Concurrent B+Trees - RCU Approach

Pranjal Patil, Manisha Dodeja, Semil Jain

1) Abstract

The project "Concurrent B+Trees - RCU Approach" seeks to address a critical challenge in multi-threaded operations within B-Tree data structures. Currently, employing locks to prevent conflicts among threads leads to performance degradation and heightened code complexity. We aim to devise a methodology that enables B+Trees to seamlessly accommodate multiple threads without relying on locks or intricate blocking mechanisms. Leveraging fundamental Operating System principles for synchronization, we aim to enhance data operation efficiency while streamlining code intricacies, thus ensuring greater accessibility and comprehension for users.

Performance bottlenecks arise in systems utilizing in-memory indexes like B+trees with multiple threads due to lock-induced thread wait times. Traditional approaches to enhancing lock functionality exist, yet our unique strategy aims to eliminate lock dependency. Our non-blocking approach seeks to outperform conventional techniques such as reader-writer locks and optimistic concurrency control. This innovation holds the promise of significantly enhancing system performance and responsiveness reliant on B+tree data structures.

Our project's primary focus is to deepen the understanding of concurrency within B+ trees and introduce a non-blocking synchronization technique mediated by the Operating System Kernel. By circumventing locks and complex blocking methods, we aim to streamline B+Tree functionality amidst multiple threads.

2) Items Proposed

- Find a Production B+Tree and Implement a simpler version of that B+Tree with important operations
- Implement Reader-Writer lock on that B+ tree
- Implement the OCC version of the RWLock
- Implement a Lock-Free version based on an existing solution
- Implement Read-Copy-Update (RCU) for synchronization
- Benchmark the code using YCSB
- Compare the results for varying workloads

3) Items Achieved

- **Our B+Tree Implementation**
  The B-tree has the following properties:
  Maximum B elements in a node
  Min B/2 -1 element in a node
  Max B +1 children
  Min B/2 children

  Insert(uint64_t key, uint64_t val):

  In a B-tree, inserting into a full node,

  1. results in a split
  2. Addition of a pivot in the parent

  At this point, if the parent is also full, this process of splitting and appending a pivot in its parent will happen recursively until we find a space in a node or we reach the root.

  Therefore, in a worst-case scenario, inserting in a full node, where all the ancestors in the path are also full results in the modification of all nodes in the path and if we later employ multiple threads on the same tree, we would have to acquire write lock on the complete path starting from the root, which will make each thread wanting to insert wait until the previous insert operation completes. Due to the excessive waiting, this multithreading approach will perform the same as being single-threaded, i.e. we wouldn't achieve any parallelism.

  To overcome the above problem, we have built a preemptive B+ tree. Here we are looking to build a situation where a split in a node only affects itself and its direct parent at a moment in time.

  To achieve this i.e. to avoid propagation of splits beyond a node's direct parent, every time, we land on a node, we are ensuring it has an empty spot available. (so that if its child splits, it will have an empty spot to occupy the new pivot)

  If it has an empty spot available, Great!, we move ahead to its child.

  If it's full, we split it to make arrangements for the future, at this instant, we would need to append a pivot to its parent, which will guarantee having an empty spot because we must have made sure it does before we arrived at the current node.

Now, if we view this in a multi-threaded scenario, we would only need to acquire locks on two nodes at a time in a particular path, i.e. Parent and child node.

Delete(uint64_t key):

Similarly, to avoid recursive movement towards the top we have performed a preemptive B+ tree deletion. The main goal is to remove an element in the leaf node. If the leaf is empty this might cause an element removal from the parent. As we want at least B/2 -1 elements in the parent we might do 2 operations on the parent.

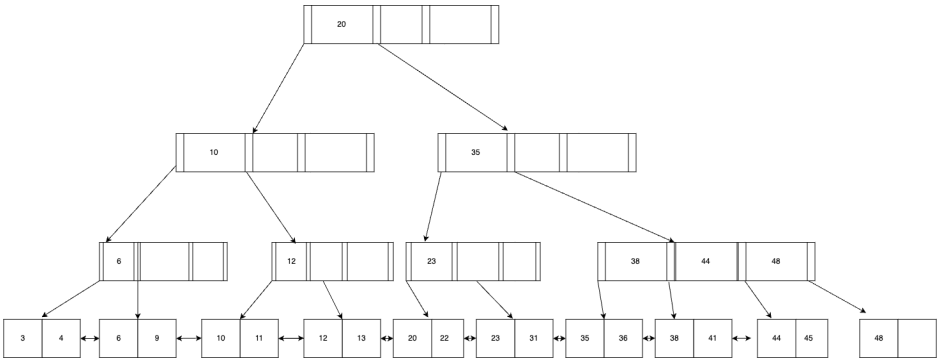
- Borrow from the same parent sibling (rotation)
- merge with its sibling.

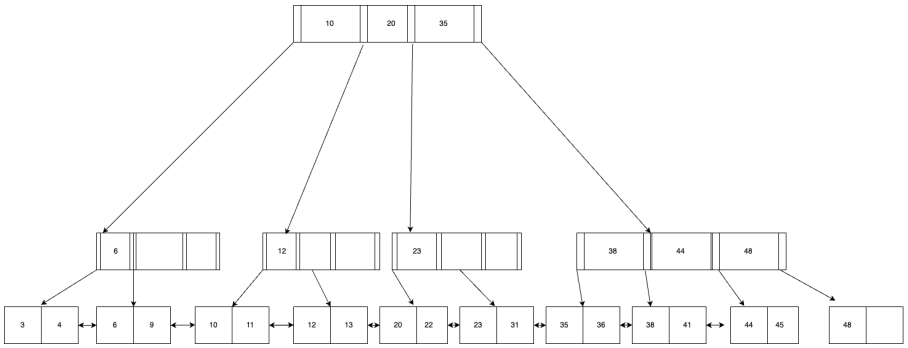
In either case, we are rebalancing the tree to ensure no intermediate node has less than a B/2-1 element.

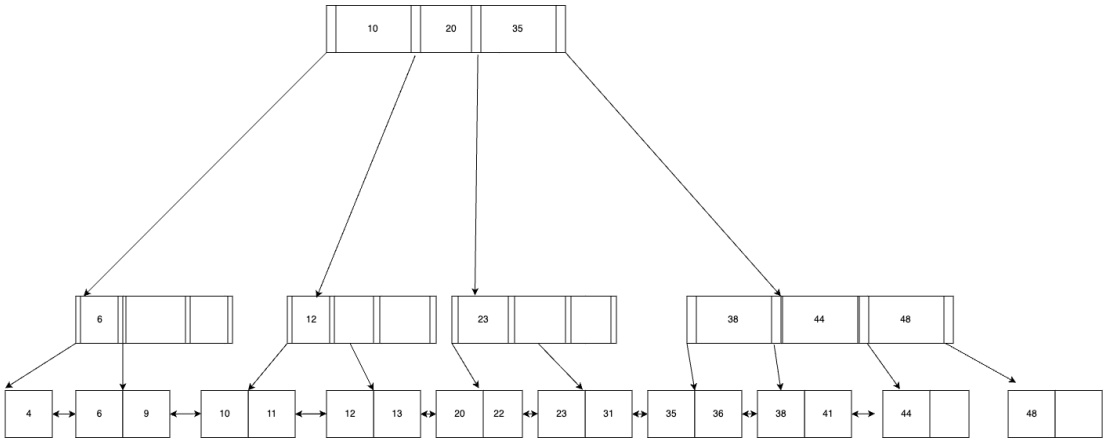
Here as we are moving down towards the leaf we are assuming the worst case and if a node has exactly min element we do either of the above two operations.

NOTE: Leaf nodes are connected as a doubly linked list and hence the insertions and deletions ensure that the linking is done correctly.

The below example represents deleting key 45.







Point Query (uint64_t key):

We start from the root and go to the child based on the key value. As we reach the child, we iterate through the node, and if we find the element return 1 0.

Range Query (uint64_t key1, uint64_t key2):

Here we are supposed to find all the data that is between key1 and key 2 inclusive. Here we first start with finding the key1 using point query. Once found we start iterating the doubly linked list from there until we find key2 or if key2 doesn't exist, all elements less than key2. Then return the vector that stored all those elements.

- **Our RWLock Implementation**

  Our chosen strategy for locking involves a hand-in-hand latching approach. At any given moment, both the parent and child nodes are latched. Once the child node is deemed safe (i.e., it will not split or merge during an update or the split/merger operation is done), the latch on the parent node is released, and the next child in the path is latched.

  Basic Idea:

  Acquire a latch for the parent node.

  Acquire a latch for the child node.

  Release the latch for the parent if the child is deemed "safe."

  A safe node will not split or merge during an update.

  Conditions for safety:

  Not full (on insertion) (< B elements)

  More than half-full (on deletion) (> B/2-1 elements)

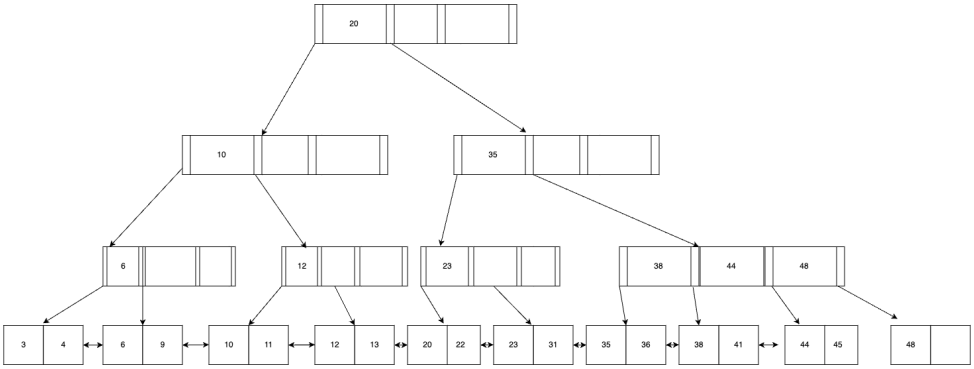
Implementation Strategy:

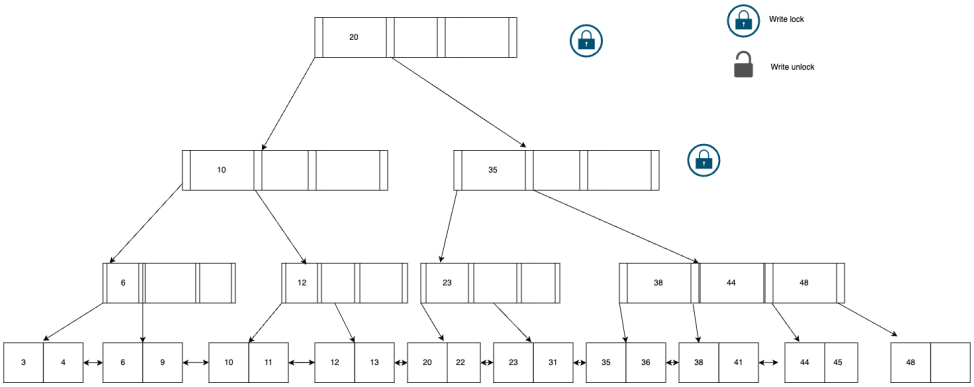
For both **Insert** and **Delete** operations, the process begins at the root and descends through the tree, obtaining Write (W) latches as necessary. Once a child node is latched, its safety is verified. If the child is safe, all latches on ancestors are released.

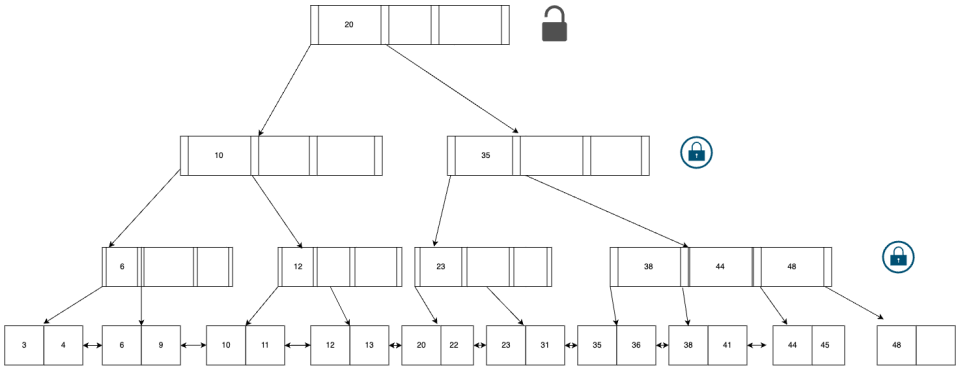
For **Point** and **Range Query** operations, the process also starts at the root, acquiring Read (R) latches on child nodes. Subsequently, the latch on the parent node is released.

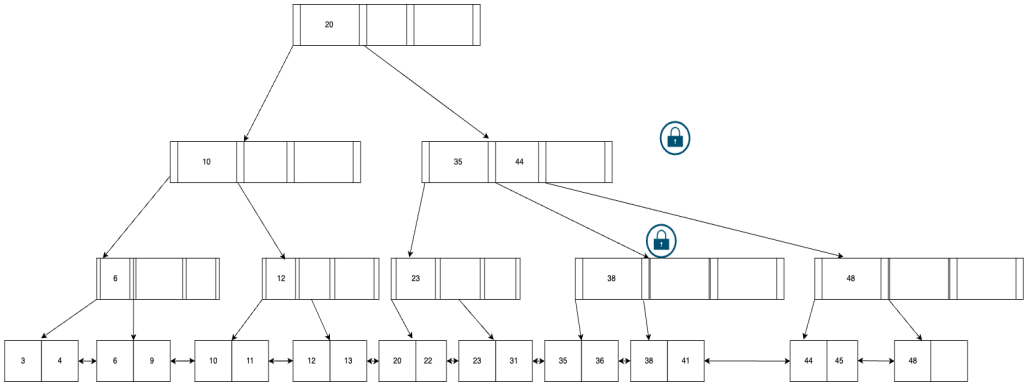
We also implemented the distributed counter as we did in Project 1.

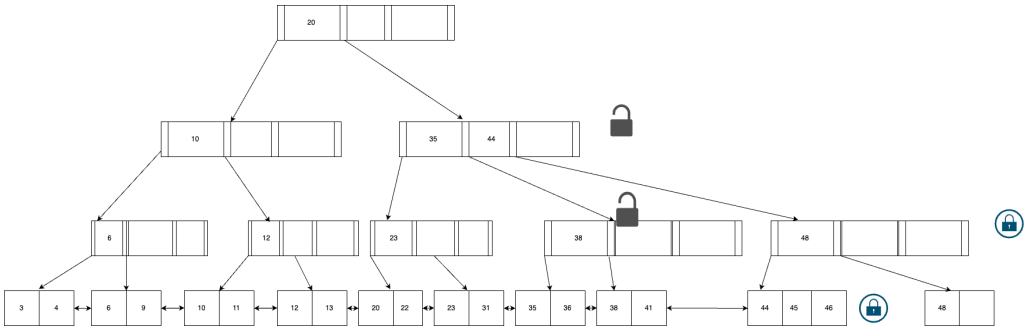
The below example represents inserting key 46.









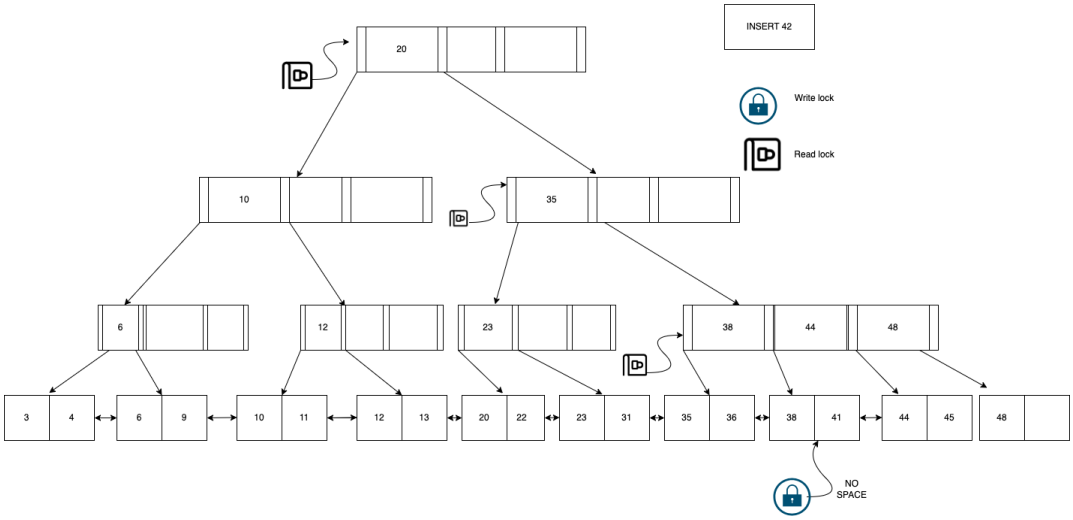


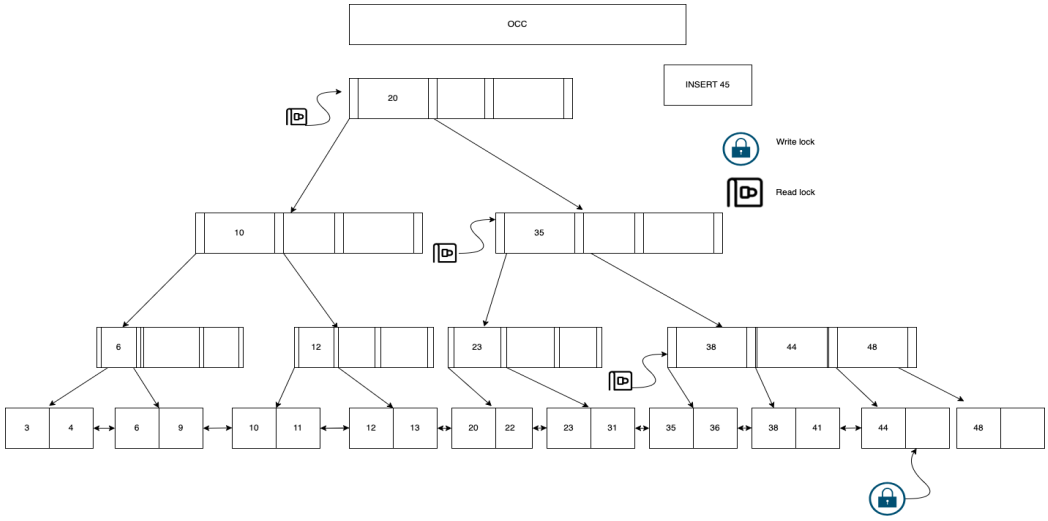
- **Our OCC Implementation**

  In OCC the basic idea is that first we go to the leaf and check if it is full or not. If it is not we directly insert our element. If it is full then we restart the insertion from the root with the preemptive implementation. Consider a case of insert.

  - First, we put a read lock on the root and traverse to the respective child. Then we release that lock and put a read on the child. This we do until we get to a leaf.
  - When we reach the leaf we put a write lock. Then we check if the node is full. If it is we start the insertion again by preemptive insertion.
  - If it is not then we directly insert it into the node.

  For removal, the process is the same but instead at the leaf, we check if there is only 1 element present or more than that. If only one is present then we start again with preemptive removal. Or else We directly remove it from the node.



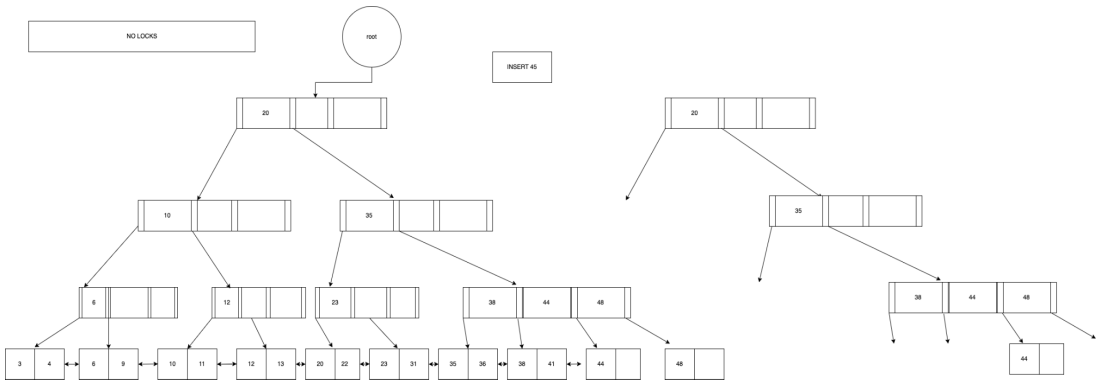


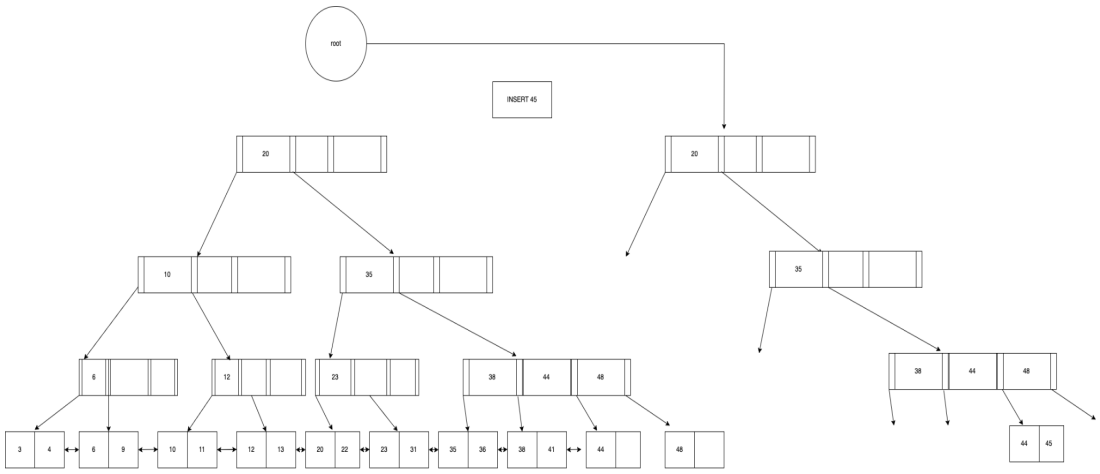
- **Our Lock-Free Implementation**

  In our lock-free implementation, reader threads are not blocked by the writer threads. Here are the steps for the insertion thread.

  - The insertion thread first copies the path from the root to the leaf where the element is to be inserted.
  - After copying those nodes it makes its changes in those copied nodes. These modifications include splitting/inserting in intermediate etc.
  - Once that is done we change the root of the tree to be the root of the copied one.
  - To ensure that no 2 write threads conflict we have used atomic compare and swap for the variable to confirm that only 1 thread is executed at a time.

  For remove it is the same process, but the modification of the copied nodes will be of merging/removing elements from the intermediate node.





- **Our RCU Implementation**

  RCU is a synchronization mechanism that was added to the Linux kernel during the 2.5 development effort and is optimized for read-mostly situations. Avoids the use of lock primitives while multiple threads concurrently read and update elements that are linked through pointers and that belong to shared data structures (e.g., linked lists, trees, hash tables).
  The basic idea behind RCU is to split updates into "removal" and "reclamation" phases. Splitting the update into removal and reclamation phases permits the updater to perform the removal phase immediately, and to defer the reclamation phase until all readers active during the removal phase have completed.

  The main APIs that we used to incorporate synchronization between threads are:

rcu_read_lock(): Marks an RCU-protected data structure so that it won't be reclaimed for the full duration of that critical section.
rcu_read_unlock(): Used by a reader to inform the reclaimer that the reader is exiting an RCU read-side critical section. Note that RCU read-side critical sections may be nested and/or overlapping.
synchronize_rcu(): Blocks until all pre-existing RCU read-side critical sections on all CPUs have been completed. Note that synchronize_rcu will not necessarily wait for any subsequent RCU read-side critical sections to complete.

Although the API name suggests locking there is no locking involved on a shared resource.

We used liburcu, a LGPLv2.1 userspace RCU (read-copy-update) library to get the above APIs and integrate them within our code. This data synchronization library provides read-side access which scales linearly with the number of cores. liburcu is available on most major Linux distributions.

Our Insert Strategy for synchronize_rcu():
- create a new node,
- copy the data from the old node into the new one, and save a pointer to the old node,
- modify the new, copied, node,
- update the global pointer to refer to the new node,
- sleep until the operating system kernel determines that there are no readers left using the old structure, in the Linux kernel, by using synchronize_rcu(),
- once awakened by the kernel, deallocate the old node.

Our Remove Strategy for synchronize_rcu():
- Remove pointers to a node, so that subsequent readers cannot gain a reference to it.
- Wait for all previous readers to complete their RCU read-side critical sections.
- At this point, there cannot be any readers who hold references to the node, so it now may safely be reclaimed (e.g., kfree()).

4) Testing and Benchmarking

Our Initial testing and benchmarking were on the Cade lab machines. Later we moved to CloudLab servers as we required sudo access on the machine for our implementations. Our node degree is 64 elements for all test cases to optimize for cache access.

Hardware Used for Testing:

We used an Intel Xeon D-1548 processor machine with 8 cores and 64 GB memory.

Below is an Image of the entire specs from the CloudLab portal:

m510	270 nodes (Intel Xeon-D)
CPU	Eight-core Intel Xeon D-1548 at 2.0 GHz
RAM	64GB ECC Memory (4x 16 GB DDR4-2133 SO-DIMMs)
Disk	256 GB NVMe flash storage
NIC	Dual-port Mellanox ConnectX-3 10 GB NIC (PCIe v3.0, 8 lanes

After the comments were received during the presentation we added new hardware for testing as well:

r650	32 nodes (Intel Ice Lake, 72 core, 256GB RAM, 1.6TB NVMe)
CPU	Two 36-core Intel Xeon Platinum 8360Y at 2.4GHz
RAM	256GB ECC Memory (16x 16 GB 3200MHz DDR4)
Disk	One 480GB SATA SSD
Disk	One 1.6TB NVMe SSD (PCIe v4.0)
NIC	Dual-port Mellanox ConnectX-5 25 Gb NIC (PCIe v4.0)
NIC	Dual-port Mellanox ConnectX-6 100 Gb NIC (PCIe v4.0)

Benchmark Used:

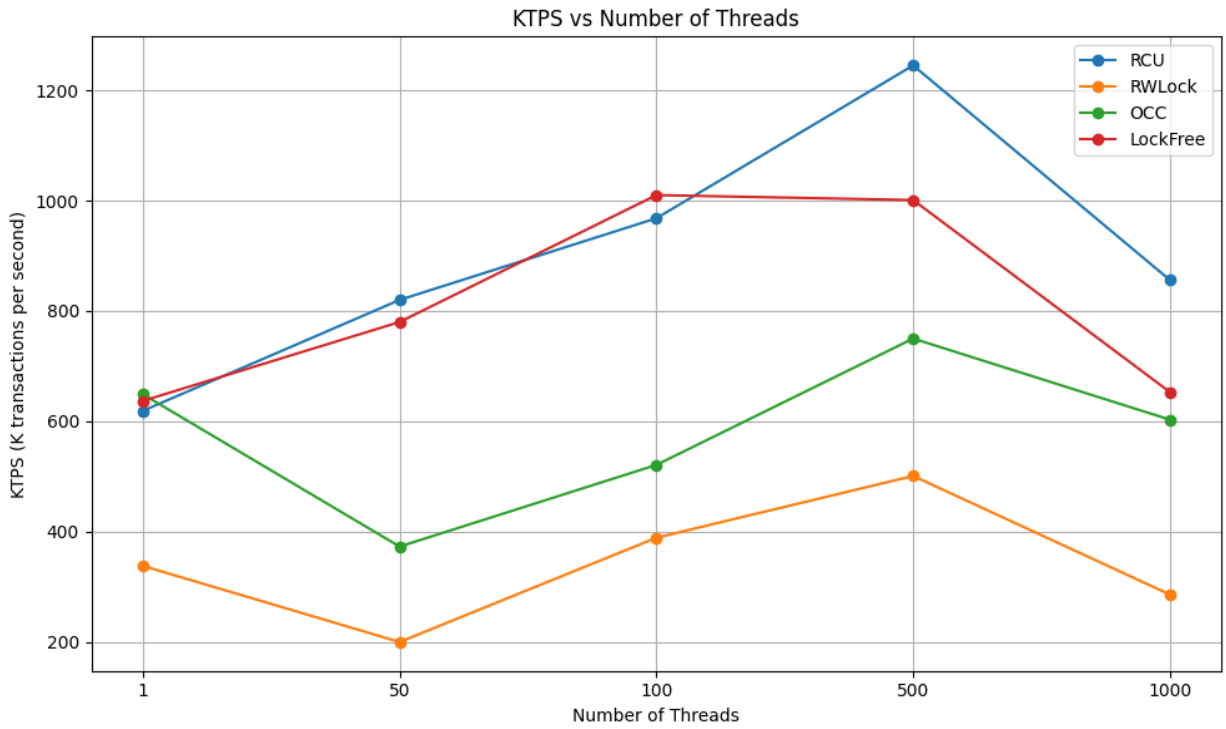
We measured the performance using two benchmarking strategies:

a. Used a benchmarking strategy similar to Project 1 and measured latency for each operation for each thread. Finally, calculated the min, max, mean, and std deviation. Also, we make sure that the threads are originated such that they contend for resources.
b. We used the YCSB benchmark for databases. We modified the benchmark to accommodate integer inputs compared to the original string key-value pairs. Below is the screenshot of the workload configurations we used.

```
# Yahoo! Cloud System Benchmark
# Workload A: Read-Heavy only workload
#   Application example: Session store recording recent actions
#
#   Read/Write ratio: 90/10
#   Request distribution: zipfian

recordcount=100000
operationcount=100000
workload=com.yahoo.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.9
updateproportion=0.05
scanproportion=0
insertproportion=0.05

requestdistribution=zipfian
```

5) Results

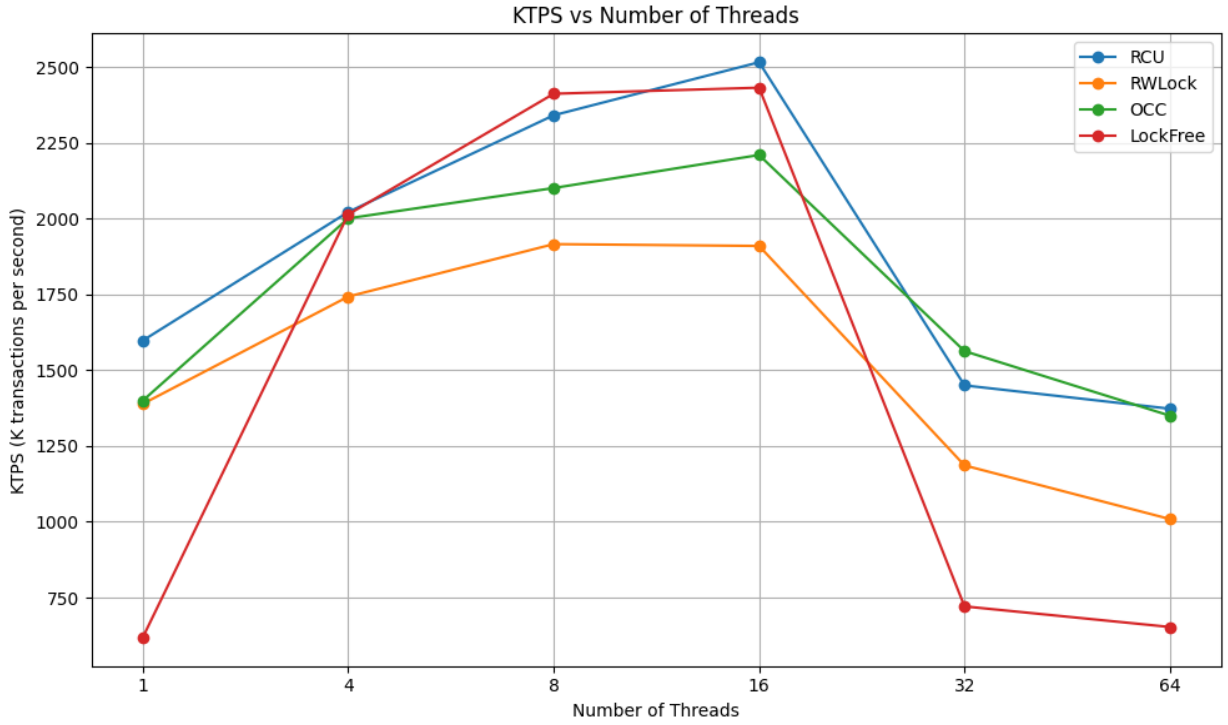
a. YCSB workload with the above configurations



Observations:

- The least throughput is of the RW Lock. Due to write locks the contention overhead is highest.
- OCC works better than RW Lock.
- In the Lockfree and RCU, we observe that in the start the throughput is the same but later the RCU throughput has increased. This is because of the while loop. Here as we are doing insert copying and updating even if there is already one insert that is still doing the operation it causes an increase of time.
- We also observed that the lines are starting to converge. This is because higher threads cause writers to increase and hence due to contention, copy update, and RCU limits, the throughput decreases.

As per the comments provided in the class we then ran our benchmark on a bigger config machine and a lower number of threads, Below is a graph on the bigger machine:



Observations:

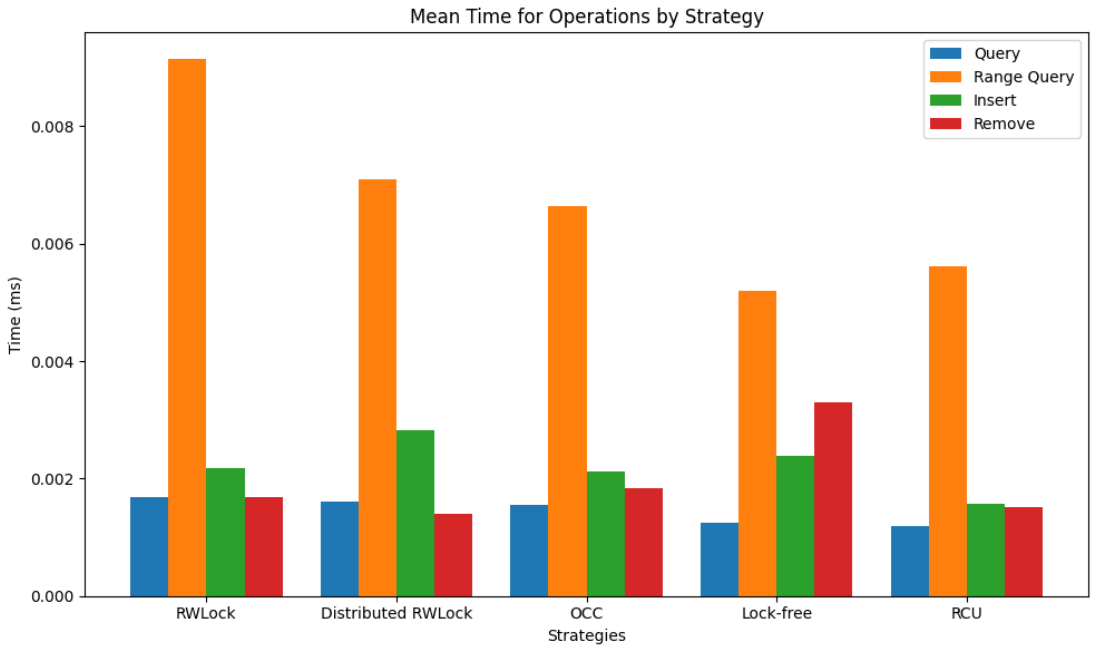
- The least throughput is of the RW Lock. Due to write locks the contention overhead is highest. But the throughput is better due to the higher number of cores.

- OCC works better than RW Lock because there is a good probability that a leaf node will not be full hence only read locks are there throughout the path. Additionally, if node size increases the probability of being full decreases. But too high also causes time for traversing nodes. Hence we need to find the most optimized size.

- In the Lockfree and RCU, we observe that both have almost the same throughput. Here the throughput of Lock free is comparatively higher than before the graph because now we are not using a while loop and instead using atomic operation swap and compare, which avoids unnecessary copies of the path.

- We also observed that the lines are starting to converge. This is because with a high number of threads, the writers also increase and hence due to contention, copy update, and RCU limits the throughput decreases.

b. Our Benchmarking is based on Project 1 with the following configurations:
#Items: 500000
#Readers: 10000
#Read Iterations per reader: 10
#Writers: 100



Observations:

- For range Query, we see that Lock-free and RCU take the least time. This is because no reader threads are getting blocked by any writer or other reader threads.

- For the same reason, we observe that the time for queries is lowest in the RCU and Lock-free.
- Insert time for OCC is slightly less than RW lock. This is because read locks are prominent in this code.

- Distributed has a higher time for insertion. This is because to acquire lock we will need to check all counters of threads in the write case. This causes an increase in time.

- In the case of Lock-free and RCU, the latter time is higher because of the while loop that we are using that continuously checks if insertion is being done or not.

- As the remove and insert working are mirror images of each other their observations are almost similar.

While running on the 72-core machine we got the following benchmark results for our two non-blocking synchronization techniques:

NoLocks:

```
semil@node:~/bplus-project_RW_OCC_rcu/NO_locks$ ./run 10000 100 500000 10 64
Running benchmark with 10000 readers, 100 writers, 500000 items, 10 iterations
Readers (Point): min 0.000285 ms, max 0.049659 ms, mean 0.000546 ms, std_dev 0.000209
Readers (Range): min 0.000000 ms, max 0.112324 ms, mean 0.002452 ms, std_dev 0.002737
Writers (Insert): min 0.000000 ms, max 4.058404 ms, mean 0.308746 ms, std_dev 0.866205
Writers (Remove): min 0.000000 ms, max 0.353543 ms, mean 0.058701 ms, std_dev 0.083574
```

RCU:

```
semil@node:~/bplus-project_RW_OCC_rcu/rcu$ ./run 10000 100 500000 10 64
Running benchmark with 10000 readers, 100 writers, 500000 items, 10 iterations
Readers (Point): min 0.000351 ms, max 0.136159 ms, mean 0.000550 ms, std_dev 0.000440
Readers (Range): min 0.000000 ms, max 0.040751 ms, mean 0.002759 ms, std_dev 0.002849
Writers (Insert): min 0.000000 ms, max 0.055498 ms, mean 0.001573 ms, std_dev 0.006501
Writers (Remove): min 0.000000 ms, max 0.009680 ms, mean 0.000641 ms, std_dev 0.001085
```

6) Conclusions and Learnings

- RCU has a lot of benefits including Concurrency Without Locking Overheads, Non-Blocking Synchronization, Reducing Contention, and Optimizing Read-Heavy Workloads. But it also has its flaws, as it was quite complex to write the update logic and needed to keep a lot of

edge cases in mind. Also, synchronization between writers is an overhead to be taken care of.
- The Lock Free and RCU are best suited for read-heavy workloads as there is no blocking for the readers, but the writers are implemented sequentially in both. Due to the less complex Lock-free version, it proves to be a better alternative for non-blocking synchronization
- Deadlocks are quite hard to debug!!!!!
- Based on the proportion of readers and writer threads we need to find the most optimistic method to execute our database operations

7) Future Work

We used the most basic APIs for RCU integration and would love to explore the more complex APIs like call_rcu(&callback) (a non-blocking writer), assign_pointer(), and rcu_derefrence() to achieve a better performance. We also would like to test this on better machines and more focused test cases to rule out the noise.

8) Contributions

All three of us contributed equally to the project. We divided the work among the three of us and then helped each other with the debugging part if stuck. All of us pitched in to brainstorm our base B+Tree algorithm strategy. We then divided the locking strategies among us. Although everyone worked on almost everything, below are the tasks and the main contributors.

- Find a Production B+Tree and Implement a simpler version of that B+Tree with important operations - **Pranjal, Manisha, Semil**
- Implement Reader-Writer lock on that B+ tree - **Manisha, Pranjal**
- Implement the OCC version of the RWLock - **Manisha, Pranjal**
- Implement a Lock-Free version based on an existing solution - **Manisha, Pranjal**
- Implement Read-Copy-Update (RCU) for synchronization - **Semil**
- Benchmark the code using YCSB - **Semil**
- Compare the results for varying workloads - **Pranjal, Manisha, Semil**

9) References and Code Links

- Code: https://github.com/SemilJain/AdvDB-Project
- https://www.kernel.org/doc/html/next/RCU/whatisRCU.html
- https://en.wikipedia.org/wiki/Read-copy-update
- https://github.com/urcu/userspace-rcu/tree/master
- https://code.google.com/archive/p/cpp-btree/
- https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture04.pdf
- https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/project1.html