Building Write-Optimized Tree Indexes on Disaggregated Memory

Qing Wang
Tsinghua University
q-wang18@mails.tsinghua.edu.cn

Youyou Lu
Tsinghua University
luyouyou@tsinghua.edu.cn

Jiwu Shu
Tsinghua University
shujw@tsinghua.edu.cn

ABSTRACT

Memory disaggregation architecture physically separates CPU and memory into independent components, which are connected via high-speed RDMA networks, greatly improving resource utilization of database systems. However, such an architecture poses unique challenges to data indexing due to limited RDMA semantics and near-zero computation power at memory side. Existing indexes supporting disaggregated memory either suffer from low write performance, or require hardware modification.

We present SHERMAN, a write-optimized B$^+$Tree index on disaggregated memory that delivers high performance with commodity RDMA NICs. SHERMAN combines RDMA hardware features and RDMA-friendly software techniques to boost index write performance from three angles. First, to reduce round trips, SHERMAN coalesces dependent RDMA commands by leveraging in-order delivery property of RDMA. Second, to accelerate concurrent accesses, SHERMAN introduces a hierarchical lock that exploits on-chip memory of RDMA NICs. Finally, to mitigate write amplification, SHERMAN tailors the data structure layout of B$^+$Tree with a two-level version mechanism. Our evaluation shows that, SHERMAN is one order of magnitude faster in terms of both throughput and 99th percentile latency on typical write-intensive workloads, compared with state-of-the-art designs.

1. INTRODUCTION

The popularity of in-memory databases and in-memory computing catalyzes ever-increasing demands for memory in modern datacenters. However, datacenters today suffer from low memory utilization ($< 65\%$) [5, 11], which results from imbalanced memory usages across a sea of servers. In response, academia and industry are working towards a new hardware architecture called *memory disaggregation*, where CPU and memory are *physically* separated into two network-attached components — compute servers and memory servers [11, 9, 18, 13]. With memory disaggregation, CPU and memory can scale independently and different applications share a global memory pool efficiently.

Since almost all CPU resources are assembled on compute servers under the memory disaggregation architecture, memory servers have near-zero computation power, which highlights the challenge that how compute servers access disaggregated memory residing on memory servers. Fortunately, RDMA (remote direct memory access), a fast network technique, allows compute servers to *directly access* disaggregated memory unmediated by memory servers' computation power with low latency, becoming an essential building block of memory disaggregation architecture [11, 13, 18].

In this work, we explore how to design a high-performance tree index, a key pillar of database systems, on disaggregated memory. We first revisit existing RDMA-based tree indexes and examine their applicability on disaggregated memory. Several RDMA-based tree indexes rely on remote procedure calls (RPCs) to handle write operations [8, 10]; they are ill-suited for disaggregated memory due to near-zero computation power of memory servers. For tree indexes that can be deployed on disaggregated memory, they also have some critical limitations: Some indexes use RDMA one-sided verbs for all index operation [19] (we call it *one-sided approach*); they can deliver high performance for read operations, but suffer from low throughput and high latency in terms of write operations, especially in high-contention scenarios (§3.1). Other indexes bake write operations into SmartNICs or customized hardware [1], which brings high TCO (total cost of ownership) and cannot be deployed in datacenters in a large scale immediately.

Our goal is designing a tree index on disaggregated memory that can *deliver high performance for both read and write operations with commodity RDMA NICs*. To this end, we further analyze what makes one-sided approach inefficient in write operations, and find out three major causes. First, due to limited semantics of one-sided RDMA verbs, modifying an index node (e.g., tree node in B$^+$Tree) always requires multiple round trips (i.e., lock, read, write, and unlock), inducing high latency and further making conflicting requests more likely to be blocked. Second, the locks used for resolving write-write conflicts are slow and experience performance collapse under high-contention scenarios. This is because ① at the hardware level, NICs adopt expensive concurrency control to ensure atomicity between RDMA atomic commands, where each command needs two PCIe transactions; ② at the software level, such locks often trigger unnecessary

©ACM 2023. This is a minor revision of the paper entitled "Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory", published in SIGMOD'22, ISBN 978-1-4503-9249-5/22/06, June 12-17, 2022, Philadelphia, PA, USA. DOI: https://doi.org/10.1145/3514221.3517824

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Copyright 2008 ACM 0001-0782/08/0X00 ...$5.00.

retries, which consumes RDMA IOPS, and do not provide fairness, which leads to high tail latency. Third, the layout of index data structure incurs severe write amplification. Due to coarse-grained consistency check mechanisms (e.g., using checksum to protect a whole tree node), a small piece of modification will result in large-sized write-back across the network.

Motivated by the above analysis, we propose SHERMAN, a write-optimized distributed B$^+$Tree index on disaggregated memory. The key idea of SHERMAN is *combining RDMA hardware features and RDMA-friendly software techniques to reduce round trips, accelerate lock operations, and mitigate write amplification*. SHERMAN spreads B$^+$Tree nodes across a set of memory servers, and compute servers perform all index operations via RDMA one-sided verbs purely. SHERMAN uses a classic approach for concurrency control: lock-free search with versions to resolve read-write conflicts and exclusive locks to resolve write-write conflicts.

To reduce round trips, SHERMAN introduces a command combination technique. Based on the observation that commodity RDMA NICs already provide in-order delivery property, this technique allows client threads to issue dependent RDMA commands (e.g., write-back and lock release) simultaneously, letting NICs at memory servers reflect them into disaggregated memory in order.

To accelerate lock operations, we design a *hierarchical on-chip lock (HOCL)* for SHERMAN. HOCL is structured into two parts: global lock tables on memory servers, and local lock tables on compute servers. Global lock tables and local lock tables coordinate conflicting lock requests between compute servers and within a compute server, respectively. Global lock tables are stored in the on-chip memory of RDMA NICs, thus eliminating PCIe transactions of memory servers and further delivering extremely high throughput for RDMA atomic commands (∼110 Mops). Within a compute server, before trying to acquire a global lock on memory servers, a thread must acquire the associated local lock, so as to avoid a large amount of unnecessary remote retries. Moreover, by adopting wait queues, local lock tables improve fairness between conflicting lock requests. Based on local lock tables, a thread can hand its acquired lock over to another thread directly, reducing at least one round trip for acquiring remote global locks.

To mitigate write amplification, SHERMAN tailors the leaf node layout of B$^+$Tree. First, entries in leaf node are *unsorted*, so as to eschew shift operations upon insertion/deletion. Second, to support lock-free search while avoiding write amplification, we introduce a *two-level version mechanism*. In addition to using a pair of *node-level versions* to detect the inconsistency of the whole leaf node, we embed a pair of *entry-level versions* into each entry, which ensures entry-level integrity. For insertion/deletion operations without split/merging events, only entry-sized data is written back, thus saving network bandwidth and making the most of the extremely high IOPS of small RDMA messages.

To demonstrate the efficacy of SHERMAN, we evaluate SHERMAN using a set of benchmarks. Under write-intensive workloads, SHERMAN achieves much better performance than FG [19], a state-of-the-art distributed B$^+$Tree supporting disaggregated memory. Specifically, in common skewed workloads, SHERMAN delivers one order of magnitude performance improvement in terms of both throughput and 99th percentile latency. For read-intensive workloads (i.e., 95% read operations), SHERMAN exhibits slightly higher throughput with 25% lower 99th percentile latency.

This work makes three main contributions. First, we analyze existing RDMA-based tree indexes on disaggregated memory (§3). Second, we design and implement SHERMAN, a write-optimized B$^+$Tree index on disaggregated memory (§4). Finally, we use a set of evaluations to demonstrate the high performance of SHERMAN (§5).

2. BACKGROUND

In this section, we provide the background on memory disaggregation (§2.1) and RDMA network (§2.2) briefly.

2.1 Memory Disaggregation

Traditional datacenters pack CPU and memory into the same hardware units (i.e., monolithic servers), leading to low memory utilization (< 65%) [11, 5] and further increasing the TCO of datacenters. To attack this problem, academia and industry are exploring a new hardware architecture called *memory disaggregation* [11, 9, 18, 13]. In such an architecture, CPU and memory are physically separated into two different hardware units: compute servers (CSs) and memory servers (MSs). CSs own a mass of CPU cores (10s - 100s), but MSs host high-volume memory (100s - 1000s GB) with near-zero computation power. CPUs in CSs can directly access the disaggregated memory in MSs via high-speed RDMA networks (§2.2). With memory disaggregation, CPU and memory can scale independently and applications can pack resources in a more flexible manner, boosting resource utilization significantly.

To reduce remote accesses from CSs to MSs, CSs are always equipped with a small piece of memory as the local cache (1 - 10 GB). Moreover, MSs own a small set of wimpy CPU cores (1 - 2) to support lightweight management tasks, such as network connection management.

2.2 RDMA Network

RDMA network is the key enabler of memory disaggregation architecture. It provides two types of verbs, namely *two-sided verbs* and *one-sided verbs*, to applications. Two-sided verbs — `RDMA_SEND` and `RDMA_RECV` — are the same as traditional Linux socket interface. One-sided verbs, i.e., `RDMA_WRITE`, `RDMA_READ` and `RDMA_ATOMIC` (`RDMA_FAA` and `RDMA_CAS`), operate directly on remote memory without involving the CPUs of receivers. The direct-access feature of one-sided verbs makes memory servers with near-zero computation power possible.

RDMA hosts communicate via queue pairs (QPs). A QP consists of a send queue and a receive queue. A sender performs an RDMA command by posting the request to the send queue. RDMA supports three transport types: reliable connected (RC), unreliable connected (UC), and unreliable datagram (UD). SHERMAN uses RC, since it supports all one-sided verbs and is reliable.

3. MOTIVATION

Designing a fast distributed tree index on disaggregated memory poses unique challenges. In this section, we first revisit two existing approaches — using one-sided verbs purely and extending RDMA interfaces — and reveal their respective issues (§3.1). Then, we analyze why using one-sided verbs is slow, which motivates the design of SHERMAN (§3.2).

		read-intensive		write-intensive	
		uniform	skew	uniform	skew
Throughput (Mops)		31.8	32.9	18.7	0.34
Latency (μs)	50th	4.9	4.7	9.5	10
	90th	6.4	6.2	14.3	68.7
	99th	14.9	15.3	19	19890

Table 1: Index performance in one-sided approach (100 Gbps ConnectX-5 NICs, 8 MSs, 8 CSs with 176 client threads, 8/8-byte key/value, 1 billion key space). The performance collapses under *write-intensive and skew* setting.

3.1 Existing Approaches

With near-zero computation power at MS side, we cannot delegate index operations to CPUs of MSs via remote procedure calls (RPCs), which is the main difference between memory disaggregation and traditional architectures. Existing work enables efficient lookup operations via lock-free search and caching mechanisms [4, 8], leading to a single `RDMA_READ` in the ideal situation (i.e., cache hit). However, write operations are in a quandary due to their complex semantics; specifically, there are two avenues to design write operations, each of which has its own issues.

Approaches #1: using one-sided verbs purely. FG [19], an RDMA-based B$^+$Tree, is the only one tree index that completely leverages one-sided verbs to perform index write operations (so it can be deployed on disaggregated memory). For write operations, it adopts a lock-based approach, where tree node modification is protected by RDMA-based spinning locks (`RDMA_CAS` for acquisition and `RDMA_FAA` for release). For read operation, FG follows a lock-free search scheme, where threads fetch tree nodes via `RDMA_READ` without holding locks and then check the consistency using checksum. We conduct an experiment to evaluate FG's performance. Since FG is not open-source, we implement it from scratch; we also cache internal tree nodes to reduce remote accesses. Table 1 shows the result, and we make two observations. First, in read-intensive workload (95% lookup and 5% insert/update), FG can achieve high throughput ($>$ 30Mops) and low 99th percentile latency ($< 16\mu$s). Second, in write-intensive workload (50% lookup and 50% insert/update), FG delivers 18Mops with 19μs tail latency under the *uniform* setting; yet, its performance collapses in case of skew setting, where the key popularity follows a Zipfian popularity with skewness 0.99: the throughput is only 0.34Mops and the tail latency is close to 20ms.

Approaches #2: extending RDMA interfaces. Another approach to designing indexes on disaggregated memory is extending RDMA interfaces. This approach offloads index write operations into memory servers' NICs via SmartNICs or other customized hardware [1, 12]. For example, by exploiting interface extensions (i.e., indirect addressing and notification), researchers propose HT-Tree (without implementation) [1], a hybrid index combining the hash table and tree. Yet, compared with commodity RDMA NICs, SmartNICs come at the price of *higher TCO* and *lower performance*. More specifically, SmartNICs have much higher market price (∼5×) than that of commodity RDMA NICs at present [13]. As for performance, at 100Gbps network environment, StRoM [12], the state-of-the-art RDMA extensions using FPGA, has 2× higher round-trip latency (4μs) against commodity RDMA NICs (∼2μs).

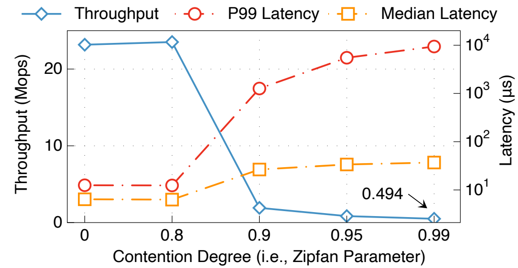

Figure 1: Performance of RDMA-based exclusive locks (ConnectX-5 NICs, 100 Gbps).

3.2 Why One-sided Approach is Slow?

The inefficiency of write operations using one-sided verbs stems from *excessive round trips*, *slow synchronization primitives*, and *write amplification*.

3.2.1 Excessive Round Trips

The most obvious cause of slow write operations is excessive round trips. For example, when modifying a tree node (not consider node split/merging), a client thread needs 4 round-trips: ①acquiring associated exclusive lock, ②reading the tree node, ③writing back the modified tree node ④and finally releasing the lock. Excessive round trips negatively impact write performance in two aspects. First, the latency of a single write operation is proportional to the number of round trips. Second, more round trips lead to the longer critical path, so conflicting write operations (i.e., requests targeting the same tree nodes) are more likely to be blocked, degrading the concurrency performance.

3.2.2 Slow Synchronization Primitives

RDMA-based locks used in these indexes cannot provide sustainable performance under a variety of workloads. We conduct an experiment to demonstrate it. In the experiment, 154 threads across 7 CSs acquire/release 10240 locks residing in an MS; we choose RDMA verbs used by FG [19], where `RDMA_CAS` for lock acquisition and `RDMA_FAA` for release. The access pattern follows Zipfian distribution and we vary Zipfian parameter to control the contention degree (0 is uniform setting, and 0.99 is the most common real-world scenario). Figure 1 shows the result: the system experiences performance collapse under high-contention settings.

The main reason behind the performance collapse is expensive in-NIC concurrency control. To guarantee correct atomicity semantic between `RDMA_ATOMIC` commands targeting the same addresses, RDMA NICs adopt an internal locking scheme [6]. More specifically, a NIC maintains a certain number of buckets (e.g., 4096), and puts `RDMA_ATOMIC` commands having the same certain bits in their destination addresses (e.g., 12 LSBs) into the same bucket. Commands in the same bucket are considered conflicting. An `RDMA_ATOMIC` can not be executed until the previous conflicting commands are finished. Unfortunately, a single `RDMA_ATOMIC` needs two PCIe transactions: ① fetching data from CPU memory into the NIC and ② writing back after modification (② can be omitted for failed `RDMA_CAS`). These PCIe transactions stretch the queueing time of conflicting `RDMA_ATOMIC` commands and thus degrade concurrency performance, especially in high-contention workloads.

Moreover, in the software design, existing locks in RDMA-based indexes blindly reties when conflicts appear, squan-

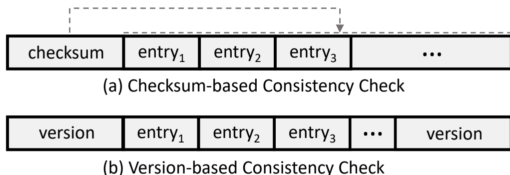

Figure 2: Two existing consistency check mechanisms.

dering limited throughput of NICs in both CSs and MSs. And they do not guarantee fairness between conflicting lock acquisition, thus causing starvation and high tail latency.

3.2.3 *Write Amplification*

Existing RDMA-based indexes always trigger large-sized `RDMA_WRITE`, suffering from write amplification. This issue stems from two causes. First, a B$^+$Tree keeps entries in each tree node sorted. When an entry is inserted/deleted to/from a node, all the entries on the right side of the insertion/deletion position need to be shifted. The shift operations cause extra data to be written via `RDMA_WRITE`.

Second, two existing consistency check mechanisms, which is proposed to detect concurrent writes for lock-free lookup, induce write amplification. In the first mechanism (Figure 2(a)), each tree node includes a checksum covering the whole node area (except the checksum itself) [7, 19]; the checksum is re-calculated when modifying the associated node, and is verified when reading the node. The other mechanism, namely version-based consistency check (Figure 2(b)), stores a version number at the start and end of each node [8]; when modifying a node via `RDMA_WRITE`, the associated two version are incremented; a node's content obtained via `RDMA_READ` is consistent only when the two versions are the same. Of note, like Cell [8], we observe that the NIC reads data in increasing address order, so we omit per-cacheline version mechanisms used by FaRM [4]. Since the granularity of the above two mechanisms is tree node, any modification to part of the node area requires to write back the whole node (include the metadata, e.g., checksum and version), leading to severe write amplification.

4. DESIGN

Motivated by our observations about root causes of inefficient write operations (§3.2), we design Sherman, a write-optimized distributed B$^+$Tree index on disaggregated memory. In this section, we begin by presenting our design principles (§4.1), proceed to give an overview of Sherman (§4.2), and then describe our key techniques (§4.3-§4.5).

4.1 Design Principles

1) Seeking solutions from hardware first. There are unexploited features of RDMA NICs that pose opportunities for index design. In Sherman, we utilize ① on-chip memory exposed by NICs to accelerate lock operations, and ② in-order delivery guarantee of RC QP to reduce round trips.

2) Applying CS-side optimization when possible. In the current multicore era, each compute server (CS) usually launches a mass of client threads (10s - 100s) to manipulate an index concurrently, leaving a lot of design space for CS-side optimization. Within a CS, we leverage local lock tables to coordinate conflicting lock requests between threads, and locks can be handed over from a thread to another quickly. Besides, Sherman maintains CS-side index cache to reduce network accesses for tree traversal.

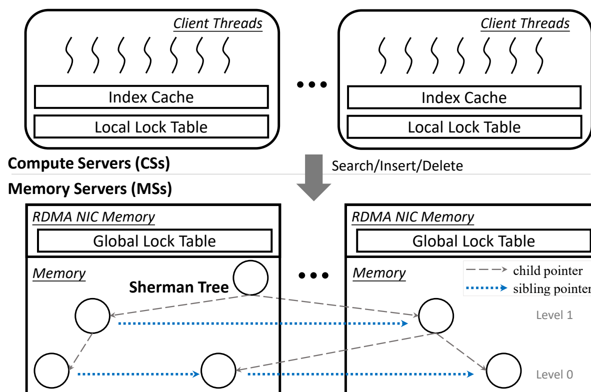

Figure 3: Sherman's architecture and interactions.

3) Tailoring data structure layout to improve RDMA friendliness. Disaggregated memory has unique profile: it is accessed via network (i.e., RDMA) rather than cache-coherent memory bus. Thus, it is necessary to tailor data structure layout of indexes to reap RDMA's full potential. To this end, Sherman ① separates locks from the tree structure, to put them into NICs' on-chip memory, and ② adopts entry-level versions as well as unsorted leaf nodes, to reduce IO size of `RDMA_WRITE` commands.

4.2 Overview

Figure 3 shows the overall architecture and interactions of Sherman, which consists of a set of memory servers (MSs) and compute servers (CSs). MSs are equipped with massive memory where Sherman tree resides. CSs run client threads that manipulate Sherman via specific interfaces (i.e., lookup, range query, insert and delete). Since the number of CPU cores continues to increase, we assume that there are a host of client threads running within a CS (dozens or even hundreds). These client threads cooperatively execute system services (e.g., RDMA-based transaction processing [15, 16]), which needs Sherman for data indexing.

B$^+$Tree structure. Sherman is a B$^+$Tree, where values are stored in leaf nodes. We record a sibling pointer for every leaf node and internal node as in the B-link tree. Client threads can always reach a targeted node by following these sibling pointers in the presence of node split/merging, thus supporting concurrent operations efficiently. Every pointer in Sherman (i.e., child/sibling pointers) is 64-bit, which includes two parts: 16-bit MS unique identifier and 48-bit memory address within corresponding MS.

Concurrency control. Sherman adopts lock-based write operations and lock-free read operations.

Write-write conflicts. Sherman uses *node-grained* exclusive locks to resolve write-write conflicts: before modifying a tree node, the client thread must acquire the associated exclusive lock. These exclusive locks are hierarchical: *local lock tables* at CS-side and *global lock tables* in on-chip memory of MSs' NICs (§4.3). Such a hierarchical structure provides high concurrency performance.

Read-write conflicts. Sherman supports lock-free search, which leverages `RDMA_READ` to fetch data residing in MSs without holding any lock. Moreover, Sherman uses versions to detect inconsistent data caused by concurrent writes. However, different from traditional mechanisms that use node-level versions, Sherman proposes a two-level version mech-

4

anism, which combines entry-level and node-level versions, to mitigate write amplification (§4.4).

Cache mechanism. To reduce remote accesses in the tree traversal, Sherman adopts a cache mechanism. Each CS maintains an *index cache*, which only makes two types of internal nodes' copies: ❶ nodes above the leaf nodes (level 1 in Figure 3), and ❷ the highest two level of nodes (including root). A client thread firstly searches type ❶ cache. On hit, it fetches the targeted leaf node directly from MSs; otherwise, it searches type ❷ cache and then traverses to leaf nodes via remote accesses. The index cache never induces data inconsistency issues, and detailed reasons are available in our original SIGMOD paper [14].

Memory management. In each MS, we reserve a dedicated *memory thread* to manage disaggregated memory. The memory thread divides memory residing in corresponding MS into fixed-length chunks (i.e., 8MB). Client threads use a two-stage memory allocation scheme to obtain memory from MSs. A client thread first chooses an MS in a round-robin manner, and requests a free chunk from the MS's memory thread via RPCs. Then, it allocates memory space for tree nodes locally within the chunk.

4.3 Hierarchical On-Chip Lock

Sherman proposes hierarchical on-chip lock (HOCL) to improve concurrency performance. HOCL leverages on-chip memory of NICs to avoid PCIe transactions at MS side; it also maintains local locks at CS side, to form a hierarchical structure, reduce retries, and improve fairness. Moreover, locks can be handed over between client threads within the same CSs, thus saving at least one round trip.

On-chip lock table. Current RDMA verbs support *device memory programming*. Specifically, an RDMA NIC can expose a piece of its on-chip memory (SRAM) to the upper applications, which can be allocated and read/written by RDMA commands. The on-chip memory eliminates PCIe transaction at receiver side, thus providing extremely high throughput ∼110 Mops `RDMA_CAS`). Sherman separates locks from tree nodes, and stores locks into on-chip memory at MS side; each tree node is in the *same* MS as the lock protecting it. These locks in each MS are structured as an array, namely *global lock table* (GLT). When locking a tree node, the client thread first hashes the address of the tree node into a position number in the associated GLT, and then issues an `RDMA_CAS` command to the lock, which tries to change it from `0` to the 16-bit CS identifier atomically. For lock release, the thread clears the lock via an `RDMA_WRITE`.

An important consideration of GLT is the on-chip memory size. In the NIC we use (i.e., ConnectX-5), 256KB on-chip memory is available. To accommodate more locks, we make the granularity of `RDMA_CAS` finer (16 bits rather than 64 bits), by applying an infrequently used RDMA verb called *masked compare and swap*, which allows us to select a portion of 64-bit for `RDMA_CAS`. Thus, an MS can maintain 131,072 locks in its GLT, enabling extremely high concurrency, particularly considering that we only lock at most one tree node at a time for a single write operation. To the best of our knowledge, Sherman is the first RDMA-based system that leverages on-chip memory of RDMA NICs.

Hierarchical structure. Sherman maintains a *local lock table* (LLT) in each CS, to coordinate conflicting lock requests within the same CSs. The LLT stores a *local lock* for each lock of all GLTs. When a thread needs to lock a tree node, it first acquires the associated local lock in LLT, and then acquires the associated lock in GLT; thus, conflicting lock requests from the same CSs are queued on the LLT at CS side, avoiding unnecessary remote retries and thus saving RDMA IOPS. Moreover, each local lock in LLT is associated with a *wait queue*. A thread that cannot acquire a local lock in LLT pushes itself into the corresponding queue; the thread can learn if its turn has arrived by checking whether it is at the head of the queue. The queue provides first-come-first-served fairness among threads within the same CSs. For lock release, the thread first releases the lock in GLT and then the local lock in LLT.

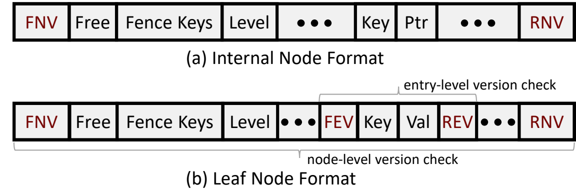

Figure 4: The format of internal nodes and leaf nodes in Sherman. **FNV/RNV** is 4-bit front/rear node version; **FEV/REV** is 4-bit front/rear entry version.

Handover mechanism. The hierarchical structure of HOCL enables a handover mechanism: handing over a lock from one client thread to another. When releasing a lock, if a thread finds out the lock's wait queue is not empty, it will hand the lock over to the one at the head of the wait queue. To avoid starving threads at other CSs, we limit the maximum number of consecutive handovers to `4`. The thread that is handed over a lock no longer needs remote accesses for acquiring the lock, thus saving at least one round trip.

4.4 Two-Level Version

To address the write amplification issue, Sherman incorporates a *two-level version mechanism*. First, Sherman uses unsorted leaf nodes so that shift operations upon insertion/deletion can be avoided. Unsorted leaf nodes complicate write operations in two aspects: (*i*) When looking up a key, the client thread needs to traverse the entire targeted leaf node; (*ii*) Before splitting a leaf node, the client thread must sort the entries in it. Given the microsecond-level network latency, the added overhead is slight.

Second, Sherman introduces *entry-level versions* to enable fine-grained consistency check, as shown in Figure 4. Specifically, in leaf nodes, each entry is surrounded by a pair of 4-bit *entry-level* versions (i.e., FEV and REV). In case of insertion without splitting, the associated entry-level versions are incremented and only the modified entry (includes FEV and REV) is written back via `RDMA_WRITE`, thus evading write amplification. Also, a pair of 4-bit *node-level versions* (i.e., FNV and RNV) is stored at the beginning and end of each leaf node, protecting the consistency at the node granularity. When splitting a leaf node, the client thread increments associated FNV and RNV, and writes back the whole node via `RDMA_WRITE`. Since internal nodes have a much lower modification frequency than leaf nodes, their format is standard: two node-level versions with a sorted layout. The extra memory space occupied by entry-level versions is acceptable: considering a B$^+$Tree storing 8-byte key and 8-byte value, each key-value pair at leaf nodes needs

extra 1-byte memory space for entry-level versions, consuming about 6% memory of all leaf nodes.

Lookup operation. Two-level version mechanism makes the lookup operation different from the standard one. After reading a leaf node from MSs, a client thread compares the two node-level versions first; mismatched versions indicate the read must be retried. Then, the client thread locates the targeted entry and compares the two associated entry-level versions; if the comparison fails, the client thread needs to re-read the leaf node via `RDMA_READ`. Wraparounds of these 4-bit versions may cause inconsistency to be undetected: two versions match but one wraps around. To handle wraparounds, we measure the time of each `RDMA_READ` command: if it takes more than $8\mu s$ (i.e., $2^4 \times 0.5$, where $0.5\mu s \ll$ the time of a single write operation), a retry is needed.

Range query. For a range query, the client thread issues multiple `RDMA_READ` in parallel to fetch targeted leaf nodes, and then checks leaf nodes' consistency in the same way as the lookup operation.

4.5 Command Combination

To enforce ordering between dependent RDMA commands (e.g., writing back a node and then releasing the lock), existing RDMA-based indexes use an expensive approach: issuing the following RDMA command only after receiving the acknowledgement of the preceding one [19]. Yet, we observe that RDMA already provides a strong ordering property at the hardware level: in a reliable connected (RC) queue pair, `RDMA_WRITE` commands are transmitted in the order that they are posted, and the NIC at receiver side executes these commands in order [2, 17]. By leveraging this ordering property, SHERMAN combines multiple `RDMA_WRITE` commands in a write operation, so as to reduce round trips.

There are two cases that SHERMAN combines multiple `RDMA_WRITE` commands. First, since a tree node and its associated lock co-locate at the same MS, the write-back of tree node and lock release can be combined through a QP, as opposed to issuing an unlock request after receiving acknowledgement of write-back; thus, one round trip is saved and the critical path shortens. Second, when a node (we call it `A` here) splits, we check whether the newly allocated sibling node belongs the same MS as `A`; if so, three `RDMA_WRITE` commands can be combined together: ① write-back of the sibling node, ② write-back of `A` and ③ release of `A`'s lock.

5. EVALUATION

Hardware Platform. Since memory disaggregation hardware is unavailable, we use a cluster of commodity, off-the-shelf servers to emulate MSs and CSs by limiting their usages of CPUs and memory [11]. Our cluster consists of 8 servers, each of which is equipped with 128GB DRAM, two 2.2GHz Intel Xeon E5-2650 v4 CPUs (24 cores in total), and one 100Gbps Mellanox ConnectX-5 NIC, installed with CentOS 7.7.1908. All these servers are connected with a Mellanox MSB7790-ES2F switch. For Mellanox ConnectX-5 NICs, the versions of driver and firmware are ofed 4.7-3.2.9.0 and 16.26.4012, respectively. Due to the limited size of our cluster, we emulate each server as one MS and one CS. Each MS owns 64GB DRAM and 2 CPU cores, and each CS owns 1GB DRAM and 22 CPU cores.

Compared Systems. FG [19] is the only B$^+$Tree that supports disaggregated memory. Since FG is not open-source, we implement it from scratch. For fair comparison, we add necessary optimizations to it: (*i*) index cache for reducing remote accesses, (*ii*) using `RDMA_WRITE` to release lock rather than expensive atomic verb `RDMA_FAA`. In order to distinguish our modified version of FG from the original one, we call it FG+ in the evaluation. The performance of FG+ is higher than that reported in FG paper [19].

Workload	Insert	Lookup	Range Query
write-only	100%		
write-intensive	50%	50%	
read-intensive	5%	95%	
range-only			100%
range-write	50%		50%

Table 2: Workloads.

Workloads. We explore different aspects of the systems by using YCSB workloads [3]. We use five types of read-write ratio, as shown in Table 2. Note that insert operations include updating existing keys (about 2/3 of all insert operations). There are two types of key popularity: *uniform* and *skewed*. In uniform workloads, all keys have the same probability of being accessed. Skewed workloads follow a Zipfian access distribution (Zipfian parameter, i.e., skewness, is 0.99 by default), which is common in production environments [3]. Unless otherwise stated, all the experiments are conducted with 8 MSs and 8 CSs. Each CS owns 500MB index cache, and launches 22 client threads (176 in total in our cluster). For each experiment, we bulkload the tree with 1 billion entries (8-byte key, 8-byte value) 80% full, then perform specified workloads. The size of a tree node (i.e., internal node and leaf node) is 1KB.

5.1 Overall Performance

To analyze SHERMAN's performance, we break down the performance gap between FG+ and SHERMAN through applying each technique one by one. Figure 5 and Figure 6 show the results under skewed and uniform workloads, respectively. In these two figures, ***+Combine*** stands for command combination technique. ***+On-Chip*** and ***+Hierarchical*** are two design parts of HOCL: leveraging on-chip memory of NICs to store locks, and hierarchical structure with handover mechanism, respectively. ***+2-Level Ver*** represents two-level version mechanism, and shows the final performance of SHERMAN.

5.1.1 Skewed Workloads

We make following observations from Figure 5. First, in both write-only and write-intensive workloads, SHERMAN has much higher throughput and lower latency against FG+. In write-only workloads, SHERMAN achieves 24.7× higher throughput with 1.2× lower median latency (50p latency) and 35.8× lower 99th percentile latency (99p latency). In write-intensive workloads, SHERMAN achieves 23.6× higher throughput with 1.4×/30.2× lower 50p/99p latency.

Second, all techniques contribute to the high write efficiency of SHERMAN. Here we analyze each technique for write-intensive workloads (write-only workloads have the same conclusions): ① Command combination improves the throughput by 3.37× and reduces the 50p/99p latency by 1.14×/3.18×, since it saves at least one round trip for each insert operation and further shortens the critical paths, decreasing the probability that conflicting requests are blocked. ② By putting locks into on-chip memory of NICs, SHERMAN gains 3.06× and 3.48× improvement in terms of through-

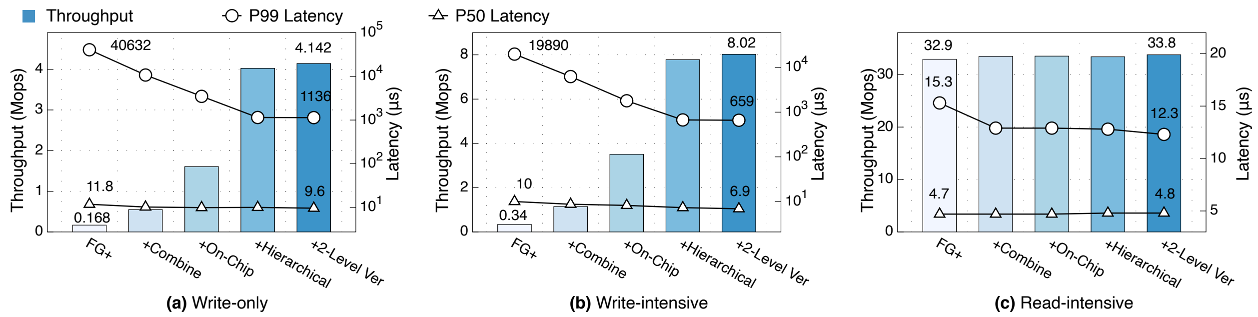

Figure 5: Contributions of techniques to performance (skewed workloads, skewness=0.99).

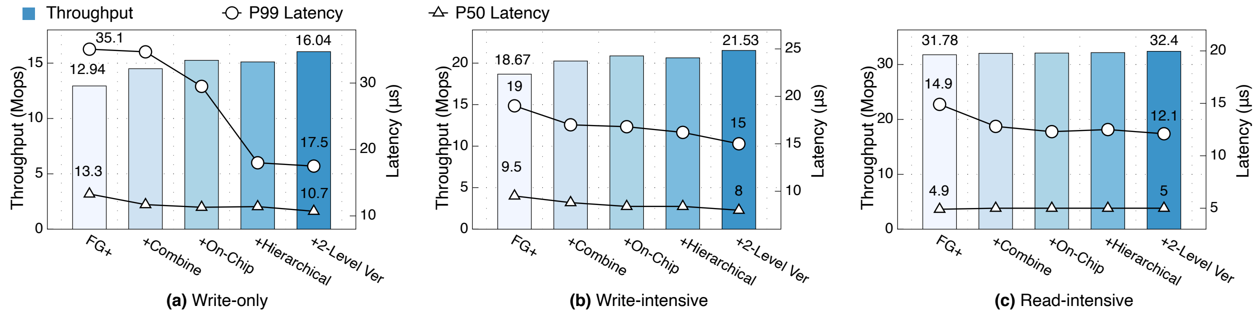

Figure 6: Contributions of techniques to performance (uniform workloads).

put and 99p latency, respectively. This is because on-chip memory avoids PCIe transactions of lock operations at MS side, which provides extremely high throughput for RDMA atomic verbs and thus can absorb more failed `RDMA_CAS` for retries. ③ Introducing hierarchical lock structure to Sherman brings 2.22× higher throughput, 1.12× lower 50p latency and 2.68× lower 99p latency. This is because unnecessary `RDMA_CAS` retries from the same CSs are avoided and the handover mechanism saves one round trip opportunistically. ④ Two-level version mechanism does not bring considerable throughput improvement (only %3), since the major bottleneck is concurrent conflicts rather than RDMA IOPS at this time; the 50p latency is reduced by 400ns (from 7.3μ to 6.9μ), since smaller `RDMA_WRITE` IO size has shorter PCIe DMA time at both CS side and MS side.

Third, in read-intensive workloads (Figure 5(c)), Sherman does not present considerable performance improvement, as expected, since all techniques we propose aim to boost performance of write operations. Yet, there are still two points worth noting here: ① By saving round trips for 5% insert operations, command combination reduces 99p latency from 15.3μs to 12.9μs; ② Sherman increases the 50p latency by 100ns (2%), we contribute it to unsorted leaf node layout, which causes traversal of the entire leaf node even for non-existing keys.

5.1.2 *Uniform Workloads*

As shown in Figure 6, compared with FG+, Sherman delivers 1.24× and 1.15× higher throughput in write-only and write-intensive workloads, respectively. These improvements mainly come from command combination and two-level version. Command combination saves round trips, so each client thread can execute more insert operations per second; two-level version reduces IO size of `RDMA_WRITE` from node size to entry size, thus giving full play to the RDMA's characteristics of extremely high small IO rate. HOCL is designed for high-contention scenarios (i.e., skewed workloads), so it does not increase throughput in uniform workloads. As for latency, Sherman reduces 50p/99p latency by 1.24×/2.01× and 1.19×/1.27× in write-only and write-intensive workloads, respectively, which mainly is contributed to command combination and HOCL: command combination saves round trips, and HOCL saves PCIe transaction time at MS side.

	range-only		range-write	
range size	100	1000	100	1000
FG+	26.09Mops	5.77Mops	7.74Mops	2.49Mops
Sherman	25.55Mops	5.76Mops	14.08Mops	3.12Mops

Table 3: Performance of range query.

5.2 Range Query Performance

In this experiment, we evaluate the performance of range query by using range-only and range-write workloads. The targeted range follows the skewed access pattern. Table 3 show the results, from which we make three observations. First, in range-only workloads, when the range size equals 100, FG+ outperforms Sherman by 2%. This is because the unsorted leaf layout in Sherman leads to unnecessary scans when targeted leaf nodes are partially occupied. Second, in range-only workloads, as range size grows (i.e., 1000), the throughput of Sherman and FG+ drops and is almost the same, since network bandwidth becomes the bottleneck. Third, in range-write workloads, Sherman outperforms FG+ by up to 1.82×. This is because Sherman's write operations save a considerable quantity of network resources for range query operations.

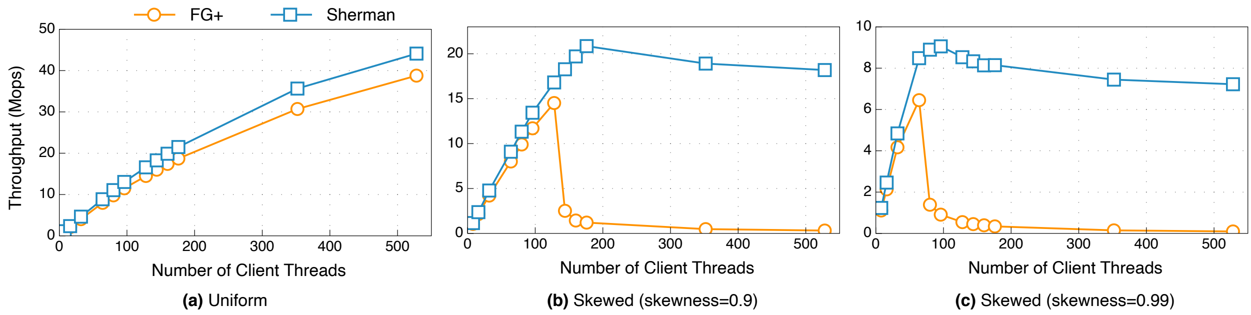

Figure 7: Scalability of SHERMAN (write-intensive workloads).

5.3 Scalability

In this experiment, we test the scalability of SHERMAN by varying the number of client threads that concurrently manipulate the tree. Due to the limited CPU cores in CSs, i.e., 22 × 8 in total, we bind multiple coroutines to every core of CSs. Each coroutine stands for a client thread, which issues index requests; a coroutine yields after initiating RDMA commands, allowing other coroutines to do useful work. We use write-intensive workloads. Figure 7 shows the results, from which we make the following observations.

First, in uniform workloads, both SHERMAN and FG+ can scale well. In case of 528 client threads, SHERMAN achieves 44Mops, 1.14× the throughput of FG+. The improvement mainly comes from two-level version mechanism: by writing back leaf nodes in a smaller granularity, more RDMA bandwidth are saved for `RDMA_READ` and lock operations.

Second, in skewed workloads, a higher contention degree (i.e., larger skewness value) leads to lower peak throughput. Specifically, SHERMAN achieves 21Mops peak throughput in case of 0.9 skewness, which is 1.44× higher than that of FG+; and 9Mops in case of 0.99 skewness, which is 1.4× higher than that of FG+. This is because the more serious the contention, the more likely concurrent write operations to be blocked, degrading the peak throughput.

Third, in skewed workloads, SHERMAN can provide sustainable throughput when more client threads are added; yet, FG+ experiences performance collapse. We attribute the SHERMAN's stable performance to a combination of all techniques we propose, as stated in §5.1.1.

6. CONCLUSION

We proposed and evaluated SHERMAN, an RDMA-based B$^+$Tree index on disaggregated memory. SHERMAN introduces a set of techniques to boost write performance and outperforms existing solutions. We believe SHERMAN demonstrates that combining RDMA hardware features and RDMA-friendly software designs can enable a high-performance index on disaggregated memory. SHERMAN is open-source at: https://github.com/thustorage/Sherman.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grant No. 61832011, 62022051) and Huawei (Grant No. YBN2019125112). Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).

7. REFERENCES

[1] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of HotOS'19*, page 120–126, New York, NY, USA, 2019. ACM.

[2] I. T. Association et al. InfiniBandTM Architecture Specification Volume 1 Release 1.3 (General Specifications), 2015.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SOCC'10*, page 143–154, New York, NY, USA, 2010. ACM.

[4] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of NSDI'14*, page 401–414, USA, 2014. USENIX Association.

[5] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of IWQoS'19*, New York, NY, USA, 2019. ACM.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of ATC'16*, page 437–450, USA, 2016. USENIX Association.

[7] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of ATC'13*, page 103–114, USA, 2013. USENIX Association.

[8] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *Proceedings of ATC'16*, page 451–464, USA, 2016. USENIX Association.

[9] P. S. Rao and G. Porter. Is Memory Disaggregation Feasible? A Case Study with Spark SQL. In *Proceedings of ANCS'16*, page 75–80, New York, NY, USA, 2016. ACM.

[10] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro. Fast General Distributed Transactions with Opacity. In *Proceedings of SIGMOD'19*, page 433–448, New York, NY, USA, 2019. ACM.

[11] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of OSDI'18*, page 69–87, USA, 2018. USENIX Association.

[12] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart Remote Memory. In *Proceedings of EuroSys'20*, New York, NY, USA, 2020. ACM.

[13] S.-Y. Tsai, Y. Shan, and Y. Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of ATC'20*, pages 33–48, USA, 2020. USENIX Association.

[14] Q. Wang, Y. Lu, and J. Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of SIGMOD'22*, page 1033–1048, New York, NY, USA, 2022. ACM.

[15] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, Feb. 2017.

[16] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of SIGMOD'20*, pages 511–526, New York, NY, USA, 2020. ACM.

[17] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.*, 12(11):1637–1650, 2019.

[18] 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. *Proc. VLDB Endow.*, 13(9):1568–1581, 2020.

[19] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of SIGMOD'19*, page 741–758, New York, NY, USA, 2019. ACM.