B-Trees for Insert Heavy Workloads

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

Ryan W
University of Michigan College of Engineering
MSE: Computer Science and Engineering
@umich.edu

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. Implementation techniques
6. Experimental setup
7. Results

Keywords

B-Trees, Reader Writer Locks, B-Link

1. INTRODUCTION/MOTIVATION

Databases systems are usually run in multiple threads or processes for a variety of reasons - so that they can handle many users at the same time, take advantage of multiple processor cores and multiple disks [2] to speedup individual queries, as well as various background tasks. We can only assume that current hardware trends will continue, and that number of threads and process within a database system will continue to increase. Systems will larger number of processors, and cores-per-processor are becoming more common, and main memories are becoming much larger. B-trees, along with many other software systems designed in the dawn of databases systems, were designed to work on very different hardware than we have available today. 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 study B-Trees for insert-heavy workloads.
PROBABLY SOME MORE HERE
It should be noted that we are not looking at transaction 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 protect against "phantom" inserts.

UofM:CoE:EECS 2015, Ann Arbor MI

ACM ISBN .
DOI:

2. PREVIOUS WORK

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

1. Maintaining elements in sorted order. This means that merge operations can be done without sort, and SMJ, and often used join in database systems, is very efficient
2. Block access. The structure should use node sizes equivalent to page sizes, and the fan out of the system should be adjusted as to maximize space utilization. This also works very well with locking - just lock the block id. This made B-Tree's very popular early on, when most databases could not fit in memory, and had to be stored on disk.
3. Modular. We can store separate nodes of the tree in different areas of memory, as well as only lock what we need to - we don't have to lock the entire tree on every operation to maintain consistency, just its children (and possibly neighbors, and in some implementations).

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 optimized 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 pools, made them the dominant 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 traversal.

CITE THE PAPERS SOMEWHERE

Other more advanced 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 neighbors in the same 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 of splitting, and the number that occur, this could provide a benefit 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 propagate 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 through 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 fan out.

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 fan out of the nodes - what size of fan out favors insert-heavy workloads? We decided to test the fan out of the inner nodes, and the number of data elements stored in a leaf node separately. We plan on seeing whether one of these fan outs has more of an input on performance. Both of these constants can be modified in `node.h`

We also want to see what the performance 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 start up.

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 concern ourselves 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.) We believe that additional lookup time is not a problem, as long as it is concurrent - the real performance limitation is when the tree structure must change, because this requires exclusive locks. If we can keep the "structure" of the tree the same, then we can maximize for inserts, and always have some number of threads doing something useful, instead of waiting as the tree is balanced.

Many B-Tree implement pointers to the neighboring nodes at the same level, both for leaf and inner nodes. While this is certainly helpful for range-based read workloads, there is a cost that has to be paid on every insert. In an insert-heavy workload, these become much more important. Also, in the inner nodes, these pointers are used to push of the tree splitting, to lower cost. This was assuming, however, that the tree was balanced. Without this invariant, there is no reason to do this. In concurrent code, managing all of these pointers become incredibly difficult to do correctly, and has minimal benefit. Therefore, we decided to forgo these pointers. This means that range queries become less efficient, because we can no longer scan across the leaf nodes. However, for an insert heavy workload, range queries are not well defined. They would require some form of predicate locking, MVCC, or some other complication that we do not want - simple put, workloads that are insert heavy should know that range based queries do not make sense, as the data is always being changed.

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
  - Fan out on inner nodes
  - Data slots on leaf nodes
- 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 a nodes children affect the performance of the tree?
- BLink Tree: This structure is different in that it insert into the leaf node, and acquires locks on the way back up

5. IMPLEMENTATION AND TEST SETUP

We implemented our B-Trees in `C++`, using new features only available in `C++14`, most notably `stdshared_time_mutex`, which allows both exclusive and shared locking. This requires a compiler with `gc -4.9` or greater. Because this is

so new, few systems have made version 4.9 available - it is not available on `bigdata.eecs.umich.edu`, for example, we decided to set up our own testing environment.

We set up a virtual machine using Oracle Virtual Box. The VM is running Ubunutu Server 15.10 with gcc 4.9.3. The host system is a Windows 7 machine with a FX-8350 (`8 cores @ 4Ghz`) and 32GB of RAM. We allocated 4 cores and 24GB of RAM to the VM. All test were run with minimal applications running on the Windows side.

We used `std::rand()` to generate input data.

6. PERFORMANCE

We tested all of our implementations under several different conditions.

6.1 Sequential

We begin with our sequential tree implementation, so that we have a baseline to compare our concurrent versions to. These are 100% insert workloads.

6.1.1 100% Insert

Sequential Tree, F = 4

Input Size	Time (ms)
1000	2
10000	12
100000	80
1000000	586
10000000	5620
100000000	59000

Sequential Tree, F = 16

Input Size	Time (ms)
1000	2
10000	20
100000	123
1000000	754
10000000	7045
100000000	70318

Sequential Tree, F = 64

Input Size	Time (ms)
1000	4
10000	31
100000	283
1000000	3607
10000000	128447
100000000	384118

Sequential Tree, F = 256

Input Size	Time (ms)
1000	7
10000	67
100000	968
1000000	14840
10000000	267131
100000000	2543760

Note the dramatic difference in performance as the fan out increases. We discuss the significance of these results in the next section.

We wanted to compare are tree to standard implementations to a standard, to verify that our concurrent versions were not built on top of something slow. To that end, we compared it to std::map on gcc 4.9.3.

`std::map`

Input Size	Time (ms)
1000	1
10000	6
100000	63
1000000	720
10000000	12780
100000000	228835

6.1.2 90% Insert, 10% Lookup

While we are primary focused on insert performance, we wanted to see how a 90% insert 10% lookup workload would perform.

Sequential Tree, F = 4

Input Size	Time (ms)
1000	1
10000	9
100000	58
1000000	527
10000000	5199
100000000	79855

Sequential Tree, F = 16

Input Size	Time (ms)
1000	2
10000	9
100000	112
1000000	667
10000000	6571
100000000	65188

Sequential Tree, F = 64

Input Size	Time (ms)
1000	11
10000	33
100000	283
1000000	4891
10000000	112838
100000000	

Sequential Tree, F = 256

Input Size	Time (ms)
1000	9
10000	69
100000	986
1000000	14619
10000000	242493
100000000	*

`std::map`

Input Size	Time (ms)
1000	1
10000	2
100000	34
1000000	716
10000000	13151
100000000	204166

We note that std::map seems to perform much better at reads.

6.1.3 80% Insert, 20% Lookup

While we are primary focused on insert performance, we wanted to see how a 80% insert 20% lookup workload would perform.

Sequential Tree, F = 4

Input Size	Time (ms)
1000	1
10000	5
100000	60
1000000	469
10000000	4701
100000000	47418

Sequential Tree, F = 16

Input Size	Time (ms)
1000	2
10000	20
100000	92
1000000	606
10000000	6839
100000000	57951

Sequential Tree, F = 64

Input Size	Time (ms)
1000	4
10000	32
100000	226
1000000	3216
10000000	82029
100000000	*

Sequential Tree, F = 256

Input Size	Time (ms)
1000	3
10000	50
100000	769
1000000	12465
10000000	182482
100000000	*

`std::map`

Input Size	Time (ms)
1000	.5
10000	1
100000	23
1000000	584
10000000	10719
100000000	180852

6.2 Concurrent, Single threaded

We then testing our concurrent versions. First we compared on its performance with one thread, and then we scale the number of threads to see how the performance changed.

6.2.1 100% Insert

Reader Writer Tree List

Input Size	10^3	10^4	10^5	10^6	10^7	10^8
Fan out						
4	13	105	1153	19787	301004	*
16	1	15	270	4694	79077	*
64	2	27	458	9124	156241	*
256	6	71	1409	25626	384319	*

Reader Writer Tree Array

Input Size	10^3	10^4	10^5	10^6	10^7	10^8
Fan out						
4	10	78	861	14650	215321	*
16	2	21	117	1326	17772	246069
64	3	25	120	1191	14523	179811
256	11	59	412	4150	45198	*

6.2.2 90% Insert, 10% Lookup

Reader Writer Tree List

Input Size	10^3	10^4	10^5	10^6	10^7	10^8
Fan out						
4	12	106	1147	17987	287852	*
16	1	15	223	4223	75675	*
64	6	53	436	7965	138397	*
256	17	94	1317	22046	342301	*

Reader Writer Tree Array

Input Size	10^3	10^4	10^5	10^6	10^7	10^8
Fan out						
4	9	79	870	13672	203731	*
16	1	10	109	1269	16874	235350
64	1	8	85	1113	12728	179811
256	3	37	347	3827	41877	*

6.2.3 80% Insert, 20% Lookup

Reader Writer Tree List

Input Size	10^3	10^4	10^5	10^6	10^7	10^8
Fan out						
4	10	76	1024	16335	266000	*
16	4	32	204	3894	65661	*
64	2	48	373	7321	122407	*
256	14	75	1117	20417	321325	*

Reader Writer Tree Array

Input Size	10^3	10^4	10^5	10^6	10^7	10^8
Fan out						
4	3	48	811	12197	191958	*
16	1	11	106	1182	16112	
64	1	22	107	1048	12947	
256	3	30	301	3426	37719	*

6.3 Concurrent, Scaling Threads

Reader Writer Tree List - Input Size of 10000

Threads	Time
1	105
2	332
4	709
8	703
16	856

Reader Writer Tree List - Input Size of 100000

Threads	Time
1	1403
2	4049
4	8569
8	9629
16	10290

Reader Writer Tree Array - Input Size of 10000

Threads	Time
1	18
2	58
4	153
8	198
16	288

Reader Writer Tree Array - Input Size of 100000

Threads	Time
1	104
2	452
4	1122
8	2629
16	3788

7. DISCUSSION

We began by testing out our sequential tree, on a variety of input sizes. We wanted to make sure that this implementation, that we would be building on top of for our concurrent versions, was efficient. We testing with various fan outs, and decided to test for 4, 16, 64, and 256. We were surprised to see the performance degrade ad the fan out increases. We assumed that a larger node size would allow faster look-ups and inserts, because the operation would have to traverse fewer levels of the tree. However,it seems that there is a difficult balance between traversal within a node, and traversal in the tree across nodes.

We saw that the fan out of 4 gave us the best performance, so we wanted to compare that to an in-memory tree. We chose to compare to `std::map`, which is typically implemented using Red-black trees. When fan out is small, either 4 or 16, the performance of our B-Tree is better. However, as the fan out increases, our sequential tree performance degrades to the point where it is slower than `std::map`.

It is curious that the fan out increases seems to slow things down

Our concurrent versions, across the board, are slower than the sequential version. This is even true when we run our concurrent versions with just one thread - the overhead of acquire locks is a huge performance hit. Each request to lock or unlock is a call to the operating system, which could result in busy-waiting. In as we scale up the number of threads, even with the same number of total inserts, we see a considerably performance degradation. This is primarily due to contention on the root node. The reader writer lock implementation still has to acquire exclusive locks on the root node when one of the inserts could result in one of the roots children splitting. While the system allows for more concurrency than the naive scheme, the contention on the root node, as well as all of the locking overhead, create a system that is slower than inserting using the sequential lock free version. This seems to be a fundamental flaw of any top down approach - the root node will always be the bottleneck, and the primary limitation to performance.

8. TAKEAWAYS

We have a few notable takeaways from our implementation and experimentation:

- **Locking** Locking is expensive. Even in our single threaded tests, our multithreaded solutions are considerably slower than our single threaded versions. The code is nearly identical, and so most of the extra time comes from locking overhead. Each call to acquire a lock, or even check if it is locked, requires a trap into the OS, which is very expensive.
- **Single Threaded** Much like Stonebraker has mentioned in multiple papers, the traditional concurrent multithreaded code with locks was built for a different era - when indexes were stored on disk, and it made since to make node's the size of a disk block. Now, it is faster to run operations in a single threaded fashion. Even when we use our concurrent code, it is faster to run 10,000 insert operations in one thread than 5000 insert operations in two threads.
- **Smaller Node** Smaller nodes seem to be faster than larger nodes. While this seems to go against what we would expect with modern cache architecture, it does make sense. Larger nodes were designed for block access devices, so it made sense to have the nodes be the same size as the disk blocks. However, if the structure fits in memory, than it is fast and cheap to traverse the nodes, so it's not as important to keep the tree's depth very small.
- **Vectors vs List** Two of our implementations are identical, except for the data structure the nodes use to store their children. We can see that the array based version beats the list based version handily on every large case, especially when fan out is 16-64. Arrays of this size fit in caches easily, and searching them sequentially is very quick. We believe the use of contiguous memory is essential to performance.
- **B-Tree vs Red black Tree** We noticed that the our sequential tree was able outperform `std::map`, but as the read percentage increased, `std::map`'s performance got dramatically better. We thing that this is because of the underlying structures of our two implementations. While we use a unbalanced B-Tree with high fan out(compared to Red-Black Trees), which allows for faster inserts. However, `std::map` must consistently re-balance, adding overhead. In higher read based workloads, it gets an advantage for that cost - look ups can be done very quickly.

9. 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
- "A paper" A paper

- GET STONEBRAKERS PAPERS WHERE HE TEARS APART TRANSACTIONS LOCKING AND TRADITIONAL DBMS

10. EXPERIMENTAL RESULTS

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

11. CONCLUSIONS

Conclusions

12. ACKNOWLEDGMENTS

Acknowledgments

13. 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.