A Lock Free B+ Tree Implementation

Title

A Lock Free B+ Tree Implementation (Chenrui Shao and Cassidy Bolio)

URL

https://cbolio.github.io/15-418-Lock-Free-B-Plus-Tree/

Summary

We are going to implement a B+ tree data structure that supports lock free search, insertion, and deletion. After implementing the B+ tree data structure, we will compare and analyze its performance against both single-thread and multi-thread search, insertion, and deletion with traditional, lock / latch-based B+ trees.

Background

B+ tree is a commonly used data structure for storage and indexing in databases. Database systems often run with many threads and need to satisfy multiple operations concurrently. Traditionally, database systems usually use locks and latches to prevent conflicts. However, this requires database systems to implement and maintain a complicated lock / latch mechanism for efficient operations. The implementation does not only need to consider the granularity of lock of the tree to enable simultaneous update on different nodes, but also need to introduce different lock modes to allow concurrent operations on the same node / neighboring nodes for operations that does not conflict with each other, while blocking operations that creates a conflict. The existence of locks and latches in a database system also means the database system needs to handle conflicts and prevent deadlocks, which increases the complexity and adds additional overhead.

Thus, a lock free B+ tree can avoid complicated latch mechanisms and may reduce the overall insertion / deletion cost for multi-threaded database operations. Therefore, this project aims to implement an efficient lock free B+ tree data structure on x86 platforms.

Challenge

A lock-free implementation of a B+ tree will present challenges in several ways. The most prominent is balancing the requirement for lock-free implementations to guarantee progress while simultaneously not damaging the structure of the tree with unsafe actions (inserts or deletes). There are three primary actions for B+ trees: search, insert, and delete. B+ trees are self-balancing and use node splitting and merging to maintain a balanced structure; while beneficial for performance, this creates additional complications for handling concurrent accesses. For example, consider the situation where one thread wants to search the tree while another wants to delete. We need the first thread to be able to successfully find its value, while also allowing the second thread to perform its action, which might trigger the tree to rebalance. If the rebalance merges together two nodes and deletes the old ones, the first thread might suddenly have its path deleted. We will need to assign states to nodes and use them to maintain safe node accesses amidst splits (creation of new nodes) and merges (deletion of old nodes).

Ensuring progress without accidentally creating unfair starvation is another challenge of this implementation. Searches are inherently “safer” as they do not modify the structure of the tree in any way. However, if we always give searches priority or try to prevent inserts/deletes from happening until no searches are occuring, we might end up starving threads that are trying to insert or delete values. Ideally, we want an implementation that allows both threads to “complete” their actions as soon as possible to have good scaling. This introduces an interesting aspect for optimization: how can we combine or organize actions to have good performance without sacrificing integrity for the tree structure? We will need to consider actions that can be “safely” performed concurrently (ex. searches, insertions/deletions that affect independent subtrees) versus those that can’t (ex. searching and deleting in the same subtree), and we will need to figure out how to indicate to the program when to perform batching versus keeping operations more independent.

Lastly, there are some details specific to B+ trees that are imperative for performance, but difficult to implement safely. For example, B+ trees store data in multiple layers of the tree, rather than just a single node. This is beneficial performance-wise as all data of the tree can be found in the leaf nodes; additionally, the leaf nodes point to each other similar to as in a linked list, allowing for full traversal of the data without needing to move through the levels of the tree each time. When splits and merges impact the leaf nodes, we need to essentially manage a safe lock-free linked list implementation within the larger B+ tree. The ABA problem that is demonstrated in linked-lists is still a concern here. We need to have several checks to ensure that we are not damaging our tree structure; for example, we want to avoid the situation where a thread is trying to insert into a specific node, then a second thread successfully inserts, causing the node to split into two distinct nodes - the first thread might think nothing has occurred as the original node is unchanged, inserting its value into that node without knowing that it is a “dead” node. Overall, the dominant challenge of implementing lock-free B+ trees is how to effectively use node ‘states’ to allow safe concurrent actions to ensure progress, while preventing actions that would damage the integrity of the tree.

Resources

We will work based on a few literatures, listed in the section below. We will not be using any starting code, but we will be using some existing code for constructing the test cases and running unit tests, such as https://github.com/tlx/tlx, https://panthema.net/2007/stx-btree/speedtest/, or https://github.com/frozenca/BTree.

Work Cited

NBTree: a lock-free PM-friendly persistent B+-tree for eADR-enabled PM systems. https://dl.acm.org/doi/abs/10.14778/3514061.3514066

PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. https://www.vldb.org/pvldb/vol4/p795-sewall.pdf

B+ review: https://web.stanford.edu/class/cs346/2015/notes/Blink.pdf

Goals and Deliverables:

Goals plan to achieve:

We will provide a lock-free implementation of B+ tree that is correct. We will prove the correctness through testing. The lock-free implementation will be benchmarked and compared with a lock / latch based implementation. We will analyze and report the performance of our implementation of the B+ tree in the final report.

Goals plan to achieve if the work goes slowly:

We will only provide a lock-free implementation of B+ tree that passes the correctness testing. We will benchmark and report the performance of our implementation result in the final report.

Goals plan to achieve if the work goes quickly:

We will provide a lock-free implementation of B+ tree that is correct. We will prove the correctness through testing. The lock-free implementation will be benchmarked and compared with a lock / latch based implementation. We will analyze and report the performance of our implementation of the B+ tree in the final report. After that, we will also attempt to make further optimizations to our B+ tree implementations so it can perform better than traditional lock / latch based B+ trees at high core counts.

Demo to show at poster session

The most important result that will be present at the poster session will be the benchmark results for our implementation and the comparison between our implementation and other latch-based implementations. We plan to have a ready-to-use interactive benchmark demo to be shown at the poster session as well.

Platform choice

This project will use C++ as the programming language, as it is commonly used in database systems and provides good performance. The project will be tested on x86-64 platforms only. It will be tested on the 8 core GHC machines and potentially higher core count servers if possible. In particular, the PSC machines may be used to analyze how the differing implementations perform under extremely high contention.

Schedule

Week	Schedule
4/8 - 4/14	Basic implementation of B+ tree, supporting insertion, deletion, and query in single thread Write milestone report due on 4/16
4/15 - 4/21	Write milestone report due on 4/16 Implement lock free B+ tree according to the literature Generate test cases
4/22 - 4/28	Performance testing of the lock free B+ tree, further optimization if needed
4/29 - 5/5	Compare performance of the lock free implementation with other implementations Write final report