatively slow storage devices (e.g., SSDs or HDDs), software overhead is negligible. However, it can be a performance bottleneck for the ultra fast networking, e.g. RDMA. Significant software overhead is present in existing disk-based LSM-tree implementations [40]. 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

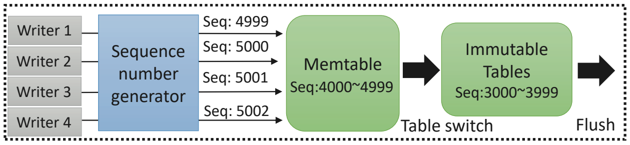


Fig. 3: Concurrent in-memory writes in `dLSM`

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 (Figure 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 MemTable 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 Figure 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 sequence number range of a MemTable is sufficiently large, then 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.

V. Near-data Compaction

We investigate the use of near-data compaction to leverage the compute capability of the remote memory node. It has the potential to improve performance by reducing data movement.

Main Idea. Near-data computing [9], [22], [25], [43], [77] moves execution closer to data to reduce data movement. `dLSM` adopts this strategy by offloading the LSM-tree compaction process entirely 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. Moreover, the compaction process does not require much compute power, which is perfect for the remote memory node.

Near-data computing has been around for decades in several contexts, e.g., database machines [30], [31], object databases [38], [69], active disks [47], [63], Smart SSDs [33], [44], [60], [70], and storage disaggregation [9], [22], [25], [43]. We investigate its use in realizing an LSM-based index for RDMA-enabled 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 [9], Nova-LSM [43], and Hailstorm [22]. Rockset [9] offloads the compaction to the compaction servers, but needs to fetch the SSTables over the network as the data is stored separately in S3.

Challenges. It is non-trivial to efficiently offload compaction to the remote memory. 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. V-A), and (2) How to perform garbage collection in the context of near-data compaction? (Sec. V-B)

A. 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. We can place this metadata 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.

Instead, `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. X-D). When the memory node receives the RPC, it compacts these SSTables. `dLSM` applies the compaction strategy as in Sec. II-C. To reduce write-stalls

[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.

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. Then, the compute node issues another RPC to copy the metadata of the compacted SSTables to the desired working space.

We include implementation optimizations in `dLSM` to avoid network round trips and memory copy by allowing the memory node to allocate memory locally. In particular, 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.

B. Garbage Collection

It is challenging to perform garbage collection for near-data compaction. The following issues need to be addressed:

- How to garbage collect SSTables 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 challenge, `dLSM` allows the compute and memory nodes to garbage collect memory allocated by each side. Memory allocated for near-data compaction is recycled by the memory node, and 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. Else, an RPC (See Section X-D) 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 challenge, 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.

VI. Optimizing for Byte-Addressability

We study how SSTables can be optimized for byte-addressability. LSM-based 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 [34], they fall short, e.g., Cuckoo Hashing does not support range query efficiently; PlainTable relies on `mmap` that cannot be efficiently applied to remote memory settings.

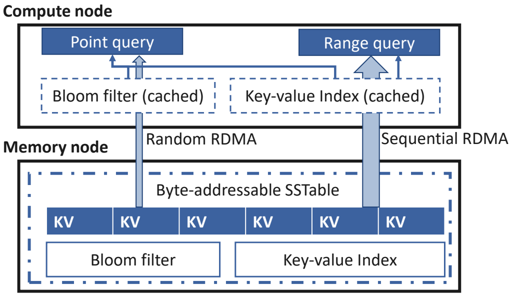


Fig. 4: Byte-addressable SSTable layout in `dLSM`

Next, we introduce a new SSTable design optimized for disaggregated memory to leverage byte-addressability.

Data layout. `dLSM` drops the notion of "blocks" to directly access a single key-value pair (Figure 4). This improves read performance by reducing read amplification as we can directly fetch a single key-value pair without fetching the whole block. Also, the design can improve write performance by eliminating extra memory copy as we do not need to wrap the key-value pairs into blocks anymore. Thus, building an SSTable is accelerated as the key-value pairs are directly serialized to the target buffer without waiting to form a block. To support range query efficiently, key-value pairs for an SSTable are sorted and stored in a continuous memory region.

Index layout. To make good use of byte-addressability, the index needs to quickly 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` stores the hot SSTables in the LSM-tree top levels into local memory.

Supporting point and range queries. Refer to Figure 4. For a point query, the compute node uses a bloom filter to check if the target key is located 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 an network read to fetch the single key-value pair from remote memory. For range queries, fetching every key-value pair one at a time has significant performance penalty as every key-value pair requires a random I/O. Instead, `dLSM` prefetches large chunks of key-value pairs by sequential I/O. Specifically, when handling a range query,

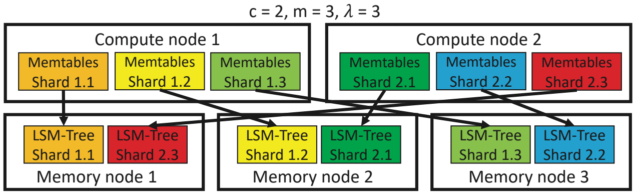


Fig. 5: Supporting multi-compute and multi-memory nodes

the compute node creates an iterator with sub-iterators across all the levels. The sub-iterators locate the first keys in the range by 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 it reaches end of range.

VII. Optimizing for Mixed R/W Workloads

We observe that, dLSM achieves moderate performance on the mixed workloads with reads and writes when compared to the performance of the 100% read or 100% write workloads. The root cause is that the background compaction in Level 0 cannot catch up with the ultra-fast write performance to MemTables, so that the SSTables in Level 0 are accumulated quickly. Thus, a read scans many SSTables in Level 0 to search a key as these SSTables have overlapping ranges. Directly limiting the number of SSTables in Level 0 does not work well because this affects the write performance.

To address this challenge, we follow Nova-LSM's approach [43] 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 Figure 10.

VIII. 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 persistence in the index part (following the prior index works including Sherman [71], an optimized B-tree index for disaggregated memory, and other indexing works, e.g., [32], [75], [85], [86]) because data persistence is achieved at the database layer. The index is rebuilt either from scratch (e.g., Hekaton [32]) or from the last checkpoint (e.g., VoltDB [11]). To illustrate, many modern main-memory databases, e.g., VoltDB [11], BatchDB [55], and PACMAN [74], do not use traditional redo/undo logging for persistence in order to achieve fast performance. Instead, they use an alternative technique termed command log [56] 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 transactional consistent checkpoint. Thus, as long as the index provides a transactional consistent checkpoint, the overall database system can be recovered by the command log. dLSM offers this LSM-based index that natively provides a transactional consistent checkpoint through snapshot isolation.

IX. 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 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], [68], software-level cache coherence protocol [61], [72] or range sharding across the compute node [43]. The first solution cannot ensure strong data consistency as other readers 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 in existing LSM-based systems [2], [3], [9], [13], [42], [66].

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 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.

Specifically, let c, m, and λ be the number of compute nodes, memory nodes, and shards within a compute node, respectively. From Figure 5, dLSM 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. The 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. XI-C8 evaluates this multi-node design for dLSM.

X. Instantiating dLSM over RDMA

RDMA-enabled disaggregated memory is well-studied [37], [71], [79], [86]. Thus, we use it to instantiate and test dLSM.

A. *dLSM Codebase*

dLSM is coded from scratch but it reuses certain data structures (e.g., concurrent skip list, bloom filters, immutable MemTables) from LevelDB and RocksDB to avoid reinventing the wheel. The codebase is open-source that contains approximately 41,000 lines of C++ code (with around 4,500 lines of code from RocksDB and 10,500 lines of code from LevelDB).

B. *RDMA Manager*

How to efficiently utilize *ibverbs* 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 *ibverbs*. All the resources for dLSM's RDMA, e.g., the queue pair, the completion queue, the protect domain, and registered memory are managed by the

RDMA manager. The RDMA manager translates read/write operations and RPCs into RDMA primitives. It provides the interfaces for registering the memory, connecting a queue pair, and invoking RDMA primitives (e.g., RDMA read, RDMA write, RDMA send, RDMA receive, and RDMA atomic). Both the compute and memory nodes build up their functions based on the RDMA manager.

For each RDMA operation, both the source and destination memory buffers are registered into the NIC through *ibv_reg_mr*. The registration pins the memory to prevent it from being swapped. Performing frequent RDMA registrations by small blocks can introduce non-negligible overhead. Thus, dLSM pre-registers large memory regions for both remote and local memories, and then reallocates memory in the user space.

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. In dLSM, every thread creates a thread-local queue pair and registered buffer in the RDMA manager to perform one-sided RDMA. Thus, threads do not collide with each other when polling the completion from the same completion queue, and no synchronization over the buffer is needed to transfer the data. dLSM's RDMA manager is generic, and can be used by other memory disaggregated systems.

C. Asynchronous IO 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 til 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, 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. Figure 6 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.

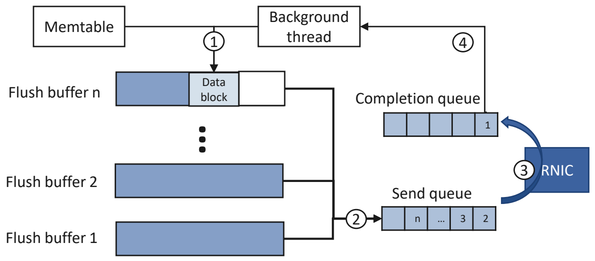


Fig. 6: Efficient flushing in dLSM

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.

D. 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.

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.

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., 10s 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., 100s to 1000s 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.

XI. Experiments

A. 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.

RocksDB relies on a file system that interacts with the underlying storage device to perform common file operations. To port RocksDB over RDMA, we implement an RDMA-oriented file system to perform data reads and writes over remote memory instead of local storage. Write-ahead logging is disabled for fair comparison (see Sec. VIII).

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 [43]. This baseline is an optimized LSM-tree for storage disaggregation (instead of memory disaggregation). We use Nova-LSM's available source code [43]. 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. For Nova-LSM to achieve high performance, the compute node contains multiple sub-ranges that allow concurrent background compaction. Besides that, write-ahead logging is also disabled.

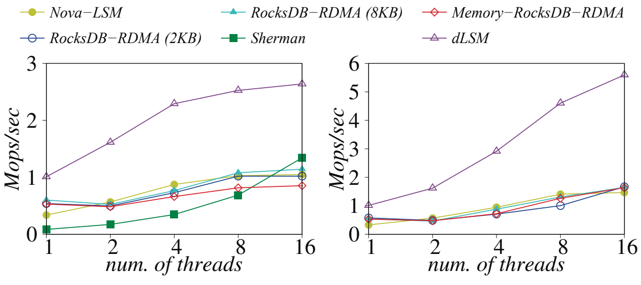
Baseline #5: Disaggregated B-tree (Sherman [71]). The last baseline is a highly optimized B-tree termed Sherman [71] for the memory disaggregated architecture. We use Sherman's available source code [71].

B. 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 [36] (as in Sec. XI-C8).

Datasets. We run the standard benchmark "db_bench" of RocksDB. We insert 100 million random key-value pairs in each system. 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. Unless otherwise stated, dLSM is configured to have 1 shard. For



(a) Normal mode (b) Bulkload mode

Fig. 7: Evaluating write performance

Nova-LSM [43], the subrange is 64 to maximize concurrency in background compaction. In Sherman [71], we follow the default block size (1KB) in the source code. To minimize RDMA remote accesses, we follow [71] to cache the internal nodes of the B-tree in local memory.

C. Results

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 experiment, 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 7(a) gives the write throughput with different number 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× ∼ 3.1× faster than RocksDB-RDMA (8KB), 1.6× ∼ 3.9× faster than RocksDB-RDMA (2KB), 1.9× ∼ 3.5× faster than Memory-RocksDB-RDMA, 2.5× ∼ 3.0× faster than Nova-LSM, and 1.8× ∼ 11.7× faster than Sherman [71]. 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 [71] 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.

Figure 7(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. IV). Specifically, dLSM is up to 4.6× faster than RocksDB-RDMA (8KB), 4.0× faster than RocksDB-RDMA (2KB), 3.5× faster than Memory-RocksDB-RDMA, and 3.8× faster than Nova-LSM. Note that Sherman [71] is not applicable to this mode.

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. Figure 8 gives the results for various numbers of threads. dLSM outperforms all other LSM-tree solutions. The reason is that dLSM is optimized for byte-addressable remote memory (See Sec. VI). 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 [71], dLSM has slightly worse read performance (up to 12.5% when the number of threads is 16). This is expected, because Sherman is a B+-tree that only involves one RDMA access for every read by caching all the internal nodes in the local memory. In contrast dLSM, being LSM-based, may issue more than one RDMA operations depending on the levels of the LSM-tree, although it uses bloom filters to skip some RDMA accesses. However, dLSM achieves 1.8× ∼ 11.7× faster writes than Sherman [71] (Figure 7a), which shows a good tradeoff.

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 9 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

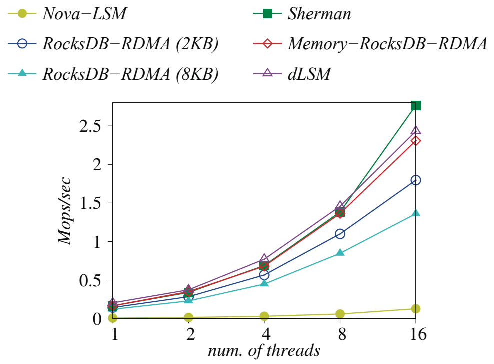
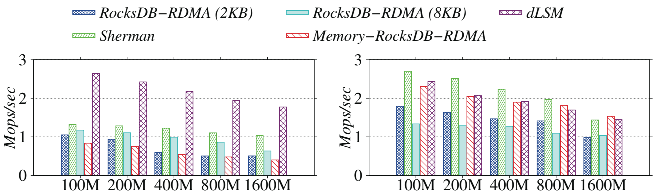


Fig. 8: Evaluating read performance



(a) Random fill (write)

(b) Random read

Fig. 9: Evaluating varied data sizes

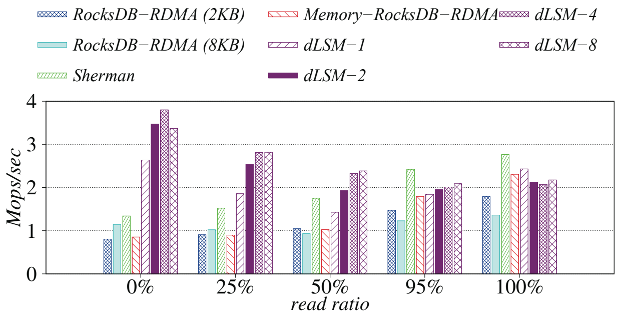


Fig. 10: Evaluating mixed read-write performance

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 Section XI-C8). 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.

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. VII, dLSM can have different number of shards. We use dLSM-λ to indicate that dLSM uses λ shards.

Figure 10 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% and 100%, which is expected since dLSM is write-optimized. It also shows that sharding improves the performance on the mixed workloads. For example, when the read ratio is 50%, dLSM-8 is 1.7× faster than dLSM-1 because there is more parallelism for Level 0 compaction and readers only need to search a smaller number of SSTables in Level 0.

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 [71] uses the cached internal nodes to accelerate the sequential read for table scan. We omit the result of Nova-LSM in this experiment due to a bug on the range index for Nova-LSM. The results in Figure 11 demonstrate that dLSM outperforms RocksDB-RDMA (8KB) by 1.3×, RocksDB-RDMA (2KB) by 1.5×, Memory-RocksDB-RDMA by 2.5× and Sherman by 1.8×. 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 [71], 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.

6) Evaluating Compaction: In this experiment, we study the impact of near-data compaction and its CPU utilizations in dLSM. We run the "randomfill" benchmark under normal mode with different remote CPU utilization and remote CPU cores. We also test the performance under different pressures of front-end insertion. We compare the results of performing

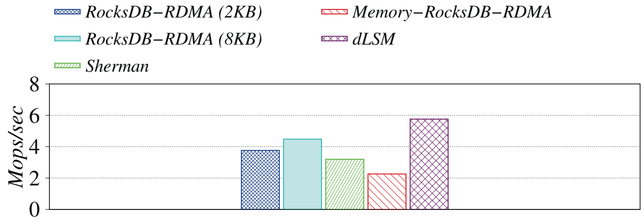


Fig. 11: Evaluating range query performance

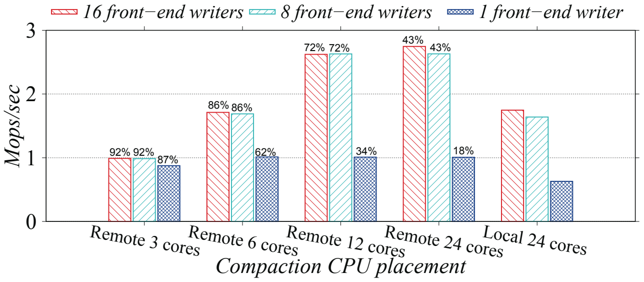
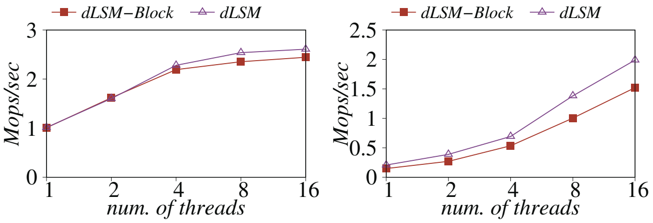


Fig. 12: The impact of remote CPU cores on near-data compaction

compaction in the compute node vs. memory node to demonstrate the effect of near-data compaction (Sec. V). Figure 12 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 the 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. As we add more remote computing power, performance enhances, 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%.

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. VI, 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 Figure 13, dLSM is faster than dLSM-Block for both writes and reads, especially for reads (up to 60% improvement). The reason is that byte-addressability can directly fetch a single key-value pair without accessing a whole block. Write performance improves due to eliminating unnecessary data copy once the notion of "block" is removed.

8) 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. IX. We use *CloudLab*[3] [36] 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.



(a) Random fill (write)

(b) Random read

Fig. 13: Evaluating byte-addressable SSTable

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 Section XI-C3, in which data is increased within a single server. From Figure 14(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 (Section XI-C3). In Figure 14(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.

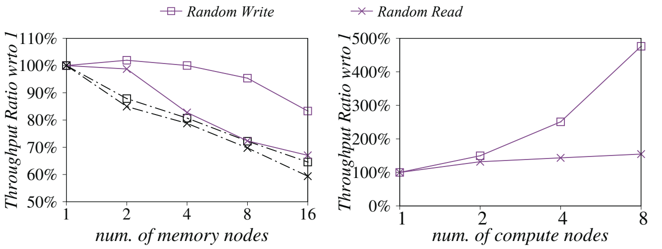
In the second experiment, we fix the number of memory nodes to 1 and scale out the compute resources from 1 to 8 nodes. We set the data size to 50 million key-value pairs. From Figure 14(b), writing has better scalability than reading. The reason is that the sequential I/Os for writes can utilize more 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 15 gives the results, indicating that dLSM scales well for multiple nodes and dLSM achieves better performance compared to the other competitors.

XII. Related Work

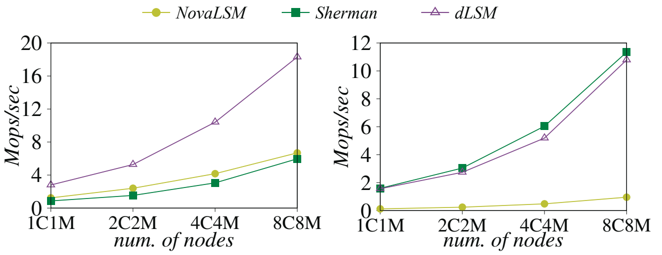
Resource Disaggregation. Resource disaggregation offers great benefits in data centers for cost efficiency, resource utilization, and elasticity. Achieving good performance in disag-

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



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

Fig. 14: Evaluating scalability



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

Fig. 15: Evaluating multi-node design

gregated architectures requires redesign of many aspects, e.g., operating systems [65], hardware [52], and networking [39], [41]. dLSM focuses on the data indexing aspect. Besides that, industrial efforts, e.g., IBM Cloud [1], Intel RSD [5], and HP ”The Machine” [46], have realized resource disaggregation in production hardware and systems.

Databases for Disaggregated Architectures. Database systems require significant rethinking to leverage disaggregated architectures. Cloud-native databases (OLTP and OLAP) are re-designed to follow this trend, e.g., Aurora [68], PolarDB [26], Socrates [14], Taurus [29], Snowflake [28], and FlexPushdownDB [77] 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., [81], [82] study the impact of memory disaggregation on OLAP databases (both disk-based and memory-based) [81], [82] and report significant performance degradation that motivates further optimization as shown in [83]. Farview [48] 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], [84] with index prefetching, optimistic locking, and optimized recovery. Zuo et al. [86] develop a hash index for disaggregated memory but it cannot support range queries as in dLSM. Sherman [71] 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], [48], [71], [81], [82], [84]. Experiments show that dLSM achieves much faster write performance over Sherman [71] while offering comparable read performance (Figure 7 and Figure 8).

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 [50] and remote cache [80] to improve query processing [19], [64], B-tree [85], hashing [57], transactions [78], and enhancing availability [79]. 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], [35], [37], [49], [58], [61], [67]. 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.

LSM-tree for Non-volatile Memory. 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], [53], [54], [73], [76]. 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 [33], [44], [70], they can perform very limited offloading. (2) dLSM has RDMA-specific optimizations that those works do not have.

XIII. Conclusion

In this paper, we investigate realizing an LSM-based index over disaggregated memory. dLSM utilizes several optimizations 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. As can be seen from this study and the optimizations introduced, ultra-fast communication technologies play an important role in the performance and optimization of indexes over disaggregated memory. It is important to abstract communication properties further, and study their effects on index realization over disaggregated memory.

Acknowledgment

Walid Aref acknowledges the support of the National Science Foundation under Grant Number IIS-1910216. M. Tamer Özsu’s research is supported by a grant from NSERC Canada.

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] Oracle Exadata , https://www.oracle.com/engineered-systems/exadata/.

[9] Remote Compactions in RocksDB-Cloud, https://rockset.com/blog/remote-compactions-in-rocksdb-cloud/.

[10] RocksDB, http://rocksdb.org/.

[11] VoltDB, https://www.voltdb.com/.

[12] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote Memory in the Age of Fast Networks. In *Proceedings of the Symposium on Cloud Computing (SoCC)*, pages 121–127, 2017.

[13] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment (PVLDB)*, 7(14):1905–1916, 2014.

[14] P. Antonopoulos, A. Budovski, C. Diaconu, A. H. Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The New SQL Server in the Cloud. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1743–1756, 2019.

[15] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 11(5):553–565, 2018.

[16] R. Balasundaram. Cuckoo Hashing Table Format (http://rocksdb.org/blog/2014/09/12/cuckoo.html), 2014.

[17] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK: preventing latency spikes in log-structured merge key-value stores. In *USENIX Annual Technical Conference (ATC)*, pages 753–766, 2019.

[18] C. Barthels, G. Alonso, and T. Hoefler. Designing Databases for Future High-Performance Networks. *IEEE Database Engineering Bulletin*, 40(1):15–26, 2017.

[19] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-Scale In-Memory Join Processing using RDMA. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1463–1475, 2015.

[20] L. Benson, H. Makait, and T. Rabl. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proceedings of the VLDB Endowment (PVLDB)*, 14(9):1544–1556, 2021.

[21] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1–10, 1995.

[22] L. Bindschaedler, A. Goel, and W. Zwaenepoel. 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)*, pages 301–316, 2020.

[23] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment (PVLDB)*, 9(7):528–539, 2016.

[24] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K. Tan, Y. M. Teo, and S. Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment (PVLDB)*, 11(11):1604–1617, 2018.

[25] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 29–41, 2020.

[26] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. 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] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 2477–2489, 2021.

[28] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 215–226, 2016.

[29] A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao, and Y. He. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1463–1478, 2020.

[30] D. J. DeWitt, S. Ghandeharizadeh, and D. A. Schneider. A Performance Analysis of the Gamma Database Machine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 350–360, 1988.

[31] D. J. DeWitt and P. B. Hawthorn. A Performance Evaluation of Data Base Machine Architectures (Invited Paper). In *International Conference on Very Large Data Bases (VLDB)*, pages 199–214, 1981.

[32] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1243–1254, 2013.

[33] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1221–1230, 2013.

[34] S. Dong. PlainTable – A New File Format (http://rocksdb.org/blog/2014/06/23/plaintable-a-new-file-format.html), 2014.

[35] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[36] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019.

[37] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *International Conference on Data Engineering (ICDE)*, pages 1477–1488, 2020.

[38] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 149–160, 1996.

[39] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–264, 2016.

[40] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In L. Réveillère, T. Harris, and M. Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, pages 32:1–32:14, 2015.

[41] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 649–667, 2017.

[42] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment (PVLDB)*, 13(12):3072–3084, 2020.

[43] H. Huang and S. Ghandeharizadeh. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 749–763, 2021.

[44] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent Distributed Storage. *Proceedings of the VLDB Endowment (PVLDB)*, 10(11):1202–1213, 2017.

[45] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, 2016.

[46] K. Keeton. Memory-Driven Computing. https://www.usenix.org/sites/default/files/conference/protected-files/fast17_slides_keeton.pdf. In *USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[47] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, 1998.

[48] D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. S. Milojicic, and G. Alonso. Farview: Disaggregated Memory with Operator Offloading for Database Engines. In *Conference on Innovative Data Systems Research (CIDR)*, 2022.

[49] H. A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 317–330, 2019.

[50] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 355–370, 2016.

[51] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation Memory Disaggregation for Cloud Platforms. *CoRR*, abs/2203.00241, 2022.

[52] K. T. Lim, J. Chang, T. N. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2009.

[53] J. Liu, S. Chen, and L. Wang. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 13(7):1078–1090, 2020.

[54] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 13(8):1147–1161, 2020.

[55] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 37–50, 2017.

[56] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking Main Memory OLTP recovery. In *International Conference on Data Engineering (ICDE)*, pages 604–615, 2014.

[57] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference (ATC)*, pages 103–114, 2013.

[58] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference (ATC)*, pages 291–305, 2015.

[59] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[60] I. L. Picoli, P. Bonnet, and P. Tözün. LSM Management on Computational Storage. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 17:1–17:3, 2019.

[61] M. Pröbstl, P. Fent, M. E. Schüle, M. Sichert, T. Neumann, and A. Kemper. 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)*, pages 17–26, 2021.

[62] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM (CACM)*, 33(6):668–676, 1990.

[63] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *International Conference on Very Large Data Bases (VLDB)*, pages 62–73, 1998.

[64] A. Salama, C. Binnig, T. Kraska, A. Scherp, and T. Ziegler. Rethinking Distributed Query Execution on High-Speed Networks. *IEEE Data Engineering Bulletin*, 40(1):27–37, 2017.

[65] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 69–87, 2018.

[66] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1493–1509, 2020.

[67] K. Taranov, S. D. Girolamo, and T. Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1811–1824, 2021.

[68] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM Conference on Management of Data (SIGMOD)*, pages 1041–1052, 2017.

[69] K. Voruganti, M. T. Özsu, and R. C. Unrau. An Adaptive Hybrid Server Architecture for Client Caching ODBMSs. In *International Conference on Very Large Data Bases (VLDB)*, pages 150–161, 1999.

[70] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson. SSD In-storage Computing for List Intersection. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 4:1–4:7, 2016.

[71] Q. Wang, Y. Lu, and J. Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1033–1048, 2022.

[72] R. Wang, J. Wang, S. Idreos, M. T. Özsu, and W. G. Aref. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)*, 16(1):15–22, 2022.

[73] T. Wang, J. J. Levandoski, and P. Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

[74] Y. Wu, W. Guo, C. Chan, and K. Tan. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 267–281, 2017.

[75] M. Xiao, H. Wang, L. Geng, R. Lee, and X. Zhang. Catfish: Adaptive RDMA-enabled R-Tree for Low Latency and High Throughput. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 164–175, 2019.

[76] B. Yan, X. Cheng, B. Jiang, S. Chen, C. Shang, J. Wang, K. Huang, X. Yang, W. Cao, and F. Li. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 14(10):1872–1885, 2021.

[77] Y. Yang, M. Youill, M. E. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proceedings of the VLDB Endowment (PVLDB)*, 14(11):2101–2113, 2021.

[78] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 511–526, 2020.

[79] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska. Rethinking Database High Availability with RDMA Networks. *Proceedings of the VLDB Endowment (PVLDB)*, 12(11):1637–1650, 2019.

[80] Q. Zhang, P. A. Bernstein, D. S. Berger, and B. Chandramouli. Redy: Remote Dynamic Memory Cache. *Proceedings of the VLDB Endowment (PVLDB)*, 15(4):766 – 779, 2022.

[81] Q. Zhang, Y. Cai, S. Angel, V. Liu, A. Chen, and B. T. Loo. Rethinking Data Management Systems for Disaggregated Data Centers. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[82] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9):1568–1581, 2020.

[83] Q. Zhang, X. Chen, S. Sankhe, Z. Zheng, K. Zhong, S. Angel, A. Chen, V. Liu, and B. T. Loo. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1345–1359, 2022.

[84] Y. Zhang, C. Ruan, C. Li, J. Yang, W. Cao, F. Li, B. Wang, J. Fang, Y. Wang, J. Huo, and C. Bi. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)*, 14(10):1900–1912, 2021.

[85] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 741–758, 2019.

[86] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *USENIX Annual Technical Conference (ATC)*, pages 15–29, 2021.