Robust Optimistic Locking for Memory-Optimized Indexes

by

Ge Shi

B.Sc., Simon Fraser University, 2021
B.Eng., Zhejiang University, 2021

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2023

Declaration of Committee

Name:	**Ge Shi**
Degree:	**Master of Science**
Thesis title:	**Robust Optimistic Locking for Memory-Optimized Indexes**
Committee:	**Chair:** Yuepeng Wang Assistant Professor, Computing Science
	Tianzheng Wang Supervisor Assistant Professor, Computing Science
	Alaa R. Alameldeen Committee Member Associate Professor, Computing Science
	Jiannan Wang Examiner Associate Professor, Computing Science

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 multicore 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 thesis 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.

Keywords: Optimistic locking; latching; robustness; index; OLTP

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Tianzheng Wang, for his invaluable guidance and unwavering support throughout my master's program. Thanks to him, I discovered the fascinating world of database systems, a field that I once mistakenly considered tedious during my undergraduate studies at my (other) alma mater, Zhejiang University. His expertise, knowledge, and critical feedback have been instrumental in shaping the direction and focus of my research. His dedication and enthusiasm has been a continual source of inspiration and encouragement for me throughout my academic journey.

I would like to thank my committee member, Dr. Alaa Alameldeen, and my examiner, Dr. Jiannan Wang, for their insightful comments and suggestions. I would also like to thank Dr. Yuepeng Wang for being the chair of my thesis defence.

My research would not have been possible without the valuable contribution of my collaborator, Ziyi Yan. His technical expertise and collaboration have been essential to the success of my research.

I would like to express my acknowledgments to the current and previous members of the Data-Intensive Systems Lab at Simon Fraser University: Kaisong Huang, Jianqiu Zhang, Xiangpeng Hao, Yongjun He, Baotong Lu, YuLiang (George) He, Miao Liu, Charley Peter Chen, Jiacheng Lu, Tianxun Hu, and Duo Lu, as well as my friends at Zhejiang University: Xuanda Yang, Zhenyun Yu, Chengyi Zhang, Ruikai Huang, Yaozhu Sun, Fan Wu, and Jiachen Li, for their support, encouragement, and friendship. Their expertise and diverse perspectives have enriched my research, and their insightful discussions and constructive feedback have been invaluable to me.

Finally, I would like to express my heartfelt appreciation to my girlfriend, Ge Chang, for her constant love and support throughout my master's program. Her consistent encouragement, understanding, and unfaltering support have been an endless source of motivation for me, and I am incredibly grateful for her presence in my life.

Table of Contents

Declaration of Committee ii

Abstract iii

Acknowledgments iv

Table of Contents v

List of Tables vii

List of Figures viii

1 Introduction 1
- 1.1 Centralized Optimistic Locking . . . 1
- 1.2 OptiQL: Optimistic Queuing Lock . . . 3
- 1.3 Contributions and Thesis Organization . . . 4

2 A Primer on (Index) Synchronization 5
- 2.1 Pessimistic Index Locking . . . 5
- 2.2 Optimistic Locking . . . 6
- 2.3 Queue-based and MCS Locks . . . 8

3 Desirable Properties and Tradeoffs 10

4 OptiQL: Optimistic Queuing Lock 11
- 4.1 Interfaces . . . 11
- 4.2 OptiQL Lock Structures . . . 12
- 4.3 OptiQL Protocols . . . 13
- 4.4 Optimistic Readers . . . 13
- 4.5 Exclusive Writers . . . 14
- 4.6 Opportunistic Read . . . 16
- 4.7 Discussions . . . 17

5 Index Locking with OptiQL 19

5.1 Memory-Optimized B+-Trees 19
5.2 Adaptive Radix Tree (ART) 20
5.3 Queue Node Management 22

6 Evaluation **24**
6.1 Experimental Setup 24
6.2 Microbenchmark Results 25
6.3 End-to-End Index Scalability 27
6.4 Tail Latency and Fairness 29
6.5 Impact of Key Space Sparsity 30

7 Related Work **32**

8 Summary **33**
8.1 Conclusion 33
8.2 Future Work 33

Bibliography **34**

List of Tables

Table 6.1 Reader success rate of `OptiQL-NOR` and `OptiQL` under varying read/write ratios and high contention (80 threads). 27

List of Figures

Figure 1.1 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. 2

Figure 2.1 Existing optimistic locks extend spinlocks (a) to support optimistic readers (b). The similar interfaces allow easy adaption in lock coupling protocols (c). 7

Figure 4.1 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. 12

Figure 4.2 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 allow other concurrent 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). 15

Figure 5.1 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. 21

Figure 6.1 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` and `OptiQL-NOR` can avoid performance under extreme and high contention. 26

Figure 6.2 Lock throughput under various contention levels and read/write ratios using 80 threads. OptiQL keeps the advantage of traditional optimistic locks under medium-low contention and read-dominant workloads, while avoiding collapsing under high contention and high write ratios. 27
Figure 6.3 Read-only throughput of B+-tree (left) and ART (right) under a skewed workload (self-similar distribution with a skew factor of 0.2). `OptiQL` and `OptiQL-NOR` perform the same as `OptLock` for read-only workloads. 28
Figure 6.4 Aggregate throughput of B+-tree (top) and ART (bottom) under the same skewed workload. 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 (first three columns). The additional atomics added by opportunistic read is negligible as shown by the update-only workload (right-most). 29
Figure 6.5 Index throughput under low contention and the balanced workload. Both centralized optimistic locks and OptiQL variants perform well when contention is rare. 29
Figure 6.6 Latency at different percentiles at 20 and 40 threads under self-similar distribution (skew factor 0.2). 30
Figure 6.7 Throughput of ART under self-similar distribution (skew factor 0.2) with sparse integer keys. 31

Chapter 1

Introduction

Concurrent indexes [5,7,14,24,26,27,30,32,33,36] are crucial for online transaction processing (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. Recent memory-optimized indexes have moved away from traditional pessimistic lock coupling [4] to lock-free approaches or optimistic locking. For example, the BwTree [30] devises lock-free B+-tree algorithms with atomic instructions such as compare-and-swap (CAS) [21] without using locks. Lock-free indexes can be fast, but are notoriously hard to implement and debug [35, 46]. Therefore, recent indexes focused more on optimistic locking.

1.1 Centralized Optimistic Locking

Compared to lock-free designs and pessimistic locking, *optimistic locks*[1] [7, 27] strike a balance between implementation complexity and performance. The basic idea is to allow read operations to proceed without exclusively acquiring the lock; only write operations need to acquire the lock exclusively. To detect inconsistencies caused by concurrent writes, read operations must check if the contents read earlier are still valid after the access. This is typically done by checking whether a version number embedded in the lock has changed after the read.

Optimistic locks greatly reduce synchronization overhead by avoiding shared memory writes caused by acquiring a shared lock. They are easy to implement and preserve the same API and compactness as traditional pessimistic locks by using only a single 8-byte

[1]Also known as "latches" to differentiate from "locks" used in logical-level concurrency control [15]. We follow the synchronization literature to use "locks."

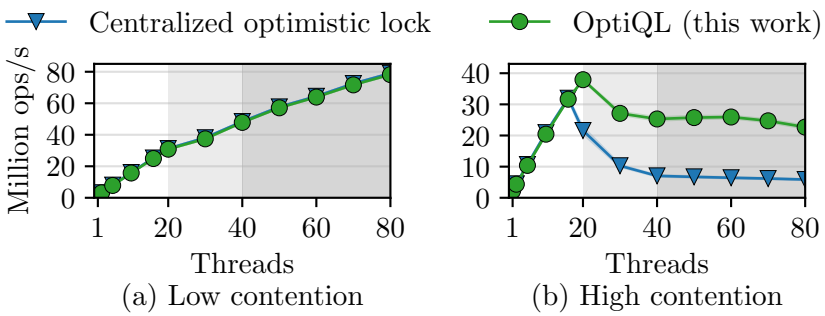


Figure 1.1: Update throughput of a memory-optimized B+-tree on a 80-thread server. Centralized optimistic locks work well under low contention (a) but can collapse under high contention (b). OptiQL performs well in both cases.

memory word as the lock. Thus, optimistic locking has been adopted by multiple recent memory-optimized indexes [25, 27, 28, 33].

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 1.1 shows the throughput of a memory-optimized B+-tree using optimistic locking (details in Chapter 6) 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 [41] that use the hardware-provided compare-and-swap (CAS) instruction [21] 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 1.1(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 [41] 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 $\sim$3$\times$ 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 [23, 44]. 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 thesis proposes OptiQL,[2] a new optimistic lock, to solve the aforementioned problems. OptiQL draws inspirations from the synchronization literature to introduce queue-based locking [9, 34, 37] to optimistic locks for both fairness and robustness. Queue-based locks are known for improved robustness under high contention [19, 41], 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 [37] to form a queue of requesters.[3] Different from existing queue-based locks which only provide mutual exclusion or blocking read [38], 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 Chapter 5 by adapting concurrent B+-trees [28] and tries [26, 27] which are based on traditional OLC to use OptiQL. From our experience,

[2]OptiQL is pronounced as "optical."

[3]At a first glance, this is similar to how lock managers for logical level concurrency control work [18]. 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 [15].

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 chapters, 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 YCSB-like [8] 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 [28] and trie (ART [26]) 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 thesis.

1.3 Contributions and Thesis Organization

This thesis 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. This thesis is based on a paper submission currently under revision at SIGMOD 2024.

The remainder of this thesis is orgranized as follows. Chapter 2 gives the necessary background. Chapter 3 summarizes the desired properties of index locking. We describe OptiQL and its use in index locking in Chapter 4–5. Chapter 6 evaluates OptiQL. We cover related work in Chapter 7 and summarize in Chapter 8.

Chapter 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 [15]. Each index node (inner or leaf) is protected 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 [15], 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 and indexes reside 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. An optimistic lock operates similarly to a reader-writer lock, however, readers can proceed optimistically assuming the lock is not held exclusively without issuing any hardware synchronization instruction (e.g., CAS) to acquire the lock in shared mode. As a result, readers do not announce their existence by writing to shared memory, avoiding most physical contention. This allows optimistic locks to scale almost perfectly on read-dominant workloads. However, a concurrent writer can acquire the lock in exclusive mode and modify the data being used by concurrent readers, without knowing the existence of any reader, So readers must validate that the data it read did not change upon completion in case a concurrent writer has modified it. Therefore, optimistic locks are in nature writer-biased because readers must validate and backoff if inconsistencies are discovered. This can be inefficient for certain operations such as long scans: a sporadic write can waste the much effort of a heavy read operation. Some systems combine pessimistic read locks [6] with optimistic locks to mitigate this problem. Nevertheless, optimistic locks have been widely used by recent work for their efficiency under read-heavy workloads [27, 28, 32, 33].

Existing optimistic locks [17, 27] realize these ideas by extending centralized test-and-test-and-set (TTS) [40] spinlocks in Figure 2.1(a). Same as TTS, centralized optimistic locks use a single 8-byte memory word as the lock, as Figure 2.1(b) shows. In addition to the lock state using the most significant bit (`1UL<<63`), the lock also carries a version number, whereas the TTS lock only needs one bit to record the lock state. 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. Using 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 to access the protected resource (e.g., 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 return a "failed to lock" result. The caller must keep track of the version

```
uint64_t lock;

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

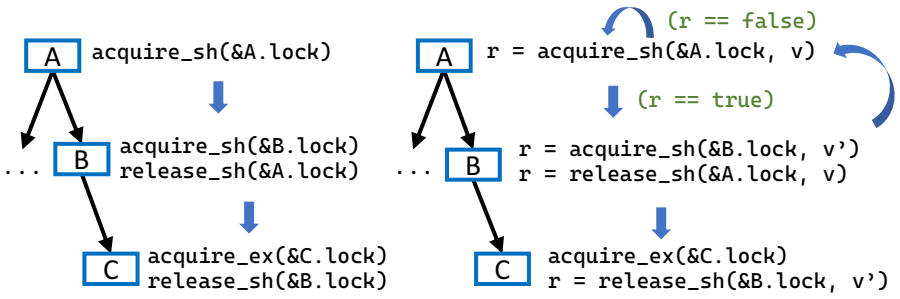
```
#DEFINE LOCKED   1UL
#DEFINE UNLOCKED 0UL

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

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 [41].

Optimistic Lock Coupling. With the interfaces in Figure 2.1(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 [25]. Figure 2.1(c) shows the same example used by prior work [25] 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 2.1(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

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.  }
7.  pred->next = qnode;
8.  while (!qnode->granted) {}
9. }
```

```
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;
10. }
```

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 Chapter 5.

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, 34, 37] 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 [37] 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, 34]) work similarly but differ in how they manipulate the queue [41]. 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 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`) [21] 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 8). 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 7 in Algorithm 1 (left) (but has

executed line 2), 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.

Chapter 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 [21].
- **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 [23, 42]. 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 [41].

Chapter 4

OptiQL: Optimistic Queuing Lock

In this chapter, 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.

The rest of this chapter describes OptiQL's interfaces and structures. We then give the detailed lock protocols of OptiQL in Chapter 4.3 and show how OptiQL can work with optimistic lock coupling (D5) in Chapter 5.

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:

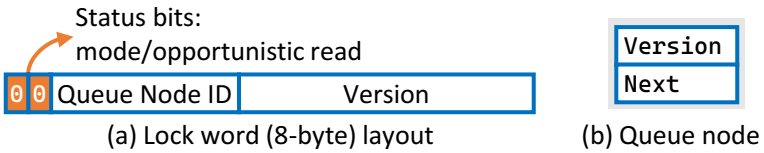


Figure 4.1: OptiQL structures. (a) The lock is still 8-byte but carries more information. (b) A queue node (for writers only) includes a version number and a pointer to the next writer.

- `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 4.1(a) 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 identifier of the queue node. Since memory-optimized database engines 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 Chapter 5), the total number of queue nodes in the entire system is not large. This means queue node IDs can be much shorter than a full virtual memory address (up to 57 bits on modern 64-bit x86 architectures [21]), 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 Chapter 4.3 describes is critical for correctness), (2) ensure the version number can be big enough to avoid early wrap-around, while (3) still maintaining a

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

```
1. #DEFINE STATUS_MASK 0xC000_0000_0000_0000ULL
2. #DEFINE LOCKED      0x8000_0000_0000_0000ULL
3.
4. bool acquire_sh(OptiQL *lock, uint64_t &v) {
5.  v = *lock;
6.  return (v & STATUS_MASK != LOCKED);
7. }
8.
9. bool release_sh(OptiQL *lock, uint64_t v) {
10.  return *lock == v;
11. }
```

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 [23]; OptiQL adopts it for maintaining the lock's compactness. The tradeoff 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 [23].

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 4.1(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.

4.3 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.

4.4 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. In Algorithm 2, the reader simply reads the lock word (line 5) to check whether the lock is already granted to a writer and opportunistic read is disallowed, by extracting the two status bits (line 6). 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

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

```
 1. #DEFINE OPREAD      0x4000_0000_0000_0000ULL
 2.
 3. void acquire_ex(OptiQL *lock, QNode *qnode) {
 4.  pred = XCHG(lock, LOCKED | to_id(qnode));
 5.  if (!pred.locked) { /* Lock is free */
 6.   qnode->version = pred.version + 1;
 7.   return; /* Lock acquired */
 8.  } else {
 9.   to_ptr(pred.qnode_id)->next = qnode;
10.   while (qnode->version == INVALID) {} /* Local spinning */
11.  }
12.  /* Lock granted, turn off opportunistic read */
13.  FETCH_AND(lock, ~(OPREAD | VERSION_MASK));
14. }
15.
16. void release_ex(OptiQL *lock, QNode *qnode) {
17.  if (qnode->next == null &&
18.      CAS(lock, LOCKED | to_id(qnode), qnode->version)) {
19.   return; /* Lock released - indeed no successor */
20.  }
21.  /* Enable opportunistic read */
22.  FETCH_OR(lock, OPREAD | qnode->version);
23.  /* Ensure the successor links after [qnode] */
24.  while (qnode->next == null) {}
25.  /* Grant successor the lock by passing version number */
26.  qnode->next->version = qnode->version + 1;
27. }
```

to retry. The `LOCKED` macro is defined similarly to that in the centralized optimistic lock in Figure 2.1(b), and the amount of work done—read the lock word, apply a mask and compare—is exactly the same. Validation (line 10) works exactly the same as `release_sh` in Figure 2.1(b). This greatly simplifies the adaptation of existing optimistic lock coupling protocols to use OptiQL (Chapter 5).

4.5 Exclusive Writers

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

Acquire. Algorithm 3 (lines 3–14) shows how a writer acquires the OptiQL lock in exclusive mode. Same as the MCS lock, 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; we discuss the details for address translation later. Figures 4.2(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 5 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 6). Figure 4.2(b) shows

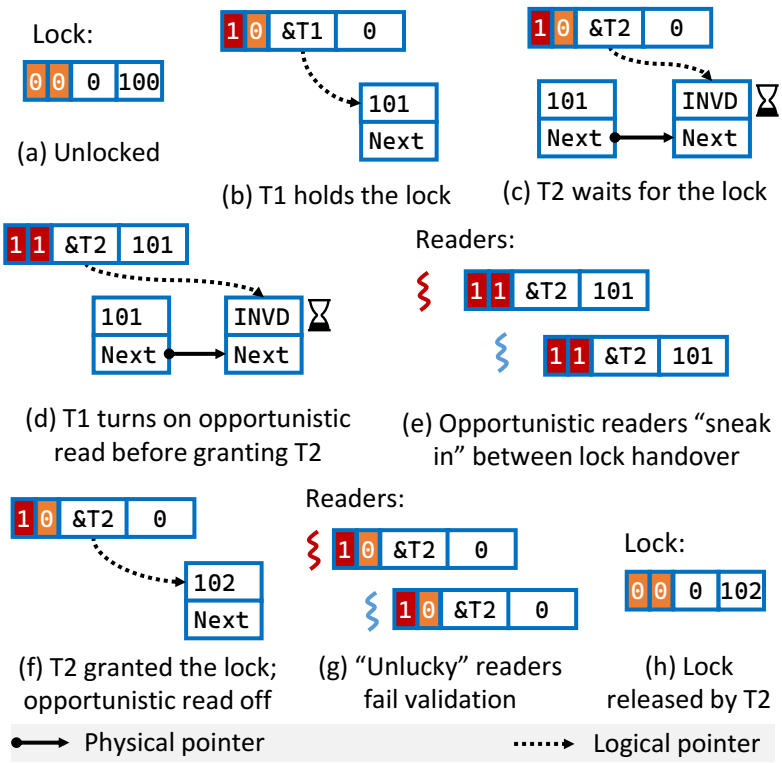


Figure 4.2: Coordination of concurrent requesters. (a–c) Exclusive writers proceed like using normal MCS locks. (d) Between lock handover, the current lock holder (T1) turns on opportunistic read to allow other concurrent readers (e). (f) Opportunistic read is turned off once the next exclusive requester (T2) is granted the lock, invalidating readers that did not finish before T2 is granted the lock (g).

the status of the lock after the first requester, $T1$, has acquired the lock which was free in Figure 4.2(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 4.2) then must (1) link itself with the predecessor (line 9), and (2) spin locally to wait for the predecessor (denoted as `pred.address`) to pass the lock (line 10). 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 4.2(c). Finally, line 13 disables opportunistic read as the last step, which we explain later.

Release. To release the lock (lines 16–27), 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 17). If the result is `NULL`, we check whether there is indeed no writer successor by issuing a `CAS` on the lock (line 18) 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 6 and 26) to reflect the new version. For example, in Figure 4.2(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 4.2(a). This is similar to the `release_ex` function in centralized optimistic locks shown in Figure 2.1(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 22 for now), we pass the lock to it, e.g., $T2$ in Figures 4.2(c) and 4.2(f), by incrementing and atomically storing the version number in the successor's queue node (line 26). This will cause the successor to break out of its local spin loop at line 10. The successor now holds the lock, and since the version number was incremented again (line 26), the successor is ready with right version number for executing its own release protocol after executing the critical section.

4.6 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 Chapter 6, this allows almost no readers as long as there are any writers.

OptiQL introduces opportunistic read to solve this problem. On top of optimistic reads, opportunistic read takes advantage of the time window of consistency during lock handover: when a writer finishes executing 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 this purpose. 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` [21] instruction in one atomic step. This way, as shown in Figure 4.2(d), the lock word will

then (1) have both the status bits set, (2) record the latest requester's queue node, and (3) carry a version number. Note that the queue node is retained on the lock word so that subsequent writer requesters can 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 4.2(e), readers can now optimistically access the data protected by the lock using the protocols described in Section 4.4. The only difference is that since readers follow Algorithm 2, they will take the entire lock word as shown in Figure 4.2(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. Therefore, 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 10. It then turns off opportunistic read (line 13) to finish lock acquisition. Opportunistic readers that validate after this point, e.g., those in Figure 4.2(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 [20]. 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.

4.7 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 Chapter 6 shows that this

can be non-negligible in certain microbenchmarks that stress the lock itself. However, we find that the overheads are negligible in realistic workloads, making it a worthwhile tradeoff.

OptiQL requires non-standard interfaces that expect a queue node parameter. Since OptiQL's lock word now records the latest requester's queue node ID, instead of its queue node address, an address translation step is needed as we mentioned earlier. Combined, these two issues require the application manage its queue nodes. Prior work [23] 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.

Chapter 5

Index Locking with OptiQL

Now we demonstrate how two representative indexes—memory-optimized B+-trees [28] and the adaptive radix tree (ART) [27]—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.

5.1 Memory-Optimized B+-Trees

Memory-Optimized B+-trees usually opt for smaller nodes (e.g., 512 bytes instead of 4KB) for better cache efficiency [47]. This emphasizes the need for compact locks that occupy 8-byte or less space. 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: (1) traverse to the target leaf node L using optimistic read (same as original OLC); (2) directly lock L; (3) validate the parent node has not changed (note this was step 2 in the original protocol); (4) search L for the target key, perform the update, release the lock and return. This allows us to truly benefit from the queue-based design in OLC protocols for B+-trees.

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.

5.2 Adaptive Radix Tree (ART)

ART [26] 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 Chapter 6 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

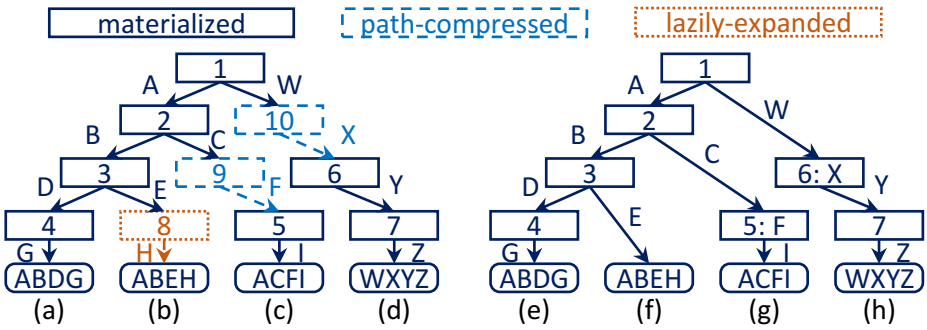


Figure 5.1: The logical (a–d) and physical (e–h) structures of ART with path compression and lazy expansion. (a, e) Path to `ABDG` is fully materialized. (b, f) Node 3 points to `ABEH` due to lazy expansion. (c, g) Path between nodes 2 and 5 is compressed, with prefix `F` in node 5. (d, h) Path between 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.

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 [26].

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.

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 5.1 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.

5.3 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 satisfy two important requirements. (1) It needs to be globally accessible. (2) In multi-socket servers queue nodes should be locally allocated; otherwise, "local spinning" will incur remote traffic. A naive approach is to track dynamically allocated queue nodes in a hash table, similar to how locks are maintained in conventional database systems in a lock manager. However, this can bring high overhead and defeat the purpose of using OptiQL; modern memory-optimized OLTP engines [10, 22, 23, 31, 42] avoid using such hash tables and lock managers for the same reason.

We observe that it is preferable to pre-allocate all the required queue nodes in a contiguous array and use indexes into the array as queue node IDs. Our solution uses interleaved NUMA memory allocation for queue node management. NUMA libraries like `libnuma` can interleave allocations across a set of NUMA nodes for a virtually contiguous chunk of memory (e.g., using `libnuma`).[1] Thus, we allocate memory pages of queue nodes in a round-robin manner across all sockets. 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. We then initialize queue nodes on these pages and allocate them at runtime. Note that since the memory is presented to the application as a contiguous array, we can still directly take the index into the array as the queue node ID, yet without dedicating bits in the queue node ID to encode NUMA node information.

[1]Specifically, the `numa_alloc_interleave` function: `https://linux.die.net/man/3/numa_alloc_interleaved`.

In essence, our approach sets up an indirection between queue node IDs (which occupy fewer bits) and the queue nodes' virtual addresses (which occupy 64 bits). This is a common approach taken by systems built for large-scale systems [23], however, previous solutions required allocating an array per NUMA node and encoding the NUMA node ID in the queue node ID. This further requires the application to pre-determine the number of sockets to support, thus increasing complexity. In contrast, our approach does not have such requirements, making it easier to implement and more practical.

Chapter 6

Evaluation

We empirically evaluate OptiQL using both microbenchmarks and YCSB-like [8] workloads. 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.

6.1 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. 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 1 (extreme contention), 5 (high contention), 30,000 (medium contention) and 1,000,000 (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 [29], 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 [47]. 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.

Each experiment runs for 10 seconds and is repeated for 20 times. We collect the throughput (lock acquires per second for microbenchmarks, and the number of index operations per second for index benchmarks) numbers and report the averages of 20 runs with error margins. We also report tail latency 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 [27, 28].
- `OptiQL`: OptiQL proposed in Chapter 4.
- `OptiQL-NOR`: Same as `OptiQL` but without opportunistic read.
- `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.

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

6.2 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 6.1 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` 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. Both `OptiQL-NOR` and `MCS` can handle contention well. OptiQL shows a similar trend but lower absolute performance (a constant amount of overhead) than `OptiQL-NOR` because it employs additional steps to support opportunistic readers. However, as we show later,

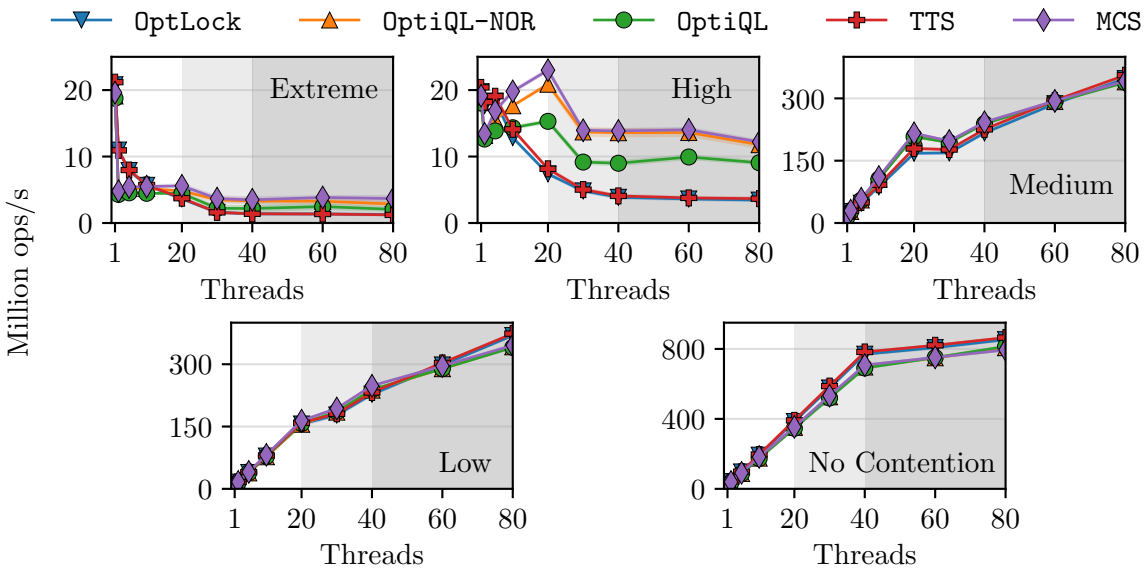


Figure 6.1: Exclusive lock throughput under different contention levels; `MCS` and `TTS` are shown as a reference to evaluate OptiQL's writer performance as they do not support readers. Queue-based `OptiQL` and `OptiQL-NOR` can avoid performance under extreme and high contention.

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.

Mixed Operations. Figure 6.2 shows the throughput of three locks under 80 threads; we excluded read-only results as all the locks scale and perform identically. In the figure, `OptLock` remains low performance due to high contention on the central lock word. `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 6.1, the successful rates of read operations of `OptiQL-NOR` are around or less than 2%, which means few readers were able to successfully acquire the lock under `OptiQL-NOR`, whereas `OptiQL` can achieve 28~32% successful rates for read operations. With more reads, the overhead of opportunistic read gradually 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

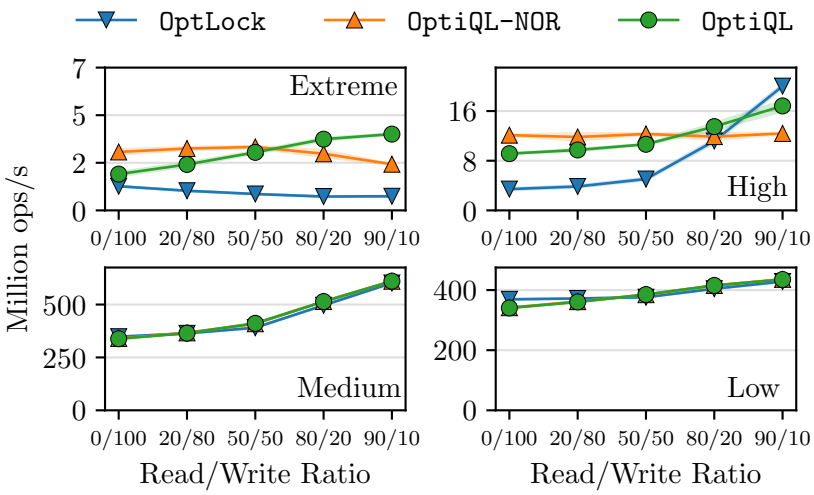


Figure 6.2: Lock throughput under various contention levels and read/write ratios using 80 threads. OptiQL keeps the advantage of traditional optimistic locks under medium-low contention and read-dominant workloads, while avoiding collapsing under high contention and high write ratios.

Table 6.1: 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`	2.02%	1.45%	0.52%	1.08%
`OptiQL`	31.63%	29.87%	30.42%	27.94%

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. Under medium and low contention, all the locks performed similarly with expected high performance for the mixed workloads.

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.

6.3 End-to-End Index Scalability

Following Chapter 5 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 [16] distribution with a skew factor of 0.2 (i.e., 80% accesses target 20% keys). In addition, we

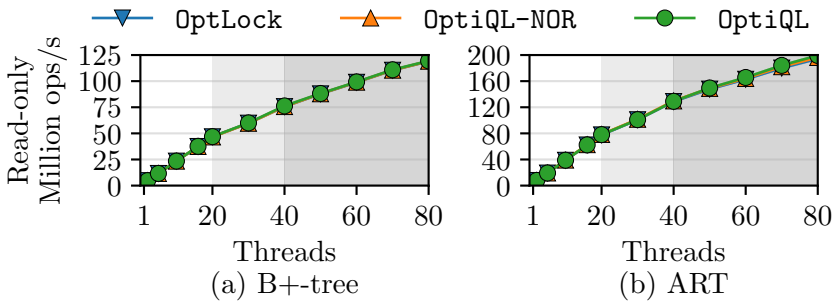


Figure 6.3: Read-only throughput of B+-tree (left) and ART (right) under a skewed workload (self-similar distribution with a skew factor of 0.2). `OptiQL` and `OptiQL-NOR` perform the same as `OptLock` for read-only workloads.

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 6.3 and Figure 6.4 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.

As Section 6.2 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 6.4), `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.

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 6.5 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.

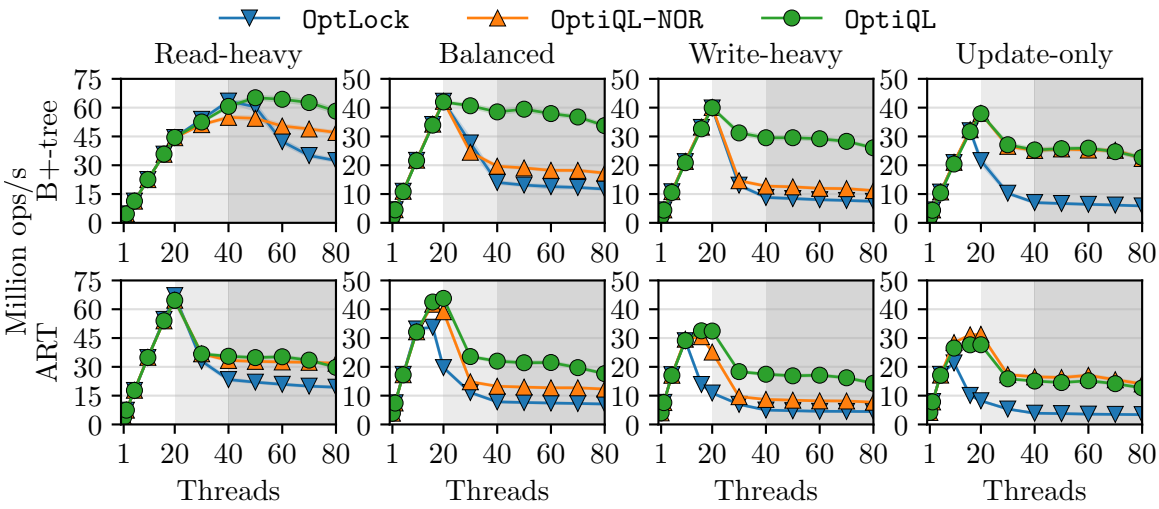


Figure 6.4: Aggregate throughput of B+-tree (top) and ART (bottom) under the same skewed workload. With more writers (left to right), `OptLock` collapses beyond one socket, while `OptiQL` maintains high performance without collapsing. Opportunistic read is critical in enabling `OptiQL` to allow readers (first three columns). The additional atomics added by opportunistic read is negligible as shown by the update-only workload (right-most).

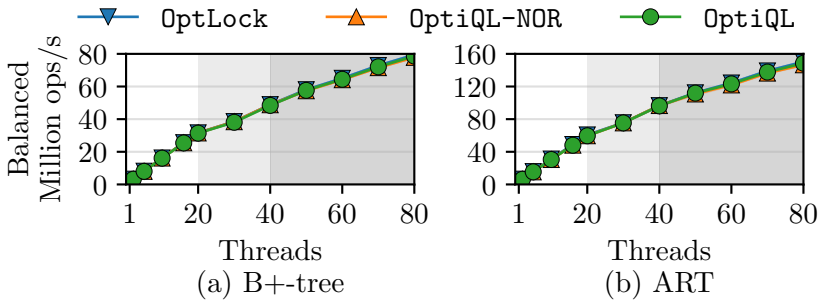


Figure 6.5: Index throughput under low contention and the balanced workload. Both centralized optimistic locks and OptiQL variants perform well when contention is rare.

6.4 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 6.6 across different workloads and under 20 (one socket) and 40 threads (two sockets). Here, `OptLock` has drastically increasing tail latencies under update-only workloads, while `OptiQL-NOR` and `OptiQL` stay relatively stable across the horizontal axis. We have also collected the variance of per-thread throughput 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

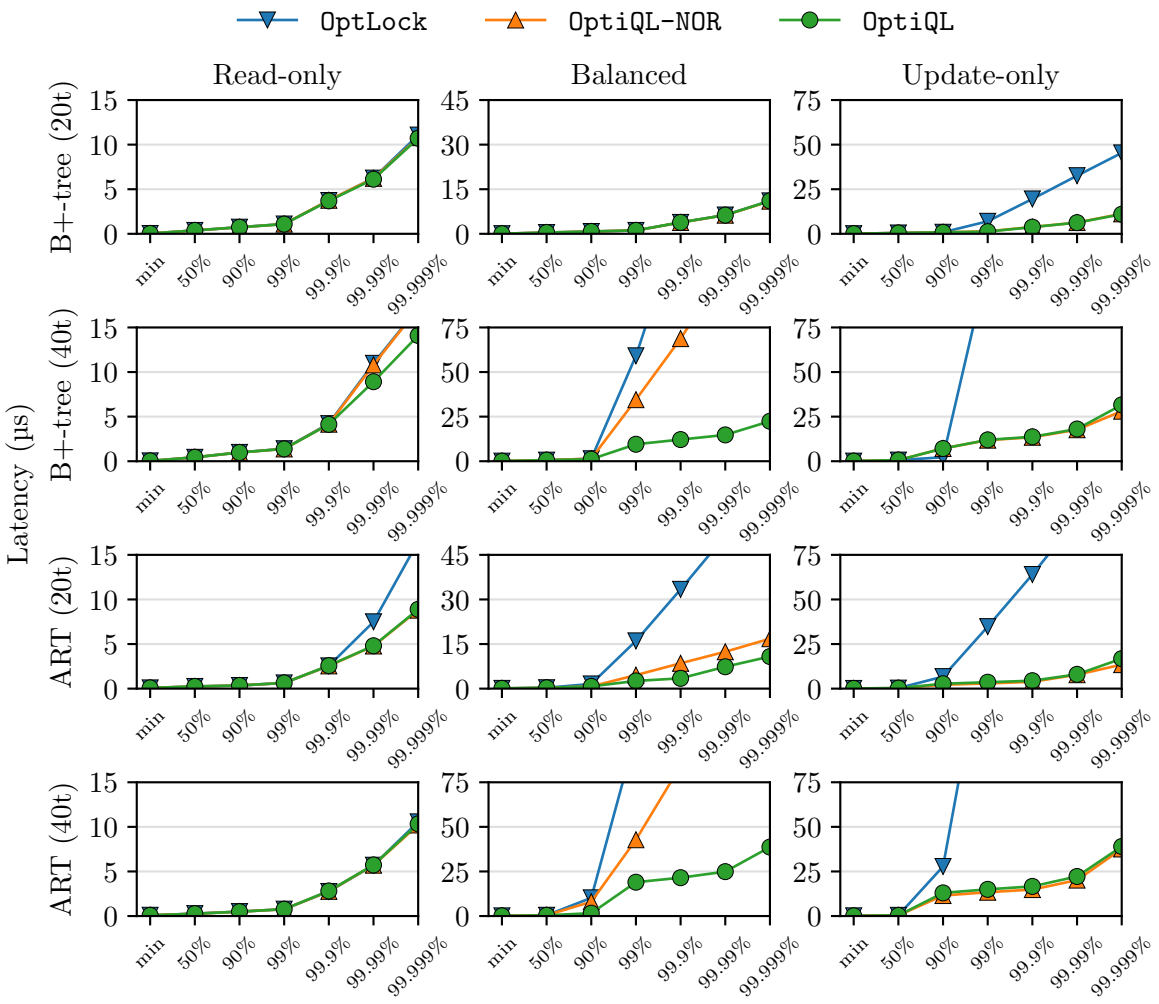


Figure 6.6: Latency at different percentiles at 20 and 40 threads under self-similar distribution (skew factor 0.2).

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

6.5 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 6.7 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 5.2 to local-spin, avoiding performance collapse.

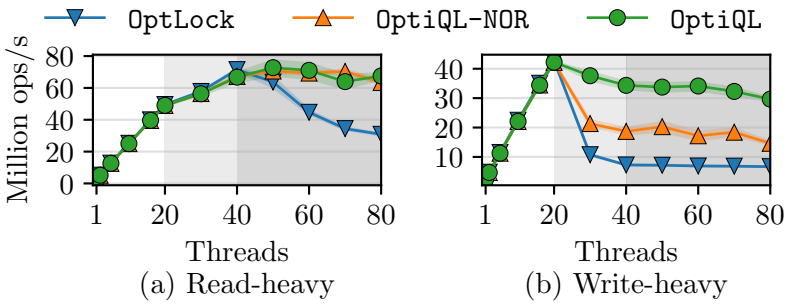


Figure 6.7: Throughput of ART under self-similar distribution (skew factor 0.2) with sparse integer keys.

Chapter 7

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 [38], more NUMA-friendly spinning [11] to reduce lock handover cost, and context-free standard lock interfaces [44]. OptiQL uniquely further adds optimistic read capabilities. In addition to queue-based locks, ticket locks [41] 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, 45] 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 [36] uses two version counters to differentiate updates and splits to reduce unnecessary retries. Some recent persistent memory (PM) indexes [32, 33, 39, 43] also use optimistic locks to improve scalability. Several indexes further avoid locking with lock-free algorithms. The Bw-Tree [30] 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 uses a multi-word CAS [46] 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 [47], making them the preferred choice in many systems.

Chapter 8

Summary

In this chapter, we conclude the thesis and discuss potential future research.

8.1 Conclusion

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 writer requesters 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 algorithms 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.

8.2 Future Work

The following section outlines potential areas of future research.

System integration and end-to-end performance. We did not explore the practicability of integrating OptiQL into existing OLTP systems [22, 42]. Opportunities for future research include supporting index structures based on OptiQL in transactional systems and evaluating their end-to-end performance.

Pessimistic reader variants. We did not compare with pessimistic readers-writer locks [38, 41], which might held its place on long-running read workloads like range scans. An evaluation for such locks and the interoperability and composability between OptiQL and them are left as future work.

Bibliography

[1] Adnan Alhomssi and Viktor Leis. Contention and space management in b-trees. In *Conference on Innovative Data Systems Research (CIDR 2021)*, 2021.

[2] T.E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[3] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *PVLDB*, 11(5):553–565, 2018.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Inf.*, 9(1):1–21, mar 1977.

[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 521–534, 2018.

[6] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. Scalable and robust latches for database systems. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, 2020.

[7] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 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, page 181–190, 2001.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. SoCC '10, page 143–154, 2010.

[9] Travis S. Craig. Building fifo and priority-queuing spin locks from atomic swap. Technical report, TR 93-02-02, Dept. of Computer Science, University of Washington, 1993.

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, 2013.

[11] Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, 2019.

[12] Dave Dice and Alex Kogan. Fissile locks. In *Networked Systems: 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3–5, 2020, Proceedings*, page 192–208, 2020.

[13] David Dice, Virendra J. Marathe, and Nir Shavit. 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, page 247–256, 2012.

[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. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 969–984, 2020.

[15] Goetz Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35(3), jul 2010.

[16] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, page 243–252, 1994.

[17] Rachid Guerraoui and Vasileios Trigonakis. Optimistic concurrency with optik. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016.

[18] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a database system. *Found. Trends Databases*, 1(2):141–259, feb 2007.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st edition, 2012.

[20] IBM. Ibm system/370 extended architecture, principles of operation, mar 1983. IBM Publication No. SA22-7085.

[21] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual. 2021.

[22] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687, 2016.

[23] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 691–706, 2015.

[24] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, dec 1981.

[25] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42:73–84, 2019.

[26] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, page 38–49, 2013.

[27] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, 2016.

[28] Viktor Leis and Ziqi Wang. Olc b+-tree. `https://github.com/wangziqi2016/index-microbench/tree/master/BTreeOLC`, 2017.

[29] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *PVLDB*, 13(4):574–587, 2019.

[30] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 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, pages 302–313, 2013.

[31] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 21–35, 2017.

[32] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. APEX: A high-performance learned index on persistent memory. *PVLDB*, 15(3):597–610, 2021.

[33] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *PVLDB*, 13(8):1147–1161, 2020.

[34] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, page 165–171, 1994.

[35] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proc. VLDB Endow.*, 8(11):1298–1309, jul 2015.

[36] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[37] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991.

[38] John M. Mellor-Crummey and Michael L. Scott. 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, page 106–113, 1991.

[39] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 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*, pages 371–386, 2016.

[40] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. *SIGARCH Comput. Archit. News*, 12(3):340–347, Jan 1984.

[41] Michael L Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Springer, 2013.

[42] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. *SOSP*, pages 18–32, 2013.

[43] Lukas Vogel, Alexander Van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. Plush: A write-optimized persistent log-structured hash-table. *PVLDB*, 15(11):2895–2907, 2022.

[44] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. 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, 2016.

[45] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, October 2016.

[46] Tianzheng Wang, Justin Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, April 2018.

[47] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 473–488, 2018.