ScaleCache: Scalable and Production-grade Buffer Management for Disk-based Database Systems

Mingyu Liu
Huawei Company
China
liumingyu15@huawei.com

Junbin Kang[*]
Huawei Company
China
kangjunbin1@huawei.com

Kai Wang
Huawei Company
China
wangkai0@huawei.com

Lu Zhang
Huawei Company
China
zhanglu214@huawei.com

Haibo Chen
Huawei Company
China
hb.chen@huawei.com

Xiuchang Li
Huawei Company
China
lixiuchang@huawei.com

Tianhong Ding
Huawei Company
China
dingtianhong@huawei.com

ABSTRACT

Buffer management is critical for DBMSs but often suffers from scalability bottlenecks and poor cache locality, which stems from centralized reference counting in page access and intensive locking in page-to-buffer translation. However, prior radical approaches like pointer swizzling or optimistic lock can hardly be adopted in production-grade DBMSs due to its inherent complexity and incompatibility.

This paper proposes *ScaleCache*, a scalable, highly-efficient and production-grade buffer management system with three key designs. *ScaleCache* first incorporates a novel compact per-group buffer reference counting technique, which enables scalable buffer pinning and unpinning by concurrent threads on many-core servers. It then devised a novel read-write lock based on copy-on-write and per-group reference counting, which is suitable for B-link tree. At last, we propose an optimistic, CPU-cache friendly and SIMD-accelerated hash table for fast and scalable page-to-buffer translation, which eliminates most contention on modern many-core hardware. ScaleCache has been adopted in *Huawei GaussDB*, a commercial high-performance DBMS. Evaluation on a 128-core server demonstrates that *ScaleCache* exhibits near-linear scalability and can significantly improve index query throughput of both classic B-link tree index and complex graph-based vector index.

PVLDB Reference Format:
Mingyu Liu, Junbin Kang, Kai Wang, Lu Zhang, Haibo Chen, Xiuchang Li, and Tianhong Ding. *ScaleCache*: Scalable and Production-grade Buffer Management for Disk-based Database Systems. PVLDB, 18(12): 5073 - 5085, 2025.
doi:10.14778/3750601.3750628

[*]Corresponding author and project leader.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750628

1 INTRODUCTION

The buffer management serves as the cornerstone of modern on-disk database management systems (DBMS) but often suffers from scalability bottlenecks and poor cache locality, which stems from centralized reference counting in page access and intensive locking in page-to-buffer translation. Specifically, the traditional buffer manager design leverages a lock-protected and chained hash table to record the mapping of cached disk page identifiers (IDs) to memory pointers within the buffer pool for page-to-buffer translation and uses centralized reference counting to prevent page eviction during page access [20, 21]. Intensive use of locking and atomic operations can lead to excessive CPU cache-line invalidation on many-core hardware. Meanwhile, the chained buffer hash table lookup, which occurs on each page access, presents noticeable overhead due to poor CPU cache locality during chain traversal.

Previous research work starting from 2008 [13, 15, 20, 21] demonstrated that the traditional buffer manager introduces significant overhead and proposed several efficient but radical buffer pool designs like pointer swizzling [13, 21] and virtual memory assisted buffer pool (*vmcache* for short) [20]. Pointer swizzling replaces disk page IDs embedded in B+ tree pages with memory pointers when they are looked up from the hash table, thus mostly eliminating the hash table translation as well as synchronization overhead on the critical page cache hit path. However, in practical usage, pointer swizzling encounters an issue: it modifies the index tuple structure, requiring data reloading in production environments and preventing seamless upgrades due to incompatibility between old and new database versions.

The recent work *vmcache* [20] proposed to map the entire database storage into anonymous virtual memory and to let the DBMS to manage page loads and evictions explicitly. *vmcache* gets rid of expensive hash-table-based page translation by leveraging the MMU hardware to translate page IDs into memory pointers within the virtual memory address space.

Both *LeanStore* and *vmcache* uses optimistic latches [3, 22, 23] during index traversal, which relies on the retry mechanism to resolve conflicts upon concurrent page modifications. *vmcache* further proposes to retry optimistic page reads upon concurrent page eviction. Practical limitations emerge when using the simple retry mechanism for optimistic page reads: (1) it may lead to potential problems such as unpredictably long iteration loops and out-of-bounds read access errors during the binary iteration over the key array in one index page due to corrupted page fields by concurrent modification, such as the key array size, index item (tuple) offset and index tuple header. (2) error signal handling (e.g., zero division) due to corrupted data incurs non-trivial overhead. The above downsides impede *vmcache* and *LeanStore* from being applied to production-grade DBMSs.

This paper proposes *ScaleCache*, a scalable, highly efficient and production-grade buffer management system with three key designs as an alternative method. First, *ScaleCache* incorporates a novel per-group buffer reference counting technique that allows multiple threads to pin and unpin a page in the buffer pool scalably on modern many-core hardware. Second, *ScaleCache* introduced an innovative non-leaf read-write page latch mechanism leveraging copy-on-write and the aforementioned per-group reference counting technique to eliminate most CPU cache-line invalidation during index traversal. Finally, *ScaleCache* introduces an optimistic, CPU cache-friendly hashing scheme with SIMD acceleration to enable high-throughput page-to-buffer translation on modern many-core architectures, eliminating cache-line contention while enhancing cache locality during translation. A comprehensive analysis of the proposed framework is presented in Section 3.

ScaleCache is now adopted by *Huawei GaussDB*, a commercial high-performance DBMS. Evaluation of the B-link tree index on a 128-core *Huawei Kunpeng* server [16] using several benchmarks show that *ScaleCache* exhibits near-linear scalability and improves the index query throughput by up to 3.3X compared to the baseline (i.e., *GaussDB* without *ScaleCache*). In terms of vector search, *ScaleCache* improves the throughput by 46% and reduces the latency by 33% on the SIFT1M dataset without any modification to the *GaussDB* DiskANN vector index [35], demonstrating that *ScaleCache* can speed up complex vector index search and is expected to benefit other complex indexes such as inverted index and spatial index in a transparent way. Implementing our per-group read-write latch for the DiskANN vector index could further improves the vector search performance significantly and we leave this as our future work.

In summary, this paper makes the following contributions: (1) We have experimentally and quantifiably identified the root sources of inefficiencies of modern disk-based buffer management and discussed the limitations of existing solutions in Section 2. (2) We propose *ScaleCache*, a scalable and production-grade buffer management system in Section 3 with three efficient techniques, namely per-group buffer reference counting, per-group read-write latch, and SIMD-accelerated, lookup-optimistic and CPU-cache friendly page-to-buffer translation, to eliminate CPU cache-line contention and to improve CPU-cache locality, thus scaling buffer pool services to many cores. (3) We have implemented *ScaleCache* in *GaussDB*, a commercial high-performance DBMS production, evaluated *ScaleCache* using several benchmarks including OLTP, OLAP and vector search workloads experimentally and showed that *ScaleCache* exhibits near-linear scalability and improves both the index query throughput and latency significantly in Section 4.

2 BACKGROUND

In this section, we first analyze the root bottlenecks of modern buffer pool design adopted by most open-source and commercial DBMSs, then introduce recent related work that aims to address such bottlenecks and that inspires our work.

2.1 The bottlenecks of traditional buffer management

We first examine the architectural limitations inherent in conventional buffer pool implementations.

Average latency breakdown of index query execution. Figure 1 shows the average latency breakdown of both multi-random point lookup of the classic B-link tree and DiskANN vector index search on a 128-core server. The experimental details will be described in Section 4. The actual index search execution time only occupies less than 40% of the overall query time on both indexes while the sum of the buffer manager cost, that mainly consists of the buffer reference counting cost (i.e., buffer pin/unpin) and hash table lookup cost for page-to-buffer translation, and index page latching cost reaches over 60%. They are three major factors that impede the scalability of buffer pool services during index traversal on modern many-core hardware.

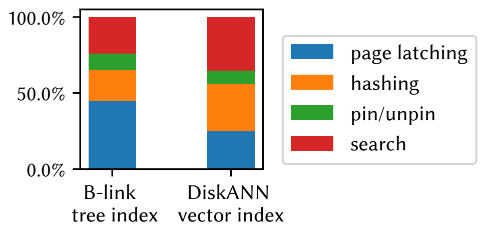


Figure 1: Average latency breakdown of parallel index query on the B-link tree and DiskANN vector index

Centralized per-buffer reference counters. The first bottleneck of traditional buffer manager stems from centralized per-buffer reference counting which is used to prevent page eviction during page access. As concurrent queries visit prefix nodes along the index search path [1], they update the corresponding shared buffer reference counters atomically, resulting in high cache-line contention and throughput drop.

Shared page latching. A classic DBMS uses shared and exclusive page latches to protect consistency of page content under concurrent readers and writers. Although shared page latching allow concurrent readers, its use of compare-and-swap (CAS) operation on shared latch state, which tries to increment the number of in-progress readers if there are no writers, can cause repeatable retries upon CAS failure and excessive cache-line invalidation on many-core hardware. We observe that concurrent queries acquiring

[1]For example, B-link tree root and inner nodes, and navigable graphs in the vector index.

shared page latches on some prefix index nodes like the B-link tree root node can result in throughput drop.

Lock-protected and chained hash table. Modern DBMSs rely on a lock-protected and chained buffer hash table to translate *PID*s into memory pointers within the buffer pool, which introduces two performance inefficiencies. First, although DBMSs use fine-grained per-bucket locking to reduce contention, there still exist locking overhead and high contention upon chaining collision. Second, pointer chain traversal may cause many random memory accesses, resulting in extra CPU cache misses on the index search path.

2.2 Related Work

Reference counting and reclamation. Prior work [2, 4, 6, 7, 17] proposes per-core reference counting for scalable kernel object reclamation in the Linux operating system on many-core hardware. These approaches leverages grace period-based synchronization to lazily reclaim non-referenced objects. Specifically, Refcache [4] devises an epoch-based reclamation mechanism. Each core completes flushing its local counter delta to the global reference counter during each epoch. Refcache can reclaim an object only if its counter drops to zero and its value remains unchanged across two epochs. Leanstore [21] also uses a similar epoch-based technique for page eviction and buffer replacement. Such epoch-based approaches face a non-trivial trade-off. Setting a small epoch interval value can result in excessive cache-line invalidation for epoch synchronization while a large epoch interval value may prevent non-referenced objects from being timely reclaimed, probably causing the system to be stuck due to out-of-memory. To avoid frequent cache-line synchronization, Refcache [4] introduces a delay of 10 milliseconds to guarantee detecting a zero true counter.

PAYGO [17] leverages Linux CPU preempt_disable()/enable() functions to devise a space-efficient and scalable counting approach, which only counts active objects in per-core hash caches to save space. Its userspace implementation needs to invoke its kernel-level PAYGO operations via OS system calls which incur expensive cost [32, 34, 39].PAYGO cooperates with the *Read-Copy-Update* (RCU) [24, 25] technique in Linux to reclaim zero-referenced pages safely for Linux page cache. Specifically, local per-core counter modification for page referencing is wrapped by *RCU* read critical section that disables CPU preemption and page deletion needs to wait for all cores to experience at least a OS context switch (i.e., grace periods) so as to guarantee all read critical sections that reference the page exited [24, 25]. However, this approach also introduces a delay before a zero-referenced page can be reused.

Vectorized hash table. Our lookup-optimistic, CPU-cache friendly and SIMD accelerated hash table design borrows some ideas from prior works that design vectorized hash table schemes to leverage SIMD acceleration [1]. A concurrent work [14] proposes a linear-chained hash scheme for efficient equi-joins. As the number of buffers is predefined when the DBMS starts, we design our new hash table with a fixed number of slots to hold all page-to-buffer translation entries without the need of considering hash table resizing as well as complex synchronization with normal operations during resizing. In cooperating with our scalable and light-weight buffer state synchronization, *ScaleCache* allows optimistic buffer hash table lookups with no locking for page-to-buffer translation within the buffer pool.

Other buffer management designs. Prior work like Kreon [28, 29] and Tucana [27] proposes to map the entire key-value store into virtual memory address to eliminate cache lookup cost and to leverage fast memory-mapped I/O for cache misses. FastMap [30] and Steroids [26] further optimize memory-mapped I/O performance. However, these approaches can consume a huge amount of OS page table memory for mapping a large database into virtual memory address like *vmcache* [20].

Due to the buffer management cost [15], main-memory databases like H-Store [31], HyPer [18], HANA [33], SQL server Hekaton [10] and Silo [37, 38] emerged in the database community. Main-memory databases support out-of-memory workloads by classifying hot/cold tuples and moving cold tuples to disk [9, 12]. Specifically, Anti-cache [9] designed for H-Store keeps indexes memory-resident and maintains a LRU-list for data tuples to facilitate cold tuple migration, which introduces large metadata memory usage. In the current big-data era, keeping all indexes memory-resident is also impracticable. Siberia [12] uses tuple access samples for hot/cold classification and employs background migration of cold tuples, which may not be able to respond to some rapidly changing workloads accurately and timely, leading to poor performance.

3 ARCHITECTURE AND DESIGN OF *SCALECACHE*

This section presents the overall architecture and design of *ScaleCache*, a scalable, highly efficient and production-grade buffer management system for disk-based DBMSs.

3.1 Per-group buffer reference counting

ScaleCache proposes a novel per-group buffer reference counting technique that eliminates the first multi-core scaling bottleneck of the DBMS buffer pool as mentioned in Section 2.

As shown in Figure 3, each buffer counter in *ScaleCache* consists of per-group counter slices, each of which lies in a separate CPU cache-line, so that pinning/unpinning in parallel by threads belonging to different groups modify different slices and hence does not cause any CPU cache-line synchronization. In order to reduce cache-line synchronization cost of pining/unpinning by threads belonging to the same group, we only organize CPU cores that at least share the same CPU last level cache (*LLC* or *L3*) into the same group (also called cache group). Since CPU cores sharing the same last level cache are physically adjacent, hardware-aware grouping can be implemented by computing the group ID through dividing the core ID by the number of cores per group, a fixed parameter given in the configuration file. Note that per-core counting is a special case where each group contains only one core. *ScaleCache* leverages the restartable sequence (*rseq*) provided by the Linux operating system (OS) to obtain one thread's running CPU core in a ultra-fast way. The *rseq* is a per-thread shared data structure and the OS scheduler would write the current CPU core ID into it each time one thread is scheduled. The *rseq* mechanism demonstrates a latency of approximately 1 nanosecond on our *Huawei Kunpeng* platform, which is orders of magnitude faster than vDSO-based approaches (~10ns) and getcpu system call overhead (~100ns). This

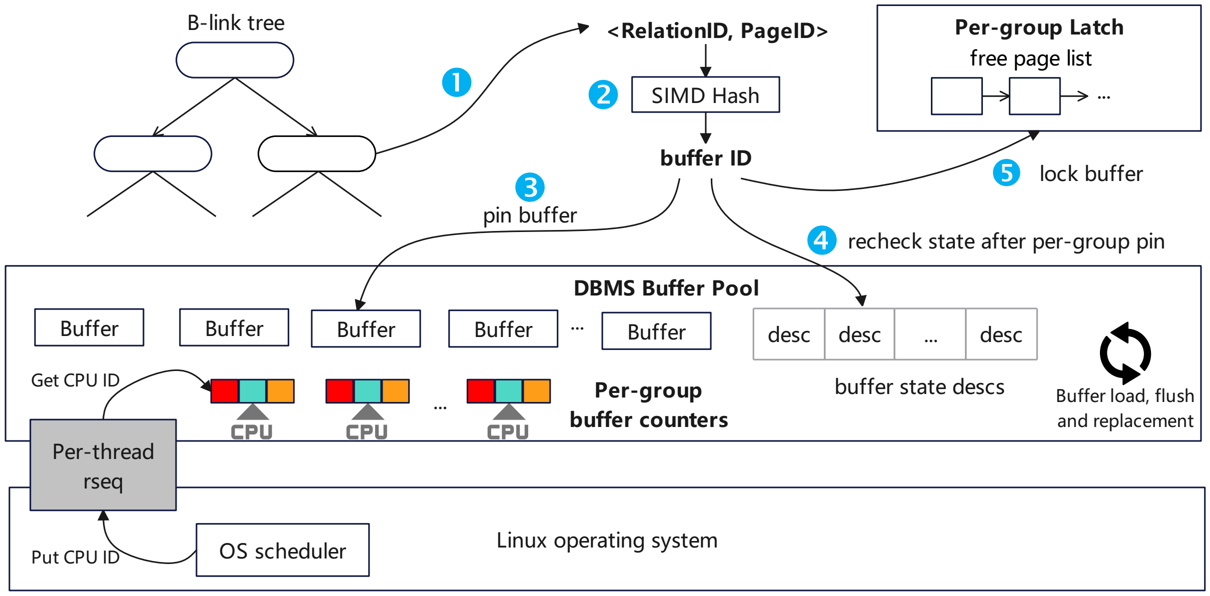


Figure 2: The overall architecture of *GaussDB-ScaleCache*

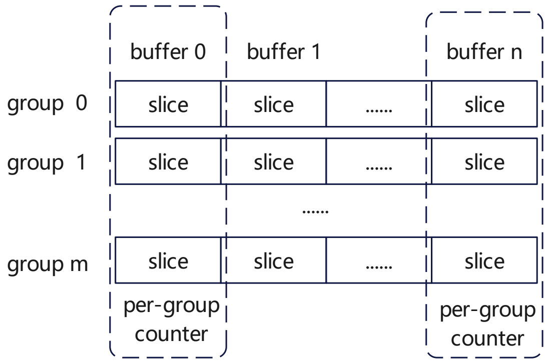


Figure 3: The memory layout of per-group buffer counters

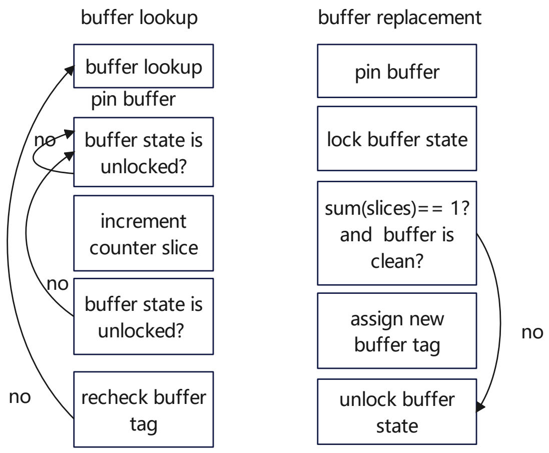


Figure 4: The light-weight and optimistic buffer state synchronization protocol

nanosecond-scale optimization is critical since if the cost of getting CPU core ID is larger than data transferring between different cores, this approach will become counterproductive.

One naive per-group counter memory design is to occupy a separate CPU cache-line for each slice, which however would lead to high memory footprint on machines with a large number of CPU cores, which is common today. In contrast, as shown in Figure 3, *ScaleCache* organizes the counter slices that correspond to the same group in a memory-compact way while avoiding false cache-line sharing between different groups. Specifically, we allocate a continuous memory slice array for each cache group. The size of each slice array is equal to the number of buffers. For each buffer counter, *ScaleCache* can use buffer ID and group ID to locate the corresponding buffer counter slice in the two-dimensional slice array. In our current implementation, each slice is an unsigned integer and its size is set to 2 bytes which can support up to 65535 DBMS thread concurrency while avoiding slice overflow [2], and we believe such high thread concurrency is enough in our customer scenarios as present-day servers now only have up to hundreds of cores.

Pin and unpin a buffer. Before (or after) accessing one page buffer, one DBMS thread obtains the CPU core ID where it runs from *rseq*, calculates its belonging group and then pins (or unpins) the buffer by incrementing (decrementing) the corresponding per-group counter slice *atomically*. Checking whether a page can be evicted from the buffer pool needs to calculate the sum of all its per-group slices. However, the computation overhead does not

[2]We limit the max concurrency of *GaussDB* by setting its max connection configuration upon startup.

Algorithm 1: Pin/unpin a buffer

```
input : bufferID
input : isOpPin
1  while true do
2      while bufferDesc[bufferID].stateLock == LOCKED
           /* read buffer state atomically to check
              if it is locked                          */
3       do
           /* loop until lock == 0                     */
4      end
5      read coreID from rseq;
6      groupID = CalculateGroupID(coreID);
7      counterSlice = perGroupCounters[groupID][bufferID];
8      if isOpPin == true then
9          counterSlice.fetch_add(1);
10     else
11         counterSlice.fetch_sub(1);
12     end
13     if bufferDescState[bufferID].stateLock != LOCKED then
14         break;
15     else
           /* The buffer state is locked during
              pinning/unpining, we retry               */
16         if isOpPin == true then
17             counterSlice.fetch_sub(1);
18         else
19             counterSlice.fetch_add(1);
20         end
21     end
22 end
```

influence performance because buffer replacement happens when no free buffers are available during which the slow storage IO dominates the overall performance even on fast NVMe SSDs (as will be demonstrated in Section 4).

As shown in Figure 4, *ScaleCache* proposes a scalable and lightweight buffer state synchronization protocol to ensure consistency between the operation of modifying a per-group counter slice for buffer pinning and unpinning and that of calculating the sum of per-group slices for buffer replacement which is protected by the buffer state lock. The algorithm details and correctness proof will be described in Subsection 3.3.

Algorithm 1 shows the detailed buffer pinning and unpinning pseudocode which is lock-free and CPU cache-line contention-free [3]. Specifically, to pin or unpin a buffer, the DBMS thread first waits for the buffer state to be unlocked by spinning (line 2-4), then tries to pin/unpin the buffer by modifying its corresponding per-group counter slice (line 5-12) and at last rechecks whether the buffer state is locked by other threads during the above process (line 13). If so, the DBMS thread undoes the change and retries the above process (line 15-21). Otherwise, the DBMS thread breaks the loop (line 14) and has successfully pinned/unpinned the buffer.

Pin and unpin on different CPU cores. Sometimes, due to the OS scheduling activities, one thread may pin a buffer on one CPU core and unpin the buffer on another CPU core belonging to a different group, leading to CPU cache-line synchronization. However, as the time interval of buffer pinning and unpinning is usually very short, which is much smaller than the OS scheduling interval (10 ms), the probability of such cases is very low.

Even if pinning and unpinning of the same buffer happen on different groups, *ScaleCache* can guarantee correctness as the modification of one per-group counter slice is serialized with the calculation of the sum of all slices in case of buffer replacement which is protected by the buffer state lock as shown in Figure 4. If one thread *T1* calculates the sum of one buffer's counter slices and does not see another thread *T2*'s pinning effect on the buffer, it would also not see *T2*'s unpinning effect as *T2*'s pinning procedure would not end until *T1* unlocks the buffer state. In such scenarios, overflow might occur where one slice counter increments to 1, while another counter becomes 65535 (equivalent to -1 in bit representation for uint16_t) following a pin and an unpin. However, this is inconsequential as the eviction check sums all counters collectively. Since 65535 and -1 are congruent modulo 65536, their sum modulo 65536 will be congruent to zero when represented as a uint16_t value. As a result, the sum will never overflow as long as the concurrency of DBMS does not exceed 65535.

Memory footprint. Each per-group buffer counter occupies $N * 2$ bytes where N is the number of groups. The original buffer counter in *GaussDB* is padded to occupy a separated 64-byte cache-line to avoid false sharing. When each group contains only one core (i.e., per-core buffer counters), *ScaleCache* performs the best across all configurations theoretically. However, as will be demonstrated in Section 4, on our 128-core *Huawei Kunpeng* server, with a configuration of each group containing 4 consecutive cores sharing the same *LLC*, *ScaleCache* performs almost the same with that of per-core buffer counters on all our experiments while the memory footprint of our per-group counting is equal to that of traditional buffer counter and is 1/4 of per-core counting. Even using per-core counting, on our 128-core *Huawei Kunpeng* server, the metadata space overhead is still reasonable 3% of the buffer pool size (8 KB page size in *GaussDB*).

3.2 Per-group page latch based on copy-on-write

Each index operation, e.g., search, insertion, or deletion, descends from the root to the leaf nodes and requires acquiring corresponding read latches on the non-leaf nodes along the traversal path. Besides, the number of non-leaf nodes of B-link tree is significantly smaller than that of leaf nodes. As a result, the concurrency contention on read latches of non-leaf nodes, particularly the root node, will remain substantially high even under uniform data accesses. A traditional reader-writer latch is implemented using an atomic variable to coordinate concurrent access, which is similar to that of buffer counters. However, per-group reference counting philosophy cannot be directly applied to read-write latches. Essentially, this approach speeds up the reference count modifying operations at the cost of checking whether the count is zero. While this trade-off

[3]More precisely, it refers that there does not exist expensive cache-line synchronization across LLCs here.

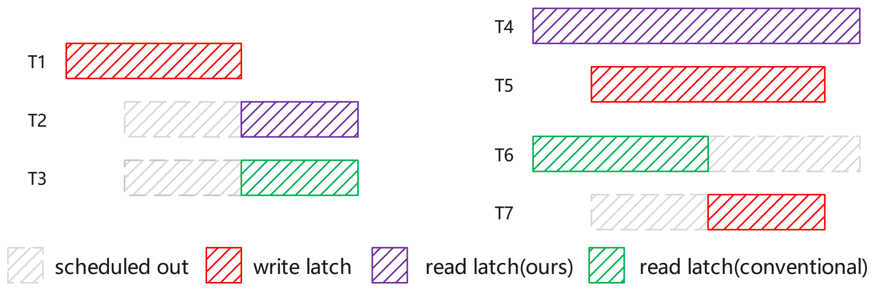


Figure 5: Example of latch in concurrency.

works in buffer managers where pin/unpin operations far outpace reference count checking, it becomes problematic for read-write latches. Acquiring a write latch may require multiple loop iterations to verify the distributed reader count, each triggering a CPU cache-line transfer, which may consequently degrade the overall system performance, necessitating the use of copy-on-write technology to eliminate frequent cache-line transfers.

In *ScaleCache*, the buffer structure within the buffer pool is redesigned with dual partitions: metadata information and data segments (i.e., buffer frames for simplicity), each independently managed via the per-group reference counting technology. The eviction of a buffer is controlled by metadata information. The operational rules are as follows:

(1) When allocating a buffer for a page, both its metadata and corresponding buffer frame are allocated and pinned (The allocation algorithm of buffer frame will be discussed later).
(2) During buffer eviction, the associated buffer frame is explicitly unpinned.
(3) Invoking buffer_pin/buffer_unpin triggers pin/unpin exclusively for metadata.
(4) Acquiring a read latch records and pins the current buffer frame, ensuring subsequent operations utilize this pinned buffer frame until the latch is released.
(5) Upon write latch acquisition, a new buffer frame is allocated, the old data is copied to it, the old segment is unpinned, and the new segment is pinned for further operations as illustrated in Figure 6.

This design ensures data observed by threads holding read latches remains valid (though potentially stale), while read latch acquisition operates solely through a lightweight per-group reference count increment, thus enabling scalability on many-core systems.

Allocation of buffer frames. The allocation of buffer frames must account for concurrent write latch acquisitions. When a write latch is obtained, multiple buffer frames associated with a buffer

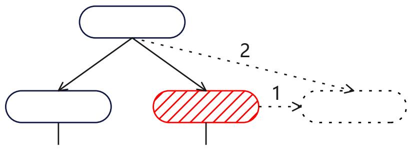


Figure 6: Schematic diagram of write latch acquisition.

might remain in use if other threads hold per-group read latches on the target buffer without releasing them, preventing the reuse of older buffer frames. Consequently, the buffer frame requirement slightly exceeds that of buffers. Since B-link tree operations can hold up to 3 write latches simultaneously, this overhead depends solely on system concurrency. For systems configured with a maximum concurrency of 512, allocating an additional 2,048 buffer frames during initialization (including redundancy for background workers) would suffice. The extra buffer frames are structured as a FIFO list. When allocating a new buffer frame, the candidate selection follows this order: 1) the buffer frame whose index matches the target buffer's index; 2) the first buffer frame in the list. For each candidate buffer frame, we verify its reference count. If reference count is not zero, it is reinserted into the buffer frame list. Otherwise, the buffer frame is pinned and assigned as the new buffer frame for the target buffer.

Concurrency correctness proof. We specify the conditions under which this approach maintains correctness in concurrent environments. First, we formally define a read-write latch as a synchronization primitive that adheres to these semantics no matter the underlying implementation:

S.1. When a write latch is held, neither read nor write latches can be acquired by other threads.
S.2. When a read latch is held, additional read latches may be acquired concurrently while write latch acquisition remains prohibited.
S.3. A write latch grants exclusive modification rights to its holder while blocking both read and write access from other threads until released.
S.4. A read latch prevents concurrent data modification.

Regarding semantics S.1, our synchronization mechanism aligns with conventional read-write latch implementations by verifying write latch occupancy prior to read latch acquisition. If a write latch is detected as held, the read latch acquisition will be blocked until the write latch is released. Examining Figure 5, thread T1 holds a write latch, while threads T2 and T3 attempt to acquire our read latch and conventional read latch respectively. Both T2 and T3 must await T1's release of the write latch, rendering them unable to differentiate between the latch types during acquisition. This demonstrates the equivalence of these two latch types regarding semantics S.1.

Regarding semantics S.2, conventional read latches block write latch acquisition, whereas our design permits write latch acquisition regardless of active read latches. While this behavioral discrepancy appears concerning, our design remains provably correct under the constraint that a thread never holds more than one latch simultaneously. Consider Figure 5: T4 holding our read latch coexists with T5 acquiring/releasing a write latch, a scenario initially suggesting inconsistency. However, comparing T6 (scheduled out after holding conventional read latch) and T7 (waiting a few time for write latch), both T4/T6 operate on stale data indistinguishably, while T5/T7 perform equivalent data updates. Since this potential inconsistency already exists in conventional implementations yet remains functionally valid, our latch maintains equivalent correctness guarantees under these boundary conditions. It's notable that the latch mechanism may encounter issues if the proposed constraint is violated. For instance, when accessing an array structure containing a data pointer and size field where each component is protected by different latches, data retrieval need acquire both read latches simultaneously. If a thread obtains an obsolete size value exceeding the actual capacity of the data pointer's allocated space, an out-of-bounds memory access error may occur.

Regarding S.3, our design preserves conventional write latch semantics without modification. Regarding S.4, while the data segment remains immutable for threads holding read latches, data segment related to corresponding buffer may undergo modifications. However, as established in our S.2 analysis, this behavioral maintains system correctness.

In conclusion, our design can effectively operate in concurrent environments under the constraint that a thread never holds more than one latch simultaneously. This constraint aligns naturally with the B-link tree's search mechanism. Specifically, during search descending, a thread releases the read latch on the parent node before acquiring a read latch on the child node, thus never holding more than one read latch.

At a first glance, there may exist two problems: (1) search operations may read stale data and (2) the child page may be deleted during decending. However, B-link tree adds a right-sibling link and high key to each node, which allows search operations to locate concurrent tree modifications [19] with at most one latch simultaneously held. For the second problem, any deleted page can be recycled only after its deletion *XID* (transaction ID) becomes visible to all active transactions under the DBMS MVCC (Multi-version Concurrency Control) mechanism such that all search operations that follow the downlink to the deleted page have completed.

Crucially, since our method is equivalent to conventional read-write latches under the above given constraint, we can guarantee the same concurrency safety for the databases that adopt the B-link tree index, e.g., PostgreSQL and *GaussDB*. If no concurrency issues are observed before adopting our method, we can guarantee their absence afterward. Our design further effectively mitigates potential concurrency-related performance degradation in B-link tree traversal operations while maintaining correctness.

It's notable that our per-group page latch design applies exclusively to non-leaf nodes. For leaf nodes, page data may be returned to upper operations (e.g., joins and aggs) and used beyond our control. Upper operation code might rely on the invariance of returned leaf nodes, or might not; to ensure correctness, leaf nodes must employ conventional read-write latches without further validation. Besides, this design intentionally introduces a read-write cost trade-off. While non-leaf nodes endure read contention across all workloads, leaf nodes bear write pressure in write-intensive scenarios, so applying our latch design to leaf nodes would consequently degrade write-heavy performance, rendering its adoption for leaf nodes of limited benefit.

3.3 Scalable and efficient page-to-buffer translation

To address the inefficiencies of the traditional page-to-buffer translation as mentioned in Section 2, we propose a scalable and highly efficient buffer lookup scheme. *ScaleCache* adopts a vectorized hashing scheme which uses a cache-friendly block-based bucket layout and leverages vector instructions (SIMD) to accelerate key lookup. Each block-based bucket consists of an array of key-value slots and an array of fingerprints that stores 8-bits key hash codes of the slots. The slot key consists of relation ID and page ID and the slot value stores the allocated buffer ID for the page. If the slot is not occupied, both key and value are set to *NULL*. The array size is set to 16 so that the fingerprints can be loaded into a SIMD vector register to compare the search key's fingerprint with them at once. The number of buffers is predefined (assuming N) when the DBMS starts. We allocate $N/2$ block-based buckets and $N/16$ overflow block-based buckets for the hash table, so that the hash table can hold all the page-to-buffer translation entries without the need of considering hash table resizing and without the corresponding synchronization cost. In case of block-based bucket overflow, we would keep its chained overflow buckets in a compact way. That is if one block bucket becomes sparse due to deletion we would move used slots from the tail bucket to it. Once the tail overflow bucket becomes empty, we would remove it from the chain and put it into the free overflow bucket list.

ScaleCache acquires per-bucket lock for hash table inserts and deletes while allowing optimistic hash table lookups without any locking. The consistency correctness is guaranteed in cooperating with our scalable and light-weight buffer state synchronization that would retry lookup upon conflicts. And if buffer lookup returns *NULL* due to concurrent modification, the buffer allocation procedure would finally find the existing entry (Algorithm 3 line 4). Such situation is rare. The details will be described later.

Fast and optimistic buffer lookup. As shown in Figure 2, to access a page, the DBMS first looks up its buffer from the buffer pool using its relation ID and page ID as key. For a search key, we first compute its hash code as well as its fingerprint, locate the bucket, and load the bucket fingerprints into a SIMD vector register for slot filtering by using SIMD instruction. For each fingerprint matching slot, *ScaleCache* performs the comparison of the search key with its key. As the probability of 8-bit collisions is very low, there exists only one matching slot at most time and the time complexity of our hash table lookup is around *O(1)*. If a key matching slot is found, we obtain its buffer ID. Otherwise, we jump to the next overflow block-based bucket if the next pointer of the current bucket is not *NULL* and then repeat the above process. If no matching slot is found, *ScaleCache* starts the buffer allocation procedure to allocate a buffer for the page to be accessed. Our buffer hashing scheme

Algorithm 2: Lookup a page in the buffer pool

```
   input  : relationID
   input  : pageID
   output : bufferID
1  while true do
2      key = relationID « SHIFT | pageID;
       /* SIMD-based and lock-free hash table lookup
          */
3      slot = SIMDHashTableSearch(key);
4      bufferID = slot->value;
5      if bufferID == Invalid then
6          return Invalid;
7      end
       /* pin the buffer                          */
8      PinBuffer(bufferID);
9      if relationID == bufferDesc[bufferID].tag.relationID &&
        pageID == bufferDesc[bufferID].tag.pageID then
10         break;
11     else
           /* The buffer is allocated to another page,
              we retry the lookup                  */
12         UnPinBuffer(bufferID);
13     end
14 end
15 return bufferID;
```

Algorithm 3: Allocate a free buffer to a page

```
   input  : relationID, pageID
   output : boolean, &res
1  key = relationID « SHIFT | pageID;
2  while bufferID = GetNextVictimBuffer() do
3      PinBuffer(bufferID);
       /* Increment the buffer counter.           */
4      if SIMDHashTableInsert(key, bufferID, &existingBufferID)
        == false then
           /* Another thread has allocated a free
              buffer to this page                  */
5          UnPinBuffer(bufferID);
6          return false;
7      end
8      LockBufferState(bufferID);
9      if Sum(perGroupCounters, bufferID) == 1 and
        bufferDesc[bufferID].dirtybit == 0 then
           /* assign this free and clean buffer to
              the page by modifying buffer desc    */
10         bufferDesc[bufferID].tag.relationID = relationID;
11         bufferDesc[bufferID].tag.pageID = pageID;
12         UnLockBufferState(bufferID);
13         break;
14     else
           /* this buffer cannot be allocated, retry
              another candidate buffer             */
15         UnLockBufferState(bufferID);
16         SIMDHashTableDelete(key);
17         UnPinBuffer(bufferID);
18     end
19 end
20 *res = bufferID;
21 return true;
```

integrates the best of the two worlds, i.e., open-addressing and chaining, and further leverages SIMD for acceleration.

After the buffer ID is obtained, *ScaleCache* reads the CPU core ID where the current thread runs from *rseq*. By using the CPU core ID and buffer ID, *ScaleCache* locates the corresponding per-group counter slice within the two-dimensional counter slice array and then pins the buffer by incrementing the slice.

As we do not acquire any locks during the buffer lookup procedure, the located buffer may be allocated to another page by other threads concurrently (i.e., buffer replacement). Our light-weight and scalable buffer state synchronization protocol would resolve such conflicts. Algorithm 2 shows the detailed buffer lookup pseudocode which is lock-free and CPU cache-line contention-free. Given the relation ID and page ID, the DBMS thread looks up the buffer ID from the buffer hash table (line 2-7). If it is found, the DBMS thread pins the buffer (line 8) and then rechecks whether the buffer state tag has been changed (line 9). If so, it means this buffer has been allocated to another page concurrently, the DBMS thread should unpin the buffer and then retry the whole buffer lookup procedure (line 11-13). Otherwise, the DBMS thread successfully obtains the buffer for page access (line 10).

Buffer allocation and replacement. Algorithm 3 shows the buffer allocation pseudocode. To allocate a free and clean buffer to a page, the DBMS thread first gets a buffer from the candidate buffer list which is maintained by background threads (line 2) and pins the buffer (line 3). Then the DBMS thread tries to insert the new page-to-buffer translation entry into the hash table (line 4). If insertion fails, it means this page has already been allocated a free buffer by another thread concurrently, and the allocation thread unpins the buffer and returns false to notify the caller to look up the allocated buffer (line 5-6). If insertion succeeds, the DBMS thread locks the candidate buffer state, calculates the sum of its per-group counter slices and checks whether it is equal to one (meaning it is only pinned by the current thread) and whether it does cache a clean page (line 9). If so, *ScaleCache* allocates this buffer to the target page by filling the target page ID and relation ID into this buffer's state tag (line 10-11). Otherwise, this buffer cannot be allocated and the DBMS thread should undo changes and retry other candidate buffers (line 15-17).

In the end, *ScaleCache* unlocks the buffer state (line 12). It is worth noting that *ScaleCache* would acquire corresponding per-bucket locks to protect the hash table against concurrent inserts and deletes and the locking detail is beyond our focus and ignored from Algorithm 3 for simplicity.

Concurrency correctness proof. The key to the concurrency correctness of *ScaleCache* is that the light-weight buffer state locking serializes concurrent lookup and replacement on the same buffer as shown in Figure 4. We now formally prove the correctness. Given one page *P1* and its allocated buffer *B1*, we consider such concurrency conflict situation where one thread *(*T1) is looking up which buffer caches *P1* (i.e., *B1*) while another concurrent thread *T2* tries to allocate *B1* to another page *P2*. After *T1* pins *B1* successfully, only two situations may exist: pinning effect by *T1* (1) has been seen by *T2* or (2) has not been seen by *T2* when calculating the sum of all the slices. For the first situation, *T2* cannot allocate *B1* to *P2* as the sum of the per-group counter would be greater than one. For the second situation, *T1* would pin the buffer *B1* successfully only after *T2* has unlocked the buffer state. Hence, upon the buffer *B1* being pinned by *T1* successfully, *T2* has already changed *B1*'s buffer state tag to record *P2* (Algorithm 3:line 10-11). As a result, *T1* would then fail in passing the rechecking of the pinned buffer's state tag (Algorithm 2:line 9) and retry the buffer lookup procedure that would finally return *NULL* and trigger the buffer allocation procedure for *P1*. Hence, our protocol can guarantee correctness of *ScaleCache* no matter which conflict situation occurs.

4 EVALUATION

We have integrated *ScaleCache* in *Huawei GaussDB*, a commercial high-performance DBMS production. In this section, we experimentally evaluate the performance of *ScaleCache* against that of the baseline (*GaussDB* with the traditional buffer manager) on the classic B-link tree lookup and complex DiskANN vector index search workloads, then quantify the performance contributions of each of our proposed techniques in this paper. Later we evaluate *ScaleCache* against the baseline on out-of-memory workloads. Lastly, we evaluate *ScaleCache* as CPU scaling and introduce a customer workload.

4.1 Experimental Setup

4.1.1 Hardware. The experiments are conducted on a machine equipped with two *Huawei Kunpeng* CPUs, each of which has 64 cores, and running the *EulerOS* operating system. Each core has a 128KB of L1 cache and 512KB of L2 cache, with every 32 cores sharing a 64MB L3 cache. The cache-line size is 64 bytes on this platform. The underlying physical machine is equipped with 2TB of DDR4 DRAM and four SSDs, each with a capacity of 7.3TB. If not explicitly mentioned, the shared buffer size of the database is set to 150 GB. The database client and server are deployed on different machines which are connected through a 25 Gbit/s network.

4.1.2 Baseline. We have integrated *ScaleCache* into *Huawei GaussDB*, a commercial high-performance and advanced DBMS. We choose *Huawei GaussDB* without *ScaleCache* as our baseline (referred as baseline in the following section) for fair comparison. Each group contains 4 consecutive CPU cores that share the same L3 cache for our per-group counting in our experiments.

To study the effectiveness of each of the techniques proposed in *ScaleCache*, we evaluate the following variants of *ScaleCache*:

- + per-group page latch: it adds the hybrid per-group page latching technique for the B-link tree index to the baseline.
- + SIMD hash: it further applies our optimistic, CPU cache-friendly and SIMD-accelerated hashing scheme for page-to-buffer translation.
- *ScaleCache*: it is a full implementation of *ScaleCache* which contains all our proposed techniques.
- Ideal: for the DiskANN vector indexing, we implement a page array in *ScaleCache* as a substitute for the buffer hash table to eliminate the hashing cost, which, we believe, can represent the theoretically optimal performance. As the page array approach needs to consume a large amount of memory, it only serves as a theoretical optimal upper bound for comparison in our paper.

It is worth noting that we have not yet applied per-group page latching to the DiskANN vector index and such optimization for vector index is expected to further improve the vector search performance significantly (as seen in Figure 1) and we leave this as our future work.

4.1.3 Benchmark. It is important to note that buffer pool and latch bottlenecks do not manifest across all workloads. Modern DBMS, as complex systems, may experience performance constraints in I/O, network, or other components. Our approach demonstrates performance improvements specifically in scenarios where buffer pool efficiency is critical, including: 1) multi-point random lookups (e.g., sysbench select in queries), 2) nested-loop join operations (e.g., TPC-C stock-level transaction queries, some queries in TPC-H), and 3) vector search workloads (e.g., DiskANN-based searches). While no performance gains are observed in comprehensive benchmarks like TPC-C where bottlenecks lie elsewhere (e.g., transaction concurrency or disk I/O saturation), our method maintains parity with baseline performance. These special scenario is also important as it appears in some real production task. These targeted optimizations address practical production needs where buffer pool contention directly impacts system responsiveness.

Multi-random point lookup. We use sysbench to generate the multi-random point lookup workload that performs 10 random point lookup at one time by using select in queries, eliminating multiple network roundtrip between the client and the DBMS and thus effectively stressing the indexing and the buffer pool layer.

TPC-C. We use TPC-C to generate typical and complex OLTP workloads using *BenchmarkSQL-5.0* and set the number of warehouses to 1000. The default configuration of TPC-C is a write-heavy workload (around 90% write) where the throughput of *GaussDB* is limited by other factors such as the write-ahead logging while index search only occupies around 15% of the overall time. Hence, we also use TPC-C to generate read-write and read-only transactional workloads where the ratio of the *new-order* write transaction and the *stock-level* read-only transaction is set to three levels in our experiments: “20:80” , “50:50” and “0:100” separately. Notably, the "0:100" configuration resembles the characteristics of one read-mostly production workload generated by one of our large financial bank customers, which executes nested-loop joins on indexed tables and involves excessive B-link tree index lookups like the TPC-C *stock-level* transaction.

TPC-H. We use the *BenchBase* [5, 11] to generate concurrent TPC-H [36] queries to the DBMS by using 512 threads. Under default configurations (SF is set to 1), TPC-H exhibits observable performance improvements in buffer pool-bound queries due to its deterministic execution pattern where each query runs exactly once per test iteration. The non-uniform query cost time distribution (with specific queries dominating execution time) allows such bottlenecks to directly manifest as measurable throughput gains when buffer pool contention is alleviated, requiring no parameter tuning to validate the optimization impact.

DiskANN vector search. We also evaluate the performance of *ScaleCache* against the baseline on the complex *GaussDB* DiskANN vector index with the SIFT1M dataset. DiskANN [35] is a graph-based vector index, which may visit many pages for each query, effectively stressing the buffer manager, i.e., page-to-buffer translation and buffer pinning/unpinning.

4.2 Multi-random point lookup

We first use sysbench to generate multi-random point lookup of B-link tree workload with varying number of client threads. The dataset size is set to 118 GB. As shown in Figure 7, *ScaleCache* exhibits near-linear scalability on our 128-core server and improves the throughput by up to 1.6X compared to the baseline. When the number of client threads exceeds 100, we can observe that the throughput of the baseline remains almost unchanged. The throughput of *ScaleCache* scales linearly to 128 cores and still increases greatly until the number of concurrency reaches around 300, which attributes to our scalable, highly efficient and practical buffer pool design while the traditional buffer manager suffers from scaling bottlenecks and poor CPU cache locality.

In Figure 7, we observe that each of all the optimizations in this paper (hybrid per-group page latching, SIMD hashing, and per-group counting) contributes to the performance improvement significantly. Compared to the baseline, the per-group page latching technique improves the throughput by 60%. When combined with our proposed SIMD hashing for page-to-buffer translation, *ScaleCache* achieves a 15% improvement in addition. Finally, after the per-group reference counting has eliminated any CPU-cache line synchronization within the buffer pool on many-core hardware, our full implementation achieves a 160% improvement in total. This breakdown result shows that all of our proposed techniques in this paper can significantly improve the performance of parallel B-link tree lookups.

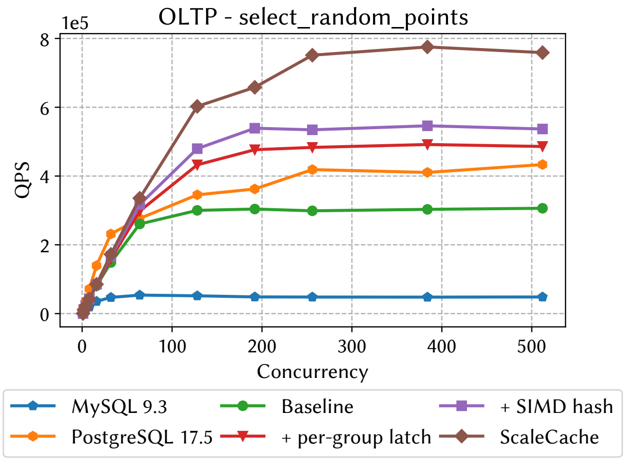


Figure 7: Throughput (Queries per second) of multi-random point lookup.

Compared to MySQL 9.3 and PostgreSQL 17.5, MySQL's use of B+ trees (vs. PostgreSQL/GaussDB's B-link trees) results in worse concurrency scaling—performance plateaus earlier as concurrency increases. While PostgreSQL 17.5 initially outperforms GaussDB due to optimized B-tree index scans for IN(...) lists, which is introduced in PostgreSQL 17.0, GaussDB significantly surpasses PostgreSQL 17.5 when leveraging *ScaleCache*. Notably, *ScaleCache* could similarly enhance PostgreSQL's performance because of the same underlying B-link tree.

4.3 TPC-C

We then evaluate the end-to-end performance of the baseline and *ScaleCache* on TPC-C with 580 client threads. Figure 8 shows the measured tpmTOTAL (total transactions per minute) under different configurations.

As shown in Figure 8, *ScaleCache* outperforms the baseline by 333% under the write-read "0:100" configuration. Under the "20:80" and "50:50" of write-read ratio configurations, *ScaleCache* improves the throughput by 136% and 18%, respectively. With the default configuration of TPC-C that is write-heavy, we observe that *ScaleCache* achieves a ~3% improvement compared to the baseline as the index search time only occupies around 15% of the overall time. We can see that our performance improvement varies with the write-read ratio. While the primary focus of *ScaleCache* is on optimizing index query performance, the optimizations can still benefit write-heavy scenarios. Figure 8 also demonstrates the performance contributions of each of the optimizations in this paper.

Compared to MySQL 9.3 and PostgreSQL 17.5, *GaussDB* implements extensive TPCC-specific optimizations (omitted here as non-core to this paper), resulting in fully outperforming both in TPCC benchmarks. Under our high-concurrency TPCC testing, MySQL exhibits inferior performance to PostgreSQL and *GaussDB*, attributing to the same reason illustrated before. Crucially, we observe a consistent pattern: without *ScaleCache*, higher *stock-level* weight correlates with TPS degradation across all databases. *ScaleCache* reverses this trend, demonstrating its exceptional capability in nested-loop join scenarios.

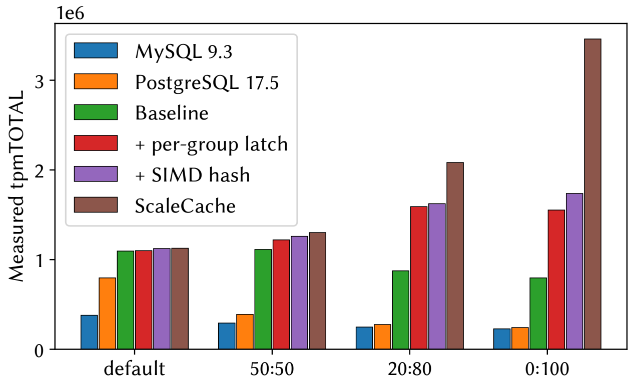


Figure 8: TPC-C performance under different configurations.

4.4 TPC-H

Figure 9 demonstrates the TPC-H throughput comparison between *ScaleCache* and the baseline. As previously noted, specific queries among the 22 TPC-H queries (particularly those containing nested-loop joins on indexed tables, analogous to TPC-C's stock-level transaction pattern) exhibit prolonged execution times. This latency concentration enables *ScaleCache* to achieve a 16% overall throughput improvement through buffer pool optimization.

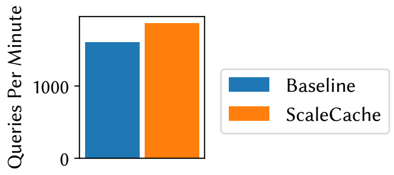


Figure 9: Throughput of TPC-H on *ScaleCache* compared to the baseline

4.5 DiskANN

In addition to the B-link tree index, we also evaluate the DiskANN vector index performance of *ScaleCache* in comparison to the baseline on the SIFT1M dataset. Figure 10 shows the latency of parallel index construction using 64 threads, and the QPS throughput as well as latency of vector search under 128 client threads. Specifically, *ScaleCache* reduces the time taken to construct vector index by 18%. In terms of vector search, *ScaleCache* improves the throughput by 46% and reduces latency by 33% compared to the baseline, nearly reaching the theoretical upper bound indicated by the "Ideal". *ScaleCache* accelerates vector search without any modification to the *GaussDB* DiskANN vector index, demonstrating that *ScaleCache* can speed up complex index searching transparently.

4.6 Per-core vs Per-group counting

As mentioned in Section 3, per-core counting is a special case that allocates a counter slice for each core. In this section, we compare two configurations of *ScaleCache*: *ScaleCache (per-core counting)* and *ScaleCache (per-group counting)*. Figure 13 presents the performance comparison of *ScaleCache* (per-core counting) and *ScaleCache* (per-group counting) in previous experiments. We observe that the performance gap is negligible, as cores within the same group share the same L3 cache, which helps reduce the cache-line synchronization cost of pinning/unpinning. Additionally, the communication latency between adjacent cores is very low for every set of four consecutive cores, as reported in [8].

As shown in Figure 13(a), the random multi-point lookup throughput using per-group counting reaches 96%-100% of that achieved with per-core counting. Both per-group and per-core counting exhibit near-linear scalability. Evaluation of TPC-C shows that the performance gap between *ScaleCache* (per-core counting) and *ScaleCache* (per-group counting) is less than 3% across different configurations, as shown in Figure 13(b). The performance difference is also minor in the DiskANN vector index experiments as shown in Figure 13(c). As discussed in Section 3, per-group counting (with 4 cores per group) reduces memory consumption to one-quarter of that required by per-core counting, making it being the same with the memory consumption of traditional centralized buffer counting while largely improving the index query performance.

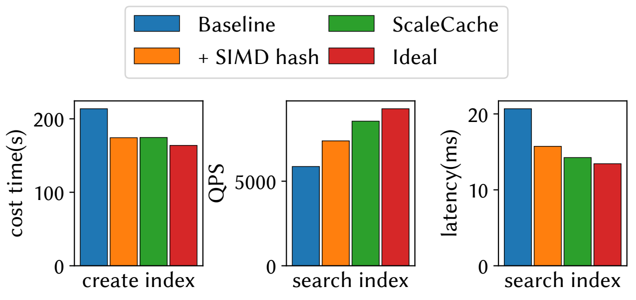


Figure 10: DiskANN vector index performance on SIFT1M, including the cost time of index construction, the QPS and latency of vector search.

4.7 Out-of-Memory Evaluation

It is important for a disk-based DBMS to process out-of-memory workloads gracefully and our optimistic protocol may retry buffer lookup procedure upon conflicts with buffer replacement. Moreover, our custom latch incurs higher acquisition costs for write operations, which occur more frequently during intensive buffer eviction scenarios. Therefore, we must demonstrate that our approach maintains performance stability under such conditions. We evaluate the performance of *ScaleCache* compared to the baseline when the dataset size exceeds the shared buffer capacity significantly, which triggers frequent page eviction and buffer replacement. In

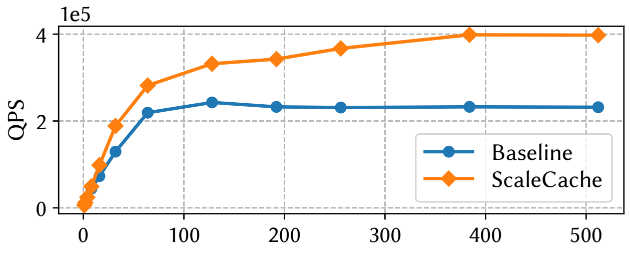


Figure 11: Throughput of *ScaleCache* compared to the baseline on the out-of-memory multi-random point lookup workload

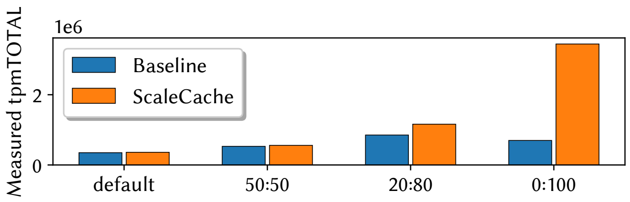
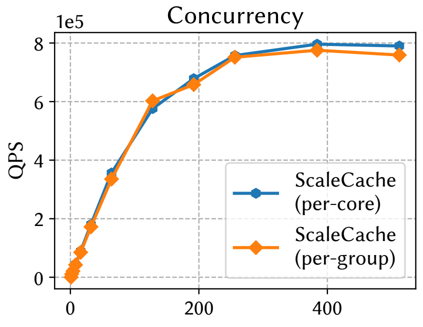
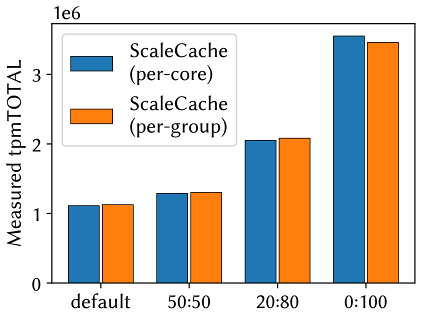


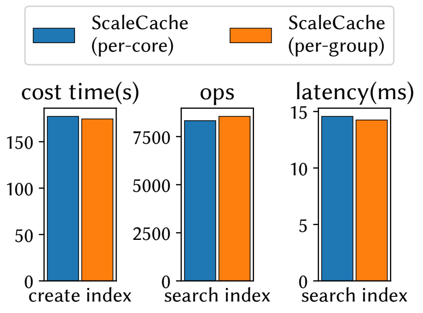
Figure 12: Throughput of *ScaleCache* compared to the baseline on the out-of-memory TPC-C workload



(a) Multi-random point lookup throughput



(b) TPC-C throughput when varying write-read ratio



(c) DiskANN index performance

Figure 13: Performance evaluation of per-group counting compared to per-core counting

the following experiments, we set the shared buffer size to 8 GB. To enforce page eviction, the dataset size is set to 280 GB for the multi-random point lookup workload. Figure 11 shows that that *ScaleCache* can also outperform the baseline by 74%. The percentage of improvement is smaller because I/O becomes a bottleneck when page eviction and load occur frequently. We observe that nearly 30% of CPU cycles are spent waiting for data to be read from disk, which reduces the performance gap between the baseline and *ScaleCache*. Like the scenario without page eviction, *ScaleCache* demonstrates better scalability on many-core hardware.

For TPC-C, the dataset size is around 232 GB. As shown in Figure 12, the throughput of *ScaleCache* consistently outperforms the baseline in the out-of-memory TPC-C workloads. Especially, *ScaleCache* achieves a 3.9X performance improvement compared to the baseline under the configuration '0:100'.

4.8 CPU Scaling Evaluation

We tested performance across different CPU socket configurations, denoting 2-socket (128-core) as 2P and 4-socket (256-core) as 4P. As shown in Figure 14, *GaussDB* exhibits performance degradation as core count increases, attributing to rising inter-core communication overhead. Conversely, *ScaleCache* achieves near-linear scaling on 4P—performance rises proportionally with concurrency up to 256 concurrency (matching 4P's core count). This demonstrates *ScaleCache* 's superior linear scalability.

4.9 Customer Workload Evaluation

We also evaluated the performance improvement brought by *ScaleCache* on a realistic read-only workload that is produced by one of our customers, a large insurance company in China. The customer deployed 9 clusters of *GaussDB* with a total of 120 TB data. The read-only workload consists of SQLs that query insurance user information, car insurance orders and car insurance claims process progress infromation,etc. In the database perspective, the workload performs many multi-point index lookups with aggregations and a few joins, which stresses the buffer pool significantly as observed in the *perf* flame graph. *ScaleCache* improves the workload throughput from 13.5 w tps to 18 w tps on the *Huawei Kunpeng* server with four processors (256 cores) compared to the previous version of *GaussDB*, better satisfying the high throughput requirement.

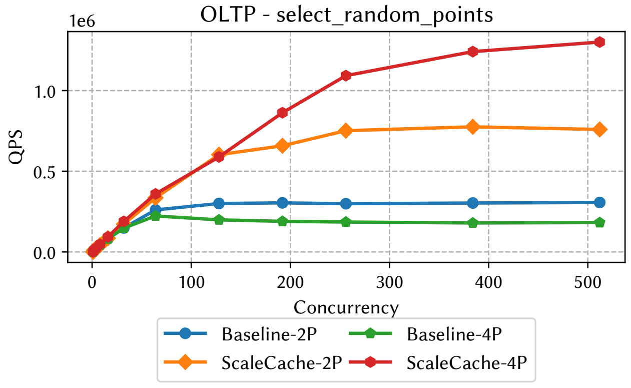


Figure 14: Throughput (Queries per second) of multi-random point lookup with different CPU sockets.

5 CONCLUSION

The buffer management serves as the cornerstone of modern on-disk database management systems (DBMS), which however suffers from severe performance bottlenecks in index query execution due to use of centralized buffer reference counting , lock-protected and chained hashing for page-to-buffer translation. In this paper, we present *ScaleCache*, a scalable, highly efficient and production-grade buffer management system on modern many-core hardware, which has been implemented in *GaussDB*. *ScaleCache* proposes three efficient techniques, namely per-group counting, per-group latch based on copy-on-write, and optimistic, CPU-cache friendly and SIMD accelerated hashing, to eliminate most cache-line contention and to improve CPU cache locality. Experimental results show that *ScaleCache* exhibits near-linear scalability and can improve index query throughput of both classic B-link tree index and complex graph-based vector index significantly.

REFERENCES

[1] Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. 2023. Analyzing Vectorized Hash Tables Across CPU Architectures. *Proc. VLDB Endow.* 16, 11 (2023), 2755–2768.

[2] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings.* 1–16.

[3] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. [n.d.]. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy.* 181–190.

[4] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: scalable address spaces for multithreaded applications. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013.* 211–224.

[5] CMU. [n.d.]. BenchBase (formerly OLTPBench) is a Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase

[6] Jonathan Corbet. 2006. The search for fast, scalable counters. https://lwn.net/Articles/170003/

[7] Jonathan Corbet. 2013. Per-CPU reference counts. https://lwn.net/Articles/557478/

[8] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. 2021. CLoF: A Compositional Lock Framework for Multi-level NUMA Systems. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021.* ACM, 851–865.

[9] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (2013), 1942–1953.

[10] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* ACM, 1243–1254.

[11] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.

[12] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *Proc. VLDB Endow.* 7, 11 (2014), 931–942.

[13] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph A. Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *Proc. VLDB Endow.* 8, 1 (2014), 37–48.

[14] Paul Groß, Daniel ten Wolde, and Peter Boncz. 2025. Adaptive Factorization Using Linear-Chained Hash Tables. In *CIDR 2025.*

[15] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008.* ACM, 981–992.

[16] HUAWEI. [n.d.]. Kunpeng 920-6426 - HiSilicon. https://en.wikichip.org/wiki/hisilicon/kunpeng/920-6426

[17] Seokyong Jung, Jong-Bin Kim, Minsoo Ryu, Sooyong Kang, and Hyungsoo Jung. 2019. Pay Migration Tax to Homeland: Anchor-based Scalable Reference Counting for Multicores. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019.* USENIX Association, 79–91.

[18] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany.* IEEE Computer Society, 195–206.

[19] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670.

[20] 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.

[21] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.* 185–196.

[22] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[23] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016.* ACM, 3:1–3:8.

[24] Paul McKenney. 2019. The RCU API, 2019 edition. https://lwn.net/Articles/777036/

[25] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. 2020. RCU Usage In the Linux Kernel: Eighteen Years Later. *ACM SIGOPS Oper. Syst. Rev.* 54, 1 (2020), 47–63.

[26] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. Memory-mapped I/O on steroids. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021.* ACM, 277–293.

[27] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *Proceedings of the 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* USENIX Association, 537–550.

[28] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018.* ACM, 490–502.

[29] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar González-Férez, and Angelos Bilas. 2021. Kreon: An Efficient Memory-Mapped Key-Value Store for Flash Storage. *ACM Trans. Storage* 17, 1 (2021), 7:1–7:32.

[30] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020.* USENIX Association, 813–827.

[31] Andrew Pavlo. 2014. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems.* Ph.D. Dissertation. Brown University, USA.

[32] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* USENIX Association, 1–16.

[33] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* ACM, 731–742.

[34] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings.* USENIX Association, 33–46.

[35] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Simhadri. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS 2019.*

[36] TPCH. [n.d.]. TPC-H is a Decision Support Benchmark. http://www.tpc.org/tpch/

[37] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013.* ACM, 18–32.

[38] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* USENIX Association, 465–477.

[39] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. 2023. User-space Bypass: Accelerating Syscall-intensive Applications. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023.* USENIX Association, 33–49.