B+Tree Concurrency Control

Yundi Bao, yundib
Yingqi Zhang, yingqizh

Link
https://t7nirvana.github.io/Concurrent-BPlusTree/

Summary
We are going to implement a few versions of B+Tree that can handle concurrent requests on a multi-core CPU platform, including locks in different granularities and a lock-free implementation. We will run benchmarks and compare their performance.

Background
B+Tree is a classic data structure used in database systems for indexes. It stores values in leaf nodes and uses internal nodes to find and direct requests to the right leaf nodes. The tree structure makes all search, insertion and deletion O(log n). In many cases there are also pointers between leaf nodes to make sequential scan possible. In order to benefit from the O(log n) operations, B+Tree has to split/merge to be balanced, which involves modifying multiple nodes.

Database systems, especially OLTP systems, need to accommodate heavy workloads and therefore B+Tree should be able to handle concurrent requests. Due to the complexity of split, merge and sequential scan, locking can be tricky and may not lead to optimal performance.

We propose three versions of concurrency mechanism:

1. A coarse-grained lock-based implementation that maintains a global lock for the entire tree. We can use a rw-lock(lock with shared and exclusive modes) to gain some level of concurrency.
2. A fine-grained lock-based implementation that maintains locks for each node. Each request only holds lock when necessary and releases it immediately when traversing further down the tree. So ideally a read request should hold at most 2 locks at a given time and a write request can hold locks along the path depending on whether it may modify the nodes.
3. A lock-free implementation that handles requests in batches. A batch of requests are assigned to worker threads who then work in phases: identifying dependencies and re-balancing requests, modifying leaf nodes, modifying internal nodes level by level.

The Challenge
The lock-free implementation would be the most challenging part of the project. Since the original requests are in a specific order and they may depend on each other, we have to distribute the requests to balance the workload and at the same time preserve that order within nodes. We also need multiple barriers while handling the batch of requests since we need to

modify a level of the tree at a time. The tree structure inherently leads to ideal workers when handling upper levels of the tree.

Another challenge is implementing sequential scans. In the fine-grained lock version, we have to handle it carefully to avoid deadlock since two requests may traverse leaf nodes in opposite directions(like “search key > 1” and “search key < 10”). In the lock-free version, we have to decide whether to let one worker do the entire scan or pass it to another worker when the scan exceeds its nodes boundary. The first choice may lead to imbalance load and bad cache locality while the second choice may involve communication and computation overheads.

We also want to evaluate the performance of each version in a reasonable way, so we need some benchmark that can really reflect the real world access patterns. Also, we may need to somehow simulate the latency from disk io as well.

Resources

We will first implement our own B+Tree data structure since this is straightforward. For the lock-based implementation, we will follow the idea covered in the 15-645 lecture and explore detailed design choices. We will make our own correctness tests for development. The idea of the lock-free implementation is proposed in the paper “PALM: Parallel Architecture-Friendly Latch-Free-Modifications to B+ Trees on Many-Core Processors” where they even implemented latency hiding and utilized SIMD execution. We will first replicate the basic implementation and then see if we can make further optimization. We found some great benchmarks from yahoo in the paper “Benchmarking Cloud Serving Systems with YCSB” and we will use some of them to evaluate our implementations.

Goals and Deliverables

PLAN TO ACHIEVE

- The B+Tree implementation with Search(), Insert(), Delete(), Scan() and split/merge functionality
- The three versions of concurrent B+Tree that support Search(), Insert(), Delete() and split/merge functionality
- The evaluation by running benchmarks and analyze the results

HOPE TO ACHIEVE

- Supporting Scan() in the two lock-based versions
- Supporting Scan() in the lock-free version
- Utilizing SIMD execution and hiding latencies in the lock-free version

We hope our lock-free version(without SIMD or latency hiding) can have linear speedup, though the factor may be smaller than 1 since we can hardly fully utilize workers in the tree structure. It should be at least significantly faster than the fine-grained lock-based implementation, especially when we have large trees and more dependent requests.

Our eventual deliverables that we will show at the poster session will be the speedup graphs of our implementations. We will also make descriptive figures to illustrate our implementations and the characteristics of the benchmarks we run.

Platform Choice

We will implement our concurrent B+Trees in C++. C++ has its own pthread library and we can easily include other languages that have richer parallelism models like OpenML, ISPC if we see necessity. We will run our benchmarks on ghc machines first and may move to psc machines if more concurrency is needed.

Schedule

Week	**Task**	**Status**
11.1 - 11.7	Proposal, Background Research	
11.8 - 11.14	B+Tree, Lock-based Implementations	
11.15 - 11.21	Lock-free Implementation	
Milestone		
11.22 - 11.28	Benchmark, Evaluation, and Extra Feature if possible	
11.29 - 12.5	Final Measurement and Report	
12.6 - 12.10	Poster	