CHAIR OF DECENTRALIZED
INFORMATION SYSTEMS & DATA
MANAGEMENT

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Concurrent Range Locking

Thua-Duc Nguyen

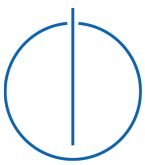
CHAIR OF DECENTRALIZED INFORMATION SYSTEMS & DATA MANAGEMENT

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Concurrent Range Locking

Author:	Thua-Duc Nguyen
Supervisor:	Prof. Dr. Viktor Leis
Advisor:	Lam-Duy Nguyen
SubmissionDate:	15.09.2024



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

München, 15.09.2024 Thua-Duc Nguyen

Acknowledgments

I would like to express my sincere gratitude to my dedicated supervisor Prof. Dr. Viktor Leis and advisor Lam-Duy Nguyen, for their unwavering support and guidance. They have consistently provided support and motivation, even amid their busy schedule.

My heartfelt appreciation to my girlfriend, Thuy-Trang Nguyen, for standing by me through the challenges of my thesis and undergraduate journey. She has consistently provided support and motivation.

Eventually, I would like to send a special acknowledgment to my family and friends whose encouragement has been a source of strength and an integral part during the course of my studies.

I would not have been able to accomplish this work without the support of each of these people.

Abstract

Large-scale distributed systems and applications often face significant challenges in managing concurrent access to shared resources. For instance, high-performance computing environments frequently require efficient handling of numerous simultaneous access requests to various parts of data sets. Similarly, disaggregated memory systems must support multiple clients accessing the same memory space concurrently, often with diverse access patterns. In such contexts, it becomes crucial to coordinate and manage these concurrent accesses effectively to ensure correctness and performance.

Concurrent range locks address these challenges by providing mechanisms for efficient and accurate management of overlapping and non-overlapping ranges of resources, thereby enabling scalable and reliable access control in complex, high-demand environments.

Previous studies have examined various range lock techniques, with the Linux kernel currently employing a range tree accompanied by a spin lock for managing access to ranges. This single spinlock, however, introduces bottlenecks. Song et al. proposed an enhancement by integrating a skip list with the spinlock, offering a more efficient and less heavyweight solution than traditional interval trees, though challenges with contention persist. Alternatively, Kogan et al. developed a lock-free range lock utilizing a concurrent linked list, each node representing an acquired range. This method addresses limitations found in existing range locks but at the cost of reduced efficiency in insertion and search operations compared to tree-based structures.

In the scope of this research, we propose a new lock-free concurrent range-locking design. We address previous bottleneck issues by leveraging a probabilistic concurrent skip list and removing the lock while maintaining high performance. The proposed mechanism will be developed and evaluated under heavy concurrent access, ensuring correctness in overlapping ranges and concurrent operations. Performance comparisons with existing state-of-the-art approaches will comprehensively assess its effectiveness.

Contents

List of Figures

List of Tables

1 Introduction

Locking mechanism is important. In modern computing environments, the efficiency of locking mechanisms is pivotal to the performance of various systems, including databases [1, 2], file systems [3, 4, 5], and operating systems [6, 7]. The need for more advanced and fine-grained locking mechanisms becomes critical as these systems continue to scale and increase complexity. A key challenge in this context is the management of concurrent access to shared resources. Traditional locking techniques, such as single-lock mechanisms, often introduce significant performance bottlenecks, especially in scenarios with high concurrency, thereby underscoring the necessity for more sophisticated approaches to locking.

Range locks boot performance through resource segmentation. Range lock [4, 8] provides a more refined approach to this challenge by partitioning a shared resource into multiple arbitrary-sized segments. Different processes can exclusively acquire each of these segments. This strategy effectively addresses the drawbacks and bottlenecks associated with single-lock methods, significantly improving the performance.

DBMS needs range locks. Range locking is crucial in database management systems, particularly for ensuring data consistency and preventing anomalies such as "phantoms" in high-concurrency environments. When transactions require strict isolation, as under the serializable isolation level, range locks are used to secure not only the individual records within a specified range but also the gaps between these records. This prevents other transactions from inserting new records or modifying existing ones in the Range until the transaction is completed, thereby maintaining the integrity of operations that depend on the stability of a data set. Implementing range locks becomes particularly challenging in systems where transaction control (TC) and data control (DC) are separated, as the TC must lock the Range before safely interacting with the DC, despite not knowing the specific keys or records involved. Effective range-locking protocols are essential to managing this complexity, ensuring that all relevant resources are locked throughout the transaction to prevent race conditions and maintain consistency across concurrent operations.

Filesystem needs range locks. In high-performance file systems, particularly those used in large-scale and distributed computing environments, managing concurrent access

to shared files is important. As file systems scale to accommodate massive data sets and numerous parallel I/O operations, traditional locking mechanisms often become bottlenecks, reducing throughput and increasing latency. Range locks offer a solution by allowing multiple processes to access different file segments simultaneously without interfering with each other. This segmentation minimizes contention and improves performance by enabling finer-grained locking at the segment level. For file systems handling concurrent access to large files, especially in high-performance computing (HPC) environments, adopting range locks can significantly enhance efficiency and scalability [4, 3].

Operating system needs range locks. There has been growing interest in using range locks within the Linux kernel community. The focus is on using range locks to alleviate contention issues associated with `mmap_sem`, a semaphore that manages access to virtual memory areas (VMAs). VMAs represent distinct sections of an application's virtual address space and are organized in a red-black tree. The `mmap_sem` semaphore controls access for various operations such as memory mapping, unmapping, protection, and handling page faults. This becomes problematic for data-intensive applications with large, dynamically allocated memory, as contention on this semaphore can become a significant performance bottleneck.

Existing range lock needs improvement. Previous implementations of range-locking mechanisms often need to improve their performance. These implementations often suffer from contention points due to the reliance on a single lock [9, 10]. Additionally, some methods may be complex and tightly coupled with lock-based concurrency control protocols, which are not applicable for general DBMS operations [2, 11]. These limitations underscore the need for more refined and scalable solutions that can better handle the demands of modern, large-scale systems.

New concurrent range-locking design. In this research's scope, we propose a new lock-free concurrent range-locking design. We address previous bottleneck issues by leveraging a probabilistic concurrent skip list [12, 13] and replace traditional locking by `compare-and-swap` methods. By doing so, we address the previous range lock bottleneck issues and maintain the lock's high level of performance.

Outline of the research. The scope of this research includes developing and evaluating the proposed range-locking mechanism. We will evaluate focusing on performance under heavy concurrent accesses, ensuring the correctness of data access in overlapping ranges and concurrent operations. Additionally, we will compare the performance of the proposed solution with existing state-of-the-art approaches to provide a comprehensive assessment of its effectiveness.

2 Related Work

2.1 Coarse-Grained Range Lock

2.1.1 Tree-Based Range Lock

Several works have explored coarse-grained range-locking mechanisms. Jan Kara introduced a range-locking mechanism for the Linux kernel [9], which utilizes a range tree (specifically a red-black tree) to manage range locks and employs a spinlock for synchronization. Each lock is represented as a node in the tree. Similarly, Kim et al. adopted a comparable range-locking mechanism in their work on pNOVA [14], a variant of the NOVA file system that uses range-based reader-writer locks to enable parallel I/O within a single shared file.

When a thread requests a range lock, it first acquires a spinlock, then traverses the tree to determine the number of locks intersecting with the requested range. Afterward, the thread inserts a node describing its range into the tree and releases the spinlock. If no intersecting locks are found, the thread can proceed with accessing the critical section. If intersecting locks are detected, the thread waits until those locks are released and the number of intersecting locks drops to zero. Upon completing its operation, the thread re-acquires the spinlock, removes its node from the tree, updates the count of overlapping locks, and releases the spinlock. This method ensures that each range is locked only after all previous conflicting range locks have been released, thereby achieving fairness and avoiding livelocks.

Drawbacks

One significant observation is that the coarse-grained spinlock of an interval tree can severely hinder parallelism, as the spinlock effectively serializes all incoming lock and unlock requests. Under heavy concurrent access, this serialization easily becomes a contention point, limiting the system's performance.

Consider three exclusive lock requests for the ranges $A = [1..3]$, $B = [2..7]$, and $C = [4..5]$, arriving in that order. While A holds the lock, B is blocked because it overlaps

with A, and C is blocked behind B. However, in practice, C does not overlap with A and could proceed without waiting. This unnecessary blocking reduces the overall efficiency and concurrency of the system.

2.1.2 List-Based Range Lock

Song et al. [10] introduced a dynamic range-locking design to enhance the implementation of the Linux kernel. Their range lock uses a skip list [15] to dynamically manage the address ranges that are currently locked.

When a thread requests a specific range `[start, start+len)`, the range lock searches the skip list. If an existing or overlapping range is found, it indicates that another thread is currently modifying that range, requiring the requesting thread to wait and then retry. If no overlapping range is found, the requested range is added to the skip list, signifying that the lock has been acquired. Releasing a range involves deleting the corresponding range from the skip list.

Compared to the interval tree, the skip list is more lightweight and efficient, allowing for quicker searches of overlapping ranges.

Drawbacks

Similar to the tree-based range lock, contention remains an issue with this approach. Additionally, it unnecessarily blocks non-conflicting requests, further reducing system efficiency and limiting concurrency.

2.2 Fine-Grained Range Lock

2.2.1 List-Based Range Lock

Kogan et al. [8] introduced a novel range lock based on a concurrent linked list, where each node represents an acquired range. This design aims to provide a lock-free mechanism, addressing critical shortcomings of previous range-locking implementations. In a lock-free system, processes can proceed without being blocked by locks held by other processes, thereby improving performance and scalability.

The proposed method involves inserting acquired ranges into a linked list sorted by their starting points, ensuring that only one range from a group of overlapping ranges can be inserted using an atomic `compare-and-swap` (CAS) operation. A significant

difference in this method compared to previous ones is that each node has two statuses: `marked` (logically deleted) or `unmarked` (present).

When a thread wants to acquire a range, it iterates through the skip list. If it encounters a marked node, it removes it using CAS and continues to iterate. If the current node protects a range that overlaps, the thread waits until that node is deleted. Otherwise, a node is inserted into the list, signaling that the range is acquired. To release a range, the thread marks the node as deleted.

Drawbacks

Linked List Inefficiency: While this design implements a lock-free mechanism that effectively addresses the limitations of existing range locks, it comes with its own set of trade-offs. In general, insertion and lookup operations in a linked list are less efficient than in tree-like structures. The average time complexity for searching in a linked list is $O(n)$, whereas it is only $O(\log n)$ for skip lists or tree-like structures [16]. Our evaluation will demonstrate that this inefficiency becomes particularly pronounced when handling multiple overlapping ranges within the list.

3 Approach

In this research's scope, we propose our new concurrent range-locking design that leverages a probabilistic concurrent skip list [12, 13]. It consists of two main functions:

- **tryLock**: The `tryLock` function searches for the required range `[start,end]` in the skip list. If an overlapping range exists, indicating another thread is modifying that range, the requesting thread must wait and retry. If not, the range is added to the list, signaling that the range is reserved.
- **releaseLock**: The `releaseLock` function releases the lock by finding the address range in the skip list and removing it accordingly.

Our range lock design is also lock-free. It relies on atomic operations (`compareAndSet()`), thus addressing the bottleneck problem of the spinlock-based range lock and maintaining the lock's high level of performance.

3.1 Skip List

A skip list is a probabilistic data structure. It allows fast search, insertion, and deletion. It is an alternative to balanced trees, such as AVL trees or red-black trees [15, 17]. The key idea of a skip list is to use multiple layers of sorted linked lists to maintain elements, where each layer is an ”express lane” for faster traversal.

How it work

Multiple Layers: A skip list consists of multiple layers where the bottom-most layer is a regular sorted linked-list. Each higher layer acts as an ”express lane” that speeds up access by skipping over multiple elements from the layer below. Nodes in higher layers provide shortcuts, allowing faster traversal across the list and effectively reducing the time complexity of search operations.

Probabilistic Balancing: Each `Node` is created with a random top level and belongs to all lists up to that level. Top levels are chosen so that the expected number of nodes in

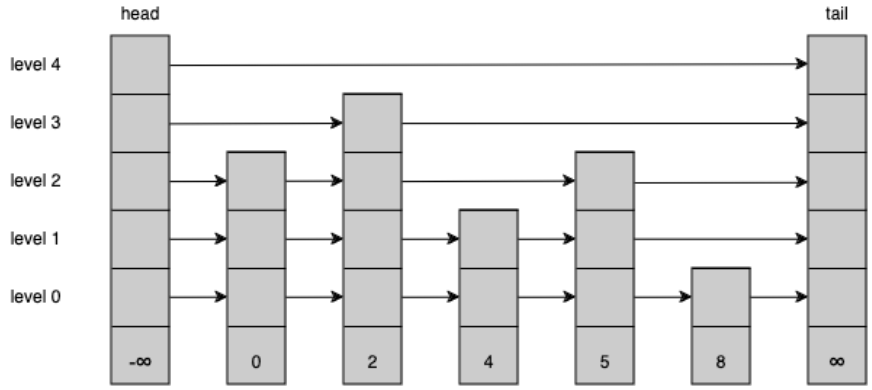


Figure 3.1: Skip List: this example has five levels of sorted linked lists.
Each `Node` has a unique key, and the head and tail have $-\infty$ and $+\infty$ keys.

each level's list decreases exponentially. Let $0 < p < 1$ be the conditional probability that a `Node` at level i also appears at level $i + 1$. All nodes appear at level 0. The probability that a `Node` at level 0 also appears at level $i > 0$ is p^i. For example, with $p = 1/2$, $1/2$ of the nodes are expected to appear at level 1, $1/4$ at level 2, and so on, providing a balancing property like the classical sequential tree-based search structures, except without the need for complex global restructuring. This random generation ensures that the list structure remains balanced. Consequently, skip list insertion and deletion algorithms are much simpler and faster than equivalent algorithms for balanced trees.

Search Operation: To search for an element, the algorithm starts at the topmost layer and moves horizontally through the elements of that layer. When it encounters an element greater than the target, it drops to the next lower layer and continues the search horizontally. This process of horizontal traversal and vertical descent continues until the target element is found or the search reaches the bottom-most layer without finding the target. The hierarchical structure allows for logarithmic search time.

Insertion and Deletion: Inserting an element involves locating the appropriate position in the bottom-most layer, placing the element there, then potentially promoting the element to higher layers. Each promotion step is independent, ensuring the probabilistic balancing of the structure. Deleting an element requires removing it from all layers in which it appears, which is straightforward once the element is located using the search algorithm. The process of updating references in multiple layers ensures that the skip list remains balanced and efficient for subsequent operations.

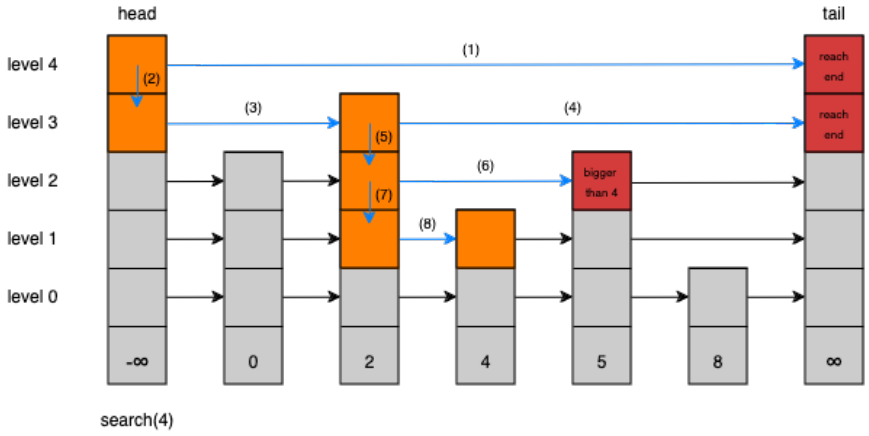


Figure 3.2: Skip List: In this example, the list searches for a `Node` with value 4. It starts on the head `Node` on the highest level, tries to move horizontally until it reaches a greater value than 4, and then goes down a level and repeats. The number noted on the arrows implies the order of the traversal.

Despite their theoretically poor worst-case performance, skip lists rarely exhibit worst-case behavior, making them efficient in most scenarios. For instance, in a dictionary with over 250 elements, the likelihood of a search taking more than three times the expected duration is less than one in a million [17]. Skip lists are ideal for implementing range locks, offering a balanced structure that improves concurrency.

Operation	**Best Case**	**Average Case**	**Worst Case**
Search, Insert, Delete	$O(1)$	$O(\log n)$	$O(n)$

Table 3.1: Time complexities of skip list operations

3.2 Concurrent Range Lock

We developed our concurrent range lock based on the design of `LockFreeSkipList` proposed by Herlihy et al. [13]. In summary, our concurrent range lock uses atomic operations (`compareAndSet()`) to manage `Node` references without locks, which enhances performance in multithreaded environments. When adding a `Node`, the process starts at the lowest level and moves upward to ensure immediate visibility. Removing a `Node` involves marking nodes from the top down before unlinking them. Furthermore, it relaxes strict structural maintenance of higher levels, focusing on the bottom-level list for set representation, which offers improved scalability and efficiency.

For the sake of simplicity, we use uint64_t for our pseudocode provided in this section. In our open-source C++ implementation, we use the template to enable generic programming.

3.2.1 Concurrent Range Lock API

The `ConcurrentRangeLock` class provides a concurrent mechanism to manage range-based locks. Its primary API includes methods for locking and unlocking ranges. Each range is stored in a single `Node`. The `tryLock` method attempts to acquire a lock for the specified range `[start, end]`, returning `true` on success and `false` otherwise. The `releaseLock` method releases the lock for the range `[start, end]`, with `true` indicating success and `false` if the range was not found or an error occurred.

The two primary methods rely heavily on `find` methods such as `findInsert`, `findExact`, and `findDelete`, which handle insertion finding, exact range finding, and physical deletion of ranges, respectively. We will discuss these methods in section 3.2.4.

```
1  class ConcurrentRangeLock {
2    public:
3      ConcurrentRangeLock();
4
5      bool tryLock(uint64_t start, uint64_t end);
6      bool releaseLock(uint64_t start, uint64_t end);
7
8    private:
9      Node *head, *tail;
10
11     int randomLevel();
12     bool findInsert(uint64_t start, uint64_t end, Node **preds, Node **succs);
13     bool findExact(uint64_t start, uint64_t end, Node **preds, Node **succs);
14     void findDelete(uint64_t start, uint64_t end);
15 };
```

Listing 3.1: Concurrent Range Lock

3.2.2 Node

Let first understand the class `Node`, the base of our `ConcurrentRangeLock` structure.

Each `Node` contains `start` and `end`, which represents range. `Node` also uses an array of `AtomicMarkableReference` (more details in section 3.2.3) to maintain forward links at each level, which allows for efficient traversal and updates. `Node` provides the following methods:

- `initialize`: sets up a `Node` with specific range and level values.
- `initializeHead`: configures the head `Node` with forward pointers directed to a provided tail `Node`, establishing the initial structure.
- `getTopLevel()`, `getStart()`, `getEnd()`: accessor methods to retrieve the Node's properties.

```
1  class Node {
2    private:
3      uint64_t start, end;
4      int topLevel;
5      AtomicMarkableReference<Node>** next;
6
7    public:
8      Node() : next(nullptr) {}
9
10     void initialize(uint64_t start, uint64_t end, int topLevel) {
11         this->start = start; this->end = end;
12         this->topLevel = topLevel;
13
14         next = new AtomicMarkableReference<Node>*[topLevel + 1];
15         for (int i = 0; i <= topLevel; ++i) {
16             next[i] = new AtomicMarkableReference<Node>();
17         }
18     }
19
20     void initializeHead(uint64_t start, uint64_t end, int topLevel, Node* tail) {
21         initialize(start, end, topLevel);
22         for (int i = 0; i <= topLevel; ++i){
23             next[i]->store(tail, false);
24         }
25     }
26
27     int getTopLevel() const { return topLevel; }
28     uint64_t getStart() const { return start; }
29     uint64_t getEnd() const { return end; }
30 };
```

Listing 3.2: Node class implementation

3.2.3 AtomicMarkableReference

`AtomicMarkableReference` is designed to maintains an object reference (in this case `Node`) along with a mark bit, that can be updated atomically [18].

The `AtomicMarkableReference` class uses a single atomic variable, `atomicRefMark`, to store a packed representation of the reference and the mark. These values are packed and unpacked using bitwise operations, where the least significant bit represents the mark. The class offers several atomic methods:

- `store`: directly sets the reference and mark.
- `compareAndSet`: tests if the current reference and mark match the expected values and, if so, updates them to new values atomically.
- `attemptMark`: sets the mark if the reference matches the expected value.
- `get`: returns both the object's reference and sets the mark reference.
- `getReference`: returns just the reference of the object.

```
1  const uintptr_t MARK_MASK = 0x1;
2
3  class AtomicMarkableReference {
4    private:
5      std::atomic<uintptr_t> atomicRefMark;
6
7      uintptr_t pack(Node* ref, bool mark) const {
8          return reinterpret_cast<uintptr_t>(ref) | (mark ? MARK_MASK : 0);
9      }
10
11     std::pair<Node*, bool> unpack(uintptr_t packed) const {
12         return {reinterpret_cast<Node*>(packed & ~MARK_MASK), packed & MARK_MASK};
13     }
14
15   public:
16     AtomicMarkableReference() {
17         atomicRefMark.store(pack(nullptr, false), std::memory_order_relaxed);
18     }
19
20     void store(Node* ref, bool mark) {
21         atomicRefMark.store(pack(ref, mark), std::memory_order_relaxed);
22     }
23
24     bool compareAndSet(Node* expectedRef, Node* newRef, bool expectedMark, bool newMark) {
25         return atomicRefMark.compare_exchange_strong(
26             pack(expectedRef, expectedMark), pack(newRef, newMark), std::memory_order_acq_rel);
27     }
28
29     bool attemptMark(Node* expectedRef, bool newMark) {
30         auto [currentRef, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
```

```
31          if (currentRef == expectedRef && currentMark != newMark) {
32              return atomicRefMark.compare_exchange_strong(
33                  current, pack(expectedRef, newMark), std::memory_order_acq_rel);
34          }
35          return false;
36      }
37
38      Node* get(bool* mark) const {
39          auto [ref, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
40          mark[0] = currentMark;
41          return ref;
42      }
43
44      Node* getReference() const {
45          auto [ref, _] = unpack(atomicRefMark.load(std::memory_order_acquire));
46          return ref;
47      }
48  };
```

Listing 3.3: AtomicMarkableReference class implementation

3.2.4 Find

Both `tryLock` and `releaseLock` methods rely heavily on `find` methods. There are several find methods in our implementation that serve different purposes:

- `bool findInsert(uint64_t start, uint64_t end, Node** preds, Node** succs)`: checks if the target range `[start, end]` is free to be inserted.
- `bool findExact(uint64_t start, uint64_t end, Node** preds, Node** succs)`: checks if the target range `[start, end]` is already present in the skip list.
- `void findDelete(uint64_t start, uint64_t end)`: finds the target range `[start, end]` from the skip list to physically delete the Node which contains the corresponding range.

These `findInsert` and `findExact` methods also fill in the `preds[]` and `succs[]` arrays with the target node's ostensible predecessors and successors at each level. Because the goal of `findDelete` is only to snip out all the deleted Node, there is no need to fill any array.

Nevertheless, these methods have to maintain the following two properties:

- During traversal, they need to skip over marked nodes. They use `compareAndSet()` (as discussed in 3.2.3) to ensure that remove all softly deleted Node on the way.
- Every `preds[]` reference is to a node with a key strictly less than the target.

```
1  bool ConcurrentRangeLock::find(uint64_t start, uint64_t end,
2      Node **preds, Node **succs) {
3      bool marked[1] = {false};
4      bool snip;
5      Node *pred, *succ, *curr;
6
7    retry:
8      while (true) {
9      pred = head;
10     for (int level = maxLevel; level >= 0; level--) {
11         curr = pred->next[level]->getReference();
12
13         while (start > curr->getStart()) {
14             succ = curr->next[level]->get(marked);
15             while (marked[0]) {
16                 snip = pred->next[level]->compareAndSet(curr, succ, false, false);
17
18                 if (!snip) goto retry;
19
20                 curr = pred->next[level]->getReference();
21                 succ = curr->next[level]->get(marked);
22             }
23             if (start >= curr->getStart()) {
24                 pred = curr;
25                 curr = succ;
26             } else {
27                 break;
28             }
29         }
30
31         preds[level] = pred;
32         succs[level] = curr;
33     }
34
35     return **condition**;
36     }
37 }
```

Listing 3.4: Find method implementation

Algorithm in details

The `find()` method starts by traversing the LockFreeSkipList from the `topLevel` of the head sentinel, which has the maximal allowed node level. It proceeds down the list level by level, filling in the `preds` and `succs` nodes. These nodes are repeatedly advanced until `pred` refers to a node with the largest value on that level that is strictly less than the target key (Lines 13–29).

While traversing, it repeatedly snips out marked nodes from the current level as they are encountered (Lines 15–22) using a `compareAndSet()`. The `compareAndSet()`

function validates that the following field of the predecessor still references the current Node.

Once an unmarked `curr` node is found (Line 23), it is tested to see if its `start` is greater than or equal to the target `start`. If so, `pred` is advanced to `curr`, `curr` is advanced to `succ`, and the traverse continues. Otherwise, the current range of `pred` is the immediate predecessor of the target node. The `find()` method then breaks out of the current level search loop, saving the current values of `pred` and `curr` (Line 26–32).

The `find()` method continues this process until it reaches the bottom level. An important point is that each level's traversal maintains the previously described properties. Specifically, if a node with the target key is in the list, it will be found at the bottom level even if nodes are removed at higher levels. When traversal stops, `pred` refers to a predecessor of the target node. The method descends to each next lower level without skipping over the target node. If the Node is in the list, it will be found at the bottom level. Additionally, if the Node is found, it cannot be marked because if it were marked, it would have been snipped out in Lines 15–22. Thus, the condition test on line 35 only needs to check if there are overlap ranges (for `findInsert`) or if the start and end of the Node match the target start and end.

- condition of findInsert
- condition of findExact

The linearization points for both successful and unsuccessful calls to the `find()` method occur when the `curr` reference at the bottom-level list is set, either at Line 11 or Line 20, for the last time before the success or failure of the `find()` call is determined at Line 35.

4 Evaluation

In this section, we will evaluate our proposed concurrent range lock under diffrerents senarios. The goal is to see the scalability, throughtput as well as the latency of our mechanism comparing to state-of-the-art range lock.

4.1 Competitor

In addition to our version of range lock, we implement two diffrerent version. One scalable range lock proposed by Kogan et al. [8], denoted as `ver2`. Other range lock proposed by Song et al. [10], denoted as `ver3`.

4.2 Microbenchmark

4.2.1 Benchmark enviroment

4.2.2 Benchmark variants

The primary objective of a concurrent range lock mechanism is to enable multiple threads to access disjoint parts of the same shared object efficiently. To simulate and evaluate the effectiveness of different range locking strategies, we utilize the `mmap()` system call to create a shared object in memory, and the `memset()` function to simulate write operations to this shared object. The use of `memset()` serves as a placeholder for actual modifications, enabling us to focus on the performance characteristics of the locking mechanism itself.

To explore various levels of contention and potential usage scenarios for range locks, we have devised three distinct variants, each designed to stress the locking mechanism under different conditions:

- `V1`: In this variant, each thread accumulates a list of ranges that it intends to modify. Once all ranges in the working queue are identified, the thread locks all these ranges simultaneously, performs the necessary modifications, and then

releases all the locks at the same time. This approach is designed to simulate scenarios where threads must work with a large number of ranges concurrently, testing the scalability of the range lock mechanism when it is required to manage a significant number of simultaneous locks. The primary goal of `V1` is to assess how well the range lock can handle heavy contention when multiple threads attempt to lock numerous ranges at once.

- `V2`**:** In this variant, each thread operates in a more granular fashion. A thread locks a single range, performs the modification (simulated by `memset()`), and immediately releases the lock before moving on to the next range in its working queue. This variant simulates a scenario with minimal contention, where the range lock mechanism is tested for its efficiency in handling rapid lock acquisition and release cycles. The objective here is to observe the overhead introduced by the locking mechanism when contention is low and to evaluate the lock's performance in scenarios that demand high throughput with minimal waiting times.
- `V3`**:** The third variant introduces a hybrid approach, combining elements of both `V1` and `V2`. In this variant, a thread locks a range and then releases it immediately with a probability of 95%. In the remaining 5% of cases, the thread holds onto the lock and not release. This variant is designed to create a scenario where the lock occasionally becomes more congested, thereby allowing us to evaluate the efficiency of the range lock under conditions where some ranges are held longer, potentially causing increased contention. The purpose of `V3` is to simulate real-world scenarios where not all operations are equal in duration, and some threads may need to hold locks for longer, testing the lock's ability to handle such situations without significant performance degradation.

Through these variants, we aim to comprehensively evaluate the performance of the concurrent range lock mechanism under different workloads and contention scenarios. Each variant provides insights into specific aspects of the lock's scalability, efficiency, and overall robustness in handling varying degrees of parallelism and contention.

4.2.3 Optimal Height for Range Locking

The height of the skip list in our range locking mechanism plays a critical role in balancing performance and resource utilization. A skip list with insufficient height may fail to optimize lookup times effectively, leading to slower performance, while an excessive height increases memory consumption and management overhead without proportionate gains in efficiency. This test aims to determine the optimal height for our

concurrent range locking mechanism before comparing it to alternative approaches. We conducted experiments using the three variants described in 4.2.2 to identify the best skip list height.

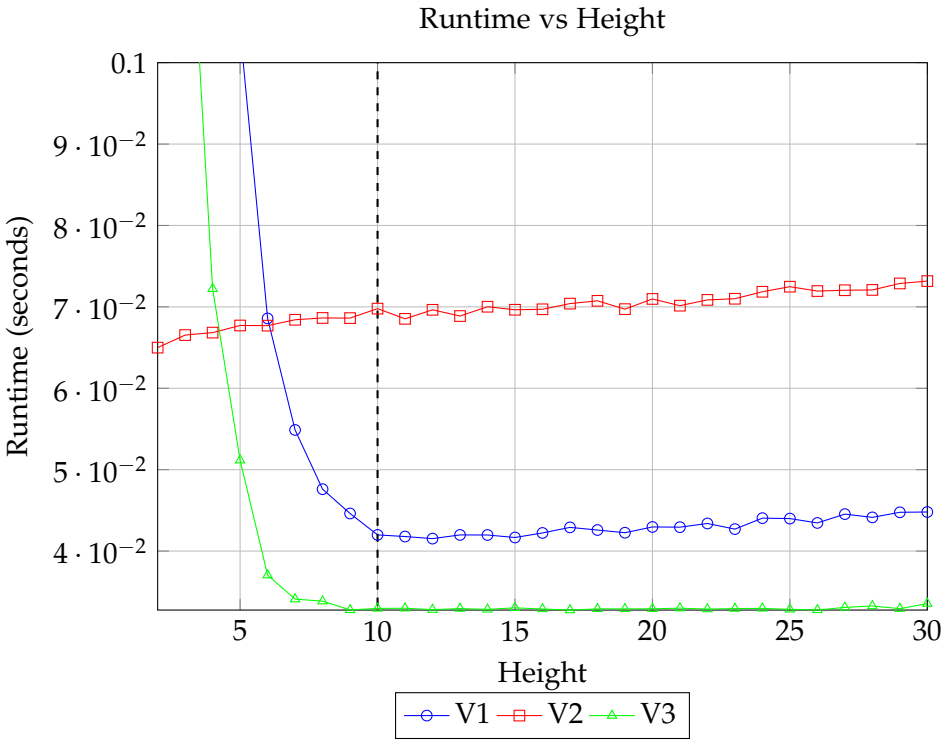


Figure 4.1: Runtime for different heights of range lock

For both **V1** and **V3**, we observe that total runtime decreases as the height of the skip list increases, which is consistent with the expected behavior of skip lists. The skip list's multi-level structure allows for faster lookups as the height increases, reducing the overall runtime. However, beyond a height of 10, we notice a plateau where increasing the height yields diminishing returns. This phenomenon can be explained by the probabilistic nature of skip list level assignment, as detailed in 3.1. Since the probability of a node being elevated to a higher level decreases linearly. For example, for level 12, the probability is approximately $\frac{1}{2^{12}} \approx 0.00024\%$. The uppermost layers of the skip list become sparsely populated, rendering them less impactful on performance. Despite this, these higher levels still contribute to increased memory usage without significant performance benefits.

For **V2**, where contention is minimal and each thread locks and releases ranges immediately, the height of the skip list has negligible impact on performance. In this scenario,

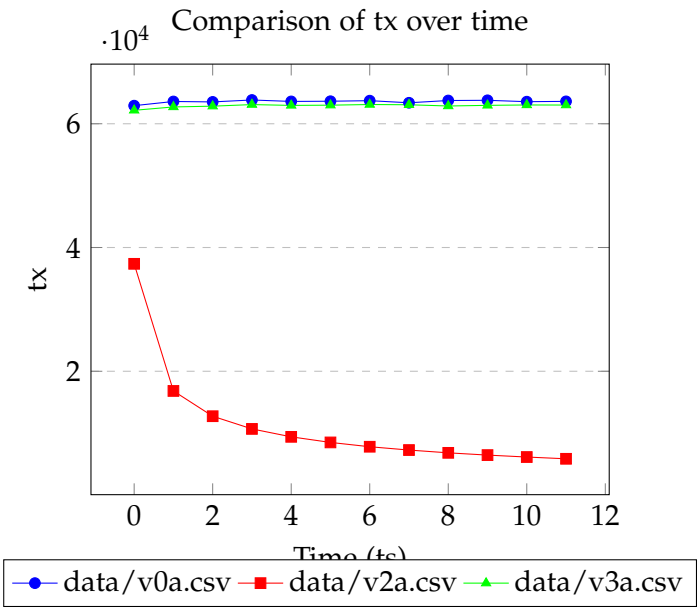
since the number of ranges locked at any given time is limited to the number of active worker threads, the benefits of a taller skip list are not realized. The overhead of managing additional levels outweighs any potential gains, leading to similar performance across different heights.

Based on these observations, we identify a sweet spot at a height of `10`, where the trade-off between performance and resource utilization is optimized. This height balances the need for efficient lookups with manageable memory overhead, making it the ideal choice for our range locking mechanism in subsequent benchmarks.

4.2.4 leanstore

In order to simulate a realistic benchmarks, we integrated all kind of range locks into `leanstore` [19], a high-performance OLTP storage engine optimized for many-core CPUs and NVMe SSDs.

The experiment, utilizing synthetic YCSB workloads, was designed to evaluate the performance of LeanStore under a read-intensive, multithreaded environment. The workload comprises exclusively read operations, executed over a 10-second duration, with each read operation employing a straightforward `memcpy()` function. Each record in the workload has a payload size of 1 MB, with a total of 1000 records being processed. The experiment also incorporates a buffer management configuration that allocates 128 GB of virtual memory alongside 32 GB of physical memory, providing a robust test of LeanStore's ability to manage large data sets under constrained physical memory conditions.


Comparison of tx over time
·10^4
6
4
2
tx
0
2
4
6
8
10
12
Time (ts)
data/v0a.csv
data/v2a.csv
data/v3a.csv