OptiQL:
Robust Optimistic Locking for Memory-Optimized Indexes

To appear at SIGMOD 2024

Ge Shi
Simon Fraser University
shiges@sfu.ca

Ziyi Yan
Simon Fraser University
zya106@sfu.ca

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

ABSTRACT

Modern memory-optimized indexes often use optimistic locks for concurrent accesses. Read operations can proceed optimistically without taking the lock, greatly improving performance on multi-core CPUs. But this is at the cost of robustness against contention where many threads contend on a small set of locks, causing excessive cacheline invalidation, interconnect traffic and eventually performance collapse. Yet existing solutions often sacrifice desired properties such as compact 8-byte lock size and fairness among lock requesters.

This paper presents optimistic queuing lock (OptiQL), a new optimistic lock for database indexing to solve this problem. OptiQL extends the classic MCS lock—a fair, compact and robust mutual exclusion lock—with optimistic read capabilities for index workloads to achieve both robustness and high performance while maintaining various desirable properties. Evaluation using memory-optimized B+-trees on a 40-core, dual-socket server shows that OptiQL matches existing optimistic locks for read operations, while avoiding performance collapse under high contention.

ACM Reference Format:
Ge Shi, Ziyi Yan, and Tianzheng Wang. 2024. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes: To appear at SIGMOD 2024. In *Proceedings of the 2024 International Conference on Management of Data (SIGMOD '24).* ACM, Santiago, Chile, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Concurrent indexes [5, 7, 14, 25, 27, 28, 31, 33, 34, 37] are crucial for OLTP engines to achieve the desirable performance and functionality on modern servers that can feature 10s–100s of CPU cores across multiple sockets. These indexes use a synchronization scheme to ensure correct concurrent accesses. Importantly, the synchronization scheme must be well-crafted to offer (1) high performance and (2) desired properties such as fairness among worker threads, low lock space consumption and backward compatibility for easy adoption. To reach these goals, memory-optimized indexes have moved away from traditional pessimistic lock coupling [4] to lock-free approaches [31, 48] or optimistic locking [26]. Lock-free indexes, however, do not always outperform those based on locking [15, 48] and are notoriously hard to implement and debug [36, 47]. Therefore, recent indexes focused more on optimistic locking.

SIGMOD '24, June 11–16, 2024, Santiago, Chile

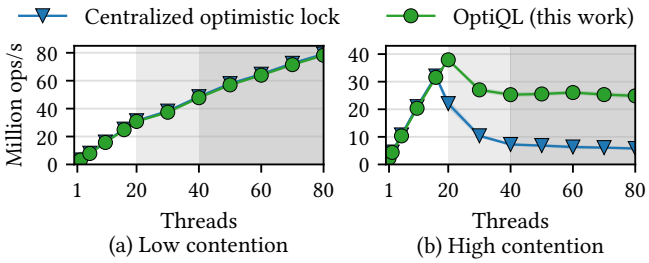


Figure 1: Update throughput of a memory-optimized B+-tree on a 80-thread server. Centralized optimistic locks work well under low contention (a) but can collapse under high contention (b). OptiQL performs well in both cases.

1.1 Centralized Optimistic Locking

Compared to lock-free designs and pessimistic locks, *optimistic locks*[1] [7, 28] are both easy to implement and high-performance. The basic idea is to allow readers to proceed assuming the lock is not held exclusively without issuing any hardware synchronization instruction to acquire the lock in shared mode. This greatly reduces synchronization overhead by avoiding shared memory writes for read operations. However, a concurrent writer can acquire the lock in exclusive mode and modify the data being used by readers, without knowing their existence. So readers must validate that the data read earlier did not change upon completion.

Optimistic locks also preserve the same API and compactness as traditional pessimistic locks by using only a single 8-byte memory word as the lock. Thus, optimistic locking has been adopted by multiple recent memory-optimized indexes [26, 28, 29, 33, 34], especially for read-heavy workloads.

However, existing optimistic locks are no panacea as they in fact achieve high performance at the cost of robustness, i.e., the ability to degrade gracefully and maintain reasonable performance under high contention. Figure 1 shows the throughput of a memory-optimized B+-tree using optimistic locking (details in Section 7) when running an update workload on a dual-socket, 40-core (80-hyperthread) server. Optimistic locking scales well as core count increases under low contention where keys are sampled randomly from a uniform distribution. But the performance collapses under a high-contention workload where 80% of the accesses focused on 20% of the keys. The reason is existing optimistic locks are designed assuming contention is rare and implemented in a *centralized* fashion as a direct extension to simple spinlocks [42] that use the

[1] Also known as "latches" to differentiate from "locks" used in logical-level concurrency control [16]. We follow the synchronization literature to use "locks".

hardware-provided compare-and-swap (CAS) instruction [22] to acquire the lock in exclusive mode. Yet in practice database workloads can exhibit various contention levels. Under high contention, many threads will issue and retry CAS on a small set of locks. The CPU then spends most cycles on retrying CAS without doing useful work, leading to performance collapse in Figure 1(b).

Such performance collapse issue is not unique to existing optimistic locks, but all centralized locks. Both the synchronization and database communities have explored solutions, but they still fall short in various scenarios. For example, exponential backoff [2] can effectively ease contention on the lock, but requires significant per-application tuning [42] and exacerbates unfairness among lock requesters which can lead to high tail latency. For example, in our experiments we observed that the "lucky" threads can be ~3× more likely to acquire the lock than others, making exponential backoff undesirable for database workloads. Other approaches, such as composite locks [12, 13], often use bigger lock words, posing non-trivial challenges to existing database engines that already assume 8-byte (pessimistic or optimistic) locks [24, 45]. Thus, it remains challenging to devise synchronization schemes for memory-optimized indexes that are both fast and robust, while maintaining various desirable properties.

1.2 OptiQL: Optimistic Queuing Lock

This paper proposes OptiQL, a new optimistic lock, to solve the aforementioned problems. OptiQL draws inspirations from the synchronization literature to introduce queue-based locking [9, 35, 38] to optimistic locks for both fairness and robustness. Queue-based locks are known for improved robustness under high contention [20, 42], by forming a queue of requesters which are granted the lock in FIFO order. Each lock requester spins locally on a private shared memory location to wait for its predecessor to grant the lock, instead of frequently retrying CAS on the lock.

OptiQL follows the classic queue-based MCS lock [38] to form a queue of requesters.[2] Different from existing queue-based locks which only provide mutual exclusion or blocking read [39], however, OptiQL forms a queue *for exclusive lock requests only*; readers still proceed optimistically based on validation without taking the lock in exclusive mode. Since the lock (in exclusive mode) is handed over from the previous lock holder to the next following the queue, there would be little to no chance for optimistic readers to observe a free lock if we blindly adopt the above approach, practically blocking most (if not all) readers. Therefore, to truly enable optimistic reads, we further propose an *opportunistic read* technique which admits readers between exclusive lock handover, i.e., after the predecessor writer has finished executing its critical section, but before the next writer is granted the lock. This way, OptiQL combines the best of both queue-based locks (for their fairness and robustness under contention) and existing optimistic locks (for their high-performance under read-dominant and low-contention workloads).

OptiQL maintains backward compatibility by keeping the lock compact as a single 8-byte word. Indexes that already use 8-byte locks can adopt OptiQL without data layout changes. In addition, OptiQL can be adopted by existing optimistic or pessimistic lock coupling protocols. However, as we elaborate later, OptiQL's queue-based design makes it necessary to tweak these lock coupling protocols for the best results. For example, blindly replacing traditional optimistic locks with OptiQL in a B+-tree may lead to unnecessary aborts and retries for update workloads due to version changes, even though the update can be applied correctly. We tackle these challenges in Section 6 by adapting concurrent B+-trees [29] and tries [27, 28] which are based on traditional OLC to use OptiQL. From our experience, these changes are important for performance reasons but straightforward. For example, we only needed to change fewer than 100 LoC out of over 600 LoC to adapt an existing OLC-based B+-tree. Moreover, as we show in later sections, OptiQL itself is also very simple to implement. These properties enable easier adoption of OptiQL in practice.

We use microbenchmarks to stress test OptiQL itself and more realistic index benchmarks to test how OptiQL performs in popular indexes. Evaluation on a dual-socket, 40-core server show that under contention, a concurrent B+-tree [29] and trie (ART [27]) enabled by OptiQL outperform their counterparts with traditional OLC by up to ~4×, while maintaining practically the same performance as traditional optimistic locking under low contention.

Our focus is index locking, but like other locks, OptiQL itself is general-purpose and can be useful beyond indexing. Similar to how we devise lock coupling protocols, however, additional changes may be needed, which is beyond the scope of this paper.

1.3 Contributions and Paper Organization

This paper makes four contributions. ❶ We revisit OLTP index synchronization and uncover several properties that are desirable but were often ignored in previous designs. ❷ We propose OptiQL, a new optimistic lock that offers robust performance under various contention levels, while maintaining efficient optimistic reads and other desirable properties. ❸ Based on OptiQL, we adapt locking protocols in representative OLTP indexes, including a memory-optimized B+-tree and the adaptive radix trie (ART). ❹ We provide a comprehensive evaluation of OptiQL and alternative synchronization primitives for OLTP indexes. OptiQL is available at https://github.com/sfu-dis/optiql.

The remainder of this paper is organized as follows. Section 2 gives the necessary background. Section 3 summarizes the desired properties of index locking. We describe OptiQL and its use in index locking in Sections 4–6. Section 7 evaluates OptiQL. We cover related work in Section 8 and summarize in Section 9.

2 A PRIMER ON (INDEX) SYNCHRONIZATION

Lock-based synchronization (optimistic or pessimistic) for OLTP indexes typically consists of two components determined at design time: (1) the lock primitive and (2) a protocol that uses the primitives to achieve concurrent accesses. Next, we give the necessary background around these two aspects.

2.1 Pessimistic Index Locking

We take B+-trees as an example to review the background. Traditional systems combine pessimistic locks (e.g., reader-writer locks) and lock coupling [16]. Each index node (inner or leaf) is protected

[2] At a first glance, this is similar to how lock managers for logical level concurrency control work [19]. However, OptiQL directly uses hardware synchronization instructions to realize the functionality, whereas lock managers usually use latches such as OptiQL and mutexes/spinlocks to build lock queues and coordinate requesters [16].

by a lock to enable fine-grained concurrency and more parallelism. The lock is typically embedded in the header part of the in-memory representation of the node, and held in the shared (exclusive) mode for read (write) accesses. With pessimistic locks, the calling thread is blocked (busy-spins or gets rescheduled) until the lock is granted; some locks also offer a "try" interface that returns without blocking if the lock cannot be granted immediately.

Although a node can be locked individually, a thread often needs to lock more than one node for structural modification operations (SMOs) like node split and merge, which in turn is often triggered as part of an insert or delete operation. The common approach is lock coupling [16], which locks nodes in the desired shared/exclusive mode as it drills down the tree, but releases the lock on a parent node after it is sure that performing the intended operation (e.g., insert) in the child node will not require revisiting the parent node (e.g., for an SMO that propagates up the tree). This way, usually the thread will only need to hold 1–2 locks, thus improving performance.

Pessimistic index locking has been the standard practice in traditional systems which are storage-centric, because in these systems storage I/O often presents a bigger bottleneck that overshadows index synchronization overhead. As we discuss next, however this is no longer the case in memory-optimized systems.

2.2 Optimistic Locking

Modern memory-optimized systems assume the working set resides in DRAM, so the relative overhead caused by synchronization is much higher than that in a storage-centric system, as both data access and synchronization are pure memory accesses with comparable latency. These indexes then opt for a different combination of optimistic locks and correspondingly, optimistic lock coupling.

Optimistic Locks. Existing optimistic locks [18, 28] are centralized and extends test-and-test-and-set (TTS) [41] locks in Figure 2(a) with optimistic reads. Same as TTS, they use a single 8-byte memory word as the lock, as Figure 2(b) shows. In addition to the lock state using the most significant bit (`1UL<<63`), the lock also carries a version number. The lock bit arbitrates mutual exclusion between writers and the version counter records how many times the lock has been acquired and released. In the `acquire_ex` function, writers synchronize with each other like with a TTS lock using a `CAS` that assumes the lock bit is unset, but additionally increment the version counter upon releasing the lock. In the `acquire_sh` function, readers start by recording the version number in a self-maintained variable `v`. If `acquire_sh` returns `true`, then the reader can start its critical section (e.g., to access a tree node). When the access is done, we invoke `release_sh` which takes the previously read version `v` as an input to validate that the lock still carries `v` *and* is unlocked. Otherwise, the reader aborts and retries. Notably, the reader functions behave similarly to try locks: they do not block and may fail. The caller must keep track of the version number for validation later, whereas the writers use exactly the same interfaces as TTS and reader-writer locks. Both TTS and optimistic locks can optionally use backoff [2] strategies to reduce contention. However, this is at the cost of fairness between lock requesters [42].

Optimistic Lock Coupling. With the interfaces in Figure 2(b), it is straightforward to adapt traditional lock coupling to use optimistic locks. For brevity, we give the high-level idea; details can be found elsewhere [26]. Figure 2(c) shows the same example used by prior work [26] that contrasts traditional (left) and optimistic (right) lock coupling. The key difference is the reader should conduct an extra validation step after invoking the `acquire_sh`/`release_sh`, and if any failed, the operation is aborted and retried. The example in Figure 2(c) shows the case where an exclusive lock is needed only at the leaf level (node *C*) for an update operation. If an inner node (e.g., *B*) needs to be locked exclusively, e.g., as part of an insert, the thread will attempt an "upgrade" operation which performs a `CAS` on *B*'s lock assuming the previously-obtained version number (`v'`). We expand later on how OptiQL can work with OLC in Section 6.

```
uint64_t lock;

void acquire() {
  while (lock == 0 ||
    !CAS(&lock, UNLOCKED, LOCKED)) {
    /* optional backoff */
    continue;
  }
}
```

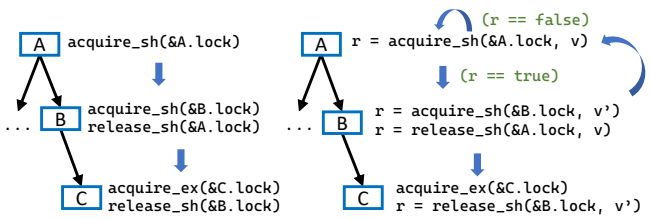
```
#DEFINE LOCKED   1UL
#DEFINE UNLOCKED 0UL

void release() {
  lock = UNLOCKED;
}
```

(a) Test-and-test-and-set (TTS) lock

```
uint64_t lock;
#DEFINE LOCKED 1UL << 63

void acquire_ex() {
  while (true) {
    uint64_t v = lock;
    if ((v & LOCKED) ||
      !CAS(&lock, v, v | LOCKED)) {
      /* optional backoff */
      continue;
    }
  }
}
```

```
void release_ex() {
  lock = (lock + 1) & ~LOCKED;
}

bool acquire_sh(uint64_t &v) {
  v = lock;
  return !(v & LOCKED);
}

bool release_sh (uint64_t v) {
  return lock == v;
}
```

(b) Existing optimistic lock based on TTS



(c) Traditional (left) and optimistic (right) lock coupling

Figure 2: Existing optimistic locks extend spinlocks (a) to support optimistic readers (b). The similar interfaces allow easy adaption in lock coupling protocols (c).

2.3 Queue-based and MCS Locks

Centralized locks like TTS and existing optimistic locks suffer from scalability collapse due to centralized spinning on the lock word. Queue-based locks [9, 35, 38] solve this problem by linking requesters in a FIFO queue, which in turn allows each thread to spin `locally` on a private space, instead of on the centralized lock itself, thus reducing contention and improving overall performance.

MCS [38] is one of the most widely used queue-based locks designed for mutual exclusion. We therefore focus on it; other queue-based locks (such as CLH [9, 35]) work similarly but differ in how they manipulate the queue [42]. To facilitate queuing, each lock requester brings a queue node that consists of (1) a `next` pointer to the next requester (successor) and (2) a boolean (`granted`) indicating whether the queue node's owner is granted the lock. The

Algorithm 1 MCS lock acquire (left) and release (right) protocols.

```
uint64_t lock;

1. void acquire(QNode *qnode):
2.  pred = XCHG(&lock, qnode)
3.  if pred == null:
4.    qnode->granted = true;
5.    return;
6.  pred->next = qnode;
7.  while (!qnode->granted) {}
```

```
1. void release(QNode *qnode):
2.  if (!qnode->next &&
3.      CAS(&lock, qnode, null))
4.    return;
5.
6.  while (!qnode->next) {}
7.
8.  /* Pass lock to successor */
9.  qnode->next->granted = true;
```

lock is an 8-byte memory word that always points to the tail of the queue, whereas the head of the queue owns the lock.

Algorithm 1 shows how a thread acquires/releases the lock. After initializing its queue node (with `next`/`granted` set to `NULL`/false), a lock requester joins the queue by issuing an atomic swap (`XCHG`) [22] on the lock word in line 2 of Algorithm 1(left). `XCHG` returns the value that was stored on the lock. If `NULL` is returned, the lock is free and the requester is granted the lock (lines 3–4). Otherwise, a predecessor already held the lock, so the requester links itself with it. Thus, the lock word always points to the latest lock requester. The requester then spins on its own `granted` field until it becomes true (line 7). To release the lock, the lock holder first determines if it has a valid successor, by reading its own `next` field at line 2 of Algorithm 1(right). If `next` is `NULL`, it changes the lock word back to `NULL` and return directly if the `CAS` succeeded (i.e., there is indeed no successor). In case the successor is yet to link itself with the lock holder by executing line 6 in Algorithm 1(left) (but has executed line 4), we wait for `next` to become non-`NULL` (line 6). Finally, the successor is granted to lock by toggling its `granted` field (line 9).

The queue-based design is the key to robustness and fairness. We describe how OptiQL leverages it for optimistic locking later.

3 DESIRABLE PROPERTIES AND TRADEOFFS

We summarize five desirable properties for OLTP index locks:

- **D1. High-Performance.** Locks are blocking in nature, so it should be lightweight, especially when contention is rare.
- **D2. Robust.** Under contention, it is expected that the overall throughput may drop, but should plateau without collapsing.
- **D3. Fair.** An OLTP system needs to serve many concurrent requests with SLAs. It is then important for indexes to ensure all the clients have a fair chance to access data.
- **D4. Compact.** OLTP engines often use fine-grained locking, with numerous locks in indexes and other components. It is then desirable for the lock to be compact, i.e., occupy at most an 8-byte machine word, which is also the unit of atomic operations [22].
- **D5. Amenable to Index Locking.** As Section 2.2 shows, existing optimistic locks can work almost seamlessly with lock coupling. A new lock should maintain this property to be useful in practice.

An ideal lock would achieve all these properties. In practice, however, this is difficult (if not impossible) because the properties can mandate tradeoffs with each other. Specifically, robustness (D2) and fairness (D3) can be at odd with high performance (D1): both D2 and D3 require maintaining ordering among *all* requesters. This needs writing to shared memory even when conflicts are completely absent (i.e., read-only), leading to poor read performance [24, 43]. So optimistic locks (queue-based or not) fundamentally trade fairness (between readers and writers) for performance. Nevertheless, OptiQL ensures fairness among writers and retains the remaining properties, whereas existing centralized optimistic locks also forgo robustness and fairness among writers, due to their centralized nature and that `CAS` does not enforce ordering [42].

4 OPTIQL: OPTIMISTIC QUEUING LOCK

In this section, we describe the detailed design of OptiQL, a queue-based optimistic lock. OptiQL is a compact lock (D4) that combines the best of both queue-based locks (such as MCS) and optimistic locks. It consists of three key designs: (1) forming a FIFO queue of writer requesters, (2) allowing optimistic read when the lock is not granted to any writer, and (3) opportunistically allowing readers during lock handover between two writers (if any). The first design enables local spinning for robustness (D2) and fairness (among writers, D3), while the remaining designs enable lock-free reads and tackle the challenge that a queue-based lock handover may starve (optimistic) readers to improve fairness (D3). Overall, these techniques allow high performance for both readers and writers (D1). To keep the implementation simple, we base OptiQL on the MCS lock, requiring only surgical changes to it with < 10 LoC.

Next, we describe OptiQL's interfaces and structures. We then give the detailed lock protocols of OptiQL in Section 5 and show how OptiQL can work with optimistic lock coupling (D5) in Section 6.

4.1 Interfaces

OptiQL provides exactly the same interfaces for readers as centralized optimistic locks (in C++ syntax). Existing OLC protocols can directly use them:

- **`bool acquire_sh(OptiQL *lock, uint64_t &v)`**: "Acquire" the lock in optimistic read mode. The caller provides an integer `v` to store the current lock version used for validation later. Returns true if the caller can proceed assuming the protected data is consistent.
- **`bool release_sh(OptiQL *lock, uint64_t v)`**: "Release" the lock in optimistic read mode. Returns true if the validation using the given version `v` succeeded; returns false otherwise.

By forming a queue of writer requesters, OptiQL requires slightly different interfaces for writers:

- **`void acquire_ex(OptiQL *lock, QNode *qnode)`**: Acquire the lock in writer mode. The function is blocking and lock is granted after it returned.
- **`void release_ex(OptiQL *lock, QNode *qnode)`**: Release the lock in writer mode. Similarly, the function is blocking and lock is released after it returned.

Compared to centralized optimistic locks, the `qnode` parameter in `acquire_ex`/`release_ex` is additional. The caller also must maintain the queue node structure by itself. We discuss later how index locking protocols can be adapted to these new lock interfaces.

4.2 OptiQL Lock Structures

OptiQL embeds several pieces of important information for functionality and correctness in its compact 8-byte lock word. From a high-level, we need to support queuing and allow optimistic readers to perform validation steps, requiring the lock word to carry a version number and serve access to the queue of writers. As Figure 3(a)

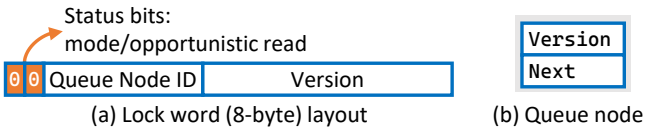


Figure 3: OptiQL structures. (a) The lock is still 8-byte but carries more information. (b) A queue node (for writers only) includes a version number and a pointer to the next writer.

shows, the lock word thus consists of (1) two bits for lock status, (2) a number of bits to record the latest writer requester (represented by its queue node ID), and (3) the remaining bits to record a version number for optimistic reads. The status bits includes a "locked" bit that indicates whether currently the lock is granted to a writer, and another bit that indicates whether opportunistic read (described later) is enabled. The numbers of bits to record the latest writer requester and version number are adjustable, depending on the total number of writer requesters the system intends to support.

Compared to the MCS lock (Section 2.3), instead of directly storing the address of the latest requester's queue node on the lock word, OptiQL stores a globally-indexed ID of the queue node. Since memory-optimized systems usually do not oversubscribe the system with more threads than hardware contexts and each thread does not need to hold many locks during index operations (details in Section 6), the total number of queue nodes in the entire system is not high. This means queue node IDs can be much shorter than a virtual memory address (up to 57 bits on modern 64-bit x86 architectures [22]), which allows us to (1) track both the latest writer *and* store the version number on the lock word at the same time (which as Section 5 describes is critical for correctness), (2) ensure the version number can be big enough to avoid early wrap-around, while (3) still maintaining a compact 8-byte lock design to ease adoption by systems that already use 8-byte locks. Our current implementation uses 10 bits for 1024 queue node IDs, leaving 52 bits for the version number. Similar techniques have been used by previous memory-optimized OLTP engines to use multiple processes [24]; OptiQL adopts it for maintaining the lock's compactness. The trade-off is that accessing to queue nodes will require address translation from queue node IDs to virtual memory pointers at runtime (described later), which however is a fixed and negligible amount of overhead as shown by our and past evaluations [24].

To facilitate queuing and local spinning, OptiQL models after the MCS lock to require each writer to provide a queue node; readers do not need to do so. As Figure 3(b) shows, an OptiQL queue node consists of (1) a `next` pointer to the writer successor, and (2) a version number. Compared to the MCS queue node, the only difference is an OptiQL queue node carries a version number, instead of a `granted` boolean. As we describe next, this allows us to easily maintain the version number that will be written back to the lock word to enable reader validation and optimistic reads in practice.

5 OPTIQL PROTOCOLS

With the aforementioned lock and queue node structures, we only need very simple changes to adapt the MCS lock protocols to support the desirable features in OptiQL.

Algorithm 2 OptiQL protocols for readers, which work in the same way as in traditional optimistic locks without queue nodes.

```
1 def acquire_sh(lock, &v):
    v = lock
3   return (v & STATUS_MASK) != LOCKED

5 def release_sh(lock, v):
    return lock == v
```

5.1 Optimistic Readers

The locking protocols for readers show a salient feature of OptiQL: they work as lightweight as existing centralized optimistic locks, with the same interfaces and semantics, without using any queue nodes or requiring address translation. In Algorithm 2, the reader simply reads the lock word to check whether the lock is already granted to a writer and opportunistic read is disallowed, by extracting the two status bits. A reader is allowed to proceed in two case: (1) the lock is free without any writers, i.e., the `locked` bit is unset in the lock word; (2) the lock is granted (or about to be granted) to a writer but opportunistic read is enabled, i.e., both status bits are set. Otherwise, `acquire_sh` returns false, which will cause the caller to retry. The `LOCKED` macro is defined exactly the same as that in the centralized optimistic lock in Figure 2(b), and the amount of work done—read the lock word, apply a mask and compare—is exactly the same. Similarly, validation (line 6) works the same as `release_sh` in Figure 2(b). This greatly simplifies the adaptation of existing optimistic lock coupling protocols to use OptiQL (Section 6).

5.2 Exclusive Writers

For writers, OptiQL works similarly to the MCS lock, but with additional care for optimistic readers.

Acquire. Algorithm 3 (lines 1–11) shows how a writer acquires the OptiQL lock in exclusive mode. Same as the MCS lock, with a newly initialized queue node whose `next` is set to null and `version` is set to a sentinel value `INVALID`, the writer first issues an atomic `XCHG` instruction to record itself on the lock word with the mode bit on and opportunistic read bit off. Following our encoding scheme in Section 4.2, the `to_ptr` function translates a queue node pointer to its ID; note that only writers need address translation which we discuss in Section 6.3. Figures 4(a–d) show this process with two requesters $T1$ and $T2$. `XCHG` returns the previous value that was stored on the lock word. Thus, the writer can be granted the lock if the returned snapshot of the lock word indicates the lock was not held (`locked` bit unset) at line 3 of Algorithm 3. In addition—and different from the MCS lock—the writer increments the returned version number and store the value in its queue node's `version` field (line 4). Figure 4(b) shows the status of the lock after the first requester, $T1$, has acquired the lock which was free in Figure 4(a). Note that here the queue node's `next` field still stores the full pointer to the successor's queue node; no address translation is required.

If the `XCHG` returned a lock word value with the `locked` bit set, then another writer requester has acted faster to acquire the lock. The current requester (e.g., $T2$ in Figure 4) then must (1) link itself with the predecessor (line 7), and (2) spin locally to wait for the

Algorithm 3 OptiQL protocols for writers. Changes on top of the original MCS lock are shaded.

```
1  def acquire_ex(lock, qnode):
2    pred = XCHG(lock, LOCKED | to_id(qnode))
     if !pred.locked: # Lock is free
4      qnode->version = pred.version + 1
       return # Lock acquired
6    else:
       to_ptr(pred.qnode_id)->next = qnode
8      while qnode->version == INVALID:
         continue # Local spinning
10   # Lock granted, turn off opportunistic read
     FETCH_AND(lock, ~(OPREAD | VERSION_MASK))
12
13 def release_ex(lock, qnode):
14   if qnode->next == NULL &&
       CAS(lock, LOCKED | to_id(qnode), qnode->version):
16     return # Lock released - indeed no successor
     # Enable opportunistic read
18   FETCH_OR(lock, OPREAD | qnode->version)
     # Ensure the successor links after [qnode]
20   while qnode->next == NULL:
       continue
22   # Grant successor the lock by passing version number
     qnode->next->version = qnode->version + 1
```

predecessor (denoted as `pred.address`) to pass the lock (lines 8–9). Note that the local spinning is done by continuously checking the queue node's `version` field which is initialized to a sentinel value `INVALID`, and will be modified when the predecessor writer releases the lock. These steps are exemplified by Figure 4(c). Finally, line 11 disables opportunistic read as the last step, which we explain later.

Release. To release the lock, the writer checks if it has a known writer successor, similar to the original MCS protocol. This is done by reading its own queue node's `next` field (line 14). If the result is `NULL`, we check whether there is indeed no writer successor by issuing a `CAS` on the lock (line 15) to change it to the new version number and expecting the lock word still records the lock holder's queue node. If the `CAS` succeeds, then there is indeed no successor and the writer simply returns, finishing the release protocol. Note that the version number is taken from the queue node's `version` field, which was already incremented (lines 4 and 23) to reflect the new version. For example, in Figure 4(h), the lock word carries version 102 since two writers have been granted and released the lock, which was initially at version 100 in Figure 4(a). This is similar to the `release_ex` function in centralized optimistic locks shown in Figure 2(b) where the lock holder increments the lock value by one. In OptiQL, since the lock word can be modified by a concurrent `XCHG` (which proceeds unconditionally, unlike `CAS`), we cannot rely on the lock word itself to calculate the next version number. Instead, OptiQL extends the queue node to maintain version information.

If there is a writer successor (ignoring line 18 for now), we pass the lock to it, e.g., *T2* in Figures 4(c) and 4(f), by incrementing and atomically storing the version number in the successor's queue node (line 23). This will cause the successor to break out of its local spin loop at lines 8–9. The successor now holds the lock, and since the version number was incremented again (line 23), the successor is ready with right version number for executing its own release protocol after executing the critical section.

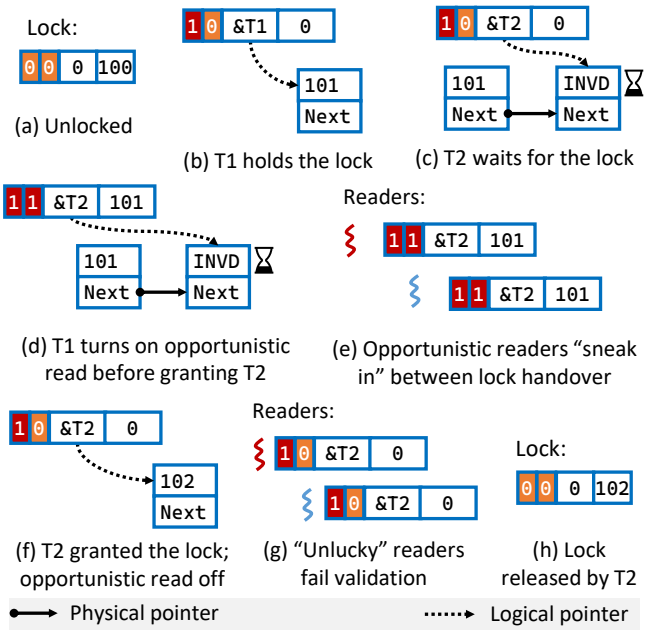


Figure 4: Coordination of concurrent requesters. (a–c) Exclusive writers proceed like using normal MCS locks. (d) Between lock handover, the current lock holder (T1) turns on opportunistic read to admit readers (e). (f) Opportunistic read is turned off once the next exclusive requester (T2) is granted the lock, invalidating readers that did not finish before T2 is granted the lock (g).

5.3 Opportunistic Read

The queue-based design avoids performance collapse under contention caused by writers. It eliminates the need to access the lock word during lock handover as the current lock holder directly passes the lock to the successor. Thus, during lock handover the lock word is always in a locked state. As a result, no reader can be admitted during lock handover. This is in stark contrast to centralized optimistic locks: when a writer releases the lock, it resets the lock word with a new version number. Before the next writer successfully acquires the lock, readers could access the data during lock handover time. We observe this is the key reason why optimistic locks still admits readers even under contention, exhibiting a certain level of fairness even between readers and writers. Yet the queue-based design eliminates such possibilities by directly passing the lock along the queue without changing lock status. As we show in Section 7, this allows almost no readers as long as there are any writers.

OptiQL introduces opportunistic read to solve this problem. On top of optimistic read, opportunistic read takes advantage of a consistency window during lock handover: after a writer finishes its critical section, until it finishes executing the release protocol, the data protected by the lock is in fact in a consistent state and can

be used correctly by a concurrent reader. The opportunistic read bit introduced earlier serves exactly the purpose to indicate this state. In Algorithm 3, after the current holder determines that there is indeed a successor, we toggle the opportunistic read bit *and* set the lock word's version number to the version number maintained by the releasing writer, using an `atomic-fetch-or` [22] instruction in one atomic step. This way, as shown in Figure 4(d), the lock word will then (1) have both status bits set, (2) record the latest requester's queue node, and (3) carry a version number. The queue node is retained on the lock word to allow subsequent writer requesters to queue up. The lock word now captures the consistent state during lock handover. The current lock holder continues to finish executing its remaining lock release steps. Meanwhile, as shown in Figure 4(e), readers can now optimistically access the data protected by the lock using the protocols described in Section 5.1. The only difference is that since readers follow Algorithm 2, they will take the entire lock word as shown in Figure 4(e) and use it for validation later. This is correct because the "version" only needs to be unique for the validation process to work properly, which is satisfied by the writer's release protocol described above. Thus, the reader protocols in Algorithm 2 remain unchanged with opportunistic read.

After the version is set by the current writer, the successor breaks from its local-spinning at line 9. It then turns off opportunistic read (line 11) to finish lock acquisition. Alternatively, it could be left to the caller (i.e., the new lock holder) to decide exactly when to turn off opportunistic read (i.e., call line 11). We refer to this as "adjustable opportunistic read" (AOR). AOR can allow lock-index co-design to potentially admit even more readers, although it may complicate API design; we explore the use of AOR in Section 6.1 and its performance in Section 7.4. Opportunistic readers that validate after this point, e.g., those in Figure 4(g), will fail as the opportunistic bit is unset, which was set when it read the lock word.

At a first glance, it may seem redundant to put the version number on the lock word, in addition to toggling the status bit for opportunistic read. However, this is necessary for correctness, to avoid a subtle ABA problem [21]. Consider a writer W that keeps repeating its critical section that increments a zero-initialized counter c: W first acquires the lock in exclusive mode, increments c to 1, and starts to execute `release_ex`. If W only toggles the opportunistic read bit but does not put the version number on the lock word, a concurrent reader R can now read c with value 1 and continue for other operations. Then W finishes its execution and starts another round, incrementing c to 2 and again starts to release the lock. Now suppose R finishes executing its critical section and starts validation: it will use the old snapshot of the lock word which only carries W's queue node and status bits values, leading R to wrongly pass the validation while the counter has been silently modified by W to a different value (2). With the version number, however, R would be able to differentiate the two rounds of critical section executions and correctly fail the validation. Therefore, it is necessary for the lock word to record both the latest writer requester and version number for opportunistic read to work properly.

5.4 Discussions

OptiQL inherits some properties of the MCS lock and combines the benefits of optimistic locks, as a result of several important design tradeoffs we consciously made. First, compared to TTS and centralized optimistic locks (Section 2.2), OptiQL's writer release path (so is MCS's) can be more expensive by mandating a `CAS` instruction when contention is rare. Second, to enable optimistic read in practice, OptiQL's opportunistic read adds two atomic (although cheap) operations on the lock word. Our evaluation in Section 7 shows that this can be non-negligible in certain microbenchmarks that stress the lock itself (e.g., the high contention cases in Figures 6–7). However, we find that the overheads are negligible in index workloads (e.g., Figure 9), making it a worthwhile tradeoff.

OptiQL allows readers to continue using standard interfaces, but requires non-standard interfaces for writers that expect a queue node parameter. Since OptiQL's lock word records the latest requester's queue node ID, instead of its address, address translation is needed as we mentioned earlier. Combined, these two issues require the application manage its queue nodes. Prior work [24] has explored practical solutions to this problem in database engines, and we observe that index operations (and database workloads in general) exhibit a sweetspot that requires only moderate and easy-to-maintain changes, which we describe in the next section.

6 INDEX LOCKING WITH OPTIQL

Now we demonstrate how two representative indexes—memory-optimized B+-trees [29] and the adaptive radix tree (ART) [28]—can use OptiQL to achieve high performance and robustness. The main challenges are (1) devising an efficient OLC protocol to work with OptiQL's queue-based design, and (2) coping with OptiQL's interfaces that use queue nodes. We begin with adapting OLC described in Section 2.2 for B+-trees and ART, and then describe a general approach to managing queue nodes.

6.1 Memory-Optimized B+-Trees

Memory-Optimized B+-trees usually opt for smaller nodes (e.g., 512 bytes instead of 4KB) for better cache efficiency [48]. This emphasizes the need for compact locks that are 8-byte or less. OptiQL maintains this property by keeping the lock word 8-byte, and shared accesses proceed in the same way as centralized optimistic locks.

For insert and update operations, a straightforward approach to adopting OptiQL in a B+-tree that already uses traditional optimistic locks is then to change the lock type in each node (inner and leaf) to OptiQL, and provide a queue node whenever the thread wants to lock a node in exclusive mode. Although easy to implement, this approach does not work well due to OptiQL's queue-based nature and the way the original OLC protocol assumes validation is performed. We use an example of updating a record to explain below. In the original OLC protocol with centralized optimistic locks, a thread attempts to update a node in four steps: (1) traverse to the target leaf node L using optimistic read; (2) read and record locally the version of the lock protecting L, and validate the parent has not change (if so, restart the operation); (3) search L for the target key (return if not found); (4) "upgrade" the lock to become a writer lock and perform the update, after which the thread unlocks L and returns. Importantly, in step 4 the upgrade operation issues a `CAS` on the lock word, expecting the lock word to still contain the version obtained in step 3. Among these, step 4 is specific to

CAS-based centralized locks described in Section 2.2. With a queue-based lock like OptiQL, a direct adaptation would be following steps 1–3, and acquire the OptiQL lock in step 4 in exclusive mode.

Compared to centralized optimistic locks where if the CAS failed the caller would retry (usually from the root of the tree), under OptiQL the thread would line up after the predecessor and wait for the lock to be granted. However, this approach is not efficient: after the lock is granted, the thread must search the node again (i.e., repeating step 3) to ensure it has the latest, correct information regarding the target key. Therefore, instead of blindly following the original OLC protocol, we revise it slightly to directly acquire the lock at the leaf level, instead of going through the upgrade step. Under our adapted OLC protocol, an updater thread still proceeds as follows in Algorithm 4: (1) traverse to the target leaf node L using optimistic read (same as original OLC, lines 2–15 of Algorithm 4); (2) directly lock L (line 16–19); (3) validate the parent node has not changed (line note this was step 2 in the original protocol) and if so, release the acquired lock in step 3 and retry (lines 22–23); (4) search L for the target key, perform the update, release the lock and return (lines 26–27). This allows us to truly benefit from the queue-based design in OLC protocols for B+-trees. Moreover, with AOR (Section 5.3), step (2) can be broken into two steps where the first step returns without calling line 11 of Algorithm 3, allowing more readers to "sneak in" until the writer conducts the update in step (4) after searching L. As we show in Section 7.4, AOR could further improve performance by up to ~30%.

So far we have assumed all the locks in a B+-trees are changed to OptiQL. Since inserts and deletes may incur SMOs that propagate to the root of the tree, the number of queue nodes needed by a thread will scale with tree depth. Although in practice most B+-trees are shallow, it is still beneficial to reduce the number of queue nodes that have to be maintained because OptiQL limits the maximum number of queue nodes by encoding their IDs in the lock word. Further, we note that (1) inner nodes experience less contention since they are updated less frequently than leaf nodes, and (2) releasing a queue-based lock (including OptiQL) can be more expensive since a CAS is involved if there is no contention. Based on these observations, instead of using OptiQL for all the nodes, we only change the leaf nodes to use OptiQL; inner nodes still use traditional optimistic locks. This effectively bounds the number of queue nodes to maintain for each thread to two (in case of node merges) and keeps the locking overhead low in inner nodes.

6.2 Adaptive Radix Tree (ART)

ART [27] is a variant of the trie structure that allows adaptive node sizes to improve utilization and performance in memory-optimized environments, on top of techniques such as path compression and lazy expansion. With centralized optimistic locks, ART is also vulnerable to performance collapse under high contention, as Section 7 shows. Similar to B+-trees, each ART node carries a lock that can be directly changed to OptiQL. Readers follow the strategy similar to that of B+-trees under both OptiQL and centralized optimistic locks. However, for modification operations such as inserts and updates, the techniques we used for B+-trees to use different locks at different levels is infeasible due to its loosely defined node type: a node in ART (and tries in general) can simultaneously serve the role of an inner and leaf node, and the type (inner or leaf) of a node will not become clear until we already access it. The former requires us to use OptiQL for all the nodes. However, this will force SMO threads to also acquire the lock in OptiQL's queue-based manner for inner nodes, which as we mentioned in the B+-tree case can be inefficient. To still allow cheap optimistic style lock acquisition, we added an upgrade interface for OptiQL that directly uses CAS to acquire the lock by comparing the current version against a provided one. The method allows readers to promote themselves to writers if there is a version match just the same as traditional optimistic locks do. Similar to MCS try-lock but different from traditional optimistic locks, however, we set the lock word to still carry a queue node such that subsequent writers whose target keys end in the node can still leverage the queue-based design to avoid performance collapse. The remaining issue then is to detect node type accurately. We discuss two solutions below, depending on whether lazy expansion and path compression are involved. We assume payloads are stored indirectly through a "TID" as previous work suggests [27].

Without lazy expansion or path compression, the thread can reliably tell if a node is at the leaf node level by checking the length of the target key. Since ART uses one byte per level, the number of levels would match the number of bytes in a key. Then, for a key of l bytes, we can take optimistic read locks for the first $l-1$ levels and only take the lock in exclusive mode directly at the last level.

Algorithm 4 B+-tree update with OLC and OptiQL.

```
1  def btree_update(root, key, new_val):
   restart:
3    node = root
     v = INVALID
5    qnode = get_qnode()
     if !acquire_sh(node.lock, v):
7      goto restart

9    # Drill down to the leaf level
     while !is_leaf(node):
11     child = find_child(node, key)
       nv = INVALID
13     if node.lock.version != v:
         goto restart
15
       if is_leaf(child):
17       acquire_ex(child.lock, qnode) # Directly lock leaf
       elif !acquire_sh(child.lock, nv):
19       goto restart

21     if !release_sh(node.lock, v):
         if child is leaf: # Release lock before retry
23         release_ex(child.lock, qnode)
         goto restart
25     node = child
       v = nv
27
     update_leaf(node, key, new_val)
29   release_ex(node.lock, qnode)
```

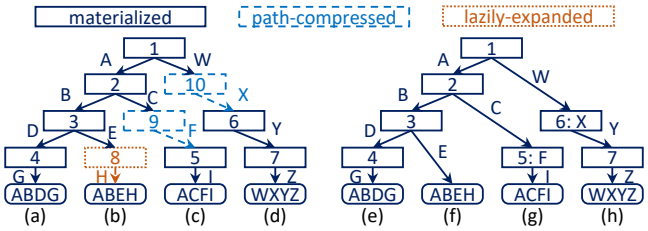


Figure 5: The logical (a–d) and physical (e–h) structures of ART with path compression and lazy expansion. (a, e) Path to ABDG is fully materialized. (b, f) Node 3 points to ABEH due to lazy expansion. (c, g) Path between nodes 2 and 5 is compressed, with prefix F in node 5. (d, h) Path between nodes 1 and 6 is compressed, with prefix X in node 6. OptiQL materializes nodes 8 and 9 under contention, so that locks on nodes 8 and 5 can be taken directly for updating ABEH and ACFI. Node 10 is safe to skip as node 7 has no prefix.

With path compression and lazy expansion, there might not be such "last-level" node at all since a higher-level node can point to the payload directly, or the last-level node is an endpoint of a compressed path and thus carries a key prefix. In both cases, an updater will only tell whether it should lock the node in exclusive mode after reading the node, yet directly acquiring the lock in exclusive mode would defeat the purpose of employing optimistic locks. Figure 5 shows several desired and pathological structures in a path-compressed, lazily expanded ART. Exclusive locks can be directly taken on nodes 4 and 7 but not on node 3 or 5. To solve this problem, we propose contention expansion, inspired by contention split in B+-trees [1]. In each node, we employ a counter to record its contention level measured by the number of exclusive lock acquisitions as a result of upgrade instead of direct, blocking acquisition. A large counter value indicates that the node is probably a leaf node that is lazily expanded or an endpoint of a compressed path. The counter is probabilistically incremented to reduce overhead. Then, to ease future accesses, if the contention level passes a threshold, we directly expand the node by materializing the last two levels. Subsequent traversals would then be able to land on the last-level node and acquire the lock directly in the exclusive mode. Our current implementation uses a sampling probability of 0.1 to increment to counter and a fixed contention threshold of 1024, although ideally they should be adapted for different workloads.

6.3 Queue Node Management

Normally, threads can allocate queue nodes directly on the stack or as a thread-local (and socket-local) variable. However, OptiQL stores queue node IDs (instead of addresses) on the lock word. This requires the application/index maintain an address translation mechanism (`to_ptr` and `to_id` in Algorithm 3) which needs to be globally accessible. A naive approach is to track dynamically allocated queue nodes in a hash table, but this can bring high overhead and defeat the purpose of using OptiQL; modern memory-optimized OLTP engines [10, 23, 24, 32, 43] avoid using such hash tables and lock managers for the same reason. Our solution follows prior work [24] to pre-allocate all the required queue nodes in a contiguous array, and queue nodes from the array are then allocated and recycled after use at runtime.[3] Since the memory is presented to the application as a contiguous array, we can directly take array indexes as queue node IDs, without dedicating bits in queue node IDs to encode NUMA node information. In essence, our approach sets up an indirection between queue node IDs (which occupy fewer bits) and the queue nodes' 64-bit virtual addresses.

7 EVALUATION

We empirically evaluate OptiQL using both microbenchmarks and index benchmarks. Through experiments, we show that:

- OptiQL matches the performance of centralized optimistic locks under low contention and read-dominant workloads.
- OptiQL achieves robust performance under various contention levels without collapsing under write-intensive workloads.
- Opportunistic read enables OptiQL to provide opportunistic read under mixed read/write workloads.

7.1 Experimental Setup

We performed experiments on a dual-socket server equipped with two 20-core (40-hyperthread) Intel Xeon Gold 6242R CPUs clocked at 3.1GHz; each CPU has 35.75MB of cache. In total the server has 40 cores (80 hyperthreads). The server is populated with 384GB (32GB×12) of DRAM occupying all the 12 channels across the two sockets. We run Arch Linux with kernel version 5.16. All the locks and indexes are implemented in C++17 and compiled with GCC 12 with the highest optimization level. We use `jemalloc` to reduce dynamic memory allocation overhead at runtime. Threads are pinned to hardware hyperthreads to avoid migrations by the OS scheduler.

Benchmarks. Following prior work [30, 31, 48], we use both microbenchmarks and index workloads to stress test the locks and measure their performance under more realistic scenarios. We devised a microbenchmark framework that can be plugged with various lock implementations. Each thread keeps issuing lock acquire/release requests on a set of pre-allocated locks following a uniform random distribution. Contention level is controlled by adjusting the number of locks to be one (extreme contention), five (high contention), 30000 (medium contention) and one million (low contention). Inside the critical section, the thread increments a `volatile` variable on the stack for 50 times.

To conduct index experiments, we use PiBench [30], a unified benchmarking framework for persistent and volatile indexes to issue the same workloads to both ART and B+-tree variants using the tested locks. For B+-trees we follow prior work to use 256-byte nodes [48]. Each index is pre-loaded with 100 million records of 8-byte keys and 8-byte values. We then issue index workloads, such as lookup, update, insert and different mixes of them.

We collect the throughput numbers (lock acquires per second for microbenchmarks, and the number of index operations per second for index benchmarks) and report the averages of 20 10-second runs with error margins of 95% confidence interval, as represented by the blurred regions surrounding the lines. We also report tail latency

[3] Our implementation uses interleaved allocation (`numa_alloc_interleave` in `libnuma`) for memory pages of queue nodes. For example, in a dual-socket system, if we request four 4KB pages, then pages 0/2 will come from from NUMA node 0; pages 1/3 will be from node 1. Our experiments show that this can slightly improve performance by up to ~2% by reducing cache coherence traffic during local spinning as the queue nodes are almost always cached.

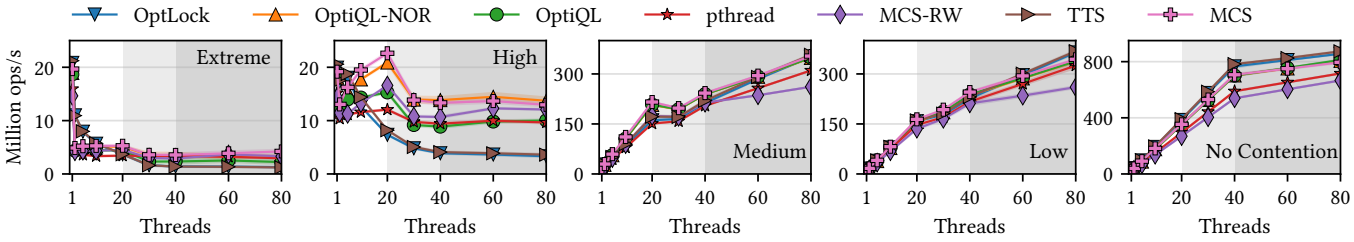


Figure 6: Exclusive lock throughput under different contention levels; `MCS` and `TTS` are shown as a reference to evaluate OptiQL's writer performance as they do not support readers. Queue-based `OptiQL`, `OptiQL-NOR` and `MCS-RW` can avoid performance collapse under extreme and high contention. `pthread` and `MCS-RW` add more overhead under low contention; `pthread` also uses a much bigger (56-byte) lock word.

numbers for index workloads to explore the effect of queuing in OptiQL.

Lock Variants. We implemented and test the following locks:

- **`OptLock`**: Centralized optimistic lock [28, 29].
- **`OptiQL`**: OptiQL proposed in Section 4.
- **`OptiQL-NOR`**: Same as `OptiQL` but without opportunistic read.
- **`pthread`**: The reader-writer lock (`pthread_rwlock_t`) in pthread (`std::shared_mutex` since C++17 [8]). Under contention it expands to form a queue-based structure.
- **`MCS-RW`**: A fair reader-writer variant of the MCS lock [39].
- **`TTS`**: Classic TTS lock as described in Section 2.2.
- **`MCS`**: The MCS lock as described in Section 2.3.

Note that among the above locks, `TTS` and `MCS` do not support shared mode; they are included as a reference to show the cost of supporting readers. `pthread` requires 56 bytes for the lock word. The original design of `MCS-RW` requires more than 16 bytes to store the lock word. However, we applied the same encoding approach in Section 6.3 to place the needed information in one 8-byte lock word, same as `OptiQL` and `OptiQL-NOR`. Both `pthread` and `MCS-RW` are pessimistic locks, so we use pessimistic lock coupling for index workloads. `OptLock`, `OptiQL` and `OptiQL-NOR` use OLC.

Next, we start with microbenchmark results to calibrate our expectations, and then report index benchmark results.

7.2 Microbenchmark Results

Our first experiment evaluates how each lock performs under different levels of contention under a varying number of threads.

Exclusive Writer Performance. We begin with a pure-write microbenchmark where each thread keeps acquiring and releasing the lock in exclusive mode. Figure 6 summarizes the results; the lightly shaded areas indicate the use of the second NUMA node and the remaining darker shaded area depict where hyperthreads are used. Under extreme contention, all the threads contend for a single lock, so the workload is inherently unscalable and the throughput for all locks is expected to drop. However, queue-based variants (`OptiQL`, `OptiQL-NOR`, `MCS-RW`, `pthread` and `MCS`) can maintain performance without completely collapsing under contention.

Under high contention, threads still have a very high chance to conflict with each other. `OptiQL-NOR`, `MCS`, `MCS-RW` and `pthread` handle contention well. OptiQL shows a similar trend but lower absolute performance than `OptiQL-NOR` with a constant amount of overhead because it uses additional steps to support opportunistic read. However, as we show later, in realistic scenarios such drop is negligible, and more importantly, it enables optimistic readers to proceed, while `OptiQL-NOR` can starve readers. Finally, note that OptiQL variants and `MCS` still outperforms `OptLock` and `TTS` by ~2×, although both exhibit low absolute performance. We therefore still recommend using a fair lock instead of relying on backoff simply attain overall high performance at the risk of starving certain requesters. Under medium, low and no contention cases, all locks scale similarly, with slightly lower scalability beyond one socket.

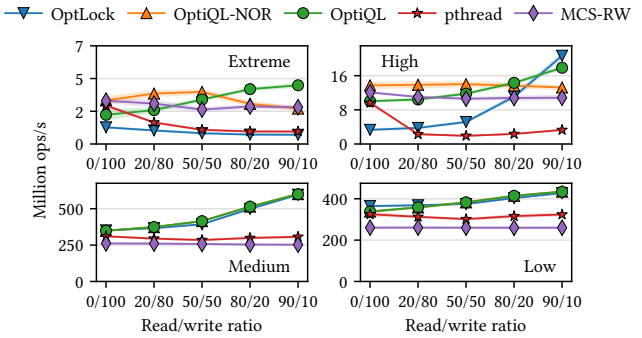


Figure 7: Lock throughput under varying contention and read-/write ratios (80 threads). OptiQL keeps the advantage of `OptLock` for medium-low contention/read-dominant workloads and avoids collapsing under high contention/high write ratios.

Mixed Operations. Figure 7 shows the throughput of five locks under 80 threads. We excluded read-only results as all the locks scale and perform identically; `TTS` and `MCS` were also excluded because they do not support readers. In the figure, under extreme contention, `OptLock` remains low performance due to high contention on the central lock word. `pthread` shows a worse trend as it is also pessimistic, whereas `MCS-RW` is able to maintain the performance thanks to its queue-based structure. `OptiQL-NOR` and `OptiQL` are able to maintain high overall performance across different read/write ratios. Also, `OptiQL-NOR` often performs better than `OptiQL` until the workload becomes read-heavy with over 50% of reads, again due to the additional atomic instructions issued for opportunistic reads. However, we note that the `OptiQL-NOR` achieved this by starving readers. In Table 1, the successful rates of read operations of `OptiQL-NOR` are <2%, which means few readers can successfully acquire the lock under `OptiQL-NOR`, whereas `OptiQL`

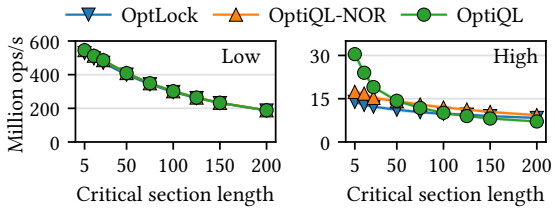


Figure 8: Lock throughput under varying critical section lengths (80 threads). Opportunistic read mainly benefits short reads.

achieves up to ~32% successful rates for reads. With more reads, the overhead of opportunistic read pays off and as more readers can successfully acquire the lock and finish their operations.

The trend is slightly different under high contention where we see most locks achieve higher throughput as read ratio increases. `OptiQL-NOR` also showed lower performance when the read ratio is increased from 80% to 90%, as threads spend most of their cycles failing and retrying to acquire the lock in optimistic read mode. `MCS-RW` showed the same trend but with lower raw performance because readers also have to issue atomic instructions, canceling out part of the benefit brought by the reader mode. Under medium and low contention, all the optimistic locks performed similarly with expected high performance for the mixed workloads. `MCS-RW` and `pthread` exhibit low performance due to their pessimistic nature.

Impact of Critical Section Length on Opportunistic Read. Since opportunistic read enables readers during exclusive lock handover, the critical section (CS) length can affect the algorithm's effectiveness, especially for readers and workloads under contention. We explore this using a read-mostly workload (80% reads and 20% writes). In the CS, the thread increments a `volatile` variable on the stack for a tunable number of times. As shown in Figure 8(left) under low contention opportunistic read is not sensitive to CS length as the logic to enable/disable opportunistic read is less executed. Under contention, in Figure 8(right), opportunistic read mainly benefits short reads (CS length $\leq$ 50). Note that we do not use AOR to give extra time for readers; we explore AOR's effect in Section 7.4.

Summary. Through microbenchmarks, we confirm that (1) OptiQL can avoid performance collapse under contention, and (2) OptiQL's opportunistic read can effectively enable readers. Next, we evaluate these locks in real indexes.

7.3 End-to-End Index Scalability

Following Section 6 we adapt a memory-optimized B+-tree and ART to use OptiQL. In the rest of this section, we use the following workloads for index experiments: (1) Read-only: 100% lookups; (2) Read-heavy: 80% lookups and 20% updates; (3) Balanced: 50% lookups and 50% updates; (4) Write-heavy: 20% lookups and 80% updates; (5) Update-only: 100% updates.

Performance under Contention. Since a major goal of OptiQL is to avoid performance collapse under contention, we first run the above workloads under a self-similar [17] distribution with a skew factor of 0.2 (i.e., 80% accesses target 20% keys). In addition, we make the key space dense such that to increase the stress on the lock. For example, following this distribution, the first 256 keys would accept 16% of the total accesses. Figure 9 shows the throughput of B+-tree and ART under this setup. Both indexes and all locks perform well under read-only workloads thanks to optimistic reads.

Table 1: Reader success rate of `OptiQL-NOR` and `OptiQL` under varying read/write ratios and high contention (80 threads).

Lock	20%/80%	50%/50%	80%/20%	90%/10%
`OptiQL-NOR`	1.67%	1.18%	0.51%	1.17%
`OptiQL`	32.19%	29.54%	27.34%	26.57%

As Section 7.2 describes, since `OptiQL` requires two additional atomic instructions on the lock word per lock handover to support opportunistic read, `OptiQL` often performs worse than `OptiQL-NOR`. This overhead is negligible even under the update-only workload where the two atomics are pure overhead (although slightly more observable in ART where `OptiQL-NOR` marginally outperforms `OptiQL`). However, as long as read operations are present (which is the case for the remaining workloads in Figure 9), `OptiQL` yields more robust and higher performance through opportunistic read than `OptiQL-NOR` in non-trivial index benchmarks. We also observe that ART scales worse than B+-tree when the workload is not pure read-only. We attribute the reason to the fact that the B+-tree implementation uses small, 256-byte nodes. This leads to a fanout of 14, effectively enabling finer-grained locking compared to ART which can have nodes containing up to 256 children/payloads. Overall, under write-heavy and balanced workloads, `OptiQL` can be over 2× faster than `OptiQL-NOR`, showing the effectiveness of OptiQL's opportunistic read design. Note that this is achieved without further trading off fairness (as optimistic locks already bias toward writers); we discuss fairness and tail latency later. As expected, `MCS-RW` and `pthread` are unable to scale. When reads dominate, their pessimistic nature require readers issue additional atomics, canceling out the benefits of having parallel readers. With more writes, although `MCS-RW` uses the similar queue-based design to OptiQL's, unlike `OptiQL`, it cannot use OLC which allowed the latter to scale and retain high performance under contention.

We also tested workloads that involve inserts and deletes, and observed the same performance characteristics for OptiQL, so for space limit we omit the details here.

Low Contention Performance. Now we evaluate OptiQL's performance under low contention, under which traditional optimistic locks perform well. We run the same experiment for both indexes but change the benchmark to follow a uniform random distribution. Figure 10 shows the results. As expected, both B+-tree and ART scale well and the difference between lock variants is minuscule. We observed similar trends under other workload mixes, so we conclude that OptiQL provides performance similar to centralized optimistic locks when contention is low.

7.4 Effect of (Adjustable) Opportunistic Read

Now we evaluate the effect of opportunistic read and AOR using the skewed workload (skew factor 0.2) under varying B+-tree node sizes. This also allows us to explore the sensitivity of opportunistic read to critical section length (i.e., node size) in index scenarios. As shown in Figure 11, without AOR, for read-heavy workloads (80% read, 20% write), using smaller nodes (e.g., 256-byte, shorter CS)

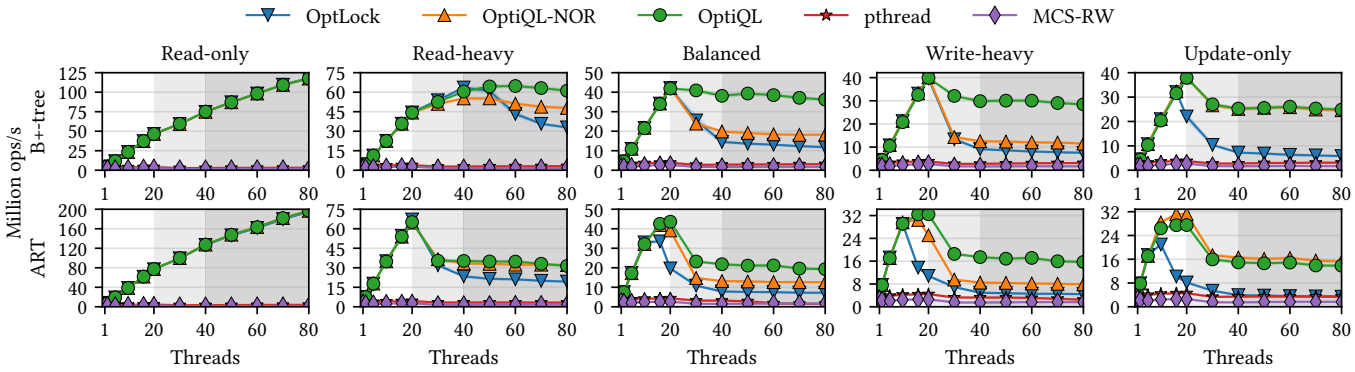


Figure 9: Throughput of B+-tree (top) and ART (bottom) under a skewed workload (self-similar distribution with a skew factor of 0.2). Pessimistic `MCS-RW` and `pthread` are unable to scale even for read-only workloads. `OptiQL` and `OptiQL-NOR` perform the same as `OptLock` for read-only workloads. With more writers (left to right), `OptLock` collapses beyond one socket, while `OptiQL` maintains high performance without collapsing. Opportunistic read is critical in enabling `OptiQL` to allow readers (second-left, middle and second-right). The additional atomics added by opportunistic read is negligible as shown by the update-only workload (right-most).

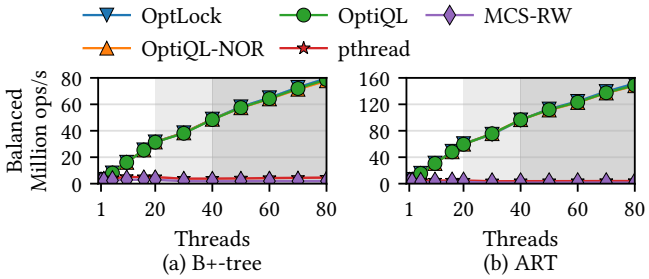


Figure 10: Index throughput under low contention and the balanced workload. Both centralized optimistic locks and OptiQL variants perform well when contention is rare.

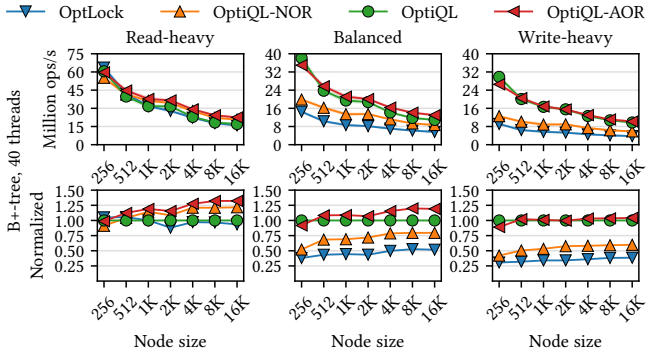


Figure 11: B+-tree throughput (top) and speedup relative to `OptiQL` (bottom) under the skewed distribution using varying node sizes (40 threads); Larger node sizes lead to longer critical sections. AOR is more necessary to make more time for readers to finish.

allows `OptiQL` to perform better than `OptiQL-NOR`. That is, opportunistic read helps allow more lookups to finish. But with larger nodes `OptiQL` becomes worse than `OptiQL-NOR`. Opportunistic read is then not beneficial, because there is not enough time for opportunistic readers to finish and in a read-heavy workload most reads are not opportunistic. This makes the atomics to enable/disable opportunistic read is mostly pure overhead. Under the balanced (50% read, 50% write) and write-heavy (20% read , 80% write) workloads, in general the gap between `OptiQL` and `OptiQL-NOR` becomes smaller (i.e., opportunistic read becomes less useful) as node size becomes bigger (longer CS). However, `OptiQL` always performed better than `OptiQL-NOR`. This is different from the read-heavy case, mainly because here with more writers, the writer-release path to enable/disable opportunistic read is much more frequently executed. Compared to the read-heavy case, many reads here are opportunistic. Although a larger CS will still make opportunistic read less beneficial, its impact is outweighed by the enabling of more opportunistic reads.

When AOR is enabled (represented by `OptiQL-AOR` in the figure), we observed up to ~30% higher throughput as the workload gets more read-intensive *and* use larger nodes. Especially, using larger nodes leads to longer critical sections, making it more necessary to use AOR to allow more time for opportunistic readers to complete. However, we note that using larger nodes also leads to lower performance and under small nodes (e.g., 256 or 512-byte) AOR does not show clear advantage on top of `OptiQL`. We therefore recommend AOR when the index needs to use larger nodes.

7.5 Tail Latency and Fairness

The choice of a locking primitive also impacts tail latency, which is related to fairness between index accessing threads. We collect up to 99.999 percentile tail latency to explore this in Figure 12 across different workloads and under 20 (one socket) and 40 threads (two sockets). Here, `OptLock` has drastically increasing tail latencies for updates, while `OptiQL-NOR` and `OptiQL` stay relatively stable across the horizontal axis. We also collected the variance which shows a similar trend (not shown here), corroborating with this result. Despite the better results than `OptLock`, we note that `OptiQL-NOR` also exhibits latency spikes under the balanced workload under 40 threads in both indexes. This is because of its reader-starving

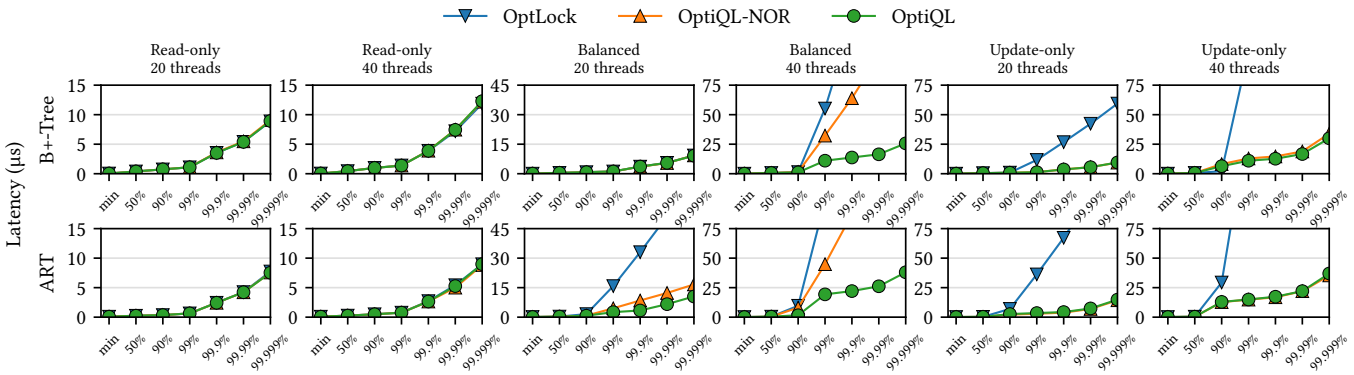


Figure 12: Latency at different percentiles at 20 and 40 threads under self-similar distribution (skew factor 0.2).

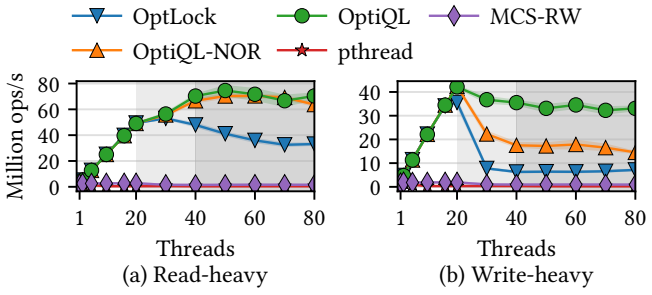


Figure 13: Throughput of ART under self-similar distribution (skew factor 0.2) with sparse integer keys.

nature without opportunistic read: most readers will have to retry many times to successfully read a record.

7.6 Impact of Key Space Sparsity

So far we have only used dense keys to stress test the indexes. This naturally causes ART to avoid lazily expanding last-level nodes which can happen in practice. We therefore conduct another experiment to populate ART with 100 million sparse integer keys, which would incur lazy expansion. As Figure 13 shows, under the self-similar distribution, indexes that use `OptLock` exhibit poor performance and collapse as we use more than one socket because they suffer from excessive retries under contention, whereas `OptiQL` and `OptiQL-NOR` can follow the contention expansion design in Section 6.2 to local-spin, avoiding performance collapse.

8 RELATED WORK

Our work is closely related to locking primitives and index designs.

Synchronization Primitives. We have discussed the basics on centralized and queue-based locks earlier, so we do not repeat here. Because of nice features such as robustness and fairness (FIFO), many proposals adapt the MCS lock, with support for pessimistic readers [39], more NUMA-friendly spinning [11] to reduce lock handover cost, and context-free standard lock interfaces [45]. OptiQL uniquely further adds optimistic read capabilities. Other queue-based locks, such as CLH [9, 35], could also be adapted with optimistic reads, which we leave for future work. In addition to queue-based locks, ticket locks [42] also enforce FIFO, They record a global "next-serving" ticket number and threads are served by the order in which they acquired their own number. However, ticket locks are still centralized and thus vulnerable to performance collapse under contention. Optimistic locks are inherently biased toward writers because readers do not announce their existence to prevent writers from clobbering the data being read. Recent work [6, 46] has combined pessimistic readers with optimistic locks. We leave it as future work to add such features in OptiQL.

Concurrent Indexes. Traditional lock coupling [4] often fails to scale, so recent indexes prefer other alternatives. Most of them detect conflicts with version numbers. OLFIT [7] combines the B-link tree idea and version-based optimistic locks. Masstree [37] uses two version counters to differentiate updates and splits to reduce unnecessary retries. Some recent persistent memory (PM) indexes [33, 34, 40, 44] also use optimistic locks to improve scalability. Several indexes further avoid locking with lock-free algorithms. The Bw-Tree [31] is a representative which is a lock-free B-tree that uses an indirection layer to allow atomic updates using single-word CAS. BzTree [3] is another lock-free B-tree designed for PM, but it leverages a multi-word CAS [47] primitive to simplify the implementation. Compared to lock-free indexes, lock-based approaches do not provide strong progress guarantee (e.g., due to OS scheduling). However, these cases are rare as memory-optimized OLTP systems typically do not oversubscribe the system. Moreover, (optimistic) lock-based approaches are usually easier to implement than lock-free ones [48], making them the preferred choice in many systems.

9 SUMMARY

Synchronization primitives play critical roles in concurrent indexes. However, existing optimistic locks in memory-optimized indexes are vulnerable to performance collapse under contention due to their centralized design inherited from centralized spin locks. To solve this problem, we propose OptiQL, a new optimistic lock that leverages a queue-based design to allow writers to line up and spin locally, instead of centrally on the lock word, to reduce unnecessary

accesses to the lock word. Readers can still proceed optimistically without writing to shared memory. With OptiQL, we adapted optimistic lock coupling to apply OptiQL in two representative indexes, ART and a B+-tree. Evaluation results show that OptiQL retains the benefits of traditional centralized optimistic locks by matching their read performance, while providing robustness without collapsing under high contention, among other desirable features such as compact lock word and fairness (FIFO ordering) among writers, which were often sidestepped or traded off by prior work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and associate editor for their constructive feedback, especially our "Reviewer 1" who suggested to make opportunistic read adjustable! This work is partially supported by an NSERC Discovery Grant, Canada Foundation for Innovation John R. Evans Leaders Fund and the B.C. Knowledge Development Fund.

REFERENCES

[1] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees.. In *Conference on Innovative Data Systems Research (CIDR 2021)*.
[2] T.E. Anderson. 1990. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (1990), 6–16.
[3] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018), 553–565.
[4] R. Bayer and M. Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Inf.* 9, 1 (mar 1977), 1–21.
[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 521–534.
[6] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and Robust Latches for Database Systems. In *Proceedings of the 16th International Workshop on Data Management on New Hardware (DaMoN '20)*. Article 2, 8 pages.
[7] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. 181–190.
[8] cppreference.com. 2022. std::shared_mutex. https://en.cppreference.com/w/cpp/thread/shared_mutex.
[9] Travis S. Craig. 1993. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. *University of Washington Technical Report 93-02-02* (1993).
[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. 1243–1254.
[11] Dave Dice and Alex Kogan. 2019. Compact NUMA-Aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Article 12, 15 pages.
[12] Dave Dice and Alex Kogan. 2020. Fissile Locks. In *Networked Systems: 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3–5, 2020, Proceedings*. 192–208.
[13] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. 247–256.
[14] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 969–984.
[15] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. 9.
[16] Goetz Graefe. 2010. A Survey of B-Tree Locking Techniques. *ACM Trans. Database Syst.* 35, 3, Article 16 (jul 2010), 26 pages.
[17] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*. 243–252.
[18] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic Concurrency with OPTIK. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Article 18, 12 pages.
[19] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends Databases* 1, 2 (feb 2007), 141–259.
[20] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.).
[21] IBM. 1983. IBM System/370 Extended Architecture, Principles of Operation. IBM Publication No. SA22-7085.
[22] Intel Corporation. 2021. Intel 64 and IA-32 Architectures Software Developer's Manual. (2021). https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
[23] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.
[24] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 691–706.
[25] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (dec 1981), 650–670.
[26] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42 (2019), 73–84.
[27] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE '13)*. 38–49.
[28] 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 '16)*. Article 3, 8 pages.
[29] Viktor Leis and Ziqi Wang. 2017. OLC B+-tree. https://github.com/wangziqi2016/index-microbench/tree/master/BTreeOLC.
[30] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *PVLDB* 13, 4 (2019), 574–587.
[31] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. 302–313.
[32] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 21–35.
[33] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *PVLDB* 15, 3 (2021), 597–610.
[34] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *PVLDB* 13, 8 (2020), 1147–1161.
[35] Peter S. Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*. 165–171.
[36] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1298–1309.
[37] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.
[38] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (feb 1991), 21–65.
[39] John M. Mellor-Crummey and Michael L. Scott. 1991. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '91)*. 106–113.
[40] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*. 371–386.
[41] Larry Rudolph and Zary Segall. 1984. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. *SIGARCH Comput. Archit. News* 12, 3 (Jan 1984), 340–347.
[42] Michael Lee Scott. 2013. *Shared-memory synchronization*. Morgan & Claypool, San Rafael, Calif. (1537 Fourth Street, San Rafael, CA 94901 USA).
[43] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. *SOSP* (2013), 18–32.

[44] Lukas Vogel, Alexander Van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *PVLDB* 15, 11 (2022), 2895–2907.

[45] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. 2016. Be My Guest: MCS Lock Now Welcomes Guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Article 21, 12 pages.

[46] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB* 10, 2 (Oct. 2016), 49–60.

[47] Tianzheng Wang, Justin Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 461–472.

[48] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 473–488.