High Concurreny B-Trees for Insert Heavy Workloads

An in-depth comparison

Isaac Bowen*
University of Michigan College of Engineering
MSE: Computer Science and Engineering
irbowen@umich.edu

G.K.M. Tobin†
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

ABSTRACT

Code can be found here Fill this in once we're done

You should try to write the best research paper that you can using the results of your project. You have read many good papers throughout this class, so by this point you should have a good idea of what makes a good research paper! Basically, your report needs to clearly present the following:

1. the problem statement
2. The motivation, why its important
3. The literature review (the previous work in this area
4. Main idea and approach
5. Implementatino techniques
6. Experimental setup
7. Results

Keywords

B-Trees, Reader Writer Locks, B-Link, Lock-Free

1. INTRODUCTION/MOTIVATION

Databases systems are usually run in multiple threads or processes, so that they can handle many users at the same time, take advantage of multiple processor cores, and using async I/O, multiple disks [2]. Systems will larger number of processors, and cores-per-processor are becoming more common, and main memories are becoming much larger. B-trees were first built for single-threaded DBMS, which stored everything on disks - including the index itself. Later systems allowed the index to fit in memory, and allowed some concurrency - multiple threads could read from the index.

We attempt to look B-Tree's for insert-heavy workloads. WE WILL ASSUME THE INDEX, OR THE WHOLE DB, CAN FIT IN MEMORY It should be noted that we are not looking at transactional level locking, but concurrent execution within a transaction, or in a transaction free environment. We are not concerned, then, with provide any kind of predicate locks to potect against "phantom" inserts.

2. PREVIOUS WORK

B-trees have long been the primary access data structure for databases, file systems, and various other systems because of their logarthimc `insert()` and `get()`. They also have a host of other properties that make them also for large systems. A short list:

1. Maintain everything inorder. This means that merge operations can be done without sort, and SMJ, and often used join in database systems, is very efficent
2. Block access. The strucutre can store as much data is it can on a single page, and doing locking on a single page. This made B-Tree's very popular early on, when database's could not fit in memory, and had to be dumped onto disk.

Over there 40 year history, there have been many different variations and flavors of B-Trees. The original version of this data structure stored data in both the leaf and inner nodes. These structures were optimzed for block access - the entire b-tree index could not fit into memory, and the fact that its nodes size could be made to match the disk block size. This, coupled with buffer bools, made them the domanit structure even in early database systems.

Later versions (B*Trees or B+Trees, depending on who you ask) stored all the data in leaf nodes, and the inner nodes served only as lookup keys. Later version would ensure the entire tree was height balanced, and put pointers on the leaf nodes so that scan could be done with only one tree taversal.

Later (fancier) techniques attempted to put links on inner nodes as well, so that they could do "overflow" nodes, to try to prolong splitting. The idea here was that when a key was inserted into a node that should then split, it would look at its neightbors in the same levle of the tree. If they have enough room, the node could offload some of its keys and values onto its siblings. This could allow the tree to become more 'full' - the average occupancy of each node will be higher - before nodes split. Depending on the cost

*A note about this author
†The secretary disavows any knowledge of this author's actions.

UofM:CoE:EECS 2015, Ann Arbor MI

ACM ISBN 123-4567-24-567/08/06.
DOI: 10.475/123_4

of splitting, and the number that occur, this could provide a benifit to performance.

There have been a few approaches to locking in B-Trees so far.

The first, naive, approach, is to lock any node that could split on an insert. However, since any split could propigate all the way back up to the root node, each insert would need to have an exclusive lock on every node that it touches - making the tree single threaded.

Another slightly better way to do locking is to acquire a shared lock at first, and then scan throught the node to find where the key will be inserted. If the child not cannot spit, than we can hold an shared lock on the node while the rest of the insert finishes in the tree below. If the child can split, we must hold an exclusive lock. The reader will notice that this can still lead to high lock contention on the root node, especially with small fannout.

3. OUR APPROACH

We first wanted to see what worked well on a sequential model, so that we can compare those same factors on our concurrent models.

We knew that we wanted to be able to test the effect of the fannout of the nodes - what size of fannout favors insert-heavy workloads? We decided to test the fannout of the inner nodes, and the number of data elements stored in a leaf node seperately. We plan on seeing whether one of these fannouts has more of an input on performance. Both of these constants can be modified in `node.h`

We also want to see what the performace is under different read and write percentage workloads. We use `std::rand()` to decide if a given operation should be a read or write in our testing setup. This value is input into the program through standard input at startup

There were a few things that we decided to change, given that this tree is to be optimized for writes instead of reads. We do not convern with keeping all the leaf nodes at equal depth. This makes sense for reads - you want to make sure each key get be access in the same amount of time, and if you are trying to get multiple elements, you can do the lookup once and then just scan across the leaf nodes. However, maintain this global depth is expensive in the face of multiple threads. We chose to abandon this requirement, and let the tree have varying depths of nodes - the root node could point to some leaf nodes, and other trees with various children. This could increase lookup time (as well as insert time, since it must do at lookup to find the insert location, however, as long as the key values are not VERY skewed, the tree should perform well.)

Many B-Tree implemnts

4. ARCHITECTURE

What all did we have to build to make this work and test out our ideas?

- Testing Framework: We want to test the system at different levels of
  - Input Size
  - Number of THreads
  - Read Write Percent Workload
  - Fannout on inner nodes
  - Data slots on leaf nodes

  To testing all of these things, I will be creating a dope ass main.ccp that can read input from test ases

- Sequential Tree: We had to build a simple, single-thread B-tree first. We built on this for our other version, and will also use it as the baseline for testing.

- ReaderWriter Tree: This tree uses different kinds of locks, shared and exclusive, and also does checking to see if the child node can split, to see if this node is "safe" from splitting. This version uses `std::list<>` within the nodes

- ReaderWriter Array: How does sequential access of nodes affect performance? We'll see. Could this show promise for further extension - putting the whole B-tree into an array?

- BLink Tree: This structure is different in that it insert into the leaf node, and acquires locks on the way back up

5. CITATIONS

Some papers that we read to give ourselves background on the topics:

- "Concurrent B-trees with Lock-free Techniques"[1] gives us the inspiration for the lock free B-tree. They, however, didn't not have access to the large memory, built in C++14 atomics, and they were not focusing on writes - they were trying to build a general purpose lock free btree for NUMA computers.

- "A survey of B-tree locking techniques"[2], much like the title says, presents an over of many of the different approaches that have been taken so far. However, it reallys skims over, almost dismissing, lock free techniques.

- "Efficient Locking for Concurrent Operations on B-Trees" [3] This one gave us the idea of the `can_split()` function, and gave background on reader writer locks and other, etc

- "Concurrent Cache-Oblivious B-Trees" [4] I need to read this one again

- "A Concurrent Blink-Tree Algorithm Using a Cooperative Locking Protocl" [5] Ryan is making this verion

6. EXPERIMENTAL RESULTS

We will need some tables. I have some data from the sequential tree

7. CONCLUSIONS

Conclusions

8. ACKNOWLEDGMENTS

Acknowledgments

9. REFERENCES

[1] Helen Cameron Afroza Sultana and Peter Graham. Concurrent b-trees with lock-free techniques. *DUNNO*, 0(0), 2010.

[2] Goetz Graefe. A survey of b-tree locking techniques. *ACM Transactions on Database Systems*, 35(2), 2010.

[3] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1987.

[4] Seth Gilbert Michael A. Bender, Jermey T. Fineman and Bradely C. Kuszmaul. Concurrent cache-oblivious b-trees. 2005.

[5] Joonseon Ahn Sung-Chae Lim and Myoung Ho Kim. A concurrent blink-tree algorithm using a cooperative locking protocol. 2003.