ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA

Tobias Ziegler
Technische Universität
Darmstadt

Carsten Binnig
Technische Universität
Darmstadt

Viktor Leis
Friedrich-Alexander-Universität
Erlangen-Nürnberg

ABSTRACT

In this paper, we propose ScaleStore, a novel distributed storage engine that exploits DRAM caching, NVMe storage, and RDMA networking to achieve high performance, cost-efficiency, and scalability at the same time. Using low latency RDMA messages, ScaleStore implements a transparent memory abstraction that provides access to the aggregated DRAM memory and NVMe storage of all nodes. In contrast to existing distributed RDMA designs such as NAM-DB or FaRM, ScaleStore stores cold data on NVMe SSDs (flash), lowering the overall hardware cost significantly. The core of ScaleStore is a distributed caching strategy that dynamically decides which data to keep in memory (and which on SSDs) based on the workload. The caching protocol also provides strong consistency in the presence of concurrent data modifications. Our evaluation shows that ScaleStore achieves high performance for various types of workloads (read/write-dominated, uniform/skewed) even when the data size is larger than the aggregated memory of all nodes. We further show that ScaleStore can efficiently handle dynamic workload changes and supports elasticity.

CCS CONCEPTS

• **Information systems → Parallel and distributed DBMSs**; **DBMS engine architectures**.

KEYWORDS

Distributed Storage Engine, Transaction Processing, Flash, RDMA

ACM Reference Format:
Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526187

1 INTRODUCTION

In-memory DBMSs. Decades of decreasing main memory prices have led to the era of in-memory DBMSs. This is reflected by the vast number of academic projects such as MonetDB [8], H-Store [34], and HyPer [37] as well as commercially-available in-memory DBMSs such as SAP HANA [24], Oracle Exalytics [26], and Microsoft Hekaton [19]. However, while in-memory DBMSs are certainly efficient, they also suffer from significant downsides.

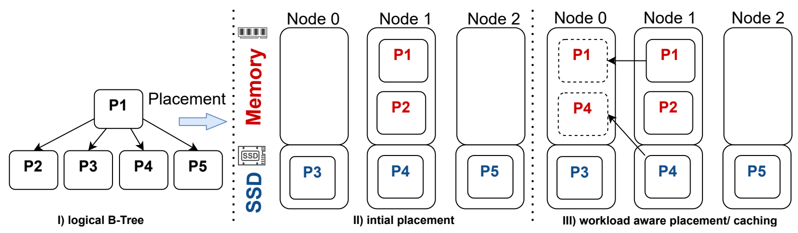
Table 1: Hardware landscape in terms of cost, latency, BW.

	Price [\$/TB]	Read Latency [μs/4 KB]	Bandwidth [GB/s]
DRAM	5000	0.1	92.0
Flash SSDs	200	78.0	12.5
RDMA (IB EDR 4x)	-	5.0	11.2

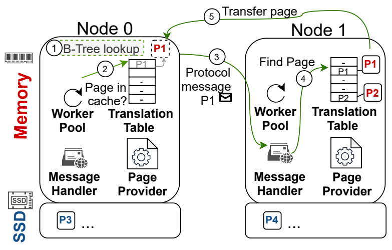
Downsides of in-memory DBMSs. An inherent issue of in-memory DBMSs is that all data must be memory resident. In turn, this means if data sets grow, larger memory capacities are required. Unfortunately, DRAM module prices do not increase linearly with the capacity, for instance, a 64 GB DRAM module is 7 times more expensive than a 16 GB module [1]. Therefore, scaling data beyond a certain size results in an "explosion" of the hardware cost. More importantly, since 2012 main memory prices have started to stagnate [27] – while data set sizes are constantly growing. This is why research proposed two directions to handle very large data sets.

NVMe storage engines. As a first direction, a new class of storage engines [38, 51] has been presented that can leverage NVMe SSDs (flash) to store (cold) data. As Table 1 (second row) shows, the price per terabyte of SSD storage is about 25 times cheaper than the price of main memory. The key idea behind such high performance storage engines is to redesign buffer managers to cause only minimal overhead on modern hardware in case pages are cached in memory. This is in stark contrast to a classical buffer manager that suffers from high overhead even if data is cache resident [28]. Recent papers [38, 51] have shown that when the entire working set (aka hot set) fits into memory, the performance of such storage engines is comparable to pure in-memory DBMSs. Unfortunately, when the working set is considerably larger than the memory capacities, the system performance significantly degrades. This is because the latency of SSDs is still at least two orders of magnitude higher than DRAM (see Table 1 second row). This latency cliff mainly affects latency-critical workloads such as OLTP.

In-memory scale-out systems. A second (alternative) direction to accommodate large data sets is to use scale-out (distributed) in-memory DBMS designs on top of fast RDMA-capable networks [7, 20, 33, 43, 53]. The main intuition is to scale in-memory DBMSs beyond the capacities of a single machine by leveraging the aggregated memory capacity of multiple machines. This avoids the cost explosion that typically arises in scale-up designs. The main observation is that scale-out systems execute latency-critical transactions efficiently via RDMA. In fact, as shown in Table 1 (third row), the latency of remote memory access using a recent InfiniBand network (EDR 4×) is one order of magnitude lower than NVMe latency. As a result, systems such as FaRM [20, 21, 56] and NAM-DB [67]



(a) Logical B-Tree and placement



(b) ScaleStore's Main Components

Figure 1: Overview of ScaleStore using a distributed B-Tree. (a) II The B-tree pages can be spread across local/remote memory and SSDs. III The caching protocol optimizes how pages are laid out across machines. (b) ScaleStore's main components.

provide high performance even for latency-sensitive OLTP workloads. However, such distributed in-memory DBMS designs still require that all data must reside in the collective main memory of the cluster. This causes unnecessarily high hardware costs when the hot set is smaller than the complete data set.

ScaleStore. Given the two directions — single-node storage engines and in-memory scale-out systems — the question remains if one can combine the best of both worlds. In this work, we propose a novel distributed storage engine called ScaleStore (code available at [15]) that provides low-latency access to the aggregated memory of all nodes via RDMA while seamlessly integrating SSDs to store cold data. In contrast to single-node storage engines, this allows ScaleStore to accommodate the hot set in the aggregated memory to avoid the latency gap. However, unlike distributed in-memory systems, ScaleStore evicts cold data to cost-efficient storage.

Challenges. While combining distributed memory with SSDs appears intuitive, it triggers several non-trivial design questions: (1) Deciding on the optimal data placement in ScaleStore is challenging due to the various storage locations (i.e., local memory, SSDs, remote memory, and remote SSDs). Clearly, a naïve static allocation scheme could be used in which the storage locations are determined upfront (in an offline manner) to support a given workload. However, this prevents efficient support for shifting workloads where the hot set is changing over time [35] or for elasticity which is a key requirement for modern scalable DBMSs, especially in the cloud. (2) Another challenge is to synchronize and coordinate data accesses. Since data in ScaleStore is distributed across storage devices and nodes it is non-trivial to achieve consistency efficiently.

Distributed cache protocol. To address these challenges, as a first core contribution, ScaleStore implements a novel distributed caching protocol based on RDMA that operates on fixed-size pages. Our distributed caching protocol provides transparent page access across machines and storage devices. As such, worker threads can access all pages as in a non-distributed system. Furthermore, an important aspect is that the distributed caching protocol dynamically handles shifts in the workload. For example, if the access pattern (i.e., which node requires which page) changes, the in-memory cache is dynamically repopulated. This means that pages are migrated from the cache of one node to another node. To enable high performance for workloads with high access locality ScaleStore dynamically caches frequently-accessed pages in DRAM on multiple nodes simultaneously. Finally, a last important aspect is that the distributed caching protocol coordinates page accesses across a cluster of nodes. This ensures a consistent view of the data despite concurrent modifications even when multiple copies are cached by several nodes.

High performance eviction. Because ScaleStore caches pages and handles workload shifts, the local DRAM buffer may fill up at very high rates, and thus unused (cold) pages have to be evicted efficiently. Existing strategies such as LRU or Second Chance are either too slow or not accurate enough. ScaleStore, therefore, employs a novel distributed high performance replacement strategy to identify cold pages and evict them efficiently. Importantly, our eviction strategy is generally applicable to arbitrary data structures, rather than being hard-coded to any particular data structure, which makes ScaleStore a general-purpose storage engine.

Easy-to-use programming abstraction. Despite being a complex system, ScaleStore offers a programming model that allows developers to implement distributed data structures in a simple manner. Typically, creating distributed data structures such as distributed B-trees is a very complex and tedious task. For instance, it is not uncommon for specialized RDMA data structures to have thousands of lines of code [73], even with hard-coded caching rules and no SSD support. The programming model of ScaleStore, in contrast, hides all this complexity and makes distributed data structure design as easy as local data structure design.

2 SYSTEM OVERVIEW

In this section, we illustrate the main concept behind our system using a motivating example and introduce the main components.

2.1 A Motivating Example

In ScaleStore, page access is transparent: any node can access any page using its page identifier (PID) and the system takes care of page placement. Consider the B-tree as shown in Figure 1(a) I, which consists of a root page (P1) and four leaf pages (P2-P5). Figure 1(a) II illustrates how the pages might initially be distributed across a cluster of three nodes: Pages P1 and P2 are cached by Node 1, while pages P3, P4, and P5 are not cached and only reside on the SSDs. P1 and P2 also have a copy on SSDs, but for brevity these pages are not shown. Note that this placement is only a snapshot, and during operation, ScaleStore dynamically re-adjusts which pages are cached based on the workload. For example, as Figure 1(a) III shows, if Node 0 performs a lookup that involves pages P1 and P4, it will replicate P1 from the remote main memory of Node 1 and P4 from the remote SSD of Node 1. Note that Node 1 did not automatically cache P4. At this point, P1 will be cached by two nodes (Node 0 and Node 1), which is highly beneficial for frequently-accessed pages like the root of a B-tree.

If Node 2 accesses page P4, then in principle, Node 2 could either obtain the page from the SSD of Node 1 or the main memory cache of Node 0. Given current hardware latencies (see Table 1), ScaleStore always chooses the latter option. As a consequence, ScaleStore is effectively capable of combining the main memories of all nodes into a combined DRAM cache connected through a low-latency RDMA network. Furthermore, since page placement is dynamic and workload-driven, other placement scenarios are possible as well, and ScaleStore will dynamically adapt to those: For example, if the read-only working set is small and fits into the cache of each node, all data will eventually be fully replicated on each node and therefore enabling high performance by avoiding the network overhead of purely distributed systems. Another possibility occurs when the workload is partitioned, i.e., each node mainly works on a different subset of the data. In this situation, over time, each node will cache specifically its subset and thus exploit locality. Finally, note that our approach naturally adapts to workload shifts at runtime.

2.2 Main Components

As Figure 1(b) shows, each ScaleStore node consists of four main components: (1) *Worker Pool*, (2) *Translation Table*, (3) *Message Handler*, and (4) *Page Provider*. In the following, we briefly illustrate how these components interact at runtime.

Consider again the example from Figure 1(a) III where Node 0 wants to access the root node (page P1) from Node 1. Figure 1(b) illustrates how the page access is implemented. The first step from the application perspective is to invoke a worker thread ① to execute an operation on a data structure (i.e., B-Tree lookup). The worker then consults the local translation table to check if page P1 is already in its cache ②. If the page is indeed in the local cache, the page ID of P1 is translated to the memory address of the cached page and returned to the application. If the page is not in the local cache, it has to be fetched from a remote node. For example, in order to request a page from Node 1, Node 0 invokes the distributed page coherence protocol. For this, a page request ③ is sent to Node 1 to request page P1. When the message handler ④ on Node 1 finds that the page is already loaded into the remote memory using its translation table, the page is ⑤ directly transferred to Node 0. Otherwise, the page is loaded from SSD into the temporary memory of Node 1 before transferring it to Node 0.

3 DISTRIBUTED PAGE COHERENCE

In this section, we describe our distributed page coherence protocol.

3.1 Protocol Overview

The basic idea of our protocol is inspired by cache coherence protocols like MESI that are used by multi-core CPUs to provide the illusion of a single unified main memory despite having multiple per-core caches and therefore duplicated copies of cache lines.

The MESI protocol. In MESI, each cache line has one of four eponymous states: (1) Modified (the cache line has been modified), (2) Exclusive (only a single copy exists), (3) Shared (there are multiple copies of this cache line), and (4) Invalidated (the cache line is out-dated). The protocol ensures coherence by using appropriate invalidation messages. For example, if a cache line is in the *Shared* state and a core wants to write to it, invalidation messages are sent to all cores that hold this cache line in *Shared* state.

Our protocol. In contrast to MESI, to work in a distributed setting, our protocol has several important differences. Instead of cache lines (usually 64 bytes), our protocol uses pages (e.g., 4 KB) as the unit of coherency to amortize network overhead and enable SSD support. These pages can be fixed in the local cache, which prevents page invalidation until an ongoing operation is finished. In a distributed cache coherence system this is required to avoid that the same page has to be fetched multiple times from a remote node due to invalidations which could quickly lead to a significant increase in overall latency. We further ensure robustness and fairness with anticipatory chaining, a technique that orders conflicts and thus ensures fairness and avoids starvation. Finally, our protocol implements sequential consistency, whereas multi-core CPUs typically implement lower memory consistency guarantees such as Total Store Order. Our protocol is separated into two distinct paths: (1) the local hot path and (2) the remote invocation. Figure 2 gives a high-level overview of the decisions in those paths. With this, ScaleStore follows the design principle "*make the common case fast*" and therefore, the local hot path is an important optimization to avoid unnecessary network messages when the page is already in the local page cache.

3.2 Local Hot Path

The local path in our protocol is a fast path that does not involve any remote messages – all decisions can be made locally. Especially for frequently-accessed pages, this obviously provides significant performance benefits. For instance, inner B-Tree pages that are rarely modified but frequently accessed are very likely to be cached on multiple nodes and can be accessed without any networking overhead. In the following, we explain the individual steps of the local hot path as shown in Figure 2.

Check ownership mode. The first step of every page access is to check the ownership mode. For this, the page translation table is used, which translates the page identifier (PID) to cache frames (similar to buffer frames in buffer managers). Besides the page data, we store the page latch, eviction information, and ownership metadata inside a cache frame. The ownership metadata describes the ownership mode for a page (i.e., what operations are allowed): (1) *node-exclusive* or (2) *node-shared*.

Before a worker thread accesses a page, it checks if the page is in the correct ownership mode. For example, the page has to be in the *node-exclusive* ownership mode for modifications (such as an update). Conversely, multiple nodes can access a page simultaneously only if the page is in the *node-shared* ownership mode. Note that node-exclusive and node-shared ownership regulate accesses on a *node-level basis*. That is, if a node owns a page in node-exclusive mode, then all worker threads can exclusively access this page.

Latch and return page. If the ownership mode is correct, the second step is that the page has to be latched for the concrete worker thread that wants to access the page. To efficiently synchronize worker threads within a node, we provide a hybrid latch [9] that combines a standard mutex with the option for optimistic access:

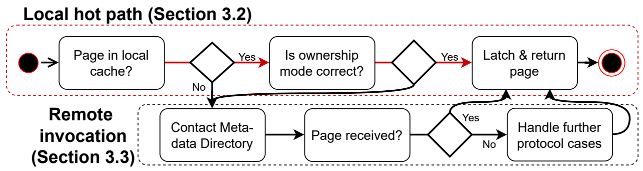

Local hot path (Section 3.2)
Page in local cache?
Yes
No
Is ownership mode correct?
Yes
Latch & return page
Remote invocation (Section 3.3)
Contact Meta-data Directory
Page received?
Yes
No
Handle further protocol cases

Figure 2: Protocol Flow.

- *Exclusive:* Acquires cluster-wide exclusive access to a page for a worker thread. Note that this first requires a transition to the node-exclusive ownership mode for this page. Once the latch is acquired the page is fixed, i.e., it cannot be evicted from the local cache until it is unlatched.
- *Shared:* Acquires shared access to a page where multiple threads (and nodes) can access it. The ownership mode for this latch can be either node-shared or node-exclusive. As before, the page is fixed in the local cache.
- *Optimistic:* Allows an optimistic page read without acquiring the latch. The ownership mode for this latch can be again either node-shared or node-exclusive. In optimistic mode, the page is not fixed and can be evicted from the local cache.

While shared and exclusive modes use a traditional OS-supported read/write mutex, the optimistic mode sidesteps the mutex. This avoids cache line invalidations for latch acquisition and is crucial for making reads scalable on multi-core CPUs. Optimistic mode relies on a version counter, which is incremented for every page modification, to detect concurrent page modifications or ownership changes. If the version has changed (or the page was evicted), the reader must restart its read operation. If restarts keep happening, one can easily fall back to the shared latching mode, which will ensure forward progress.

3.3 Remote Invocation

In cases where the page is not in the local cache, ScaleStore triggers the remote invocation path. The remote path consists of several steps as shown in Figure 2 (lower part).

To query the current ownership mode and the location of a page, we first need to contact the *directory* node. To avoid a single directory node from becoming a performance bottleneck, every node is a directory for some of the pages. This also ensures that the SSD capacities are equally utilized because pages are only persisted at the directory nodes. In Figure 1(a), for example, the directory node of page P3 is Node 0. The directory has full knowledge about the state of the page such as which other nodes currently cache the page and in which ownership mode.

Which node is the directory? In our current implementation, the directory is the node on which the page is initially created at. During the allocation of the page, a unique page id (PID) is assigned to it. The directory node id is encoded in the first 8 bits of the PID to identify the directory node from the page ID. The remaining 56 bits indicate its page slot on the SSD at the directory node. For instance, `0x010000000000000F` encodes node id 1 as the directory, and this page would be written to slot 15 on the SSD of Node 1.

Base case. Now that we know how to identify the directory node of a page from its page ID, we can request the page as shown in Figure 3 (which illustrates the base case). Node 0 – the requester – sends an *ownership request* ① to the directory. The ownership request describes to the directory what page is needed and whether

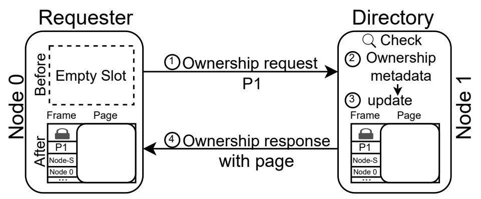


Figure 3: Base case of the Remote Invocation.

node-exclusive or *node-shared ownership* is required. In the example of our B-Tree lookup in Section 2.1, Node 0 needs node-shared ownership for page P1. The directory then checks the ownership metadata ②. This is necessary because conflicts could happen, but for now, let us assume there is no conflict. Subsequently, the ownership metadata in the cache frame is updated ③ on the directory node to reflect where the page is located. Finally, an *ownership response* ④ is sent and the page is copied to Node 0 which is then called a *node-shared owner.* Note, the node-shared owner stores the ownership mode in its local cache frame to support the local hot path as discussed before.

Conflict cases. As mentioned in step ② in Figure 3, two types of conflicts can happen during an ownership request: (1) exclusive and (2) shared conflict. A conflict in our protocol is always related to incompatible ownership modes, i.e., a requesting node (Requester) wants an ownership mode and another node already holds the page in an incompatible mode as shown below:

Requester wants:	Other Node has:	
	Node-exclusive	Node-shared
Node-exclusive	exclusive conflict	shared conflict
Node-shared	exclusive conflict	no conflict

In the following, we explain the two conflict cases.

Handling exclusive conflicts. An exclusive conflict occurs if one node requests a page of which another node is a node-exclusive owner, as Figure 4 illustrates. Regardless of whether the requester needs node-exclusive or node-shared ownership, both are incompatible with a node-exclusive ownership of another node. The first three steps are identical to the base case except that in step ② the directory will detect the conflict. Note that the directory does not necessarily have the up-to-date page content but only the metadata. In our example, Node 2 owns the page in node-exclusive mode, which implies that the page has been modified. Therefore, the directory does not store the old-page content and replies in ④ with the conflicting node id. The requester then sends an ⑤ *ownership transfer request* to Node 2, the current node-exclusive owner of the page. Node 2 transfers the page with the response ⑥ to the requester. Afterwards, Node 2 ⑦ removes the page from its cache. Notably, no acknowledgment message to the directory is required because we employ a technique called *immediate metadata updates* which we will discuss in Section 3.4.

Handling shared conflicts. In shared conflicts, a page that is in node-shared mode is requested in node-exclusive mode by another node. To handle this case, the directory node detects the conflict and sends a list of node-shared owners to the requester. The requester then chooses one of the node-shared owners (at random) and sends a transfer ownership request to transfer the page to its own cache. Afterwards, the requester sends ownership invalidation requests to all other node-shared owners, which invalidate the

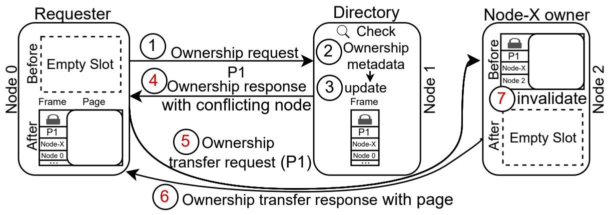

Requester
Directory
Node-X owner
Node 0
Node 1
Node 2
Before
After
Empty Slot
Frame
Page
P1
Node-X
Node 0
Node 2
1 Ownership request P1
2 Check Ownership metadata
3 update
4 Ownership response with conflicting node
5 Ownership transfer request (P1)
6 Ownership transfer response with page
7 invalidate

Figure 4: Exclusive Conflict and its Resolution.

page and ensure that the requester is the node-exclusive owner. Handling this conflict could seem expensive due to the invalidation messages, but often there are only a few node-shared owners of a page. Additionally, invalidation messages are sent in parallel to multiple nodes via low-latency messaging, as we will explain in Section 3.5.

3.4 Robustness and Fairness

After discussing the basic conflict resolution, we now describe how our protocol can efficiently handle conflicting requests coming from multiple nodes at the same time. For instance, when the B-Tree root is split or contended pages are accessed, multiple nodes compete for the same page. Such scenarios are not uncommon, and the challenge is to ensure fairness, avoid starvation, and keep latencies low.

3.4.1 ***Busy Polling.*** The naïve approach for handling multiple conflicts is to let the nodes compete through busy polling on the directory node. As illustrated in Figure 5(a), if three nodes ($N0$, $N1$, and $N2$) want to access a page in exclusive mode, all three nodes send ownership requests to the directory (D). In the example, D responds with the current exclusive owner ($N3$) to all three nodes, which in turn send ownership transfer requests to $N3$. However, only the first of the three requests can succeed (exclusive mode). Therefore, the page is sent to $N0$ and D is informed, which triggers a metadata update to reflect that $N0$ is the new exclusive owner. The other requests fail, i.e., $N1$ and $N2$ back off and contact the directory again to get the new owner ($N0$). The above sequence is then repeated until all requesters finally get the page. As Figure 5(a) shows, this approach suffers from two issues: (1) many unnecessary messages are sent (highlighted in red), and (2) starvation can happen since a node might always lose to another concurrent request.

3.4.2 ***FIFO-Queue.*** To avoid flooding the system with repeated requests for the same page, the requests can instead be queued at the directory as discussed in [10]. The requests are then served one at a time as determined by the queue. Figure 5(b) illustrates this approach in the same setting as before. The second request ($N1$) is dequeued and continued after the previous request of $N0$ succeeded, i.e., the metadata has been updated. The FIFO-queue might give a false impression that requests are ordered and fairness is achieved. However, for efficiency reasons, local workers can bypass the queue as discussed before (to enable the hot path). Therefore, as soon as the page can be accessed, the local worker threads can "steal" the page from the remote requester. For instance, in Figure 5(b), if the directory node also accesses the same page, it can intercept the page requests of the other nodes $N0 - N3$ leading to an unfair access pattern. On the other hand, enqueueing the local workers also in the FIFO-queue may solve this problem but incurs high overhead on the hot path which is prohibitively expensive.

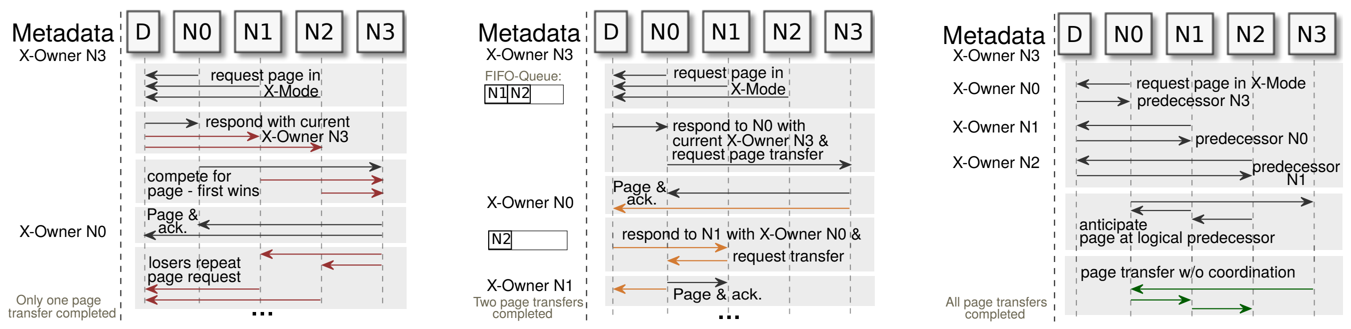
3.4.3 ***Anticipatory Chaining.*** To provide high efficiency and fairness at the same time, we propose *anticipatory chaining*. In this approach, a requester anticipates where the page will be just before its own turn (i.e., it anticipates its logical predecessor).

Immediate metadata updates. The key idea is that the metadata on the directory is directly updated once a page is requested. That way, nodes that request a page receive the anticipated page owner who will hold the page before them.

Figure 5(c) shows the general flow of our approach. Here, we see that the directory is updated immediately when the first request from $N0$ arrives at D. Moreover, when the second request from $N1$ arrives, D directly forwards $N1$ to the predecessor $N0$ (i.e., the anticipated owner who holds the page before) even though the page might not have been physically moved yet. The same holds for the third requester $N2$ which is directed to $N1$. An important aspect of our protocol is that the page is moved strictly according to the order of the requests, and the directory cannot interfere; instead, the directory is treated like any other node, and thus we guarantee fairness. For instance, if D wants to access the page after the ownership request from $N3$, it is guaranteed that $N3$ receives the page before the directory. Overall, the approach is thus in stark contrast to the previous approaches (busy polling and FIFO-queue), which update the metadata only once the page is actually transferred to the new owner. To achieve this, the other approaches send acknowledgment messages once the page is transferred. Our approach saves this coordination step entirely. Additionally, because we eschew the FIFO-queue altogether, the local hot path is very efficient without sacrificing fairness.

Owner stability. While immediate metadata updates help to achieve fairness, they can become very costly if the page is not where the requester expects it to be. For instance, if $N1$ expects the page at the predecessor $N0$ but $N0$ evicts the page in the meantime, $N1$ would need to retreat to D to get the new page location. This would require additional communication, which results in increased latencies not only for $N1$ but for all chained requests. Therefore, an important aspect that anticipatory chaining provides is what we call *owner stability*; i.e., it guarantees that the page will be at the predecessor node where the requester expects it to be.

To decide when owner stability must be guaranteed we track a *conflict epoch* per page. This conflict epoch is incremented whenever the directory detects a conflict. Every node that requests a page remembers the current conflict epoch for this page. Let us consider the following example, initially, the conflict epoch for the page, which is requested by $N0$ is 5. Therefore, when $N0$ requests the page exclusively it remembers the conflict epoch at that time, i.e., 5. When $N1$ requests the page in exclusive mode, D detects the exclusive conflict, increments the conflict epoch to 6, and responds with the current exclusive owner ($N0$). If $N0$ then tries to evict the page, an eviction request is sent to D with the conflict epoch 5. D can thus detect that the current conflict epoch does not match the epoch from the eviction request. When such a mismatch is detected, owner stability must be guaranteed. Therefore, in this case, D declines the eviction request and $N0$ simply waits until the ownership transfer request from $N1$ invalidates the page as part of the exclusive conflict resolution. This guarantees that when $N1$ expects $N0$ to be the owner, this assumption holds true.


Metadata
X-Owner N3
D
N0
N1
N2
N3
request page in X-Mode
respond with current X-Owner N3
compete for page - first wins
Page & ack.
X-Owner N0
losers repeat page request
Only one page transfer completed
FIFO-Queue:
N1 N2
respond to N0 with current X-Owner N3 & request page transfer
respond to N1 with X-Owner N0 & request transfer
X-Owner N1
Two page transfers completed
X-Owner N2
request page in X-Mode predecessor N3
predecessor N0
predecessor N1
anticipate page at logical predecessor
page transfer w/o coordination
All page transfers completed

(a) Busy Polling Approach. Red arrows highlight messages which are optimized in the FIFO approach.

(b) FIFO-Queue Approach. Orange arrows highlight messages that are optimized in Anticipatory Chaining.

(c) Anticipatory Chaining. Green arrows highlight the page transfer w/o coordination.

Figure 5: Handling multiple conflicts: Nodes N0, N1, and N2 request a Page in Exclusive (X-)Mode from Directory D.

Deadlock avoidance. Lastly, our protocol also ensures that no deadlocks occur. For example, one scenario which may lead to a deadlock is if a node upgrades from shared to exclusive ownership for a page. In this example, one node might want to copy the page from the upgrading node, but the upgrading node wants to invalidate the copying node at the same time (due to the exclusive conflict resolution). Therefore, both nodes are waiting for each other to finish and unlatch the page. In ScaleStore, we resolve such scenarios using the conflict epoch mentioned before. In the example, the upgrading node has the higher conflict epoch, and thus we detect the potential deadlock and the node with the lower conflict epoch needs to back off. Other edge cases, which we cannot describe due to space constraints, can be solved using conflict epochs as well.

Micro-benchmark. To show the effect of anticipatory chaining we execute a micro-benchmark with five nodes and one worker thread per node. In the micro-benchmark, workers either access multiple pages uniformly (*uncontended*) or all workers access a single page (*contended*). Figure 6 compares anticipatory chaining and the FIFO-queue approach. Anticipatory chaining performs generally better even in the uncontended scenario because fewer messages are sent. Moreover, in the contended scenario, anticipatory chaining achieves a 30% higher system throughput. This is because we achieve fairness between all participating nodes. In contrast, the FIFO approach has an unfair access schedule for the directory has two times higher throughput than the remote nodes. That is because the directory worker intercepts the remote requests as described in Section 3.4.2. This essentially leads to starvation and performance degradation, which is also reflected in the latency plot on the right-hand-side. Furthermore, we can observe that the latencies for anticipatory chaining are lower and the variance is significantly smaller compared to the FIFO-queue approach.

3.5 Low-Latency RDMA Messaging

We use Remote Direct Memory Access (RDMA) for all inter-node communication – including protocol messages and page transmissions. RDMA offers low latency and high bandwidth between nodes but requires careful engineering to achieve its potential. A number of RDMA implementations are available – most notably InfiniBand and RDMA over Converged Ethernet (RoCE) [62]. We use InfiniBand and Reliable Connection (RC), which guarantees that packets are delivered in order and without any loss.

RDMA implementations provide several communication primitives (so-called verbs) that can be categorized into the following two classes: (1) One-sided verbs (read/write) provide remote memory access semantics, in which one node accesses a remote node's memory over the network. The CPU of the remote node is not actively involved in the data transfer and thus is often used to save CPU cycles on the receiver. (2) Two-sided verbs (send/receive), in contrast, provide channel semantics. In order to transfer data between two nodes, the receiving node first needs to publish a receive request; thus the remote CPU is actively involved.

Can we access pages via one-sided verbs directly? There are several systems (e.g., FaRM [20] or NAM-DB [67]) that use one-sided verbs to read or even write remote data directly. This often requires careful engineering to keep data consistent when multiple updates are applied concurrently. For instance, NAM-DB uses RDMA atomic fetch-and-add operations to lock and unlock remote objects. This operation allows one to atomically modify 8 byte values from remote memory. Unfortunately, compared to CPU atomics, the network atomics are very slow and even affect other non-atomic RDMA operations [32].

An even larger problem is that RDMA atomics and CPU atomics are not compatible [4, 32, 66]. While RDMA atomics work reasonably well in NAM-DB, which has been designed with decoupled storage and compute in mind, our design provides fast local accesses, which requires synchronization between remote and local accesses. The only option would be to use RDMA atomic operations for local accesses as well. However, this incurs a latency penalty in the order of 1 μs, making local accesses slow. Consequently, ScaleStore relies on remote procedure calls (RPC) for most operations. We exploit one-sided RDMA to (1) implement efficient RPC-based message passing and (2) transfer pages between nodes.

One-sided RPC. Using one-sided RDMA to build an RPC framework is very common [3, 20, 25, 59, 72]. In ScaleStore, we use a mailbox system very similar to the one from L5 [25]. In L5 every incoming message is written to a pre-specified memory area. This area is called the mailbox and incoming requests are detected when they are written to this region. To reduce the number of connections in ScaleStore, every worker is connected to a message handler on every remote node. The message handler provides a private mailbox for every worker which is continuously monitored to detect incoming requests. When a new request arrives, the message handler processes this request and replies to the worker's

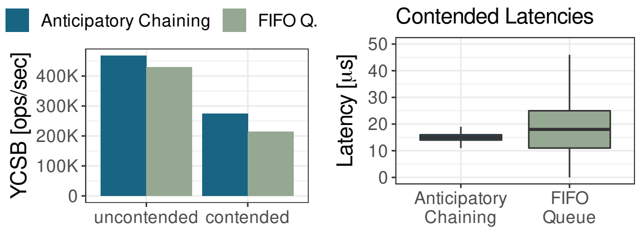


Figure 6: Effect of Anticipatory Chaining on throughput (contended and uncontended workload) and latencies (contended workload).

thread-local mailboxes. In ScaleStore all messages are cache-line size (64 byte) which makes this design very efficient as we do not need to handle variable-length messages. To avoid that the message handler becomes the bottleneck, we carefully optimized our design. For instance, we offload conflict resolution as seen in Section 3 to the worker threads of the requester instead of doing this inside the directory message handler. More specifically, the workers on the requester side implement the logic for conflict resolution. We also tried to resolve conflicts at the directory, however, it turned out that this impacts the overall performance negatively.

Page transfer using RDMA writes. The ultimate objective of our protocol is to transfer a page to the local cache. For the page transfer itself, we analyzed two possible strategies, one using one-sided reads and another using one-sided writes: (1) For one-sided reads, the message handler responds with a remote memory pointer to the page. The worker then reads (RDMA read) the page into its local cache. The benefit is that with one-sided reads the CPU of the remote message handler is not involved and that the page transfers are spread across all workers, i.e., threads. (2) The message handler directly writes the page to the cache of the worker.

Both strategies provide similar performance for the page transfer. However, we decided to use the second one for two reasons: First, the message handler can efficiently transfer pages in the background with RDMA since RDMA writes can be executed asynchronously. Second, the message handler can detect when a page transfer has been completed and immediately unlatch the page. In the first strategy, the worker however needs to send an additional message to indicate that the page transfer has finished.

Interestingly, one would assume that a single message from the worker would be enough, i.e., half roundtrip. However, due to the mailbox design, the message handler also needs to acknowledge this message. Otherwise, the previous message from the same worker could be overwritten before the message handler actually processes the first message and only sees the second message. All in all, the second strategy saves a full roundtrip.

4 HIGH PERFORMANCE PAGE EVICTION

Modern RDMA networks are fast – which means that new pages may be added to the caches at very high rates. To avoid the overall performance from being throttled due to the lack of free pages, ScaleStore needs to evict cold pages quickly while making sure that hot pages stay in the cache. To achieve these goals, we separated our eviction process into two components: (1) a low-overhead strategy to track page accesses, and (2) a dedicated background thread – the page provider – which utilizes the access information to actually evict pages. This separation of concerns allows worker threads to focus solely on query processing while the page provider handles the eviction in the background.

4.1 Epoch-Based LRU Approximation

Downsides of existing strategies. To distinguish between hot and cold pages, the access pattern has to be tracked in some way. A well-known eviction strategy is Least Recently Used (LRU), which orders pages based on their access recency by maintaining an LRU-list. Cold pages are stored at the end of the list while hot pages are at the front, and thus one can reliably classify cold pages. While LRU would evict the right pages, maintaining an LRU list incurs high overhead for every page access and can easily become a scalability bottleneck in multi-core systems. An LRU-approximation such as *Second Chance* (or Clock) may be employed to avoid this overhead. Unfortunately, *Second Chance* only classifies hot pages well (since they are typically accessed very often), but it often evicts *warm* pages instead of cold pages. Evicting warm pages may lead to a significant slowdown in ScaleStore because pages that are required need to be read from remote memory or SSD, or even worse, pages may bounce between nodes. To identify cold pages more reliably, we need a more fine-grained distinction between the degree of hotness without incurring the overhead of LRU.

Epoch-based LRU approximation. The basic idea of our LRU approximation is to use a periodically-growing global epoch counter. This global epoch counter is used to track the access time for each page. When accessing a page the current epoch is determined from the global epoch counter and stored in the cache frame. This conceptually clusters pages that are similarly cold, warm, or hot and thus creates equivalency classes. In other words, some pages may share the same last accessed epoch and hence are treated equally by the eviction strategy. For instance, the B-tree root was likely accessed in the current epoch, whereas some leaf pages might have been accessed in an earlier epoch. This approach significantly reduces the cost to track access information because the global epoch is rarely incremented and the check if a page has been accessed in the current epoch is a mere `if`-statement:

```
if(gEpoch > frame.lastEpoch) // if avoids unnecessary
    frame.lastEpoch = gEpoch; // cache-line invalidations
```

Compared to LRU, our approximation results in much higher performance (and multi-core scalability) while still accumulating enough access history to identify cold pages.

4.2 Page Provider

The main goal of the page provider is to maintain a sufficient number of free cache frames per node even under workload changes.

Sampling-based approach to find cold pages. With our epoch-based LRU approximation, the workers track access information efficiently, but what is left to discuss is how the page provider utilizes this information to find cold pages. To identify cold pages, we apply a sampling-based approach that iterates over the translation table (thus randomized). We sample N pages, sort them and determine a configurable epoch eviction threshold, e.g., 10% smallest epochs from the sample. Based on this epoch eviction threshold, pages are evicted from the cache. The sampling phase is repeated when the epoch changes or if the rate at which free pages are needed cannot be satisfied.

Evicting pages. When the page provider evicts a page, it is important to know whether it is dirty (modified) or not. As we know from Section 3, only the directory has full knowledge about the state of the page and, inter alia, if it is dirty. Therefore, there are two cases when the page provider evicts a page: (1) the evicting node is the directory, (2) or not. When the current ScaleStore node is the directory, it knows if the page is dirty. Typically, dirty pages are persisted to SSD, but when the page is replicated on other nodes, the directory can evict the page immediately. When the current ScaleStore node is not the directory of the selected page, we need to inform the (remote) directory node: An eviction request is sent to the directory, which then latches the page exclusively and checks if there are ongoing concurrent page requests with the conflict epoch. In the case of a concurrent page request, the eviction process for this page is deferred to maintain the owner stability mentioned in Section 3.4.3. When the directory discovers that the page is dirty, it copies the page to its cache and then eventually evicts it to SSD.

4.3 RDMA and NVMe Optimizations

The page providers communicate via RDMA messages with each other. To achieve high performance, we use the same setup as in the message handler, except that every page provider has a private mailbox on the remote page provider to ensure efficient communication. This allows one page provider to contact the page provider of the directory node directly.

RDMA optimizations. In contrast to the protocol messages for which low latency was required, we now optimize for high throughput. Therefore, we batch eviction candidates and send the batch to the directory. A batch has up to 100 pages that are all managed by the same directory. The batches are filled opportunistically in that the page provider tries to fill the batch, but if it does not find enough pages the batch is sent out earlier. Batching reduces the number of messages drastically and additionally enables another important optimization: For every page in a batch, the page provider on the directory checks if the page is dirty. If that is indeed the case, then the page must be persisted, i.e., first copied to the local cache of the directory. To optimize this copy request, we (1) use one-sided reads to directly copy the message from the remote node back to the directory node, and (2) we link multiple RDMA reads together. Instead of registering every single RDMA read with the NIC, we register a linked list of RDMA reads once. With that technique, we save precious CPU cycles because every operation which is explicitly registered with the NIC costs CPU cycles and the NIC can be better utilized (see doorbell batching in [32]).

NVMe optimizations. We use `libaio`, the asynchronous I/O API of the Linux Kernel, to saturate the bandwidth of high-speed NVMe SSDs when evicting pages to SSDs. To avoid OS caching effects, we open the database file with `O_DIRECT`.

5 PROGRAMMING ABSTRACTION

Designing and implementing efficient and scalable distributed data structures is a difficult task. Our abstraction allows programmers to port single-node data structures with minimal changes and ensures that all operations to the same page are sequentially ordered.

5.1 Interface

The key abstraction that ScaleStore's interface offers for application developers are different latch guards.

Latch guards for page access. In ScaleStore, we introduce latch guards that wrap around page accesses in a way that distribution is fully transparent and sequential consistency is guaranteed. As such, page guards act as a proxy for the translation from PID to the memory address and the acquisition of the correct ownership and latch modes. For each latch mode, we provide a guard:

- `ExclusiveGuard(PID)`: Latches the desired page in exclusive mode and ensures that the page is in node-exclusive ownership.
- `SharedGuard(PID)`: Latches the desired page in shared mode and ensures that the page is in node-exclusive or node-shared ownership mode (as both are compatible with reads, see Section 3).
- `OptimisticGuard(PID)`: Latches the desired page in optimistic mode, i.e., saves the version, and ensures the same ownership modes as the `SharedGuard(PID)` guard. Due to the optimistic nature a programmer needs to ensure that no concurrent changes occurred. Therefore, this guard provides a `hasChanged` method which indicates if a restart is necessary.

We further provide update and downgrade guards, e.g., an existing `SharedGuard` can be passed to a new `ExclusiveGuard` in order to upgrade from shared to exclusive. All guards have in common that they provide a `data` method to conveniently access the underlying page and the object encapsulated in that page, e.g., a B-Tree node. Moreover, with the destruction of the guards, the pages are unlatched.

5.2 Example: B-Tree Lookup

We now explain how the abstractions are applied in practice to develop the lookup operation in a distributed B-tree. In total, our B-tree code has only about 900 lines and thus is comparable to a single-node implementation. More importantly, the implementation is as easy as for a non-distributed B-tree. In our evaluation in Section 6 we use this distributed B-Tree implementation. The abbreviated C++ lookup code looks as follows:

```cpp
bool lookup(KeyType key, ValueType& returnValue) {
   restart:
   OptimisticGuard g_parent(catalogPID); // get catalog
   // get rootPID from catalog ...
   OptimisticGuard g_node(rootPID);
   if (g_parent.hasChanged()) goto restart;
   auto node = g_node.data<NodeBase>();
   if (g_node.hasChanged()) goto restart;
   while (node->type.isInner()) { // traverse inner nodes
     auto& inner = reinterpret_cast<Inner&>(node);
     if (g_parent.hasChanged()) goto restart;
     PID nextPid = inner.children[inner.lowerBound(key)];
     if (g_node.hasChanged()) goto restart;
     g_parent = std::move(g_node); // node becomes parent
     g_node = OptimisticGuard(nextPid);
     node = g_node.data<NodeBase>(0); // get next node
     if (g_node.hasChanged()) goto restart;
   }
   auto& leaf = reinterpret_cast<Leaf&>(node);
   SharedGuard sg_node(std::move(g_node));
   // leaf latched; search key and return it ...
}
```

Synchronization is done using optimistic lock coupling [39, 40], and consequently we traverse pages using two optimistic guards: For the parent page `g_parent` (Line 2) and `g_node` for the current page (Line 4). The root node is stored in a catalog page, which is buffer-managed the same way as the B-tree itself. After every optimistic page access (Lines 5, 7, 10, 16), we validate whether that node was modified concurrently and restart if it was. Lines 8-17 traverse the inner nodes, at each level replacing the parent with the current node. Once we arrive at the leaf page (Line 18), we upgrade its optimistic guard to a shared guard.

Other data structures. As long as data is stored on fixed-size pages, the programming interface can be used to implement arbitrary data structures, e.g., hash table, barrier, or columnar storage.

6 EVALUATION

In this section, we investigate the performance and scalability of ScaleStore (available at [15]) and compare it with other systems.

6.1 Experimental Setup

We conducted our experiments on a 5-node cluster running Ubuntu 18.04.1 LTS, with Linux 4.15.0 kernel. Each node is equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores), 512 GB main-memory split between both sockets, and four Samsung SSD 980 Pro M.2 1 TB SSDs connected via PCIe by one ASRock Hyper Quad M.2 PCIe card. We use the Linux' md software RAID 0 implementation and use direct block device access. The nodes are connected with an InfiniBand network using one Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x, 100 Gbps) per node.

When not noted otherwise, we configured ScaleStore as follows: Every node has a 150 GB in-memory cache, and we use 4 KB pages. We use 20 worker, 4 message handler, and 2 page provider threads (pinned to NUMA 0). We use an optimistically-latched B-Tree implemented on top of ScaleStore using the programming abstractions from Section 5. In all experiments, the benchmark drivers are implemented in C++ and compiled together with ScaleStore into one binary.

Workloads. We use YCSB, a widely-used OLTP-style benchmark [17]. For all experiments, the key-value pairs use 8 byte keys and the values are randomly generated strings of 128 bytes. We use the following workloads: *100% Reads*, *95% Reads & 5% Writes*, *50% Reads & 50% Writes*, and *5% Reads & 95% Writes*. Reads are *point lookups* and writes are *point updates*. While the default distribution is uniform for most experiments, we also evaluate skewed workloads with a C++ Zipf generator [16]. Furthermore, the number of clients is defined by the number of worker threads, i.e., 20 clients per node. We execute one operation on a single client until completion, i.e., we do not batch operations nor execute them asynchronously. We only limit the throughput to achieve varying target throughput in our latency experiment in Section 6.7.3. We used a 30 second warm-up phase to measure the steady-state performance followed by three experiment phases of 30 seconds each. When not noted otherwise, we report the average system throughput, i.e., the accumulated average performance of every node. Finally, we use a distributed B-Tree which is implemented on top of ScaleStore for serving all operations.

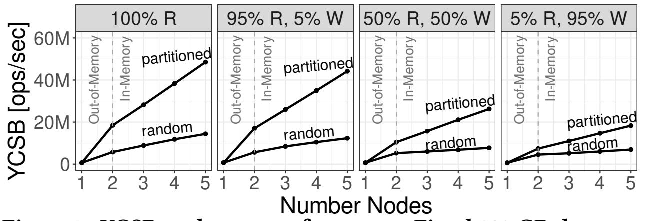


Figure 7: YCSB scale-out performance. Fixed 280 GB data set.

6.2 Scale-Out

We first conduct a scale-out experiment with the different YCSB workloads where we scale from 1 to 5 nodes and keep the data set size, i.e., hot set, constant at 280 GB. Hence, we show the different scenarios ScaleStore is designed for: if only one node is used, the data does not fit in the cache (150 GB). From 2 nodes on, however, the data can be fully cached. Furthermore, we use two access patterns: (1) a partitionable workload where nodes only cover distinct key ranges (and thus only parts of the data need to be cached per node) and a (2) random workload where workers access the full key range. This random access pattern is certainly extreme since all data is requested by all nodes. For that access pattern, the data set is too large that it can be fully cached at each node and therefore the performance is bound by the network latency.

Figure 7 shows the results of the experiment. When the workload is partitionable, ScaleStore achieves its peak performance with 5 nodes of up to $50M$ ops/sec in the read-only workload and $20M$ ops/sec in the update heavy workload. When the accesses are randomly spread across the cluster ScaleStore still scales and achieves around $15M$ ops/sec for read-only and $8M$ ops/sec for the update heavy workload. An interesting finding is that the performance for the update heavy workload (95% writes) is strictly bound by network latency. In fact, this is already the case for the 50% writes workload, which is why both workloads have similar performance. However, the relative performance difference between read-only and more write-heavy workloads can be observed in both access patterns partitioned and random. For random accesses it is slightly higher due to the average cache utilization, i.e., more cache misses. With 5 nodes the average cache utilization in the update heavy workload is just 35.1% compared to 98.7% in the read-only workload which implies that pages are frequently invalidated.

When looking at the speedup between single-node performance of $800K$ ops/sec and 5 nodes we can observe that this is an increase between 10 and 60 times. This shows that when the hot set outgrows the memory, the performance can be considerably increased when scaling-out to avoid the latency cliff of SSDs.

Ablation study. To better understand how ScaleStore achieves its performance, we now dissect how each optimization affects the performance of ScaleStore. For the experiment, we use the random read-only performance with 5 nodes from Figure 7. First, we disable all optimizations and then enable them step-by-step. The baseline system implements the high-level ideas of the protocol without any RDMA optimizations, a classical queue-based LRU-eviction strategy, a traditional single latched translation table, and a pessimistic lock-coupled B-Tree. The baseline system only achieves $1M$ ops/sec which already increases to around $3M$ when enabling the RDMA and message handler optimizations. Interestingly, already under this load, the standard translation table became

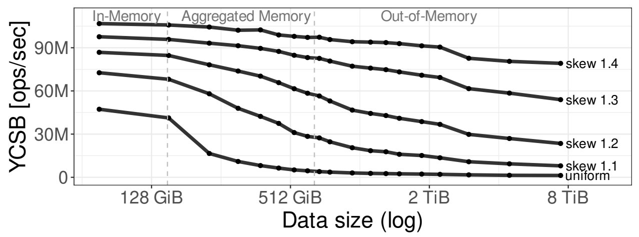


Figure 8: Data scalability with varying skew.

a bottleneck due to the cache-line invalidations with a single reader-writer latch. Therefore, we designed our own optimistically-latched translation table which increased performance to 8*M* ops/sec. We resolved the next bottleneck, LRU-eviction, with our epoch-based LRU-approximation to increase performance to around 12*M* ops/sec. We finally partitioned our translation table to remove the single latch when inserting and deleting pages and used the B-Tree implementation as shown in Section 5 to achieve the final performance of around 15*M* ops/sec. All in all, to build a high performance distributed storage system many bottlenecks needed to be resolved before efficiently leveraging RDMA.

6.3 Data Scalability

Let us now focus on data scalability. We use all 5 nodes and analyze the read-only performance when increasing the YCSB data set size from 75 GB to 7.5 TB. Furthermore, we examine the effect of varying degrees of locality (skew). The results are shown in Figure 8. Overall, the results demonstrate that with ScaleStore we can gracefully bridge the latency cliff when data spills out from the local to aggregated cluster memory and then to SSDs.

We now look in more depth at the uniform (random) access pattern. What we can see is that with a data size of 150 GB and smaller, the performance is at its peak because the data set can be fully cached on all nodes. When increasing the data size, the performance degrades as expected. If the data does not fit into the local caches, pages need to be fetched from remote nodes (either from remote memory or SSDs) which increases page access latency as shown in the following table:

Median	**99.9th Percentile**	**Data size**	**Storage**
1.7 μs	3.0 μs	75 GB	in-memory
12.1 μs	30.6 μs	350 GB	aggregated memory
79.3 μs	178.8 μs	7500 TB	out-of-memory

Finally, ScaleStore benefits tremendously from locality in the access pattern as shown in Figure 8. This is because the hot path of our protocol can be exploited more frequently and our epoch-based eviction can reliably find cold pages. We see that even with a low skew of 1.2, ScaleStore achieves around 22*M* ops/sec with a data set size of 7.5 TB. The more locality the higher the performance resulting in around 85*M* ops/sec with 1.4 skew and 7.5 TB data size.

6.4 ScaleStore vs. GAM

Closest to the functionality of ScaleStore is GAM [10] (available at [12]), which is a state-of-the-art Distributed Shared Memory system using an RDMA-based cache coherence protocol. Different from ScaleStore, GAM is a pure in-memory system that provides a unified address space across a cluster of nodes but it does not provide the possibility to evict data to SSDs. Unfortunately, we needed to reduce the total data set size to 30 GB since GAM's hash-table performs suboptimally with larger data sets. Furthermore, we split the evaluation in a single-node scale-up and a single-thread scale-out experiment. We did not use a multi-threaded scale-out experiment as before, since GAM did not work in this setup even after thorough investigations (possible reasons are the different Mellanox drivers and compiler versions). The results of the experiment are shown in the following table:

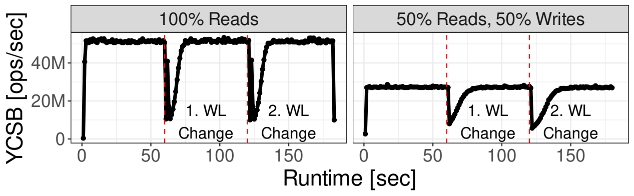


Figure 9: In-memory performance with workload shift.

Configuration		100% Reads ScaleStore	100% Reads GAM	50% Reads/Writes ScaleStore	50% Reads/Writes GAM
Scale-up (1 Node)	1 Thread	**712K**	558K	**690K**	534K
	20 Threads	**12,465K**	2,654K	**11,566K**	2,752K
Scale-out (1 Thread)	2 Nodes	**1,449K**	513K	**396K**	250K
	4 Nodes	**3,075K**	850K	**625K**	546K

When looking at the results, we can observe that ScaleStore outperforms GAM in all workloads, most notably is ScaleStore's performance with 20 threads. One important aspect is that GAM uses a slightly modified YCSB runner with a hard-coded value for updates instead of randomly generating 128-byte values; thus, we use the same setup (see here).

6.5 Workload Change

Next, we show ScaleStore's ability to adapt to changing access patterns. Therefore, we partition the data uniformly across our 5-node cluster, such that every partition is 120 GB.

Initially, every node accesses only data from its local partition which results in a situation where the partition can be kept in the local cache. Then, in regular intervals, we reassign the partitions. For instance, we reassign Node 0 to access data in the partition of Node 1 instead of accessing its local data. This procedure is repeated multiple times to show that ScaleStore can reliably handle workload changes. While this workload may not be realistic, it shows the extreme case in which the cache needs to be re-filled completely while the eviction strategy handles this sudden workload shift.

The results are shown in Figure 9 for the YCSB read-only and mixed workload. As we can see, ScaleStore recovers extremely fast already after a couple of seconds to the baseline performance of 50*M* respectively 28*M* ops/sec.

6.6 Elasticity

In the previous section, we showed how our protocol handles workload changes but with a fixed cluster size. In the following, we evaluate the elasticity in a decoupled storage and compute architecture which is typically used by cloud DBMSs today. We use two

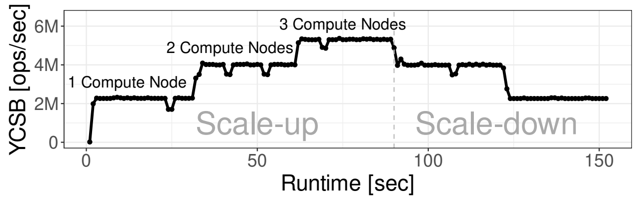


Figure 10: Elasticity in a decoupled architecture.

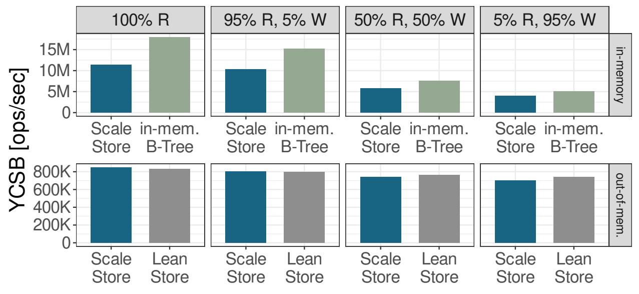


Figure 11: Performance comparison on a single node system.

storage nodes that accommodate a 250 GB data set. To show the elasticity, we scale the number of compute nodes at runtime. Every compute node only has 10 GB of cache to avoid that most data is replicated to the compute layer. We start with a single compute node and add another compute node every 30 seconds until we reach three nodes, after which we disable them one by one again. In Figure 10 we can see that ScaleStore leverages the additional compute resources and that performance scales. Overall, the adaptation happens in a few seconds after a new compute node has been added.

6.7 System Comparison

In this section, we provide a system comparison to specialized systems in the following settings: single-node in-memory, single-node out-of-memory, and distributed in-memory.

6.7.1 ***Single-node In-Memory.*** In the single-node in-memory scenario, we compare ScaleStore to a state-of-the-art in-memory B-tree [14, 64] that uses optimistic-lock-coupling (OLC). We use 400M records (120 GB) to ensure that the entire B-Tree fits in memory. Moreover, both the pure in-memory B-tree and ScaleStore's B-tree have the same page layout and synchronization protocol. This allows us to quantify the overhead of ScaleStore being able to scale-out and handle out-of-memory workloads.

The results of the comparison are shown in Figure 11 (upper row). We can observe that the overhead in the read-only workload is more prominent than in the write-heavy workloads. This is because the optimistic scheme for read-only workloads is very efficient and thereby highlights the extra work of ScaleStore. In ScaleStore, for every page access, a page identifier is translated to the corresponding memory pointer. While this indirection adds some overhead, it is necessary to be able to scale-out and handle out-of-memory workloads (i.e., pages on SSDs). The overhead diminishes with higher write ratios, for which CPU cache line invalidations and lock contention dominate.

6.7.2 ***Single-node Out-of-Memory.*** Next, we focus on experiments with data sets that are larger than memory and compare

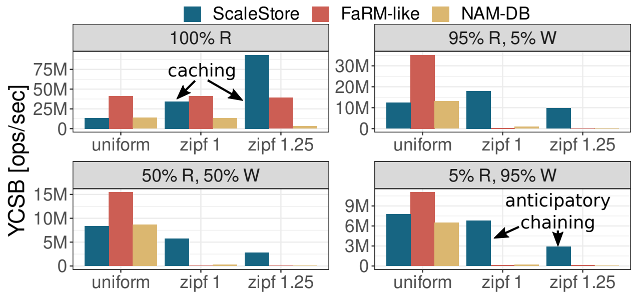


Figure 12: Distributed in-memory throughput comparison.

ScaleStore with LeanStore (available at [13]), a recent high-performance storage engine. For LeanStore and ScaleStore, the cache size is set to 150 GB while we use 1B YCSB records (300 GB). As such, the data size is twice the in-memory capacity. Figure 11 (lower row) shows that both systems perform nearly identical because both systems are I/O bound. This shows that ScaleStore integrates NVMe SSD efficiently.

6.7.3 ***Distributed In-Memory.*** Moving beyond a single-node deployment to a distributed setting.

Competitors. In this section, in contrast to ScaleStore, all competitors are pure distributed in-memory systems that do not come with the overhead of integrating SSDs. One key feature of ScaleStore is that we do not require static partitioning but dynamically adapt on runtime. For this reason, we compare with *NAM-DB* [67]. NAM-DB does not rely on static partitioning either as it reads remote data with one-sided reads and updates remote data with one-sided writes respectively. To be fair we disabled transactions to cleanly compare the protocol overhead with ScaleStore.

Different from ScaleStore, NAM-DB uses shared and exclusive latches which are (un)latched with RDMA atomics, i.e., one-sided. The second competitor is *FaRM* [20] which in contrast to NAM-DB uses an optimistic synchronization scheme. Since FaRM is not publicly available, we re-implemented a FaRM-like approach based on lock-free one-sided reads as described in [20]. The lock-free one-sided read retrieves the remote value and to check if the read was consistent, we validate the versions stored in every CPU cache line. For updates, the lock-free one-sided read is used to copy the record to a thread-local buffer, the record is then modified and sent back to the owner. The owner checks the version(s) of the modified record against the local version(s), if they are equal the modified record is installed. Finally, for both competitors, we use a pre-allocated array (key-value pairs collocated), i.e., no collisions, which allows them to use a single one-sided read to fetch remote data. This is different from ScaleStore which uses a distributed B-tree that also supports range queries.

Throughput. For this experiment, we use 1B YCSB-records, distributed across 5 nodes. Figure 12 shows the throughput for uniform and skewed accesses. First, we focus on the uniform access pattern. When looking at the read-only workload, one can see that FaRM outperforms NAM-DB and ScaleStore. This is because only a single very efficient one-sided lock-free read is needed to read a remote record, while NAM-DB requires 3 operations, i.e., two RDMA atomics and one one-sided read, and is thus bound by the latency of those operations. Similar to NAM-DB, ScaleStore is also latency bound. This is because only 50% of the data can be cached in this workload, and thus pages are read from remote nodes. Additionally, hot pages cannot be predicted with a uniform workload reliably.

With the skewed access patterns, the results look different. For the read-only workload, we can observe that with more locality (skew) the performance of ScaleStore increases. With a light skew of 1, the performance is comparable to FaRM, and with 1.25 we outperform them. In contrast to FaRM, NAM-DB suffers from locality even in the read-only scenario. The reason is that RDMA atomics limit the performance drastically; the higher the skew the more requests are routed to a subset of the nodes which cannot sustain the number of RDMA atomic-operations [32]. Finally, when looking at the skewed write-heavy workloads, we can see that ScaleStore outperforms NAM-DB and FaRM and also provides a robust performance for higher skews (contention). In the write-heavy workload (95% writes), the performance of NAM-DB and FaRM drops significantly whereas the performance of ScaleStore degrades gracefully. While FaRM and NAM-DB compete in a busy polling manner (with exponential back-off), we use anticipatory chaining as discussed in Section 3.4.

Latency. In addition to throughput, we additionally analyzed the latencies for the read-only and the write-heavy workload with uniform and skewed (1.25) access patterns. For this experiment, we increase the target throughput until the system cannot sustain the required throughput anymore. Based on the target throughput, every worker is assigned a schedule when it needs to send the next operation. This schedule is generated based on a Poisson process, where the time between two operations is exponentially distributed. When a worker misses a scheduled operation, we send it as soon as possible and include the wait time, i.e., we correct the latencies. This means that latencies spike as soon as the target throughput is not sustainable anymore.

We show the median and the 99th percentile latencies in Figure 13. As expected in the uniform distribution the latencies of NAM-DB and ScaleStore spike before FaRM since FaRM can sustain the target throughput longer. ScaleStore's latencies are comparable with NAM-DB even though NAM-DB has a very simple access scheme, i.e., two RDMA atomics and one RDMA read, whereas our protocol is more complex and uses larger transfer units (4 KB pages due to the SSD integration), yet it only adds a marginal overhead. Moreover, as mentioned before, in the read-only workload we only cache 50% of the workload and constantly fetch new pages from remote nodes. The eviction batches pages to sustain the rate of incoming pages and to maintain enough free space. Batching increases the 99th percentile latency because a worker may wait for a page that is currently in the eviction process. However, the median latency is not affected and thus comparable to NAM-DB.

In the skewed read-only workload, we can observe the benefit of caching in ScaleStore. Whereas in the skewed write-heavy workload, anticipatory chaining allows ScaleStore to achieve a higher target throughput than both competitors.

Discussion. To summarize, the results show that our system provides robust performance across different workloads and even outperforms pure distributed in-memory systems for skewed workloads due to caching and anticipatory chaining. This is especially prevalent in the write-heavy contended scenarios in which the performance degrades gracefully, compared to our baselines.

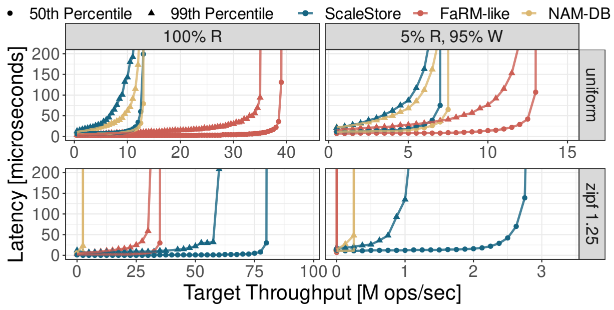


Figure 13: Distributed in-memory latency comparison.

Let us mention that we handle out-of-memory workloads and as a result require an additional indirection. This prevents ScaleStore to use FaRM's one-sided lock-free reads and thus have a slightly higher latency in the uniform workloads. Moreover, important to note is that we measured the best case scenario (upper-bound) for NAM-DB and FaRM, where we know exactly where data needs to be read; i.e., they only require a single one-sided read to fetch the data. In the case where deletes and inserts need to be supported, typically direct reads can not be used in NAM-DB and FaRM since additional indexes are required to find the memory location. Hence, we argue that the benefits of our implementation which also supports such cases outweigh the slight performance overhead.

Another interesting finding was that ScaleStore can benefit from the locality. In this case, our system can leverage caching and exploit local DRAM. In contrast, NAM-DB cannot fully use locality, even when data is fully partitioned, because the RDMA atomics are not compatible with local CPU-atomics, which means that NAM-DB requires two RDMA atomics even when accessing local data [67]. While FaRM can fully exploit locality they do not provide a general-purpose caching solution. FaRM has hard-coded rules for some data structures, i.e., cache inner nodes of their B-Tree implementation. Since they do not have a general caching protocol, they can only exploit locality if such hard-coded rules are provided. ScaleStore instead provides a general-purpose caching solution combined with a high performance eviction that works for arbitrary data structures.

7 RELATED WORK

This paper is related to the following distinct lines of work:

RDMA-enabled Systems. Recent work on RDMA-enabled in-memory DBMSs [5, 20, 21, 33, 46, 47, 56, 67–70, 73] and key-value stores [31, 44, 48, 54] achieve unprecedented performance for distributed systems. However, most do not support high performance NVMe SSDs to economically store cold data as ScaleStore.

In a similar spirit as ScaleStore, some of the other systems [20, 21, 56, 67] expose a distributed shared address space in which every node can access every data item. To exploit the shared address space, one-sided RDMA operations are used to directly access remote data and traverse data structures such as B-Trees or hash tables. To prevent multiple round-trips such systems rely on different approaches: Either data is merged with the index [20, 21, 56] or the index is fully or partially replicated [11, 49] at all servers, which has other downsides such as keeping data consistent under replication. Kalia et al. [33] thus avoid replication of the index and partition the data set. However, when workload changes they need to re-partition the data to exploit locality or fall back to expensive distributed transactions. In contrast to these approaches, ScaleStore instead provides a dynamic caching protocol that automatically adapts the placement and replication of data to the workload. Redy [70] is a cloud service that provides high performance caches by extending the local memory with remote memory of unutilized remote machines. Besides these research proposals, several industrial-strength products have adopted RDMA in existing DBMSs [30, 52, 53]. Oracle RAC [53], for example, has RDMA support, including the use of RDMA atomic primitives. Microsoft SQL Server [43] uses RDMA to extend the buffer pool of a single node. They leveraged the lower latencies of RDMA to evict pages to remote memory instead of SSD but do not discuss the effect on distributed databases at all.

Distributed Shared Memory (DSM). Distributed shared memory exposes the memory of a collection of machines as a shared memory space. There are two flavors of DSM: (1) systems using coherence protocols [10, 36, 57] and (2) systems based on partitioning the global memory space (PGAS) [50]. From the first category, GAM is closely related to ScaleStore because it builds an RDMA-based cache coherence protocol. However, while the protocol may be related we employ multiple optimizations and outperform GAM in our evaluation. Additionally, unlike ScaleStore, GAM does not support transparent access to local or remote NVMe SSDs. In contrast to ScaleStore which uses sequential consistency, GAM uses a weaker consistency model – partial store order (PSO) – which in turn requires explicit placement of fences to achieve sequential consistency. Furthermore, ScaleStore scales better and its protocol enables anticipatory chaining. Pröbstl et al. [55] extended GAM to enable file I/O. The second system category (PGAS) does not abstract away remote data accesses, instead, remote accesses are explicit. However, this makes programming much more challenging compared to ScaleStore's abstraction. Additionally, PGAS systems often do not employ caching and thus cannot profit from locality.

Dynamic Re-partitioning & Live Reconfiguration. To dynamically re-partition data a number of papers [35, 45] focused on software transactions with ownership semantics. Zeus [35], for example, acquires all objects involved in a transaction and first moves them to the same server. This allows Zeus to avoid complex distributed transaction protocols and instead execute a single-node transaction on the re-partitioned objects. A key part of Zeus' contribution is an ownership protocol that bears similarities to our protocol, thus Zeus' underlying idea is related to ScaleStore. However, while ScaleStore handles NVMe SSDs and provides a general caching strategy, Zeus is in-memory only and does not discuss eviction strategies at all. Instead, they focus on transactions and their custom commit protocol. Because NVMe SSDs require different design decisions, such as fixed-size pages or asynchronous I/O, ScaleStore's protocol implementation is, despite some similarities, quite different. Live reconfiguration systems [23, 58, 65] only re-partition the shards online when (severe) workload imbalances are detected. This process is done to balance load at runtime while minimizing the impact on running transactions.

Cloud DBMSs. Cloud DBMSs such as Snowflake or Aurora [18, 61] often decouple the storage from the compute layer. To improve performance both systems deploy a caching solution. For instance, Snowflake uses consistent hashing to assign queries to the nodes which already cached the data. Queries are executed against a fixed set of immutable blocks and thus the cached content is not modified. Instead, a new block with the updated data is created and old blocks are evicted by an LRU-scheme. Crystal [22] improves caching in disaggregated architectures by reusing computations across multiple queries. We think of ScaleStore as a distributed storage engine that simplifies the development of disaggregated systems. ScaleStore can include NVMe SSDs on the storage layer and with our unified caching strategy speed up the compute layer.

Single-node Storage Engines. Now moving from distributed settings to modern single-node storage engines [2, 6, 38, 41, 42, 51, 60, 71]. Some systems leverage NVMe SSDs while others focus on non-volatile memory (NVM) to efficiently handle larger-than-memory workloads. NVM performance is quite different from NVMe SSDs and closer to DRAM performance, however, NVM is also more expensive [27] thus we focus on NVMe SSDs in ScaleStore. Single-node high performance storage engines with NVMe SSDs suffer from performance degradation if the hot set outgrows the memory due to the high latencies of NVMe SSDs.

8 CONCLUSION AND FUTURE WORK

This paper introduced ScaleStore, a novel distributed storage engine for scalable DBMSs designed for DRAM, NVMe, and RDMA. As we have shown in our evaluation, ScaleStore can scale to large data sets efficiently even if the data set outgrows the aggregated main memory capacity of the full cluster. Furthermore, due to its carefully-designed distributed caching and eviction strategies, ScaleStore can not only efficiently adapt to workload changes but also support very different distributed DBMS architectures and requirements such as elasticity. In addition, we provide an easy-to-use programming abstraction for implementing distributed data structures managed by ScaleStore.

We think that ScaleStore can serve as the foundation for a distributed DBMS. In the following, we thus sketch ideas for future work. For example, a two-phase-locking concurrency control scheme can be implemented by physically storing locks within the tuples. This way, just like the data itself, locks are automatically distributed and managed through cache coherency. Another relevant observation is that because our protocol always moves the data to the processing node, we do not need a two-phase commit – even though we are a distributed system. Instead of a two-phase commit, each node would have its own distributed ARIES-style WAL for recovery [63]. To avoid having to flush all distributed logs on commit, a technique called Remote-Flush Avoidance [29] can be used to commit non-overlapping transactions without any inter-node synchronization. Finally, a distributed setting also raises the question of high availability when nodes fail. We leave the modification of our cache protocol to support replicas for future work.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation priority program 2037 (DFG) under the grants BI2011/1 & BI2011/2, the DFG Collaborative Research Center 1053 (MAKI), the BMBF and the state of Hesse as part of the NHR Program. We also thank hessian.AI for the support.

REFERENCES

[1] Industry Perspectives | Nov 12. 2015. Don't forget about Memory: DRAM's Surprising role in the high cost of data centers. https://www.datacenterknowledge.com/archives/2015/11/12/dont-forget-memory-drams-surprising-role-high-cost-data-centers

[2] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013).

[3] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. 2019. DPI: The Data Processing Interface for Modern Networks. In *CIDR*.

[4] InfiniBand Trade Association. 2000. InfiniBand Architecture Specification, Release 1.0, 2000. http://www.infinibandta.org/specs.

[5] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *SIGMOD*.

[6] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *PVLDB* 14, 9 (2021).

[7] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *PVLDB* 9, 7 (2016).

[8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[9] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*.

[10] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *PVLDB* 11, 11 (2018).

[11] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *EuroSys*.

[12] GAM Code. 2018. https://github.com/ooibc88/gam

[13] LeanStore Code. 2022. https://github.com/leanstore/leanstore

[14] OLC B-Tree Code. 2018. https://github.com/wangziqi2016/index-microbench/blob/master/BTreeOLC/BTreeOLC.h

[15] ScaleStore Code. 2022. https://github.com/DataManagementLab/ScaleStore

[16] Zipf Generator Code. 2021. https://github.com/opencog/cogutil

[17] 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*.

[18] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*.

[19] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*.

[20] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*.

[21] Aleksandar Dragojevic, 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 *SOSP*.

[22] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *PVLDB* 14 (2021).

[23] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *SIGMOD*.

[24] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4 (2011).

[25] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *ICDE*.

[26] Gabriela Gligor, Silviu Teodoru, et al. 2011. Oracle exalytics: engineered for speed-of-thought analytics. *Database Systems Journal* 2, 4 (2011), 3–8.

[27] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[28] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD*.

[29] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*.

[30] IBM. [n.d.]. Moving from a TCP/IP protocol network to an RDMA protocol network. https://www.ibm.com/docs/en/db2/11.1?topic=tfsai-moving-from-tcpip-protocol-network-rdma-protocol-network

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *SIGCOMM*.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. *login Usenix Mag.* 41, 3 (2016).

[33] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*.

[34] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB* 1, 2 (2008).

[35] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: locality-aware distributed transactions. In *EuroSys*.

[36] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory. In *HPDC*.

[37] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*.

[38] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*.

[39] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* (2019).

[40] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.

[41] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. 2019. Persistent Buffer Management with Optimistic Consistency. In *DaMoN*.

[42] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *ICDE*.

[43] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *SIGMOD*.

[44] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*.

[45] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *SIGMOD*.

[46] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*.

[47] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. In *SIGMOD*.

[48] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*.

[49] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *USENIX ATC*.

[50] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *USENIX*. Santa Clara, CA.

[51] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[52] NVIDIA. 2012. Mellanox InfiniBand Helps Accelerate Teradata Aster Big Analytics Appliance. https://www.mellanox.com/news/press_release/mellanox-infiniband-helps-accelerate-teradata-aster-big-analytics-appliance

[53] Oracle. 2012. Delivering Application Performance with Oracle's InfiniBand Technology.

[54] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2009. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Oper. Syst. Rev.* 43, 4 (2009).

[55] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. 2021. One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA. In *ADMS*.

[56] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *SIGMOD*.

[57] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *SoCC*.

[58] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing. *PVLDB* 8 (2014).

[59] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: The Data Flow Interface for High-Speed Networks. In *SIGMOD*.

[60] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*.

[61] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice,

Tengiz Kharatishvili, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD*.

[62] Jérôme Vienne, Jitong Chen, Md. Wasi-ur-Rahman, Nusrat S. Islam, Hari Subramoni, and Dhabaleswar K. Panda. 2012. Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems. In *HOTI*.

[63] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *PVLDB* 7, 10 (2014).

[64] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*.

[65] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *USENIX*.

[66] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*.

[67] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The End of a Myth: Distributed Transactions Can Scale. *CoRR* abs/1607.00655 (2016).

[68] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-aware Partitioning in Parallel Database Systems. In *SIGMOD*.

[69] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2021. Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks. *SIGMOD Rec.* 50, 1 (2021).

[70] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2021. Redy: Remote Dynamic Memory Cache. *CoRR* (2021).

[71] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD*.

[72] Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2020. RDMA Communciation Patterns. *Datenbank-Spektrum* 20 (2020).

[73] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *SIGMOD*.