ScaleEvict: Altruistic Eviction for RDMA-Enabled Distributed Storage Engines

Till Steinert
Technical University of Munich
Munich, Germany
till.steinert@tum.de

Muhammad El-Hindi
Technical University of Munich
Munich, Germany
muhammad.el-hindi@tum.de

Tobias Ziegler
TigerBeetle
San Francisco, USA
tobias@tigerbeetle.com

Viktor Leis
Technical University of Munich
Munich, Germany
leis@in.tum.de

Carsten Binnig
TU Darmstadt
Darmstadt, Germany
DFKI
Darmstadt, Germany
hessian.AI
Darmstadt, Germany
carsten.binnig@cs.tu-darmstadt.de

Abstract

Modern hardware and economic trends are driving the adoption of distributed storage engines that expose a transparent, shared-cache abstraction: any node can access both the cluster's aggregate DRAM and its NVMe storage over a fast (RDMA) network. To sustain performance under changing workloads, these systems must continuously evict and re-cache pages at very high rates. However, most designs still rely on node-local eviction algorithms such as LRU, which waste aggregate DRAM by retaining redundant page copies. We propose *ScaleEvict*, an altruistic eviction strategy implemented in the state-of-the-art ScaleStore engine. ScaleEvict efficiently coordinates eviction decisions across nodes to reduce redundant replication and prioritize globally valuable pages. ScaleEvict matches ScaleStore's throughput using only two-thirds of the DRAM. Alternatively, at equal DRAM capacity, ScaleEvict improves throughput by up to 2× while reducing 99th percentile latency by up to 3×.

CCS Concepts

• **Information systems** → *Parallel and distributed DBMSs.*

ACM Reference Format:
Till Steinert, Muhammad El-Hindi, Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2026. ScaleEvict: Altruistic Eviction for RDMA-Enabled Distributed Storage Engines. In *22nd International Workshop on Data Management on New Hardware (DaMoN '26), May 31–June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3789237.3809123

1 Introduction

Distributed page eviction. Page eviction (buffer replacement) has a long and active literature in the data management and systems communities, from classical DB buffer management to modern storage-aware designs [10, 15, 17–19, 26, 30, 32, 33]. In a local setting, a database management system (DBMS) evicts pages from an in-memory buffer pool to directly attached storage, aiming to minimize I/O and latency. In distributed settings, multiple nodes cache pages whose durable or authoritative copies reside remotely (e.g., on another node or a separate storage layer) [3, 7, 27, 28, 34, 35, 40, 42, 46]. These settings introduce cluster-level objectives (e.g., minimizing expected access cost and avoiding redundant replication), that purely local eviction decisions cannot satisfy [12, 16].

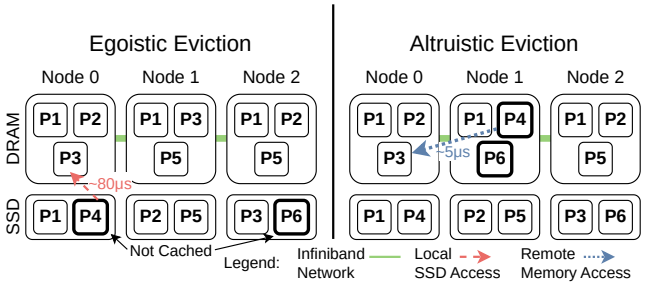


Figure 1: RDMA-enabled shared-cache architecture with different eviction strategies. *Egoistic* eviction fails to maintain the entire working set in aggregate DRAM (Page 4 and 6 must be read from SSD). *Altruistic* eviction reduces access latency by caching the entire working set in aggregate DRAM.

Motivating example. Figure 1 shows the architecture of a distributed shared-cache system like TaurusDB [13] or ScaleStore [50], where pages may reside in the cluster's aggregate DRAM and can be accessed remotely. Because the aggregate DRAM capacity (9 pages) exceeds the dataset size stored on SSD (6 pages), every page could be cached on at least one node. Most systems, however, make *egoistic*, per-node eviction decisions, resulting in suboptimal utilization of the aggregate DRAM, as illustrated in the left half of Figure 1: While pages P1, P2, P3, and P5 are redundantly replicated across nodes, P4 and P6 can only be accessed through (remote) SSD, leading to high access latency (80μs). Instead, a globally-coordinated *altruistic* strategy could reduce access latency to that of a remote memory

DaMoN '26, Bengaluru, India

ACM ISBN 979-8-4007-2455-8/26/05
https://doi.org/10.1145/3789237.3809123

copy (5μs on modern networks) by evicting redundant copies (P3 and P5 on Node 1), as shown in the right half of Figure 1.

In theory, altruistic eviction dominates. The example shows that altruistic eviction can result in substantially better utilization of aggregate DRAM capacity—an increasingly important consideration given the high cost of DRAM on-premises [23, 47] and in the cloud [39]. These utilization improvements could also translate into substantial performance gains. Yet existing shared-cache and disaggregated-memory DBMS designs typically rely on egoistic, per-node replacement or simple cluster-level heuristics (e.g., using remote capacity before falling back to storage), rather than utility- and copy-aware altruistic eviction [7, 27, 28, 34, 46].

In practice, altruistic eviction on modern hardware is hard. The reason modern high-performance shard-cache systems implement local eviction strategies is efficiency. Prior work that tried to solve altruistic eviction implemented this via global synchronization and coordination. However, this approach fails on modern hardware. Modern hardware SSDs and NICs support massive parallelism and very high transaction rates [20, 22, 25], demanding much higher eviction throughput from the storage engine, which makes global coordination a scalability bottleneck [24, 45]. This raises the question of whether a scalable altruistic eviction strategy can be efficiently implemented on modern hardware.

Contributions. In this work, we answer this question affirmatively by designing, implementing, and evaluating ScaleEvict. ScaleEvict is a highly efficient altruistic eviction strategy designed for the latency and bandwidth characteristics of today's shared-cache systems. We implement and evaluate ScaleEvict in ScaleStore [50], a state-of-the-art shared-cache storage engine that leverages RDMA and NVMe SSDs. ScaleEvict reduces redundant replication and prioritizes pages with high global value. It combines the efficiency of local eviction with the benefits of global altruistic eviction through a two-phase algorithm: (1) a carefully engineered, sampling-based local candidate-selection procedure tuned for modern CPUs to sustain high eviction rates; and (2) low-overhead inter-node coordination that piggybacks on the existing directory-based coherence protocol, leveraging partitioned (rather than global) knowledge to efficiently correct poor local decisions. The first phase enables high node-local eviction throughput, while the second prevents costly mistakes by ensuring pages valuable to other nodes are not evicted to SSD. At the same time, the second phase feeds back into the first by synchronizing the state, helping avoid redundant copies and evicting them locally when appropriate. As a result, ScaleEvict matches ScaleStore's baseline throughput with only two-thirds of the DRAM. Conversely, at equal DRAM capacity, it improves throughput by up to 2× while reducing 99th percentile latency by up to 3×.

2 Background: ScaleStore

We implement our altruistic eviction strategy ScaleEvict in the open-source storage engine ScaleStore [50]. In this section, we briefly describe the ScaleStore architecture to introduce the necessary background for the rest of this paper.

Bridging the NVMe latency cliff with RDMA. ScaleStore is a distributed multi-writer storage engine that combines fast RDMA networking with NVMe SSD storage using a page-based data layout. Many design choices, like optimistic latches, are inspired by LeanStore [31], and it achieves competitive performance to similar systems like GAM [8], NAM-DB [48], and FaRM [14].

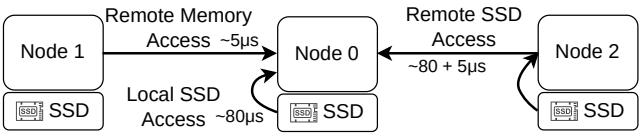


Figure 2: Not cached pages have different access costs depending on their current location. Remote pages can be retrieved via RDMA with low latency (5 μs). SSD retrieval is more expensive and should be minimized by the eviction policy.

Invalidation-based coherence protocol. Instead of relying on static partitioning, ScaleStore caches pages dynamically based on the workload. This approach solves the skewed reader problem [50] by dynamically replicating hot pages on multiple nodes. ScaleStore's caching enables traversing hierarchical data structures from local memory instead of sequentially fetching pages over the network [52]. To ensure correctness of concurrent reads and writes, ScaleStore implements a MESI-like coherence protocol where nodes hold pages in shared (S) or exclusive (X) mode, guaranteeing that operations on the same page are sequentially consistent.

Directory per storage node. ScaleStore enforces cache coherence through directories located on every storage node. Each directory owns a distinct subset of all pages and is responsible for enforcing cache coherence of these pages. Other nodes can identify and contact the directory using ownership information encoded in the upper bits of the page's unique page identifier (PID). When a node wants to request a page, it must contact the directory responsible for that page. Depending on the request mode and current state, the directory will reply with a page copy, revoke existing copies, or queue subsequent exclusive requests. The directory tracks the current possessor metadata for the page (i.e., which nodes hold a copy and in what mode) using a bitmap stored in the buffer frame.

Page access paths. Figure 2 shows the access paths through which a page can be retrieved: If Node 0 is the page's directory and the page is currently not cached, Node 0 has to read it from its local SSD. If the page is cached by Node 1, Node 0 can directly request it from Node 1 rather than reading it from the SSD. This reduces access latency because ScaleStore uses fast RDMA networking, which is an order of magnitude faster than (remote) SSD reads. By forwarding page copy requests to a current page holder, remote memory effectively becomes a cheap extension of the local buffer pool. If Node 0 is not the page's directory and the page is currently not cached, Node 0 must contact the page's directory (Node 2), which reads the page from its local SSD and sends it to Node 0. After receiving the page, the receiver (Node 0) stores it in its local page table. This means that if the access mode is shared, both the sender and receiver keep a copy of the page. If it is exclusive, the sender's copy must be invalidated and is subsequently deleted.

High-performance eviction. In ScaleStore, each node independently evicts the least-recently used pages based on its local access pattern. To avoid page eviction on the hot path when worker threads access pages, eviction is managed by a dedicated background thread

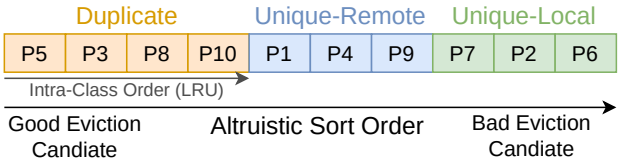
(*page provider*) that ensures that sufficient free buffer pool capacity [24]. This thread becomes active when the buffer pool reaches a high watermark and starts batch-evicting pages until it reaches a pre-configured free capacity (low watermark). If evicted pages are not owned by the evicting node, they must be returned to their respective directory's page provider to preserve cache coherence and ownership guarantees. The directory then asynchronously writes evicted dirty pages back to the SSD.

3 ScaleEvict: Altruistic Eviction in ScaleStore

In this section, we replace ScaleStore's current egoistic eviction with an efficient altruistic strategy to empirically validate the benefits described in Section 1. We introduce *ScaleEvict*, an altruistic eviction algorithm designed for modern hardware. *ScaleEvict* consists of two stages, which we implement in the page provider: In Section 3.1, we adapt local eviction on worker nodes to efficiently generate eviction candidates. In Section 3.2, we describe how the directory facilitates altruistic eviction while minimizing the required coordination overhead.

3.1 Worker Node Eviction

Worker nodes must efficiently score pages and select eviction victims. This requires (1) defining an eviction priority (i.e., scoring candidates) and (2) selecting victims that satisfy the eviction policy.
A measure for altruism. Instead of evicting pages solely based on local recency, as in egoistic eviction, the idea of altruistic eviction is to evict pages that contribute the least to fulfilling the cluster-level objectives. As shown in the figure below, we express this through an *altruistic sort order* that encompasses both access recency and replication state:



We distinguish between three classes of pages: *Duplicate* pages are cached on more than one node, *Unique-Local* pages are only cached at the directory, and *Unique-Remote* pages are only cached by one node that is not the directory. Because modern networks are fast, duplicate pages have a strictly lower utility to the cluster than unique pages: For instance, we consider P5 a better eviction candidate than P1 because, if evicted, P5 can be copied from aggregated DRAM, whereas P1 needs to be read from SSD. One would expect that Unique-Remote pages should rank higher than Unique-Local pages, because they incur an additional network round-trip for fetching the page. However, because the replication state may be inaccurate, Unique-Remote pages have lower expected access latency. The intuition here is that a Unique-Remote page could be replicated, whereas evicting a Unique-Local page will definitely result in an eviction to SSD. Within each class, we select eviction candidates that were least recently used to prioritize hot over cold pages. Thus, P5 is a better eviction candidate than P10, because it was accessed less recently.

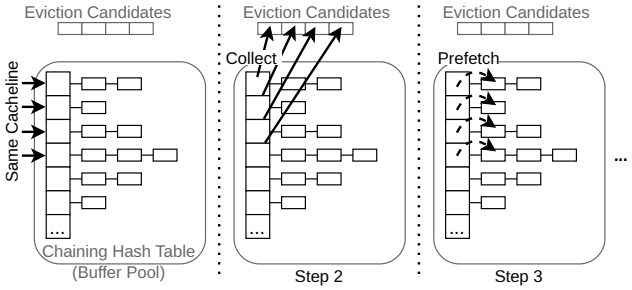


Figure 3: Vectorized buffer pool traversal to efficiently select eviction candidates. We hide data-dependent cache misses by advancing multiple pointers to eviction victims in lockstep.

Naïve approach for selecting eviction victims. A prior approach proposed by Venkataraman and Naughton maintains separate priority queues for each class [41]: When a page is accessed, it is moved to the back of its current (LRU) queue. And when its replication state changes, it is moved from the unique to the remote queue or vice versa. As long as the duplicate queue is not empty, eviction candidates are selected from it. Otherwise, the eviction reverts to the unique queue. However, this approach adds substantial overhead for accessing and maintaining multiple data structures, which is prohibitive given that each ScaleStore node caches over 10 million pages and must evict in the order of 500 thousand pages per second.
Efficiently selecting eviction victims. To efficiently scale to many-core CPUs, the eviction strategy must avoid lock contention and expensive computations on the worker node's hot path. We therefore implement ScaleEvict as a probabilistic eviction strategy in ScaleStore's page provider, allowing pages to be evicted *just-in-time*. This strategy employs an efficient buffer pool traversal and an altruistic scoring function. ScaleStore uses a chaining hash table with slot inlining [53] to translate page IDs to buffer frames.

Naïvely traversing the hash table by following the chain pointers individually (scalar) would suffer from poor cache utilization and data-dependent cache misses [2]. Instead, we use a vectorized traversal [54] by using a pointer array as an index. The main idea of this algorithm, illustrated in Figure 3, is to maintain an array of multiple chains to hide data-dependent cache misses that we traverse in lockstep. First, we initialize the chain array with pointers to the hash table bucket heads (step 1). Afterward, we iterate over the array and collect a pre-determined number of samples (step 2). Before continuing with the next iteration, we update the array's current position to the chain's next pointer (step 3). This triggers prefetching of the next pointer and hides the main memory latency while we continue iterating the next positions in the array, which are already CPU cache-resident. As shown in the following table, this reduces the stalled cycles caused by memory latency by ≈ 4×:

traversal	cycles	stalled cycles	cache misses
scalar	2770	**2349**	17
vectorized	860	**612**	10

We considered two strategies for populating the chain array: Randomly sampling over the buckets and a partition-wise linear scan (e.g., starting with partition [0; k], then [k+1; 2k], and so on).

The latter has fewer instructions by avoiding random number generation and fewer cache misses due to a more predictable access pattern. Although the predictable access pattern appears to come at the cost of a less randomized sample, we still find it sufficiently random for our application, because the page identifier is decorrelated from its location in the hash table via the hash function.

Eviction candidate scoring. We prioritize eviction candidates using a scoring function that emulates the previously described altruistic sort order. The scoring function assigns each page a numeric score based on its normalized LRU timestamp and a per-page-class weight. We apply this scoring function to the collected victim samples to approximate the distribution over the buffer pool. Instead of precisely evicting the candidates with the lowest scores, we aim to evict pages that fall in a low percentile of the score distribution. Therefore, we select an eviction threshold based on a pre-configured percentile (e.g., the lowest 25%). We continue traversing the hash table and evict pages that have an altruism score below the threshold until we reach the pre-configured free capacity (cf. Section 2).

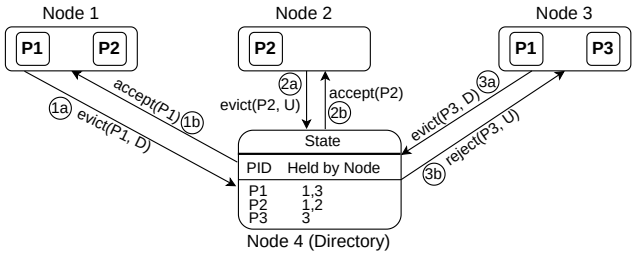
3.2 Directory Eviction

Worker nodes score and evict pages on a best-effort basis. As a result, they may inadvertently evict a page to SSD if a page that is assumed to be replicated is in fact unique. To prevent such suboptimal decisions, the directory must validate eviction requests before they are executed.

Altruistic eviction requires global knowledge. The primary objective of altruistic eviction is to avoid evicting unique pages whenever possible. Achieving this goal requires knowledge of which pages are replicated on other nodes. Because nodes communicate over the network, they cannot directly inspect each other's memory state. Prior work has proposed several mechanisms to maintain replication state, including centralized coordinators and gossip-based dissemination [12, 36, 38, 42]. However, maintaining precise global knowledge is costly in high-performance distributed systems.

Exploiting the directory's partitioned knowledge. Although maintaining fully consistent global knowledge is expensive, ScaleStore's directory nodes already maintain *partitioned* global knowledge. Each directory node manages metadata for a subset of the working set as part of the invalidation-based cache-coherence protocol. In particular, the directory tracks which nodes cache a page and in which coherence mode. Furthermore, evicted pages are returned to their responsible directory node (e.g., to flush dirty data to SSD). Consequently, the directory is naturally positioned on the eviction path. We leverage this partitioned knowledge to validate eviction requests without introducing an additional global coordination mechanism. As long as the directory can reject harmful evictions, worker nodes only require an eventually consistent view of page replication state.

Directory eviction algorithm. To address inconsistencies in the worker node's replication knowledge, we extend the directory's page provider to reevaluate eviction requests issued by worker nodes. Using its partitioned but globally informed view of page replication, the directory either approves or rejects each request. The figure below illustrates scenarios in which the directory and worker nodes agree or disagree on a page's replication state:



In the ideal case, both the directory (Node 4) and the worker nodes agree on the victim page's replication state. For example, in case (1a), Node 1 requests eviction of page P1, assuming that it is replicated. The directory approves the request (1b), as P1 is indeed replicated on Node 3. Next, in case (2a), Node 2 requests eviction of P2, which it believes to be unique. However, P2 is actually replicated on Node 1; thus, evicting P2 would not trigger an SSD write. Because the page's true utility is lower than Node 2 assumes, the directory accepts the request (2b). Finally, Node 3 wants to evict the presumed duplicate page P3 (3a). Finally, in case (3a), Node 3 attempts to evict page P3 under the assumption that it is replicated. The directory detects this as a false positive: P3 is in fact unique, and eviction would therefore result in the page becoming SSD-resident. Consequently, the directory rejects the request and updates Node 3's replication metadata for P3 to reflect its unique status. This incurs no additional messaging overhead, as the directory already sends acknowledgments for eviction requests.

4 Experimental Evaluation

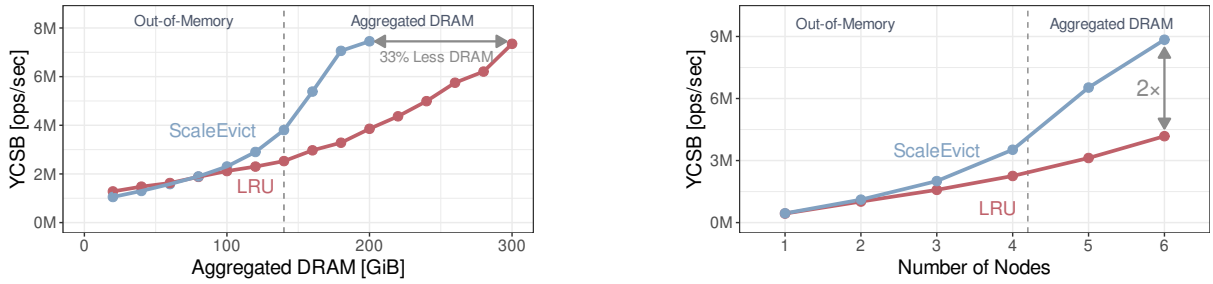
Having shown how altruistic eviction can be implemented efficiently, we now evaluate its benefits empirically in ScaleStore. We first quantify how optimizing cluster-level objectives, such as reducing redundant replication, improves aggregate DRAM utilization and performance. We then study ScaleEvict's robustness across workload characteristics and under workload drift.

4.1 Experiment Setup

Baseline. We implement ScaleEvict in ScaleStore and compare it against ScaleStore's existing LRU implementation [50]. Both eviction strategies use the same epoch-based access tracking and page provider configuration. This isolates the performance impact and runtime overheads attributable to ScaleEvict.

Hardware. Unless stated otherwise, we run experiments on a 4-node cluster, each with a 50 GiB buffer pool, running Ubuntu 24.04.3 LTS (Linux 6.8.0-88-generic). Each node has two Intel Xeon Gold 5220 CPUs (18 cores each) and 512 GiB of DRAM split across sockets. Storage consists of four Samsung 980 Pro 1 TB NVMe SSDs per node, attached via an ASRock Hyper Quad M.2 PCIe adapter and configured as a Linux `md` RAID-0 device. ScaleStore uses `O_DIRECT` I/O and a 4 KiB page size to match the SSDs' logical block size. Before each experiment, we reset the SSDs using `blkdiscard`. Nodes are connected via InfiniBand using one Mellanox ConnectX-5 MT27800 NIC (EDR 4x, 100 Gbps) per node.

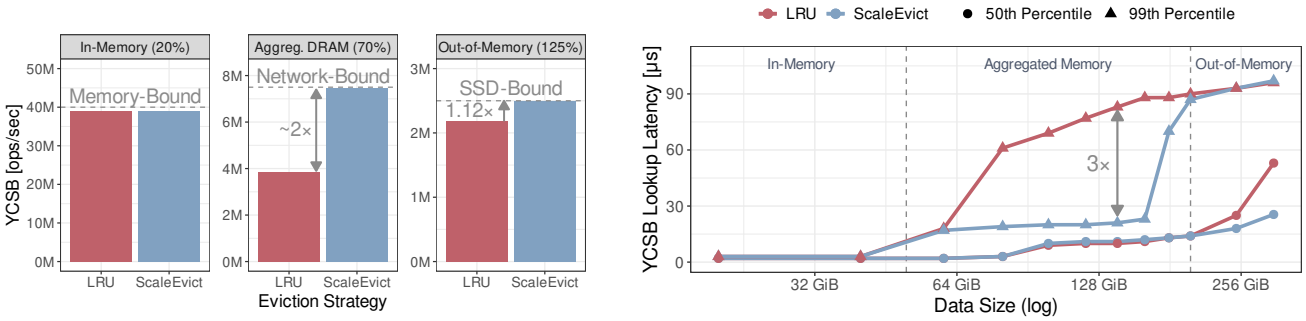
Benchmark. We evaluate eviction policies using YCSB [11] with 8-byte keys and randomly generated 128-byte string payloads. Operations access keys through an optimistically latched B-tree whose



(a) Buffer pool scalability with a fixed 140 GiB working set

(b) YCSB scale-out performance with a 210 GiB working set

Figure 4: ScaleEvict utilizes resources more efficiently than LRU: (a) it achieves comparable performance with fewer resources; (b) it scales more effectively with additional (compute) resources.



(a) Aggregate throughput under YCSB uniform point lookups: ScaleEvict is limited by hardware constraints, whereas LRU is constrained by inefficient utilization of aggregate DRAM.

(b) Closed-loop operation latency (median and 99th percentile): Replication awareness reduces ScaleEvict's tail latency when the dataset fits entirely in aggregate DRAM.

Figure 5: ScaleEvict improves throughput and latency across varying working-set sizes under YCSB.

nodes are also stored on ScaleStore pages. Each worker node runs 20 clients in a closed loop (i.e., each client issues the next request only after the previous one completes). We use a 30 s warm-up phase followed by 30 s of steady-state measurement. Unless otherwise noted, we report the average aggregate throughput (summed across all worker nodes) during the measurement phase.

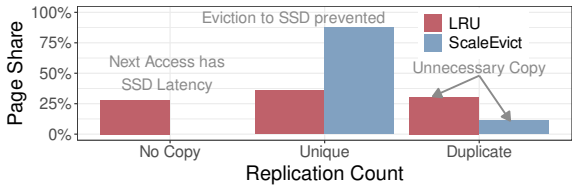
4.2 Improved Resource Utilization

ScaleEvict reduces required DRAM for a target throughput. As illustrated in Section 1, the key benefit of altruistic eviction is improved utilization of *aggregate* cluster memory. This can increase performance but also reduce the amount of DRAM required to reach a target throughput. To demonstrate this effect, we use a 4-node cluster with a fixed 140 GiB working set and gradually increase the buffer pool size per node. Figure 4a reports the results. Initially, the working set does not fit into the aggregated cluster memory, and system throughput is bound by frequent storage accesses in both strategies. Once aggregate DRAM approaches the working set, ScaleEvict achieves higher throughput than the LRU baseline (starting at 120 GiB aggregate DRAM). Even before the current DRAM shortages and price peaks, main memory has been a significant contributor to server hardware costs [47]. A 33% reduction in main memory usage can therefore translate to significant cost savings.

ScaleEvict scales with aggregate DRAM via scale-out. Beyond increasing buffer capacity per node, ScaleEvict can exploit additional aggregate DRAM by scaling the cluster. In the next experiment, we vary the cluster size from 1 to 6 nodes (Figure 4b), where each node contributes 50 GiB of buffer capacity. We set the working set [1] to 210 GiB (70% of the maximum aggregate DRAM capacity across 6 nodes) and reuse the same read-only YCSB workload as above. ScaleEvict benefits from added memory capacity and outperforms LRU once aggregate DRAM reaches roughly 150 GiB (3 nodes). In contrast, egoistic LRU shows only limited benefit from scale-out. With 6 nodes, ScaleEvict improves throughput by 2×.

ScaleEvict reduces redundant replication in the buffer pool. To highlight that these gains stem from optimizing for cluster-level objectives, we inspect the per-node buffer-pool contents after convergence. Because this introspection is expensive, we use a smaller configuration (3 nodes, 10 GiB buffer pool per node) and set the working set to 60% of aggregate DRAM. We run uniform YCSB point lookups until steady state, then take a snapshot of the buffer pools. The resulting replication distributions are shown below.

[1] We use working set and database size interchangeably. While ScaleStore could store databases of several TiBs, we only allocate as much data as needed for each experiment.



As the figure shows, LRU leaves 25% of pages resident only on SSD (i.e., not cached anywhere in the cluster). ScaleEvict prioritizes evicting redundant copies, increasing the fraction of pages cached exactly once (Unique), and reducing SSD-resident pages. In contrast, LRU has a large fraction of duplicated pages, limiting effective utilization of aggregate DRAM.

4.3 Improved Performance

The previous experiments showed that ScaleEvict makes more effective use of *aggregate* cluster memory, translating into higher throughput. We now study how these benefits depend on working-set size and affect operation latency. We use a 4-node cluster with 200 GiB of aggregate DRAM and vary the working set size of a read-only YCSB uniform point-lookup workload.

ScaleEvict doubles throughput for data in aggregate DRAM. Figure 5a shows the impact of an increasing workload size on the throughput. When the working set fits in DRAM, neither policy needs to evict pages. In this case, ScaleEvict matches LRU's performance because both are limited by CPU and DRAM access. ScaleEvict is most effective when the working set exceeds per-node DRAM but fits in *aggregate* cluster memory. In this setting, ScaleEvict avoids evicting uniquely cached pages and improves throughput by up to 2× over LRU. As the working set grows further, YCSB operations increasingly incur unavoidable SSD reads, which dominate latency and cap throughput. At a working set of 1.25× aggregate DRAM, ScaleEvict still outperforms LRU by ≈ 12%. For larger working sets, the throughput converges under a uniform workload because most requests require SSD accesses.

ScaleEvict reduces tail latency for out-of-memory workloads. For OLTP workloads, latency is as important as throughput. Figure 5b reports the median and 99th percentile latency for the same read-only YCSB workload. When the working set fits in DRAM, B-tree traversals are dominated by DRAM accesses. Since all operations read from memory, there is no need for evictions, and there is little variation between median and 99th percentile latency for both eviction policies.

When the working set fits in *aggregate* DRAM (middle region), ScaleEvict's replication-aware eviction reduces tail latency. LRU is oblivious to replication and begins evicting unique pages, leading to occasional SSD reads. ScaleEvict instead evicts redundant copies and avoids SSD reads entirely, maintaining a stable 99th-percentile latency of ≈ 23 μs. Median latency is similar for both policies because most accesses are served from local or remote DRAM.

Once the working set exceeds the aggregate DRAM capacity, SSD reads become unavoidable, and the 99th percentile latencies of ScaleEvict and LRU converge to ≈ 100 μs. However, we observe that the median latency of ScaleEvict degrades more slowly than that of LRU (2× lower). This is because the egoistic LRU policy will still implicitly replicate pages, resulting in a lower aggregated DRAM hit rate and thus more SSD reads. ScaleEvict, on the other hand, will retain more distinct pages in the buffer pool, resulting in a higher rate of remote memory hits.

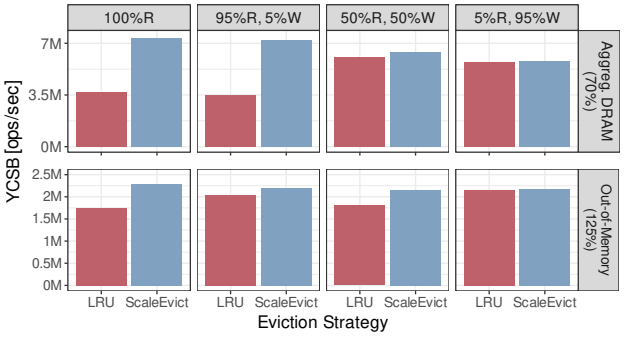


Figure 6: YCSB throughput under varying read/write ratios for two working-set sizes. The top figure corresponds to a working set that fits in aggregate DRAM; the bottom figure corresponds to a working set that exceeds aggregate DRAM.

4.4 Improved Robustness

So far, we evaluated read-only workloads. We now study robustness under mixed read/write workloads and dynamic workload drift.

ScaleEvict remains effective under mixed workloads. We first analyze mixed workloads in which a write occurs with a given probability. Figure 6 reports throughput for different read/write ratios on a 4-node cluster, for a working set that fits in aggregate DRAM (top) and one that exceeds aggregate DRAM (bottom). When the write ratio exceeds 50%, ScaleEvict and LRU perform similarly. Write operations naturally reduce replication, thereby reducing the opportunity for altruistic eviction. To ensure consistency, a write requires exclusive page ownership, invalidating existing replicas and lowering memory pressure without additional eviction work.

For a fixed working-set size, ScaleEvict provides comparatively predictable performance across read/write ratios. In the in-aggregate case, ScaleEvict remains close to the network/memory bound. In the out-of-memory case, it approaches the SSD bound. In contrast, LRU is SSD-bound at low write ratios even when the working set fits in aggregate DRAM, leading to larger performance variability for the same working-set size.

ScaleEvict avoids sharp drops due to workload drift. Next, we evaluate adaptation to workload drift. We use a skewed YCSB workload (Zipf factor = 1.2) with a fixed working set of 70% of aggregate DRAM. To avoid identical hot pages across nodes, we apply a uniform random offset that permutes the Zipf ranks of keys on each node, resulting in different page-popularity orders. After 60 s, we induce drift by selecting a new random offset, so that previously hot pages may become cold (and vice versa). We repeat this procedure three times. Figure 7 shows throughput over time, with workload drifts marked by vertical lines. Under a stable workload, both strategies show a stable throughput. After each drift, both recover within a few seconds. However, LRU experiences a larger relative throughput drop (≈ 30%) than ScaleEvict (≈ 15%) compared to their respective pre-drift medians. Intuitively, ScaleEvict maintains broad coverage of distinct pages in aggregate DRAM

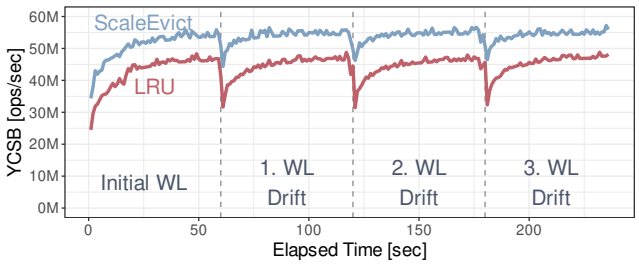


Figure 7: Throughput under workload drift induced by changes in access skew (vertical lines denote drift events). ScaleEvict mitigates performance cliffs during skew shifts.

rather than over-replicating the current hot set. After a drift, nodes can therefore fetch newly hot pages from remote DRAM instead of falling back to SSD, reducing the transient performance loss.

5 Related Work

Altruistic eviction. Cooperative caching and distributed memory systems in the 1990s already recognized that eviction should optimize cluster-level utility rather than per-node recency [12, 16, 29, 38, 41, 44]. These works introduced ideas such as replica-aware cost models, random forwarding (e.g., N-chance), and duplicate elimination via separate queues. Related hint-based approaches pursued similar goals using lightweight metadata exchange rather than full global synchronization [36, 37]. ScaleEvict follows the same high-level goal but targets modern RDMA+NVMe systems and a much higher eviction rate by using sampling-based victim selection and directory-side lazy correction rather than queue- or synchronization-heavy designs.

Page eviction on modern hardware. As the eviction bottleneck has moved from I/O to compute, page eviction must become cheap (a few CPU cycles per eviction) and highly scalable. To solve these issues, sampling-based eviction approaches have been proposed [5, 6, 31, 45, 50]. Our design complements this line of work: we retain a lightweight, sampling-friendly implementation but make replacement replication-aware to optimize aggregate DRAM utilization in distributed shared caches. We believe that our sampling algorithm also extends to other high-performance buffer managers that use hash tables to translate page IDs to buffer frames.

Shared Caches, Disaggregation, and CXL. Recent shared-cache and disaggregated-memory DBMSs demonstrate the benefits of remote-memory accesses and dynamic data placement [7, 13, 28, 34, 35, 40, 42, 46]. However, they typically rely on local replacement or coarse heuristics and do not implement utility- and copy-aware altruistic eviction. The in-memory system GAM [8] does not suffer from evictions to SSD. In contrast, by evicting cold data to NVMe SSDs, ScaleStore can economically scale to large datasets. Related discussions also appear in CXL memory management [9]; unlike cache-line-granular CXL coherence, ScaleEvict operates at page granularity and integrates directly with the directory protocol.

Distributed OLTP in the Cloud. Despite the conceptual advantages of shared-writer systems [49], the cloud has largely converged toward single-writer systems such as Aurora [43] and Socrates [4]. One reason for this is the availability of the necessary hardware: Although most cloud providers use RDMA for their internal infrastructure, it is usually not available to customers. Instead, cloud providers rely on proprietary network fabrics such as AWS EFA [1]. Prior work has shown that, although more efficient than TCP/IP, EFA's network latency is an order of magnitude higher than RDMA's, reducing the potential gains of cooperative eviction [51]. Likewise, cloud SSDs have not kept up with on-premises trends: Instance-attached cloud SSDs suffer from higher latency [21] and lower throughput [39] than their on-premises counterparts. For shared-caching systems to realize their full potential in the cloud, cloud hardware needs to keep pace with modern hardware trends.

6 Summary

In this paper, we revisit the page eviction problem in the context of distributed storage engines. We develop the altruistic eviction strategy, ScaleEvict, and implement it in the RDMA-enabled storage engine ScaleStore. Our evaluation shows that ScaleEvict improves aggregated DRAM utilization and increases robustness to workload drifts. Furthermore, ScaleEvict can increase system throughput by 2×, while reducing 99th percentile latency by up to 3×.

Acknowledgments

Funded/Co-funded by the European Union (ERC, CODAC, 101041375). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. This work has been partially funded by the LOEWE Spitzenprofessur (III 5-519/05.00.003-(0005)) of the state of Hesse. We also thank hessian.AI and DFKI for their support.

References

[1] 2026. Elastic Fabric Adapter. https://aws.amazon.com/hpc/efa/.
[2] Anastasia Ailamaki, Erietta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis. 2015. How to stop under-utilization and love multicores. In *ICDE*. IEEE Computer Society, 1530–1533.
[3] Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. 1995. Serverless Network File Systems. In *SOSP*. ACM, 109–126.
[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD Conference*. ACM, 1743–1756.
[5] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *NSDI*. USENIX Association, 389–403.
[6] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *USENIX ATC*. USENIX Association, 499–511.
[7] William Bridge, Ashok Joshi, M. Keihl, Tirthankar Lahiri, Juan Loaiza, and N. MacNaughton. 1997. The Oracle Universal Server Buffer. In *VLDB*. Morgan Kaufmann, 590–594.
[8] 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. *Proc. VLDB Endow.* 11, 11 (2018), 1604–1617.
[9] Jeronimo Castrillon, Jana Giceva, Yu Hua, Kimberly Keeton, Akhil Shekar, Kevin Skadron, Tianzheng Wang, and Huanchen Zhang. 2026. Declarative Memory Services. In *CIDR*. www.cidrdb.org.
[10] Hong-Tai Chou and David J. DeWitt. 1985. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB*. Morgan Kaufmann, 127–141.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[12] Michael Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. 1994. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *OSDI*. USENIX Association, 267–280.

[13] Alex Depoutovitch, Chong Chen, Per-Åke Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: bringing multi-master to the cloud. *Proc. VLDB Endow.* 16, 12 (2023), 3488–3500.

[14] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*. USENIX Association, 401–414.

[15] Wolfgang Effelsberg and Theo Härder. 1984. Principles of Database Buffer Management. *ACM Trans. Database Syst.* 9, 4 (1984), 560–595.

[16] Michael J. Feeley, William E. Morgan, Frédéric H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. 1995. Implementing Global Memory Management in a Workstation Cluster. In *SOSP*. ACM, 201–212.

[17] Prudhvi Gadupudi and Suman Saha. 2025. Evolution of Buffer Management in Database Systems: From Classical Algorithms to Machine Learning and Disaggregated Memory. *CoRR* abs/2512.22995 (2025).

[18] Goetz Graefe. 2007. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN*. ACM, 6.

[19] Jim Gray and Gianfranco R. Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *SIGMOD Conference*. ACM Press, 395–398.

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

[21] Gabriel Haas, Bohyun Lee, Philippe Bonnet, and Viktor Leis. 2025. SSD-iq: Uncovering the Hidden Side of SSD Performance. *Proc. VLDB Endow.* 18, 11 (2025), 4295–4308.

[22] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (2023), 2090–2102.

[23] Scharon Harding. 2026. Ram now represents 35 percent of Bill of materials for HP PCS. https://arstechnica.com/gadgets/2026/02/ram-now-represents-35-percent-of-bill-of-materials-for-hp-pcs/

[24] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD Conference*. ACM, 877–892.

[25] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2025. A Wake-Up Call for Kernel-Bypass on Modern Hardware. In *DaMoN*. ACM, 14:1–14:5.

[26] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*. Morgan Kaufmann, 439–450.

[27] Jeffrey W. Josten, C. Mohan, Inderpal Narang, and James Z. Teng. 1997. DB2's Use of the Coupling Facility for Data Sharing. *IBM Syst. J.* 36, 2 (1997), 327–351.

[28] Tirthankar Lahiri, Vinay Srihari, Wilson Chan, N. MacNaughton, and Sashikanth Chandrasekaran. 2001. Cache Fusion: Extending Shared-Disk Clusters with Shared Caches. In *VLDB*. Morgan Kaufmann, 683–686.

[29] Avraham Leff, Joel L. Wolf, and Philip S. Yu. 1993. Replication Algorithms in a Remote Caching Architecture. *IEEE Trans. Parallel Distributed Syst.* 4, 11 (1993), 1185–1204.

[30] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25.

[31] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE Computer Society, 185–196.

[32] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD Conference*. ACM Press, 297–306.

[33] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *ICDE*. IEEE, 1326–1339.

[34] 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@VLDB*. 17–26.

[35] Uwe Röhm, Klemens Böhm, and Hans-Jörg Schek. 2001. Cache-Aware Query Routing in a Cluster of Databases. In *ICDE*. IEEE Computer Society, 641–650.

[36] Prasenjit Sarkar and John H. Hartman. 1996. Efficient Cooperative Caching Using Hints. In *OSDI*. ACM, 35–46.

[37] Prasenjit Sarkar and John H. Hartman. 2000. Hint-based cooperative caching. *ACM Trans. Comput. Syst.* 18, 4 (2000), 387–419.

[38] Markus Sinnwell and Gerhard Weikum. 1997. A Cost-Model-Based Online Method for Distributed Caching. In *ICDE*. IEEE Computer Society, 532–541.

[39] Till Steinert, Maximilian Kuschewski, and Viktor Leis. 2026. Cloudspecs: Cloud Hardware Evolution Through the Looking Glass. In *CIDR*. www.cidrdb.org.

[40] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. 2001. Cache-on-Demand: Recycling with Certainty. In *ICDE*. IEEE Computer Society, 633–640.

[41] Shivakumar Venkataraman and Jeffrey Naughton. 1996. *Global memory management for multi-server database systems*. Ph. D. Dissertation. AAI9631416.

[42] Shivakumar Venkataraman, Jeffrey F. Naughton, and Miron Livny. 1998. Remote Load-Sensitive Caching for Multi-Server Database Systems. In *ICDE*. IEEE Computer Society, 514–521.

[43] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD Conference*. ACM, 1041–1052.

[44] Geoffrey M. Voelker, Hervé A. Jamrozik, Mary K. Vernon, Henry M. Levy, and Edward D. Lazowska. 1997. Managing Server Load in Global Memory Systems. In *SIGMETRICS*. ACM, 127–138.

[45] Demian E. Vöhringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *Proc. VLDB Endow.* 16, 11 (2023), 3323–3334.

[46] Ruihong Wang, Jianguo Wang, and Walid G. Aref. 2025. Cache Coherence Over Disaggregated Memory. *Proc. VLDB Endow.* 18, 9 (2025), 2978–2991.

[47] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. In *ASPLOS*. ACM, 609–621.

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

[49] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem?. In *CIDR*. www.cidrdb.org.

[50] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD Conference*. ACM, 685–699.

[51] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. 2022. EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?. In *DaMoN*. ACM, 10:1–10:5.

[52] 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 Conference*. ACM, 741–758.

[53] Michael Zinsmeister, Lam-Duy Nguyen, Viktor Leis, and Thomas Neumann. 2026. Predictive Translation: High-Performance Buffer Management Without the Trade-Offs. (2026).

[54] Marcin Żukowski et al. 2009. *Balancing vectorized query execution with bandwidth-optimized storage*. SIKS.