Efficient, Scalable, and Fair Locking on Disaggregated Memory with Decentralized Coordination

Hanze Zhang
Shanghai Jiao Tong University
hanzezhang@sjtu.edu.cn

Ke Cheng
Shanghai Jiao Tong University
chengke608@gmail.com

Rong Chen
Shanghai Jiao Tong University
rongchen@sjtu.edu.cn

Xingda Wei
Shanghai Jiao Tong University
wxdwfc@sjtu.edu.cn

Haibo Chen
Shanghai Jiao Tong University
haibochen@sjtu.edu.cn

ABSTRACT

Databases on disaggregated memory (DM) rely heavily on locking for concurrency control. However, we find that under contention, existing lock implementations can significantly degrade database performance because they overload the network interface controllers (NICs) of memory nodes (MNs) and provide poor fairness among competing clients on compute nodes (CNs).

This paper presents DecLock, an efficient, scalable, and fair locking mechanism for DM. DecLock decouples centralized *state maintenance* on MNs from decentralized *ownership transfer* across CNs. Its cooperative queue-notify locking atomically queues waiters on MNs and then transfers lock ownership through direct message-based notifications between CNs, rather than repeated retries to MNs. This design preserves precious MN-NIC resources for data access while ensuring fair lock handoff. Evaluations show that DecLock improves throughput by up to 43.37×, 4.35×, and 1.81× over state-of-the-art RDMA-based spinlock, ticket lock, and MCS lock, respectively. Moreover, DecLock helps a NoSQL data store, a transaction engine, and a real-world database index avoid severe performance degradation under high contention, improving throughput by up to 1.48×, 1.59×, and 2.31× over prior solutions, respectively.

PVLDB Reference Format:
Hanze Zhang, Ke Cheng, Rong Chen, Xingda Wei, and Haibo Chen. Efficient, Scalable, and Fair Locking on Disaggregated Memory with Decentralized Coordination. PVLDB, 19(9): 2248 - 2261, 2026.
doi:10.14778/3819518.3819548

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dracit7/declock.

1 INTRODUCTION

The DM architecture, which detaches CPU and memory resources into physically disaggregated CNs and MNs, attracts growing interest from both academia [10, 12, 21, 26, 28, 37] and industry [2, 38, 50]. Beyond academic efforts to redesign database indices [29, 30, 44] and transaction engines [53, 54] for DM, production databases such as PolarDB [11, 50] have adopted the DM architecture to independently scale CPU and memory resources. In practice, data on a single MN can be accessed concurrently by hundreds of clients across multiple CNs [29, 30, 38], making the NICs on MNs (referred to as **MN-NICs**) a performance bottleneck for DM-based databases.

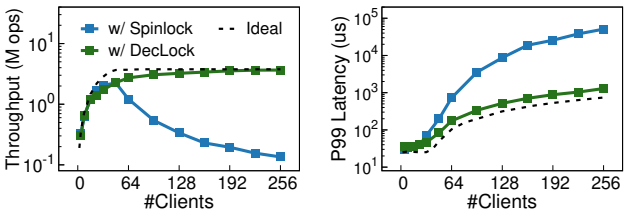

Fig. 1: ***Throughput and tail latency of update operations in a DM database index using spinlock and DecLock as #clients increases.*** **Workload:** ***10 million objects w/ a Zipf access distribution.*** **Testbed:** ***8 CNs and 1 MN, connected with 100 Gbps RDMA NICs.***

DM databases extensively utilize locks stored on MNs to serialize conflicting data accesses from different CNs [4, 27, 29, 30, 43, 44, 46, 54]. Database clients on CNs acquire and release these locks by directly manipulating lock states on MNs with remote operations (e.g., one-sided RDMA). Since locks are colocated with the data they protect, lock operations compete with data accesses for MN-NICs, amplifying the performance impact of locks on DM databases. Under high-contention workloads—common due to large numbers of concurrent clients [29, 30, 38]—lock operations can easily saturate MN-NIC IOPS, drastically damaging database performance.

To demonstrate the performance degradation caused by inefficient locking, we study the performance of update operations in a database index on DM. We simulate the ideal performance by running all clients on a single machine as coroutines, serializing operations using local locks that incur negligible overhead compared to RDMA locks. As shown in Fig. 1, when using RDMA spinlocks, the operation throughput collapses to below 5% of the ideal as the number of clients scales to 256, and the tail latency surges to 69× of the ideal.

This significant overhead stems from applying the fail-and-retry locking approach—specifically, spinlocks—in DM environments [27, 30, 44, 54]. To acquire a spinlock, clients must repeatedly retry remote operations to check or update lock states on MNs until they obtain lock ownership, leading to **inefficient** use of MN-NIC IOPS. As the number of clients increases, retries become more frequent due to longer wait times, making databases **unscalable**. For example, in the index evaluated in Fig. 1, with 256 clients, each client averages 28 retries to acquire a lock, consuming over 95% of MN-NIC IOPS. Spinlocks are also notoriously **unfair**, which greatly exacerbates the number of retries for a few unlucky clients, leading to poor tail latency. Even applying classic optimizations from multicore systems [6, 19, 24, 31, 51] cannot fully address these three issues.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819548

Key idea. We argue that *locking should minimize MN-NIC usage to prevent performance interference with data accesses in disaggregated memory environments.* Following this principle, we decouple the costly ownership transfer procedure from lock state maintenance. This approach enables **efficient** MN-NIC usage, requiring at most two remote operations to acquire a lock. Ownership transfer leverages decentralized coordination across CNs to enhance **scalability** as the number of clients increases. Meanwhile, lock states like the identity of waiters remain centralized on MNs to ensure **fairness**.

Our approach. We introduce DecLock, an efficient, scalable, and fair reader-writer lock that serializes conflicting DM data accesses with near-ideal performance (see Fig. 1). The core design is *cooperative queue-notify locking* (CQL), which enables lock clients to *enqueue* themselves on MNs and *notify* each other between CNs. When acquiring a lock, the client enqueues itself to a centralized array (on MNs) and waits for notification from the client that releases the lock (between CNs). When releasing the lock, the client dequeues itself, retrieves information of the next waiter (from MNs), and notifies this waiter to transfer lock ownership (between CNs).

A centralized queue recording waiters is essential for maintaining fairness between readers and writers, but manipulating both the lock mode and the queue in a single atomic operation is challenging. To address this, the CQL protocol splits the queue into an *atomic control plane* and a *non-atomic data plane*. The control plane encodes the queue metadata and the lock mode into an atomic header, allowing clients to (de)allocate a queue entry and update the lock mode using a single fetch-and-add (FAA) operation. Since FAA operations always succeed, this design ensures that clients obtain the lock in a FIFO manner. Although the data plane is non-atomic, the atomic allocation of entries ensures that clients always write to different entries in the data plane, eliminating write-write conflicts. Additionally, the protocol incorporates versions within queue entries to identify invalid entries, resolving read-write conflicts.

Although CQL efficiently uses MN-NIC IOPS and preserves fairness, its centralized queue design can create significant memory and bandwidth overhead on MNs for locks acquired by many clients. Hierarchical locking reduces queue size by enqueuing waiters at the CN level rather than for individual clients. However, existing hierarchical designs compromise fairness between waiters on different CNs. To address this, we employ *timestamp-based hierarchical locking*, which records the acquisition time of each waiting CN in its queue entry. When a lock is released, ownership is transferred locally only if local waiters have an earlier acquisition time than remote CNs, thereby preserving fairness across CNs.

We evaluated DecLock through a microbenchmark and three DM database components: a NoSQL data store, a transaction engine, and a real-world database index (Sherman [44]). Experimental results show that DecLock achieves throughput improvements of up to 43.37×, 4.35×, and 1.81×, while reducing tail (99^{th}-percentile) latency by up to 98.2%, 67.8%, and 43.7%, compared to state-of-the-art RDMA-based spinlock [13], ticket lock [51], and MCS lock [19], respectively. Furthermore, DecLock effectively mitigates throughput collapse in DM database components, improving throughput for the NoSQL store, transaction engine, and database index by up to 1.48×, 1.59×, and 2.31×, respectively.

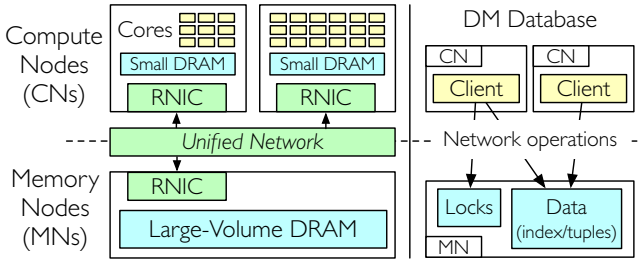

Fig. 2: ***The architecture of DM (left) and DM-based databases (right).***

Contributions. We summarize our contributions as follows:

- A detailed analysis of the performance implications of RDMA-based locks on disaggregated memory (§2).
- A new CQL protocol that enables efficient, scalable, and fair locking on DM by minimizing the usage of MN-NICs (§4).
- A *timestamp-based hierarchical locking* design that allows local ownership transfers while ensuring fairness across CNs (§5).
- An evaluation demonstrating the performance advantages that DecLock offers to DM databases (§6).

2 BACKGROUND AND MOTIVATION

2.1 DM Architecture and Systems

Disaggregated memory (DM) is gaining increasing traction in modern data centers [2, 12, 21, 28, 37, 38, 50]. DM decouples CPUs and memory into CNs and MNs, improving scalability and resource utilization (see Fig. 2 (left)). CNs host powerful CPUs for computation and only a small amount of DRAM (typically a few GB [30, 39]) that serves as a cache, whereas MNs provide large memory capacity for data storage but offer negligible computing power. CNs and MNs are typically equipped with RDMA-capable NICs (RNICs) and connected through a unified high-speed network, enabling direct communication between any pair of nodes [2, 20, 36, 37].

Numerous in-memory database components have been redesigned for DM, including transaction engines [48, 53, 54], NoSQL data stores [25, 38, 39, 43, 55], and tree-based indices [4, 27, 30, 44, 46]. Due to the lack of computing power on MNs, these systems heavily rely on CPU-bypassing techniques (e.g., one-sided RDMA) to directly access data in the MN memory (see Fig. 2 (right)). In practice, RNICs on hotspot MNs often bear significant loads generated by hundreds of clients on multiple CNs, rendering their processing power (IOPS) a bottleneck for DM databases [29, 30, 35, 38].

To serialize conflicting data accesses, DM databases rely on locks to protect shared data [27, 29, 30, 43, 44, 50, 54]. These locks are typically stored on MNs alongside the data they protect, allowing for piggybacking of lock operations and data accesses. Database clients acquire and release these locks through atomic remote operations, such as RDMA compare-and-swap (CAS) and fetch-and-add (FAA),

before and after accessing the protected data. Since lock operations are on the critical path and contend with data accesses on MN-NICs, they can drastically degrade DM database performance.

2.2 Performance Issues of DM Spinlock

Existing DM databases commonly adopt the *spinlock* design for its simplicity [29, 30, 44, 50, 53, 54]. The spinlock state is a 64-bit value that is either 0 (indicating the lock is *free*) or a non-zero ID representing the client holding the lock (i.e., *lock holder*). Clients acquire the lock by swapping the value with their ID using CAS operations, which succeed only when the original value is 0. This ensures at most one client holds the lock at a time, while other clients (i.e., *lock waiters*) must blindly retry the CAS operation until they obtain lock ownership. When the holder releases the lock by setting the value to 0, ownership transfers randomly to whichever client successfully swaps the value first. Our microbenchmark experiments demonstrate that under high contention workloads, this fail-and-retry mechanism severely impedes both performance and fairness.

Issue #1: MN-NIC contention. When a spinlock is held, waiters must repeatedly retry the CAS operation on the lock state to gain ownership. As more clients compete for locks, the average number of CAS retries per acquisition rises to up to 37.8 due to extended wait times (see Fig. 3 (left)). These retries contend with other RDMA operations for limited MN-NIC IOPS resources, leading to higher median latency for both lock acquisitions and remote data accesses (see Fig. 3 (middle)). Consequently, acquisition throughput drops sharply from a peak of 1.61 M ops to merely 0.20 M ops, which explains the throughput collapse observed in the database index.

Surprisingly, adding more MNs does not alleviate the MN-NIC contention issue. This is because real-world workloads typically follow a Zipfian distribution [5, 14, 49], where a few "hot" locks receive most accesses. In our microbenchmark with this distribution, over 95% of CAS retries target the hottest lock. This makes the hotspot MN a bottleneck regardless of how many MNs exist or how locks are partitioned. Consequently, spinlock throughput remains flat even when scaling up the number of MNs (details in §6.2).

Issue #2: Poor fairness. Fairness among lock clients refers to whether clients can obtain lock ownership in the same (or a similar) order as they initiated acquisition requests. CAS-based spinlocks cannot enforce fairness, as their ownership can be obtained by any client attempting to acquire the lock when it is released. This lack of fairness leads to (99^{th}-percentile) tail latency that is over 1000× higher than the median latency (see Fig. 3 (middle)), which explains the substantial tail latency observed in the database index.

2.3 Existing Scalable and Fair Locks

Previous research has applied classic lock algorithms to improve the scalability and fairness of RDMA locks [6, 16, 19, 35, 51]. However, none of them fully addresses the above locking issues on DM.

Ticket locks track the order of clients using a ticket value that atomically increments with each acquisition [24, 51]. These locks improve scalability by maintaining separate tickets for readers and writers, allowing multiple readers to hold the lock simultaneously. They also provide optimal fairness by ensuring readers and writers obtain ownership in the exact order of their acquisition requests (i.e., *task-fair* [7]). However, waiters still need to repeatedly poll the latest ticket value from MNs, leading to MN-NIC contention. Although adding delays between reads (i.e., *backoff* [1, 35]) can mitigate this contention, properly tuning the backoff interval for varying contention levels is notoriously hard [34].

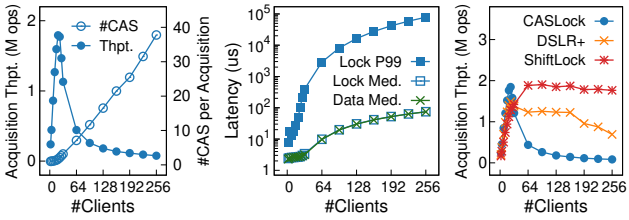

Fig. 3: ***Acquisition throughput of spinlock and the average number of CAS operations per acquisition (left), Median and 99^{th}-percentile latency of spinlock acquisition and median data access latency (middle), and Acquisition throughput of different lock mechanisms (right). Detailed experimental setup can be found in §6.***

To demonstrate this issue, we integrated the latest backoff approach for DM [35] into the state-of-the-art RDMA ticket lock (DSLR [51]). We compared this enhanced version (DSLR+) against CAS-based spinlocks (CASLock) using a workload with equal readers and writers. As shown in Fig. 3 (right), DSLR+ maintains stable throughput with up to 160 clients because its read operations consume fewer MN-NIC IOPS than CAS operations. However, DSLR+ still saturates MN-NIC IOPS and suffers throughput degradation when adding more clients.

MCS locks record waiters in a linked list, with the tail pointer embedded in the lock state [6, 19, 31]. Recent work like ALock [6] and ShiftLock [19] also identifies NIC contention issues in RDMA locks and adopts MCS locks to reduce NIC usage. After joining the list, waiters can request lock ownership directly from their predecessors instead of repeatedly polling lock states on MNs. This approach significantly reduces MN-NIC usage and prevents throughput collapse compared to CASLock and DSLR+, as shown in Fig. 3 (right).

However, linked waiters spread across CNs cannot see each other, preventing readers from joining the list without blocking other readers [19]. ShiftLock addresses this by using a counter in the lock state to track readers, but this still requires repeated polling when waiting for ownership. To minimize these checks, ShiftLock transfers lock ownership to all waiting readers after several consecutive transfers between writers. While this reduces wait times for readers, it compromises fairness (*phase-fair* [7]) and introduces significant latency variance in mixed read-write workloads [8]. Despite these efforts, ShiftLock still requires an average of 2.34 checks per acquisition, limiting throughput when MN-NIC IOPS becomes saturated (see Fig. 3 (right)). This issue becomes more severe with longer critical sections (e.g., transaction processing [42, 54]), potentially requiring dozens of checks and further increasing MN-NIC contention.

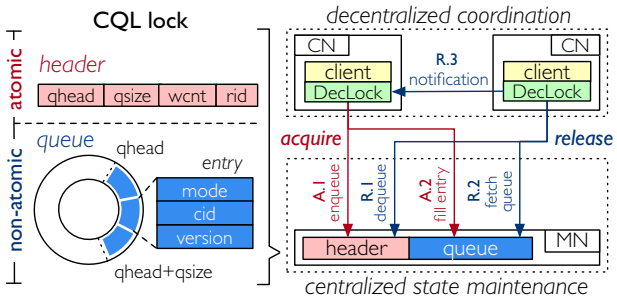

Fig. 4: ***Architecture, data structure, and workflow of DecLock.***

3 APPROACH AND OVERVIEW

MN-NICs, which execute remote operations from numerous clients, have become a significant performance bottleneck in DM databases. Therefore, we propose a fundamental design principle for the lock service on DM: *minimize the use of MN-NICs to reduce contention with data accesses*.

Our approach. Following this principle, we propose cooperative queue-notify locking (CQL), which decouples *ownership transfer* from *state maintenance*. CQL completely eliminates repeated operations to MNs while preserving strong fairness between waiting readers and writers. Lock states are maintained as a *centralized queue* on MNs that orders all waiting clients, enabling ownership transfer through *decentralized coordination*—directly notifying waiters via messages between CNs rather than retries towards MNs.

CQL requires maintaining multiple lock states, including lock mode (free/shared/exclusive), queue metadata, and queue entries, which cannot be updated with a single remote atomic operation. To tackle this challenge, CQL locks are split into an **atomic control plane** (a 64-bit header) and a **non-atomic data plane** (a circular queue of clients), as shown in Fig. 4 (left). The control plane records the used portion of the queue (`qhead` and `qsize`) and the number of writers in the queue (`wcnt`), along with a reset ID (`rid`) for exception handling. The header's value implicitly represents the lock mode: the lock is free if `qsize==0`, shared if `qsize>0` and `wcnt==0`, and exclusive if `qsize>0` and `wcnt>0`. The data plane consists of multiple queue entries, each storing the acquisition mode (`mode`), the client ID (`cid`), and a `version` for resolving entry-level race conditions (details in §4.3).

As illustrated in Fig. 4 (right), the client acquires the lock using an FAA operation on the header, which atomically enqueues the client and returns the original header value to determine the lock mode (A.1). If the client needs to wait for lock ownership, it then fills its information into the queue entry (A.2). Upon releasing the lock, the client dequeues itself (R.1), fetches the queue (R.2), and proactively notifies waiting clients in the queue to transfer lock ownership (R.3).

CQL is novel in three ways. First, for **efficiency**, it moves the costly lock ownership transfer off the CN-MN interconnect. Locks can be acquired with at most two remote operations to MNs—typically just one—substantially reducing MN-NIC IOPS compared to existing RDMA locks. Second, for **scalability**, its decentralized coordination needs only one message between CNs for each ownership transfer. Third, for **fairness**, both readers and writers are enqueued and notified strictly in their order of acquisition.

System architecture. DecLock is a lock mechanism that implements the CQL lock abstraction and provides lock operation APIs, assuming that databases are non-malicious and properly use its API for lock operations. DM databases store CQL locks in MN memory and associate them with protected shared data. DecLock updates the CQL lock state through RDMA operations to MNs while transferring lock ownership through two-sided RDMA notifications between CNs. Although Fig. 4 shows only one MN, DecLock scales to multiple MNs by assigning a subgroup of CQL locks to each MN. Each lock is exclusively managed by one MN and identified jointly by the MN ID and its remote address on that MN.

4 CQL PROTOCOL

4.1 Lock Header Encoding

CQL encodes queue metadata and lock states in a 64-bit atomic header that records the queue head (`qhead`), size (`qsize`), writer count (`wcnt`), and reset ID (`rid`).

Field choice and order. DecLock tracks the occupied portion of the circular queue using head and size rather than head and tail because size has a fixed upper limit (i.e., the queue capacity), while both head and tail increment infinitely. Since field overflows in less significant bits would corrupt the content in more significant bits, the most significant field in the header can safely overflow. Therefore, `qhead` occupies the most significant bits, ensuring its overflows do not affect other fields. The next bits contain `qsize` and `wcnt`, both having limited maximum values. The least significant bits serve as `rid`, a reset ID that indicates whether the lock is being reset due to exceptions like entry overwrites (details in §4.4).

Field size. Both `qsize` and `wcnt` fields are allocated the same number of bits (`N`), as the number of writers never exceeds the queue size. `N` is larger than the bits required to represent queue capacity (e.g., `N=12`), ensuring that `qsize` and `wcnt` do not overflow even if queue size temporarily exceeds capacity (i.e., lock queue overflow). Such overflow is quickly detected and fixed by the reset process. `qhead` keeps increasing and is wrapped within the queue capacity when used as the queue index. In addition to the bits for indexing, `qhead` contains extra bits that indicate how many times the queue buffer has been traversed. Therefore, `qhead` is allocated more bits (`M`) than `qsize` and `wcnt` (e.g., `M=32`). Finally, the reset ID is allocated with sufficient bits (`K`) to identify every CN (e.g., `K=8`).

Field manipulation. During lock operations, a requester may need to update multiple header fields with a single FAA. This is done by constructing a 64-bit addend with 1 in the fields to increment and 0 elsewhere. Decrements are implemented by adding the two's complement of the addend. Any resulting overflow can be canceled by subtracting 1 from the adjacent more significant field.

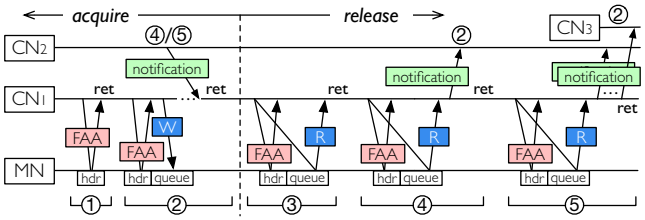

Fig. 5: ***Workflows of acquiring and releasing a CQL lock. "FAA", "W", and "R" denote RDMA FAA, WRITE, and READ operations.***

4.2 Lock Operations

Fig. 5 illustrates the five possible workflows of acquiring (①–②) and releasing (③–⑤) a CQL lock. The branches executed in each workflow are marked on the pseudocode in Fig. 6. For brevity, we assume all queue entries contain up-to-date content when read. The mechanism for correcting obsolete entries is detailed in §4.3.

Lock acquisition workflow. To acquire a lock, the client first enqueues itself by issuing an FAA on the lock header, which atomically increments `qsize` by one (Line 1). For exclusive acquisition (i.e., when `EX(mode)` is 1), the FAA also increments `wcnt` by one. The client then examines the original header value returned by the FAA to determine whether it obtains lock ownership immediately. If the request is exclusive and there are other queued clients, or if there are writers in the queue, the client becomes a *waiter* (②). It then populates its queue entry with its ID, acquisition mode, and updated entry version (Lines 2–4), so that its predecessor can notify it upon release; the client then waits for this notification (Line 5). Otherwise, the client becomes a *holder* immediately (①), and its queue entry need not be populated because holders do not wait for notifications.

Lock release workflow. To release a lock, the client first dequeues itself by issuing an FAA on the lock header, which increments `qhead` by one and decrements `qsize` by one (Line 6). If the client is a writer, `wcnt` is also decremented by one. To hide network latency, DecLock piggybacks this FAA with a READ that fetches the full lock queue (Line 7). If the queue becomes empty after the release (i.e., the `qsize` returned by the FAA is 1), the operation finishes because no waiter needs to be notified (③). Otherwise, DecLock examines the entry immediately after the current client (*successor*). If the successor is a writer (④), it is directly notified as it must be waiting (Lines 11–12). If the successor is a reader and the current client is a writer (⑤), the lock can be shared by all consecutive readers in the queue, so all of them are notified (Lines 14–19). If both the current client and its successor are readers (③), no notification is needed since the successor already holds the lock.

4.3 Queue Entry Versions

When fetching the lock queue during release operations, clients may encounter obsolete queue entry contents populated during previous traversals of the lock queue. This problem arises from the circular behavior of the lock queue, which allows previously dequeued entries to be reused by subsequent clients. To distinguish obsolete entries

```
# lid|cid: lock ID|client ID

# RDMA operations:
#   FAA(address)[field += value] -> old_value
#   CAS(address, expected, swap) -> old_value
#   WRITE(address, value) -> nil
#   READ(address, size) -> value

# Remote address:
#   HDR_ADDR(lid): lock header's address
#   QADDR(lid, i): the ith queue entry's address

EX(mode) := (mode == SHARED) ? 0 : 1 # wcnt
VERSION(idx) := (idx / QUEUE_CAPACITY) # version

cql_acquire(lid, cid, mode):
1   hdr = FAA(HDR_ADDR(lid))[qsize+=1,wcnt+=EX(mode)] ①②
2   if (mode == EXCLUSIVE and hdr.qsize > 0) or
       hdr.wcnt != 0 then
3   | i = hdr.qhead + hdr.qsize                          ②
4   | WRITE(QADDR(lid, i % QUEUE_CAPACITY),
    |               {mode, cid, VERSION(i)})
5   | wait_for_notification(lid)

cql_release(lid, cid, mode):
6   hdr = FAA(HDR_ADDR(lid))
          [qhead+=1,qsize+=-1,wcnt+=-EX(mode)]          ③④⑤
7   queue = READ(QADDR(lid, 0), QUEUE_CAPACITY)
8   if hdr.qsize > 1 then
9   | next = hdr.qhead + 1                              ③④⑤
10  | successor = queue[next % QUEUE_CAPACITY]
11  | if successor.mode == EXCLUSIVE then
12  | | send_notification(successor.cid, lid)           ④
13  | else        # the successor is SHARED
14  | | if mode == EXCLUSIVE then                       ③⑤
15  | | | for i in range(next, next+hdr.qsize-1) do     ⑤
16  | | | | idx = i % QUEUE_CAPACITY
17  | | | | if queue[idx].mode == EXCLUSIVE then
18  | | | | | break
19  | | | | send_notification(queue[idx].cid, lid)
```

Fig. 6: ***Pseudocode of CQL acquisition and release operations in DecLock. The destination of all RDMA operations is the MN containing the CQL lock, which is omitted for convenience.***

from valid ones, DecLock uses the version of queue entries to represent the freshness of their contents. The version is determined by the division of the queue index and the queue capacity, which represents the number of traversals within the queue buffer. Clients update the version and other entry fields using a single RDMA WRITE operation, which ensures atomicity because RDMA operates at cacheline granularity [18], and all queue entries are cacheline-aligned. When another client reads the entry, it compares the version in the entry with the version calculated using the queue index it is using. The entry value is considered valid only if the two versions are equal. All entries are initialized with a version value of `-1`.

When encountering obsolete entries, the CQL protocol *refetches* the entries from MNs. Notably, since obsolete entries are rare, the refetching incurs minimal additional READ operations—at most 1.8% in our experiments (see §6.4). In the protocol, there are two possible causes of obsolete entries:

Read-write conflicts. Although the atomic enqueuing mechanism prevents write-write conflicts on queue entries, read-write conflicts can occur due to concurrent acquisition and release operations. Specifically, when fetching the lock queue during a release, some entries might not have been populated during acquisition, resulting

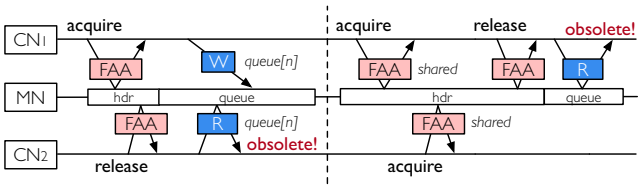

Fig. 7: ***Two causes of obsolete entries: read-write conflicts (left) and shared holders (right).***

in obsolete contents (see Fig. 7 (left)). To resolve the conflict, the client refetches the entries.

Shared holders. If a reader is enqueued when the queue contains no writers, it becomes a *shared holder* without populating its queue entry, making the entry obsolete (see Fig. 7 (right)). Lock release operations cannot distinguish between obsolete entries belonging to shared holders and those caused by read-write conflicts. To resolve this ambiguity, the CQL protocol identifies shared holders in the queue by excluding writers—since waiting writers always populate their entries. Accordingly, when a reader releases the lock, it locates all writers in the queue immediately after fetching the queue. Since entries of writers might not be populated yet, it repeatedly fetches the queue until the number of valid writers in the queue matches `wcnt` in the header. Once the position of writers is determined, the remaining entries certainly belong to shared holders.

4.4 Lock Reset

DecLock resets the CQL lock states when a waiter is unable to obtain lock ownership within a finite time period due to corrupted lock states or node failures. The reset process is exclusively executed by the client initiating it. A lock is considered to be undergoing a reset when its header contains a non-zero reset ID. Any lock operations that encounter a non-zero reset ID will be aborted. Databases are required to retry aborted acquisition operations and can safely ignore aborted release operations. Note that lock resets rarely occur—only 0.0014% of acquisitions in our evaluation (see §6.6).

Reset occasions. Clients may wait infinitely for a CQL lock in the following cases, which are handled through resets.

Queue entry overwrite. If the content of an entry is overwritten (e.g., due to queue overflows) before the client releasing the lock fetches it, the original content is lost. Overwrites are detected when the fetched entry's version exceeds the version calculated using the queue index.

Version overflow. If the version value keeps increasing after experiencing an integer overflow, it will eventually increase to the initial version of queue entries (`-1`). Since only waiters populate their queue entries, some entries may remain unpopulated and retain their initial values, which cannot be distinguished from valid entries with a version equal to `-1`. Using 16-bit versions, version overflow occurs less than once in every 100,000 acquisitions of the same lock.

Node failure or deadlock. Similar to spinlocks, CQL locks held by failed CNs or involved in a deadlock are never released, blocking waiters infinitely. To ensure the liveness of locks, DecLock sets a timeout for acquisitions and resets locks that experience timeouts. Databases then resolve deadlocks by aborting transactions that experience lock resets and release all their acquired locks. Upon MN failures, all locks on the failed MN are reset once the MN recovers.

Reset process. The reset process includes three sequential steps:

Step 1: Set the reset ID. The client sets the reset ID to its CN ID via a CAS operation on the lock header. When the CAS operation fails due to concurrent lock header updates by other lock operations, the client retries the CAS. During retries, if the client detects a non-zero reset ID, indicating an ongoing reset process, it abandons the reset attempt to ensure only one client executes the remaining steps.

Step 2: Notify participants. The client broadcasts a reset signal to all clients and waits until it has collected all responses. Responses from failed clients are not awaited. Lock holders respond to the signal after releasing the lock, whereas other clients respond immediately. Lock waiters abort the acquisition operation after responding.

Step 3: Reset the lock states. The client reinitializes the lock queue and resets the lock header value with two WRITE operations issued in sequence, clearing the reset ID.

4.5 Protocol Correctness

We validate the correctness of the CQL protocol by proving it preserves two essential lock properties when no failures occur: mutual exclusion and liveness [15, 32]. Mutual exclusion eliminates all read-write and write-write conflicts, which we verify by establishing that two invariants always hold: *the ownership of locks held by a writer cannot be obtained by any other clients* (M1), and *the ownership of locks held by readers cannot be obtained by writers* (M2). Both invariants are verified by enumerating all possible lock acquisition workflows. Liveness guarantees that each client obtains lock ownership within a finite time frame (as long as all holders release the lock in a finite time), which we prove by showing that *all waiters' information remains in the queue until they are notified or the lock is reset* (L1), and *all waiters recorded in the queue are notified in a finite time* (L2). To prove L1, we demonstrate that version overflows and queue entry overwrites can always be detected in finite time and fixed by lock resets. The correctness of lock resets is verified by proving that the reset process terminates in finite time, and the lock has no holder or waiter when the reset process terminates. The detailed proof of the CQL protocol can be found in [52].

4.6 Failure Handling

Failure model. DecLock preserves the mutual exclusion property when experiencing CN, MN, and network failures, but it does not guarantee strong liveness upon MN and network failures. Fairness is not preserved in case of failures because clients may abort ongoing lock operations and acquire the lock in a new order. DecLock uses a reliable coordinator to detect failures (e.g., using heartbeats [9, 23]) and handle network partitions [3, 40]. Clients check failure detection results when waiting for the completion of RDMA operations or the responses of the reset signal. Aligning with DM databases [29, 30, 53], DecLock uses the Reliable Connection (RC) of RDMA to

handle packet loss and re-ordering at the transport layer. Details about how failures are handled in DecLock can be found in [52].

Failures during reset. If the CN executing a reset process fails, surviving clients detect that the reset ID belongs to a failed CN, ignore its reset signals, and initiate a new reset. Clients use CAS operations to swap the reset ID to their own CN ID, with the first successful client exclusively executing the reset process. This recovery mechanism is idempotent, as the final successful reset will reliably restore the lock to its initial state.

Message interleaving. Messages in DecLock include notifications, reset signals, and failure detection results. If failure detection is delayed, notifications and reset signals sent to the failed CN are lost. Loss of notifications is equivalent to failure of the lock holder CN, which is handled by a reset process triggered by subsequent waiters. Loss of reset signals is harmless, as the reset workflow does not wait for responses from failed CNs.

Notifications sent before the reset but arriving after reset signals may cause the receiver to incorrectly obtain lock ownership after reset. To handle this, DecLock maintains a reset counter on all CNs for each lock, which increments by 1 when the lock is reset. The reset signal broadcast in Step 2 synchronizes the counter value. Clients embed the local reset counter value in notifications and ignore any notification with a reset count smaller than the local reset counter.

5 TS-BASED HIERARCHICAL LOCKING

Challenge: queue size variability. The CQL protocol uses a static queue capacity because atomic circular queues with variable capacity are hard to implement [33]. However, in DM databases, the degree of lock contention can vary widely because compute resources are elastic. Consequently, a small queue may overflow frequently under high contention, whereas a large queue wastes MN memory and MN-NIC bandwidth.

Solution: timestamp (TS)-based hierarchical locking. We observe that DM databases tend to utilize CPU cores on the same CN to exploit data locality, rather than spreading clients across more CNs [29, 30, 44, 54]. Following this insight, we address the challenge with a *hierarchical* design that maintains CQL lock waiters at the CN level and employs local locks on each CN to resolve lock conflicts among its clients. This approach reduces the required queue capacity to the number of CNs allocated to the database.

However, hierarchical designs often compromise fairness among clients across different CNs [17]. Specifically, local waiters may preempt remote waiters who acquired the lock earlier, which cannot be addressed by simply limiting the maximum consecutive ownership transfers between local clients [6, 44]. We ensure fairness by embedding *timestamps* of lock acquisition in the lock queue entries, which help maintain the order between local and remote waiters.

5.1 Lock Data Structures

In DecLock, each lock consists of a CQL lock on the MN that stores the data and a local lock on each CN.

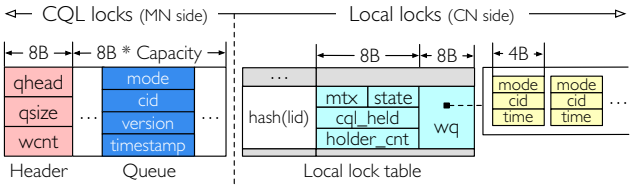

Fig. 8: ***Data structures of CQL locks (left) and local locks (right).***

CQL locks. As shown in Fig. 8 (left), each CQL lock consists of an 8-byte header and several 8-byte queue entries. In addition to the mode, the client ID, and a version required by the protocol, each entry also records a timestamp indicating when the acquisition operation is issued (details in §5.3).

Local locks. As shown in Fig. 8 (right), each CN maintains a hash table of local locks. Each local lock includes a mutex to protect its metadata (`mtx`), the local lock state (`state`), the holder count (`holder_cnt`), a flag indicating whether the CQL lock is held by the local CN (`cql_held`), and a pointer to a dynamically allocated queue of local waiters (`wq`). For each waiter, the queue records the acquisition mode (`mode`), the client ID (`cid`), and an acquisition timestamp (`time`). When a lock is acquired, the corresponding local lock is created and inserted into the table if it is not already present. Throughout our experiments in §6, local lock tables consume less than 20 MB of memory on each CN, which is negligible given that CNs typically have GBs of memory [30, 39].

5.2 Hierarchical Lock Operations

Lock acquisition. To acquire a lock, the client first tries to acquire the corresponding local lock in the local lock table. If the client obtains local lock ownership (i.e., the lock state is `FREE`, or both the lock state and acquisition mode are `SHARED`), it increments the holder count; otherwise, it appends itself to the wait queue and waits for notifications from the current local lock holders. Once a writer enters the wait queue, the lock state transitions to `EXCLUSIVE`, which blocks subsequent readers and prevents writer starvation.

After obtaining local lock ownership, the acquisition operation finishes immediately if the CQL lock is already held by the local CN (`cql_held` is true). Otherwise, the client proceeds to acquire the CQL lock, waits until it obtains the CQL lock ownership, and then sets `cql_held` to true. If the client is a reader, it further shares lock ownership with eligible readers in the wait queue by notifying them and incrementing the holder count accordingly.

Lock release. To release a lock, the client first releases the corresponding local lock. If the holder count is greater than 1—indicating lock ownership is shared with other local clients—the client simply decreases the holder count and finishes the release operation. Otherwise, the client selects a local waiter from the wait queue and determines whether to transfer ownership directly to it based on specific policies (detailed in §5.3). If ownership should be transferred to remote waiters before the selected local waiter, the client releases the CQL lock, clears the `cql_held` flag, and notifies the selected

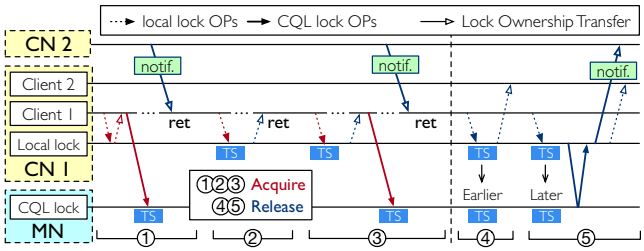

Fig. 9: ***Acquisition and release workflows of hierarchical locking where the acquisition timestamp (TS) is recorded and compared.***

waiter to reacquire the CQL lock later. Otherwise, the client notifies the selected local waiter directly. In cases where no local waiters are present, the client resets the local lock state back to `FREE`. The pseudocode for hierarchical lock operations is available in [52].

5.3 Timestamp-based Fairness

A key challenge in applying the hierarchical design to DecLock is maintaining fairness between local and remote waiters. Fortunately, in CQL locks, waiter information is recorded in a centralized queue visible to clients across all CNs. By embedding acquisition timestamps in this information, clients can determine the correct ordering among waiters on different CNs, thereby preserving fairness.

Use cases of timestamps. Fig. 9 illustrates how the acquisition timestamp is recorded during lock acquisition (①–③) and compared during lock release (④–⑤). When a client needs to wait for the CQL lock (①/③), it writes the timestamp along with other waiter information to the CQL queue entry (Line 4 in Fig. 6). Similarly, if the client needs to wait for the local lock (②/③), it records the timestamp in its wait queue entry. When releasing the lock, the client compares the timestamp of local and remote waiters. If the first local waiter has an earlier timestamp than remote waiters, or if there are no remote waiters (④), the local waiter can directly obtain ownership without releasing the CQL lock (②). Otherwise (⑤), the client releases the CQL lock to transfer ownership to remote waiters, requiring the local waiter to reacquire the CQL lock later (③). Notably, the CQL lock must be reacquired to update its state if the local waiter's acquisition mode differs from the local lock state.

Fairness policies. DecLock supports both task-fair and phase-fair policies for intra-CN lock scheduling to accommodate different workloads [8]. In the task-fair design, when a reader acquires the CQL lock and shares ownership with other waiting readers, it scans the wait queue to collect adjacent readers and stops upon encountering a writer or a waiter with a later timestamp than remote waiters. In contrast, the phase-fair design allows the reader to share lock ownership with all readers in the wait queue. Moreover, when a writer releases the CQL lock, the task-fair design assigns the first local waiter to reacquire the CQL lock, while the phase-fair design gives this privilege to the first reader in the wait queue. In summary, the phase-fair design trades fairness for higher concurrency by allowing waiting readers to preempt waiting writers.

Synchronized time. DecLock synchronizes time on all CNs regularly to address clock skew and clock drift problems in distributed environments. This is achieved through a counter on the MN. During synchronization, each CN atomically increments the counter by one using an FAA operation. When the counter matches the number of CNs, that CN resets it to 0. All CNs repeatedly read the counter until it becomes 0 and subsequently record the current time. This mechanism achieves precision within the latency of one RDMA READ (around 3 µs), which is sufficient to ensure accurate ordering of lock acquisitions. DecLock adopts a statically configured interval (e.g., one minute), as this parameter is determined solely by the clock drift rate of the hardware platform, and is irrelevant to workload characteristics such as operation latency.

Timestamp encoding. Each timestamp is a 16-bit unsigned integer recording the number of microseconds that have passed since the last time synchronization. To address overflows, if the difference between two timestamps is greater than half of the maximum timestamp value (around 32 ms), the timestamp with the larger value is considered to be the earlier timestamp. Notably, even if timestamp comparisons occasionally produce wrong results (e.g., when the difference is too large to be represented by 16 bits—which did not occur in our experiments), this only affects fairness among the current waiters and does not compromise correctness.

Prefetched remote timestamp. To check the timestamp of remote waiters, the client must fetch the remote lock queue, introducing extra latency to ownership transfer. DecLock hides this latency by delegating the task to clients waiting for the local lock. Specifically, when a client becomes the first waiter of the local lock, it prefetches the remote lock queue and stores the earliest timestamp among remote waiters in the local lock. When the current holder releases the lock, it can directly use the prefetched timestamp. If no timestamp has been prefetched, this indicates that either there are no remote waiters, or the remote waiter acquires the lock later than all local waiters. Thus, lock ownership can be transferred to local waiters.

The prefetched remote timestamp becomes invalid once the CQL lock is released. Therefore, clients must update this prefetched remote timestamp after reacquiring the CQL lock. To facilitate this, when a client releases the CQL lock, it identifies the earliest timestamp in the lock queue and appends this timestamp to the notification. Waiters that receive this notification can then update their prefetched timestamp without fetching the queue again.

6 EVALUATION

6.1 Experimental Setup

Testbed. We conducted experiments using 16 machines in a single rack, each equipped with two 24-core Intel CPUs, 128 GB of RAM, and two ConnectX-4 100 Gbps RNICs. All RNICs are connected to a Mellanox 100 Gbps switch. Each machine hosted two logical nodes, with each node exclusively using one CPU and one RNIC. We additionally used two 64-core servers connected via ConnectX-7 100 Gbps RNICs to investigate intra-CN scalability.

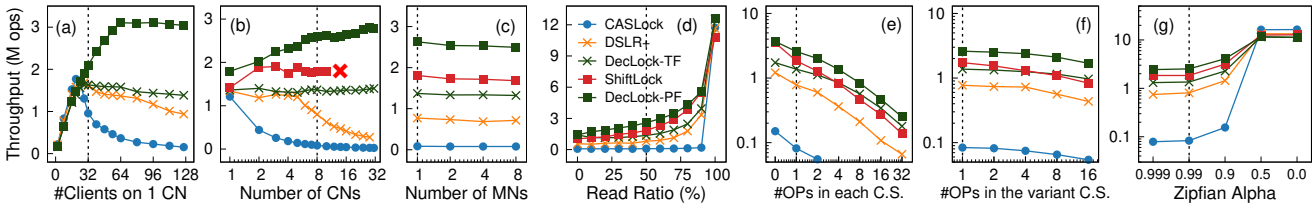

Fig. 10: ***The operation throughput of the microbenchmark using different lock mechanisms under various workload parameters. Dotted lines denote the default parameter values.***

Comparing targets. We compare DecLock with three RDMA-based locks: a CAS-based spinlock (CASLock), an improved version of DSLR [51] (DSLR+), and ShiftLock [19]. CASLock represents the conventional spinlock design widely used in DM databases [29, 30, 44, 50, 53, 54]. For a fair comparison, we use a recent implementation that supports reader-writer semantics [13]. DSLR+ combines ticket locking [24] with truncated exponential backoff [35] to improve spinlock performance. ShiftLock implements a reader-writer variant of the MCS lock [31] to reduce spinning retries. We exclude other MCS-based locks [6, 16] because they do not support reader-writer semantics. In all experiments, DecLock uses lock queues with 32 entries and 16-bit versions.

Workloads. To evaluate the performance of DM lock mechanisms across various workloads, we build a microbenchmark that simulates typical usage scenarios [29, 30, 44, 54]. Specifically, each operation acquires a lock in shared (or exclusive) mode, fetches (or updates) the corresponding remote object, and then releases the lock. Additionally, to demonstrate end-to-end benefits, we integrate DecLock into three DM database components—a NoSQL data store, a transaction engine, and a B$^+$Tree index [44]—and evaluate them using real-world traces [49] and representative benchmarks [22, 42, 44].

6.2 Lock Operation Performance

We first study the lock operation performance of DecLock with the microbenchmark. DecLock is evaluated with both task-fair (TF) and phase-fair (PF) policies for intra-CN scheduling, which match the fairness levels of DSLR+ and ShiftLock, respectively.

Scalability. To evaluate the scalability of DecLock, we run the microbenchmark with varying numbers of CNs and MNs. All clients access 100,000 locks and corresponding objects following a Zipfian distribution (α=0.99) with a 50% read ratio. These parameters mirror the access patterns in DM databases under write-intensive workloads [30, 44]. Fig. 10 (a–c) presents the resulting lock throughput.

Intra-CN scaling (a). We evaluate intra-CN scalability with one 64-core server as the CN and the other as the MN; the CN runs two client coroutines per core. CASLock saturates MN-NIC IOPS at 20 clients, where lock operations already consume 88.5% of IOPS. More clients increase contention and retries, collapsing throughput and raising lock-related IOPS to 98.1%. DSLR+ and DecLock-TF are both limited by task-fair scheduling. However, DSLR+'s retry-driven IOPS keeps rising, eventually saturating the MN-NIC and degrading throughput, with lock operations using up to 91.6% of IOPS. In contrast, DecLock-TF keeps IOPS usage stable (up to 35.2%) and thus sustains high throughput. DecLock-PF scales further by relaxing intra-node fairness to phase-fair, which allows higher reader concurrency. It peaks at 3.10 M ops when the MN-NIC saturates, with lock operations consuming 74.5% of IOPS, and then remains stable. We exclude ShiftLock because its artifact is incompatible with the RDMA driver on our 64-core servers. In later experiments on 24-core nodes, we use 32 clients per CN.

Inter-CN scaling (b). We fix the number of MNs at 1 and scale to 31 CNs. As in the intra-node results, CASLock and DSLR+ degrade sharply from MN-NIC contention, whereas DecLock maintains high throughput. DecLock-PF scales more slowly than in the intra-CN case and never reaches its peak, as more CNs reduce opportunities for local ownership transfers. ShiftLock uses 2.29× more acquisition IOPS than DecLock (Fig. 11 (a)), saturates the MN-NIC at 2 CNs, and spends 83.7% of IOPS on lock operations. Other experiments use 8 CNs by default, since ShiftLock gets stuck beyond 10 CNs.

Scaling to more MNs (c). Partitioning locks across multiple MNs does not noticeably improve throughput, despite higher aggregated MN-NIC IOPS. The baselines remain bottlenecked by the hotspot MN's NIC IOPS (see §2.2), while DecLock stays stable because, with the default 8 CNs, it does not reach MN-NIC saturation. Given the limited impact of MN count, other experiments use 1 MN.

Influence of various workloads. We vary workload parameters to simulate various workloads and examine their impact on lock throughput, as shown in Fig. 10 (d–f).

Read ratio (d). DecLock-TF consistently outperforms DSLR+ (by 1.18×–2.50×) and CASLock (by 15.31×–20.80×) across all read ratios except 100%, due to reduced MN-NIC IOPS consumption. ShiftLock exceeds DecLock-TF by up to 1.55× because its writers occasionally transfer ownership to all waiting readers, sacrificing fairness to improve concurrency. DecLock-PF, which uses a similar strategy among local waiters, outperforms ShiftLock by 1.12×–1.55×. All lock mechanisms have similar performance in read-only scenarios, as all acquisitions obtain the lock without retries.

Length of critical section (C.S.) (e). We vary the number of remote accesses in critical sections to emulate different C.S. lengths. As C.S. length increases, throughput drops for all lock mechanisms because lock conflicts intensify. The longer waits cause retries to surge for CASLock (387.9), DSLR+ (349.2), and ShiftLock (15.6) (see Fig. 11 (a)). In contrast, DecLock needs at most 1.10 RDMA

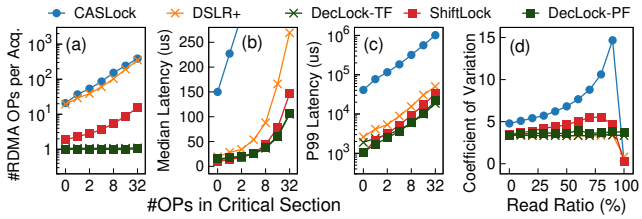

Fig. 11: ***The average number of RDMA operations used by each acquisition (a), median latency (b), 99th-percentile latency (c), and latency coefficient of variation (d) of lock mechanisms under the microbenchmark.***

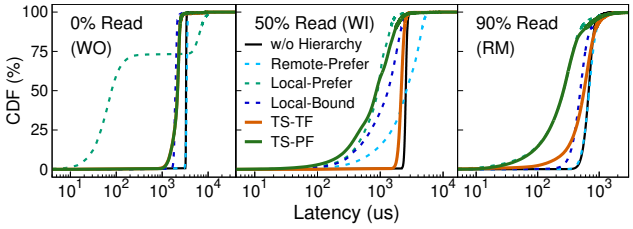

Fig. 12: ***The acquisition latency distribution of the most contended lock in the microbenchmark with different hierarchical designs.***

operations per acquisition regardless of C.S. length, because fully decentralized ownership transfer eliminates retries. The extra 0.10 operations come from filling the queue entry while waiting for ownership. This lower MN-NIC IOPS usage yields throughput gains over CASLock (43.37×), DSLR+ (4.35×), and ShiftLock (1.81×).

Variability of C.S. (f). We increase the C.S. length on a single CN while fixing the C.S. length of all other CNs to 1 to evaluate the impact of heterogeneous critical sections. DecLock maintains its performance gains over baselines as the C.S. length on the variant CN increases, with performance trends nearly identical to those observed when scaling the C.S. length of all CNs simultaneously.

Skewness (g). We decrease the α parameter of the Zipfian distribution to reduce workload skewness, where α=0 indicates a uniform distribution. As the workload becomes less skewed, the throughput of CASLock, DSLR+, and ShiftLock converges to that of DecLock, since the number of retries decreases due to fewer lock conflicts.

Latency. We investigate how lock mechanisms affect the end-to-end latency of microbenchmark operations, as shown in Fig. 11 (a–c).

Median latency. Median operation latency captures operations that acquire the lock immediately, without retries. DecLock's median latency scales with C.S. length, whereas the baselines' median latencies rise much faster, widening the gap. This is because retries on a few hot locks saturate MN-NIC IOPS and slow down all RDMA operations, including retry-free acquisitions. Overall, DecLock reduces the median latency of CASLock, DSLR+, and ShiftLock by up to 95.8%, 64.3%, and 28.5%, respectively.

Tail (99th-percentile) latency. Tail operation latency reflects the time to acquire heavily contended locks. CASLock has the highest tail latency because it lacks fairness. DSLR+ and ShiftLock also have higher tail latency than DecLock, as higher RDMA latency lengthens critical-section execution and aggregate wait time. Initially, DecLock-PF has lower tail latency than DecLock-TF because of higher concurrency. However, as C.S. length grows, weaker fairness causes DecLock-PF to exceed DecLock-TF on tail latency. Overall, DecLock reduces the 99th-percentile latency of CASLock, DSLR+, and ShiftLock by up to 98.2%, 67.8%, and 43.7%, respectively.

Fairness. We evaluate fairness using the latency coefficient of variation (CV), defined as the standard deviation divided by the mean; lower CV implies more stable lock acquisition latency and thus stronger fairness. As shown in Fig. 11 (d), DecLock-TF has nearly the same CV as DSLR+, since both mechanisms provide task fairness. By contrast, CASLock offers no fairness guarantee and therefore has the highest CV. ShiftLock uses phase fairness, which relaxes reader-writer fairness and raises CV in mixed workloads. Meanwhile, DecLock-PF enforces phase fairness within CNs while preserving task fairness across CNs, keeping its CV close to that of DecLock-TF. In read-only workloads, all baselines achieve near-zero CV, whereas DecLock's CV stays similar to that under other read ratios because of latency differences between local and remote acquisitions.

6.3 Hierarchical Designs

To study our timestamp-based mechanism (TS), we compare it with three ownership transfer policies in hierarchical locking: Remote-Prefer (prioritizes remote waiters over local ones), Local-Prefer (prioritizes local waiters over remote ones), and Local-Bound (transfers ownership to remote waiters after at most N consecutive local transfers [6, 44], where N=4 as in previous work [44]). We evaluate TS with both TF and PF policies within CNs. Due to space constraints, we only present the performance of baselines with PF policy, omitting similar results obtained with TF policy. We run the microbenchmark using three workloads: write-only (WO, 0% reads), write-intensive (WI, 50% reads), and read-mostly (RM, 90% reads). To highlight the impact of fairness, we analyze the latency distribution of the most contended lock.

As shown in Fig. 12, prioritizing either remote or local waiters can introduce significant latency variance due to the starvation of the other type of waiter. The local-bound policy closely matches TS-PF's performance in WO workloads, but its latency gap with TS-PF widens as the read ratio rises because it allows only N local readers to share the lock, which inherently limits concurrency. Across all read ratios, TS achieves lower median and tail latencies compared to the non-hierarchical design, as it accelerates ownership transfer among local waiters while preserving fairness across CNs. Compared with TS-TF, TS-PF further reduces latency in WI and RM workloads by boosting reader concurrency. Overall, compared to other policies, TS cuts median and 99th-percentile acquisition latencies by up to 66.7% and 70.8%, respectively.

6.4 Entry Refetching Overhead

We measure the additional RDMA READ operations required to handle obsolete entries (see §4.3) under various microbenchmark parameters, as shown in Fig. 13 (left). In all configurations, DecLock

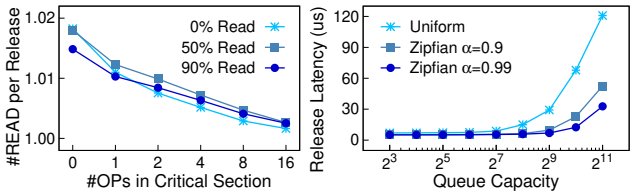

Fig. 13: ***The average number of extra RDMA READs per lock release caused by refetching obsolete entries (left) and the median lock release latency with different capacities of the lock queue (right).***

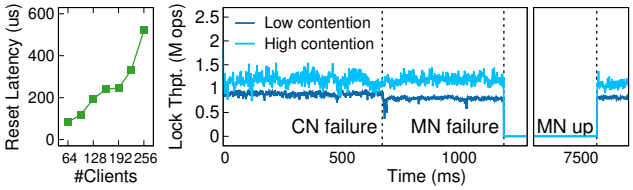

Fig. 14: ***The latency of resetting a lock with different numbers of clients (left) and the timeline of DecLock's lock operation throughput in the presence of CN and MN failures (right).***

incurs at most 0.018 extra RDMA operations per release on average, consuming negligible MN-NIC resources. Moreover, the refetching frequency is inversely proportional to the length of critical sections because when clients hold the lock for longer periods, waiters have more time to update their entries.

6.5 Sensitivity Analysis

Impact of lock queue capacity. We investigate the impact of lock queue capacity on DecLock by measuring its release latency across various capacities, as shown in Fig. 13 (right). As capacity increases, release latency rises more quickly in workloads with lower skewness because these workloads have higher operation throughput and saturate MN bandwidth earlier. Nevertheless, release latency remains stable below 8.80 µs when the queue capacity is up to 128, even in uniform workloads. This result indicates that DecLock can support 128 CNs (i.e., thousands of clients) with little performance loss. Notably, this capacity far exceeds the typical number of CNs assigned to a single DM database [29, 30, 50, 53].

Impact of time synchronization interval. We vary DecLock's synchronization interval from 10 ms to 60 s while running the microbenchmark, finding that this parameter has a negligible impact on DecLock's performance. Owing to the low clock drift rate on our testbed, even a coarse interval (60 s) is sufficient to align timestamps across CNs and maintain fairness in hierarchical locking.

6.6 Lock Reset Overhead

We measure the average reset latency with varying numbers of clients, as shown in Fig. 14 (left). The reset latency is proportional to the number of clients due to the process of sending reset signals to clients and awaiting their responses. Despite their high latency, lock resets have minimal impact on the performance of DecLock because of their infrequent occurrences. DecLock uses sufficiently large lock queues to prevent queue overflows and employs 16-bit versions to reduce version overflows. Thus, across all experiments conducted in §6.2, at most 0.0014% of acquisitions are reset.

6.7 Fault Tolerance

To evaluate DecLock's liveness and correctness under CN and MN failures, we manually shut down 1 CN and 1 MN in sequence while running the microbenchmark. We run the workload in two configurations: one using 1 core (Low contention) and the other using 16 cores (High contention) on each of 8 CNs. As shown in Fig. 14 (right), under low contention, the lock throughput quickly stabilizes at around 88% of the original throughput after 1 CN fails as one-eighth of the clients are terminated. Under high contention, the throughput remains unchanged after the failure since DecLock maintains its peak throughput even with fewer clients. When the MN fails, all operations pause and the lock throughput drops to 0 ops until the MN recovers.

6.8 DM Database Performance

We evaluate DecLock using three representative DM database components to demonstrate its effectiveness in mitigating performance degradation caused by lock contention. DecLock provides phase-fair locking for all components.

NoSQL store. We build a NoSQL object store that allows clients on CNs to perform get and set operations on objects stored on MNs. Conflicting accesses to objects are serialized by reader-writer locks on MNs. Each client performs operations based on object size, get-set ratio, and key distribution derived from traces of Twitter [49]. We evaluate the store using two traces: one with a small object size (414 B) and a low get ratio (65%) to represent IOPS-bound scenarios (No. 14), and another with a large object size (9213 B) and a high get ratio (89%) to represent bandwidth-bound scenarios (No. 53).

As shown in Fig. 15 (left), CASLock and DSLR+ cause throughput collapse in both scenarios because retries consume most MN-NIC IOPS and slow other RDMA operations. ShiftLock mitigates this problem by removing retries from waiting writers, but reader retries still dominate MN-NIC IOPS and cap peak throughput. Its weaker fairness also yields higher tail latency than DSLR+ in BW-bound scenarios. By contrast, DecLock eliminates retries, allowing MN-NICs to operate efficiently and sustain stable throughput after reaching IOPS or bandwidth bottlenecks. As a result, DecLock improves object-store throughput by up to 35.60×, 3.33×, and 1.48× over CASLock, DSLR+, and ShiftLock, respectively.

Transaction engine. We build a transaction engine based on the two-phase locking protocol and evaluate it with different lock mechanisms, using the SmallBank [22] and TPC-C [42] benchmarks as workloads like prior work [47, 54]. SmallBank comprises short transactions that check or update bank accounts, with each transaction acquiring 1 to 3 locks. TPC-C consists of complex transactions implementing order-entry operations, where each transaction acquires up to tens of locks. Both workloads are write-intensive, with over 85% of transactions being read-write. We configure 100,000 accounts for SmallBank and 8 warehouses for TPC-C.

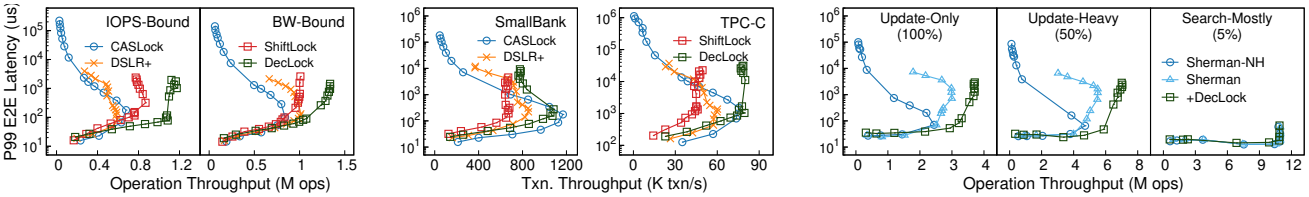

Fig. 15: ***The NoSQL store's throughput and 99th-percentile operation latency in IOPS-bound and bandwidth (BW)-bound scenarios (left), the transaction throughput and 99th-percentile latency under SmallBank and TPC-C benchmarks (mid), and the update/search operation throughput and 99th-percentile operation latency of Sherman under different workloads (right) when using different lock mechanisms.***

As shown in Fig. 15 (mid), DecLock improves transaction throughput by up to 16.91×, 3.08×, and 1.59× compared to CASLock, DSLR+ and ShiftLock, respectively. While CASLock and DSLR+ achieve high peak throughput owing to their simple acquisition and release logic, they exhibit a sharp throughput decline after reaching the peak under both workloads due to MN-NIC IOPS saturation. Under the SmallBank workload, DecLock's throughput drops after scaling to more than 1 CN due to the lower chance for lock ownership transfers among local clients, but remains stable afterwards due to its efficient usage of MN-NICs. This throughput decline does not occur under the TPC-C workload because of its higher locality.

Database index. Sherman [44] is a state-of-the-art B$^+$Tree index for DM. It uses CAS-based locks to serialize tree node updates and adopts hierarchical locking to reduce the overall rate of RDMA retries. We integrate DecLock into Sherman and compare it with vanilla Sherman and Sherman without hierarchical locking (Sherman-NH). We use the workloads from Sherman's paper [44], including Update-Only (100% updates), Update-Heavy (50% updates and 50% searches), and Search-Mostly (5% updates and 95% searches). We also evaluate the impact of range scans by replacing point searches in these workloads with 100-key range scans. Lock mechanisms unsupported by Sherman (e.g., ShiftLock) are excluded due to significant integration complexity. All systems use up to 512 clients distributed across up to 31 CNs.

As shown in Fig. 15 (right), DecLock improves throughput by up to 27.71× compared to Sherman-NH and 2.31× compared to Sherman. It also reduces the 99th-percentile latency by up to 97.6% and 82.1%, respectively. Under Update-Only and Update-Heavy workloads, Sherman-NH experiences significant throughput collapse because RDMA retries occupy the MN-NIC. While hierarchical locking reduces retry frequency because only one client on each CN acquires the remote CASLock, MN-NIC usage still increases as more CNs are added, eventually causing throughput collapse beyond 16 CNs. In contrast, DecLock enables Sherman to sustain high throughput with more clients by using MN-NICs efficiently. Under the Search-Mostly workload, all lock mechanisms perform similarly since search operations in Sherman are lock-free. All evaluated systems deliver nearly identical performance for range scan and point search workloads. This is because B$^+$Trees use fat leaf nodes that store dozens of keys, allowing each range scan to access only two leaf nodes and incur an RDMA footprint similar to that of point searches. We omit the range scan results from Fig. 15 for brevity.

7 RELATED WORK

Disaggregated memory (DM) databases. Providing elastic compute and memory resources, the DM architecture has attracted widespread interest and is being adopted by production databases [11, 50]. Numerous database components have been redesigned for DM, including hash tables [25, 38, 39, 43, 55], transaction engines [48, 53, 54], and tree-based indices [4, 27, 29, 30, 44]. Most of them use CAS-based spinlocks to serialize concurrent data accesses [29, 30, 44, 45, 48, 50, 53, 54]. Under high contention, these locks can significantly hamper performance due to inefficient MN-NIC IOPS usage and lack of fairness. DecLock introduces decentralized coordination that minimizes MN-NIC usage while ensuring fairness.

RDMA-based distributed locks. Several studies explore distributed locks built solely on one-sided RDMA operations [6, 13, 16, 19, 41, 47, 51]. Some adopt ticket locks [51] or MCS locks [6, 16, 19] to improve scalability and fairness over spinlocks. Ticket locks provide strict reader-writer fairness, but still repeatedly poll lock state on MNs and thus do not resolve MN-NIC contention. MCS locks either do not support readers [6, 16] or track them with counters [19], which wastes MN-NIC resources and weakens fairness between readers and writers. Some RDMA locks use lock-cohorting-inspired hierarchical designs to improve scalability [6, 44]. However, they sacrifice fairness across CNs because they cannot track a consistent acquisition order for local and remote waiters. They also lack reader-writer semantics, limiting concurrency. DecLock instead uses fully decentralized coordination across CNs to reduce MN-NIC usage while preserving fairness by ordering all waiters chronologically with acquisition timestamps.

8 CONCLUSION

This paper presents DecLock, a scalable, efficient, and fair lock mechanism designed for the disaggregated memory architecture. Evaluation using both a microbenchmark and DM database components confirms the efficiency and efficacy of DecLock.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 62272291, 62572302) and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM122).

REFERENCES

[1] A. Agarwal and M. Cherian. 1989. Adaptive backoff synchronization techniques. *SIGARCH Comput. Archit. News* 17, 3 (apr 1989), 396–406. https://doi.org/10.1145/74926.74970

[2] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. 2023. Memory Disaggregation: Why Now and What Are the Challenges. *SIGOPS Oper. Syst. Rev.* 57, 1 (June 2023), 38–46. https://doi.org/10.1145/3606557.3606563

[3] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, Virtual Event, USA, 351–368.

[4] Hang An, Fang Wang, Dan Feng, Xiaomin Zou, Zefeng Liu, and Jianshun Zhang. 2023. Marlin: A Concurrent and Write-Optimized B+-Tree Index on Disaggregated Memory. In *Proceedings of the 52nd International Conference on Parallel Processing* (<conf-loc>, <city>Salt Lake City</city>, <state>UT</state>, <country>USA</country>, </conf-loc>) *(ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 695–704. https://doi.org/10.1145/3605573.3605576

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. Association for Computing Machinery, New York, NY, USA, 53–64.

[6] Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. 2024. ALock: Asymmetric Lock Primitive for RDMA Systems. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3626183.3659977

[7] Björn B Brandenburg and James H Anderson. 2009. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *2009 21st Euromicro Conference on Real-Time Systems*. IEEE Computer Society, Brussels, Belgium, 184–193.

[8] Björn B Brandenburg and James H Anderson. 2010. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems* 46, 1 (2010), 25–87.

[9] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 335—-350.

[10] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/3445814.3446713

[11] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2477—2489. https://doi.org/10.1145/3448016.3457560

[12] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. 2023. Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1060—1073. https://doi.org/10.1145/3603269.3604833

[13] Yeounoh Chung and Erfan Zamanian. 2015. Using RDMA for Lock Management. arXiv:1507.03274

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[15] Pierre-Jacques Courtois, F. Heymans, and David Lorge Parnas. 1971. Concurrent control with "readers" and "writers". *Commun. ACM* 14 (1971), 667–668.

[16] A. Devulapalli and P. Wyckoff. 2005. Distributed queue-based locking using advanced network features. In *2005 International Conference on Parallel Processing (ICPP '05)*. IEEE Computer Society, Washington, DC, USA, 408–415. https://doi.org/10.1109/ICPP.2005.34

[17] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 247–256. https://doi.org/10.1145/2145816.2145848

[18] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[19] Jian Gao, Qing Wang, and Jiwu Shu. 2025. ShiftLock: Mitigate One-sided RDMA Lock Contention via Handover. In *23rd USENIX Conference on File and Storage Technologies (FAST '25)*. USENIX Association, Santa Clara, CA, 355–372.

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

[21] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. USENIX Association, Boston, MA, 649–667.

[22] H-Store. 2013. SmallBank Benchmark. Retrieved May 18, 2026 from https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[23] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIX ATC '10)*. USENIX Association, USA, 11.

[24] Leslie Lamport. 1974. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* 17 (1974), 453–455.

[25] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, K. Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. *Proc. VLDB Endow.* 15 (2022), 4023–4037.

[26] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. ACM, New York, NY, USA, 488–504. https://doi.org/10.1145/3477132.3483561

[27] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST '23)*. USENIX Association, Santa Clara, CA, 99–114.

[28] Quanxi Li, Hong Huang, Ying Liu, Yanwen Xia, Jie Zhang, Mosong Zhou, Xiaobing Feng, Huimin Cui, Quan Chen, Yizhou Shan, and Chenxi Wang. 2025. Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*. USENIX Association, Philadelphia, PA, 167–187.

[29] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *The 30th ACM Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 110–126. https://doi.org/10.1145/3694715.3695959

[30] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. USENIX Association, Boston, MA, 553–571.

[31] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9 (1991), 21–65.

[32] John M Mellor-Crummey and Michael L Scott. 1991. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices* 26, 7 (1991), 106–113.

[33] Narasinga Rao Miniskar, Frank Liu, and Jeffrey S. Vetter. 2021. A Memory Efficient Lock-Free Circular Queue. In *2021 IEEE International Symposium on Circuits and Systems*. IEEE, Melbourne, Australia, 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401239

[34] Zoran Radovic and Erik Hagersten. 2002. Efficient synchronization for nonuniform communication architectures. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, Baltimore, MD, USA, 13–13.

[35] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. 2024. Scaling Up Memory Disaggregated Applications with SMART. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 351–367. https://doi.org/10.1145/3617232.3624857

[36] Berkeley Architecture Research. 2015. Berkeley Architecture Research: FireBox. Retrieved May 18, 2026 from https://bar.eecs.berkeley.edu/projects/firebox.html

[37] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI '18). USENIX Association, Carlsbad, CA, 69–87.

[38] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 675–691. https://doi.org/10.1145/3600006.3613144

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

[40] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36 (2013), 21–27.

[41] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1—12. https://doi.org/10.1145/2213836.2213838

[42] TPC. 1992. TPC-C. Retrieved May 18, 2026 from https://www.tpc.org/tpcc/

[43] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, Virtual Event, USA, 33–48.

[44] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

[45] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proc. VLDB Endow.* 16, 1 (sep 2022), 15–22. https://doi.org/10.14778/3561261.3561263

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

[47] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

[48] Xingda Wei, Haotian Wang, Tianxia Wang, Rong Chen, Jinyu Gu, Pengfei Zuo, and Haibo Chen. 2023. Transactional Indexes on (RDMA or CXL-based) Disaggregated Memory with Repairable Transaction. arXiv:2308.02501

[49] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, Virtual Event, USA, 191–208.

[50] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/3626246.3653377

[51] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1571–1586. https://doi.org/10.1145/3183713.3196890

[52] Hanze Zhang, Ke Cheng, Rong Chen, Xingda Wei, and Haibo Chen. 2025. DecLock: A Case of Decoupled Locking for Disaggregated Memory. arXiv:2505.17641 [cs.DC]

[53] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. USENIX Association, Santa Clara, CA, 801–819.

[54] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST '22)*. USENIX Association, Santa Clara, CA, 51–68.

[55] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference (USENIX ATC '21)*. USENIX Association, Virtual Event, USA, 15–29.