CHAIR OF DECENTRALIZED
INFORMATION SYSTEMS & DATA
MANAGEMENT

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Concurrent Range Locking

Thua-Duc Nguyen

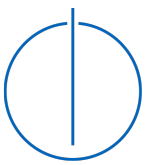
CHAIR OF DECENTRALIZED INFORMATION SYSTEMS & DATA MANAGEMENT

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

In modern computing environments, efficient locking mechanisms are vital for the performance of databases, file systems, and operating systems. Traditional single-lock techniques often lead to performance bottlenecks in high-concurrency scenarios. Range locks offer a refined approach by partitioning shared resources into multiple segments, each of which can be exclusively acquired by different processes, thus improving performance.

As database sizes grow, locking the entire database is impractical due to poor throughput and high latency. Existing key-range locking methods are complex and not suitable for general DBMS operations. Therefore, a new technique, such as range locks, is necessary.

Previous approaches, including the Linux kernel's range tree with an internal spinlock and skip lists with spinlocks, face contention issues. A lock-free range lock using a concurrent linked list has been proposed, but it suffers from slow insertion and lookup operations.

This research proposes a new concurrent range-locking design leveraging a probabilistic concurrent skip list with per-node locks, addressing previous bottlenecks and 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 provide a comprehensive assessment of its effectiveness.

Contents

List of Figures

List of Tables

1 Introduction

Locking mechanism is important. In modern computing environments, the efficiency of locking mechanisms play a crucial role in the performance of various systems, including databases [1, 2], file systems [3, 4, 5], and operating systems [6, 7]. As these systems grow in scale and complexity, the demand for more sophisticated and efficient locking mechanisms becomes crucial. One of the fundamental challenges in this context is managing concurrent access to shared resources. Traditional locking techniques, such as single-lock mechanisms, often lead to significant performance bottlenecks, particularly in high-concurrency scenarios.

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

DBMS needs range locks. As database sizes increase exponentially, locking the entire database becomes impractical. This approach will prevent concurrent transactions from progressing, resulting in poor throughput and high latency. The previous key-range locking in DBMS is complex and tightly coupled with lock-based concurrency control protocols [2, 9]. Consequently, this technique is not applicable to general DBMS operations, such as variable-sized page allocation. Therefore, a new technique, such as range locks is desirable.

Existing range lock need 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 [10, 11]. Additionally, some methods may be complex and tightly coupled with lock-based concurrency control protocols, which are not applicable for general DBMS operations [2, 9]. 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 concurrent range-locking design that leverages a probabilistic concurrent skip list [12, 13]. Our range lock design also utilizes the per-node lock technique instead of an interval lock, thus addressing the previous range lock bottleneck problem and maintaining the lock's high level of performance.[1]

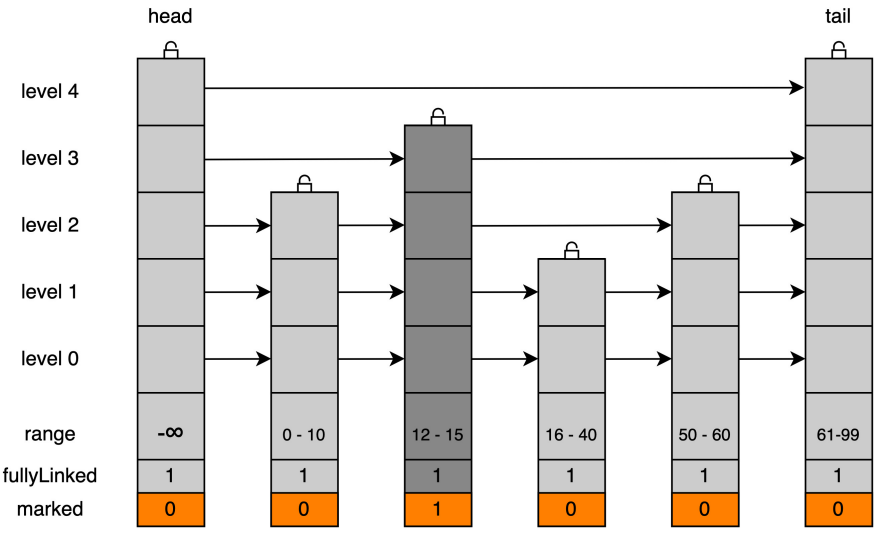


Figure 1.1: Concurrent range lock

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.

[1] More details in approach chapter

2 Related work

This chapter reviews the existing research on range locks, focusing on different aspects of the problem and the solutions researchers have proposed. The following sections provide a detailed examination of different implementations, highlighting their limitations. This analysis aims to show the evolution of range-locking mechanisms and the ongoing efforts to improve their performance and scalability.

2.1 Range Lock in the Linux Kernel

Jan Kara introduced a range-locking mechanism for the Linux kernel [10]. The implementation leverages an interval tree to manage range locks and employs a spin lock for synchronization. Each lock is represented as a node in the tree.

How it works

When a thread wants to acquire a range, it first acquires a spinlock. Then, it traverses the tree to determine the number of locks intersecting with the given range and find the. Next, the thread inserts a node describing its range into the tree and releases the spinlock. If the number of intersecting locks is zero, the thread can access and write to the critical section it acquires. Otherwise, it waits until those locks are released when the number of intersecting locks drops to zero. When a thread completes its range, it acquires the spin lock, removes its node from the tree, updates the number of overlapping locks, and frees the spinlock. It guarantees that each range is locked only after all previous conflicting range locks requested have been unlocked, thus achieving fairness and avoiding livelocks.

Drawbacks

Single Point of Contention. On this range lock, all operations relies on a single spinlock to protect the entire tree. Under heavy concurrent access, this easily becomes a contention point, limiting the system's performance.

Limitation of FIFO Order. 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 the FIFO restriction. This unnecessary blocking reduces the overall efficiency and concurrency of the system.

2.2 Skip List-Based Range Lock

Song et al. [11] introduced a dynamic and fine-grained range-locking design to enhance the implementation of the Linux kernel. Their range lock utilizes a skip list [14] to dynamically manage the address ranges that are currently locked.

How it works

When a thread requests a specific range `[start, start+len)`, the range lock searches for it in the skip list. If an existing or overlapping range is found in the skip list, it means that another thread is currently modifying the specific range, and the requesting thread must wait and then retry. If no overlapping range is found, the range is added to the skip list, indicating that the range 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, and it can efficiently perform searches for overlapping ranges.

Drawbacks

Single Contention Point: Similar to the one found in the Linux kernel, this range lock is also protected by a spinlock. Hence, the same bottleneck issue still need to be addressed

2.3 Concurrent Linked List-Base 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 some 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.

How it works

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 the previous one is that a 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 reaches a marked node, it simply removes it using CAS and continues to iterate. If the current node is protecting a range that overlaps, it simply 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.

Drawbacks

Linked List Inefficiency. This design implements a lock-free mechanism that effectively addresses the limitations of existing range locks. However, this approach comes with its own set of trade-offs. In general, insertion and lookup operations in a linked list are less efficient than 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 [15].

2.4 Comparative Analysis and Trade-offs

The analysis of range lock implementations reveals distinct trade-offs in terms of performance, scalability, and complexity. The interval tree-based range lock in the Linux kernel, despite its fairness and simplicity, suffers from a significant bottleneck due to its reliance on a single spinlock, which can severely limit performance under high contention. The skip list-based range lock offers improved efficiency and dynamic management of address ranges, leveraging the lightweight and fast nature of skip lists. However, it still retains the single contention point, similar to the interval tree approach, which hinders its scalability under heavy concurrent access. The concurrent linked list-based range lock by Kogan et al. presents a lock-free mechanism that significantly enhances scalability by eliminating blocking behavior. Nonetheless, it introduces

inefficiencies in insertion and lookup operations, as linked lists inherently have a higher average time complexity compared to skip lists and tree-like structures.

2.5 Future Directions

Future research on range-locking mechanisms should focus on hybrid approaches that combine the advantages of different data structures to mitigate their individual weaknesses. For instance, exploring the integration of lock-free principles with more efficient data structures, such as combining skip lists with lock-free operations, could enhance both performance and scalability. Additionally, investigating adaptive range-locking mechanisms that dynamically switch between different data structures based on the current contention level and workload characteristics could provide optimized performance across varying conditions. Finally, further studies on distributed range-locking techniques could extend these mechanisms to multi-node environments, addressing the growing need for scalable synchronization in distributed systems. By advancing these directions, researchers can develop more robust and efficient range-locking solutions that cater to the demands of modern computing environments.

3 Approach

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

- **try_lock**: The `try_lock` function searches for the required range `[start, start+len)` 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.
- **release_lock**: The `release_lock` function releases the lock by finding the address range in the skip list and removing it accordingly.

Our range lock design also utilizes the per-node lock instead of an interval lock, 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 that allows fast search, insertion, and deletion. It is an alternative to balanced trees, such as AVL trees or red-black trees [14, 16]. 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

Layers: A skip list consists of multiple layers. The bottom-most layer is a regular sorted linked list. Each higher layer acts as an "express lane" to speed up access by skipping over multiple elements from the layer below.

Probabilistic Balancing: When a new element is inserted, a node with a random height is generated. 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.

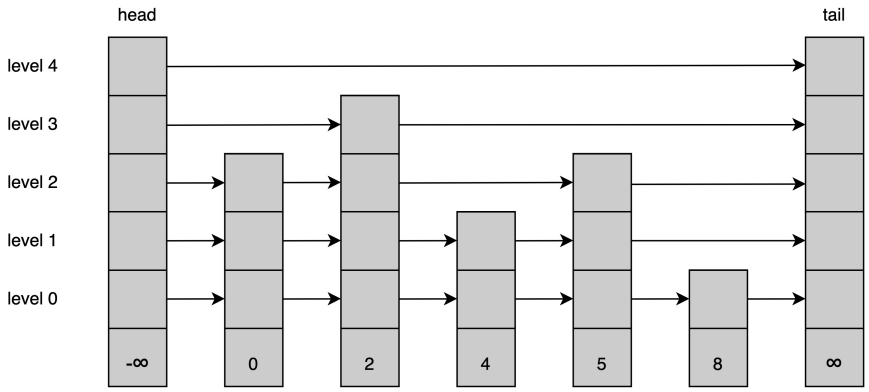


Figure 3.1: Skip List: this example has five levels of sorted linked lists.
Each node has an unique key, and the head and tail have $-\infty$ and $+\infty$ keys.

Search Operation: To search for an element, the search starts at the top-most layer and moves horizontally until it finds an element greater than or equal to the target element. If it finds an element greater than the target, it drops to the next lower layer and continues the search. This process repeats until the element is found or the search reaches the bottom-most layer without finding the target.

Insertion and Deletion: Inserting an element involves placing it in the appropriate position in the bottom-most layer and then possibly promoting it to higher layers based on the coin flips. Deleting an element involves removing it from all layers in which it appears.

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 [16]. 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

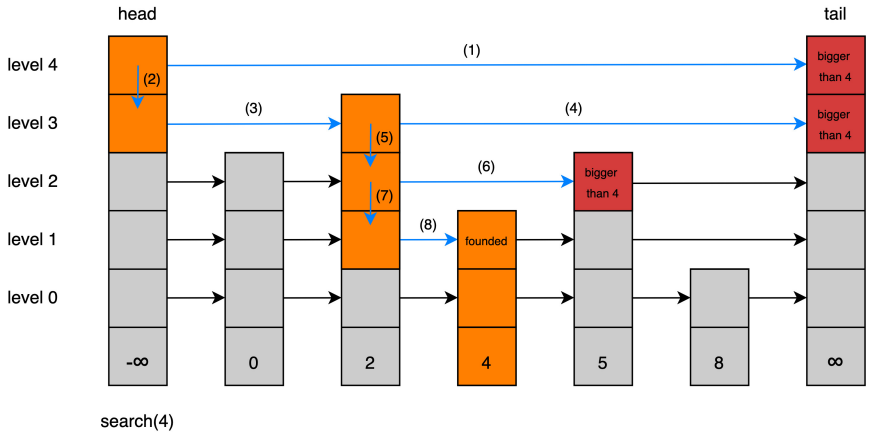


Figure 3.2: Skip List: In this example, the list searches for a node with value 4. It starts on the head node on the highest level, tries to move horizontally until it reaches a greater value than 4, and then goes down a level and repeats. The number noted on the arrows implies the order of the traversal.

3.2 Concurrent Range Lock

We developed our concurrent range lock based on the LazySkipList technique proposed by Herlihy et al. [13]. In their work, the authors introduced a LazySkipList. To summarize, the LazySkipList holds lock on all nodes to be modified, validates that nothing important has changed, completes the modifications, and releases the locks (in this context, the fullyLinked flag acts like a lock). We made some modifications to adapt this structure to our concurrent range lock.

3.2.1 Node

The structure of a node is designed to facilitate efficient concurrent operations. Each node contains several fields: start and `end`, which represent the aquired range associated with the node; `topLevel`, an integer indicating the highest level of the skip list in which this node appears; and `next`, an array of node pointers, where each element points to the next node at the corresponding level of the skip list. A node's i^{th} next pointer points to the next node at level i or higher.

Additionally, the node has two boolean flags, `marked` and `fullyLinked`. The `marked` flag indicates whether the node has been logically removed from the skip list, while the `fullyLinked` flag signifies whether the node has been fully integrated into all its

intended levels in the skip list. Moreover, to ensure thread safety during concurrent operations, each node includes a recursive mutex to control access.

The node's methods include a constructor for initialization, a destructor for cleanup, and lock and unlock methods to manage the mutex. There are also methods to retrieve the node's topLevel, start, and end values, ensuring that the node's properties can be accessed and modified safely in a multi-threaded environment. This structure supports the node's participation in multiple levels of the skip list, enabling efficient and safe concurrent modifications.

```
struct Node {
    Node **next;
    bool marked = false;
    bool fullyLinked = false;

  private:
    T start; T end;
    int topLevel;
    mutable std::recursive_mutex mutex;
};

Node<T>::Node(T start, T end, int level) : start{start}, end{end},
    topLevel{level} {
    next = new Node<T>*[level + 1];
}

void Node<T>::lock() { mutex.lock(); }

void Node<T>::unlock() { mutex.unlock(); }

int Node<T>::getTopLevel() const { return topLevel; }

T Node<T>::getStart() const { return start; }

T Node<T>::getEnd() const { return end; }
```

Listing 3.1: Node structure

4 Evaluation

The proposed approach will be evaluated under these evaluation criteria:

- **Performance:** We will test the range lock mechanism under increasing load and concurrent accesses to measure its performance.
- **Correctness:** We will ensure the consistency and correctness of data accesses, especially when there are overlapping data ranges and concurrent operations.
- **Comparison:** We will compare the performance of the proposed solution with existing state-of-the-art approaches.

5 Result

We aim to develop a scalable range lock that performs better than the existing range locks. The evaluation results will provide insights into the performance characteristics and potential trade-offs of the proposed mechanism.

6 Conclusion

Conclusion

Bibliography

[1] D. B. Lomet. *Key range locking strategies for improved concurrency*. Digital Equipment Corporation, Cambridge Research Laboratory UK, 1993.

[2] G. Graefe. "Hierarchical locking in B-tree indexes". In: *On Transactional Concurrency Control*. Springer, 2007, pp. 45–73.

[3] C.-G. Lee, S. Noh, H. Kang, S. Hwang, and Y. Kim. "Concurrent file metadata structure using readers-writer lock". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, pp. 1172–1181.

[4] J. Gao, Y. Lu, M. Xie, Q. Wang, and J. Shu. "Citron: Distributed Range Lock Management with One-sided {RDMA}". In: *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 2023, pp. 297–314.

[5] L. Chang-Gyu, B. Hyunki, N. Sunghyun, K. Hyeongu, and Y. Kim. "Write optimization of log-structured flash file system for parallel I/O on manycore servers". In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. 2019, pp. 21–32.

[6] J. Corbet. "Range reader/writer locks for the kernel". In: *LWN.net* (2022). Accessed: 2024-04-21. URL: https://lwn.net/Articles/724502/.

[7] L. Dufour. "Replace mmap_sem by a range lock". In: *LWN.net* (2017). Accessed: 2024-04-21. URL: https://lwn.net/Articles/723648/.

[8] A. Kogan, D. Dice, and S. Issa. "Scalable range locks for scalable address spaces and beyond". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.

[9] A. Pavlo. "Two-Phase Locking". In: *15445.courses.cs.cmu.edu* (2022). Accessed: 2024-04-21. URL: https://15445.courses.cs.cmu.edu/fall2022/slides/16-twophaselocking.pdf.

[10] J. Kara. "Implement range locks". In: *lkml.org* (2013). Accessed: 2024-04-21. URL: https://lkml.org/lkml/2013/1/31/483.

[11] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. "Parallelizing live migration of virtual machines". In: *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2013, pp. 85–96.

[12] H. Maurice, L. Yossi, L. Victor, and S. Nir. "A provably correct scalable concurrent skip list". In: *Conference On Principles of Distributed Systems (OPODIS). Citeseer*. Vol. 103. 2006.

[13] H. Maurice, S. Nir, L. Victor, and S. Michael. *The art of multiprocessor programming*. Newnes, 2020.

[14] W. Pugh. *A skip list cookbook*. Citeseer, 1990.

[15] M. Fomitchev and E. Ruppert. "Lock-free linked lists and skip lists". In: *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. 2004, pp. 50–59.

[16] W. Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676.