B-Trees for Insert Heavy Workloads

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

Ryan Wawrzaszek
University of Michigan College of Engineering
MSE: Computer Science and Engineering
ryanwawr@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

Most traditional database management systems (DBMSs) use structures called B-Trees for storing indexes. B-Trees were designed to allow for efficient insertion and lookup of key-value pairs. However, this structure was developed decades ago, when DBMSs were run on single machines with small amounts of main memory, distributed applications were rare, and high availability was not a major concern. These circumstances influenced the design and structure of the B-Tree in critical ways, but database technology is a rapidly changing field. We have seen huge increases in processing power and main memory sizes, distributed applications have become commonplace, and clients have come to expect ”always-on” systems with high availability for concurrent access. All of this means that many of the needs and assumptions that motivated the design of the B-Tree no longer hold true.

UofM:CoE:EECS 2015, Ann Arbor MI
© 2015 Copyright held by the owner/author(s).
ACM ISBN .
DOI:

Our particular focus in this paper is on the performance of B-Trees in in-memory databases. In the early days of relational databases, main memory was often too small to hold the full index. Therefore, B-Trees were designed to have a large fanout, since each level of the tree that had to be traversed incurred an additional disk I/O. Multi-threading was introduced to B-Trees, in part, to allow the processor to continue doing useful work while the next node of the tree was read into memory. Modern databases have advanced to the point where entire databases, even those containing several terabytes of information, can be stored in main memory. Our aim for this project was to analyze several different B-Tree implementations under an insert-heavy workload similar to those of modern OLTP databases.

2. PREVIOUS WORK

B-trees have long been the primary access data structure for relational databases. They have many properties that make them ideal structures for storing, retrieving, and using large amounts of data.
These include:

1. Efficient operations - Since all data is stored in the leaf nodes of the tree, the `update()` and `get()` operations only need to access one node at each level of the tree. Therefore, their runtime is logarithmic with respect to the fanout. The `insert()` operation is slightly more complex, as splits occurring in leaf nodes may propagate back up the tree, but the runtime is still logarithmic.

2. Maintaining elements in sorted order - In addition to allowing for efficient operations, maintaining elements in a sorted order allows for efficient joins on clustered indexes. Since selected tuples are retrieved in order, the sort-merge-join (SMJ) algorithm can be used without the expensive preprocessing step, allowing for improved query response times.

3. Block access - B-Trees often set the size of each node in the tree to be equivalent to the size of a memory page, adjusting the fan out of the system correspondingly in order to maximize space utilization. This also allows for efficient locking, as one needs only to lock the block id in order to lock the associated page. This made

B-Tree's very popular early on, when most databases could not fit in memory, and had to be stored on disk.

4. Granular locking - B-Tree nodes can be locked individually. On each access, the DBMS only needs to lock the nodes that might be affected by the current operation. So long as the locking scheme maintains consistency within the tree, multiple threads are able to access the tree at once, improving concurrency and throughput.

Many different structures and modifications of B-Trees have been proposed over the last 40 years. In the original version of this data structure, data was stored in the inner nodes as well as in the leaf inner nodes. This was seen as an optimal solution at the time as there was no wasted disk bandwidth, an important factor in an era when many DBMS applications were still single-threaded. A structure that limited data to the leaves would have wasted disk bandwidth by reading in a full page of data for each inner node, even though only the list of child nodes was needed.

As applications became multi-threaded, new versions of the structure, such as B*Trees or B+Trees, switched to storing data exclusively in leaf nodes, storing only keys in the inner nodes. This loss of some amount of disk bandwidth associated with reading non-leaf nodes into memory was less critical in a multi-threaded environment. This shift in organization also allowed more keys to be stored in inner nodes, which gave trees a wider fanout and led to less total disk I/Os. Later versions of B-Trees also guaranteed that the tree structure was height balanced in order to provide good worst-case performance. They also added pointers linking leaf nodes together so that tables scans could be accomplished with only one traversal of the tree.

CITE THE PAPERS SOMEWHERE

Later implementations added "sibling pointers" to inner nodes as well in an attempt to prolong splitting of nodes, which was a more expensive operation that a simple insert or retrieval. When a key was inserted into a node that would ordinarily be forced to split, it would instead check if it's sibling nodes had room for an additional key. If the sibling nodes were not full, keys were redistributed between the nodes in an order-preserving fashion. This allowed for delayed splitting of nodes and resulted in nodes that were, on average, closer to capacity. This served to minimize the depth of the tree which, in turn, decreased the amount of total disk I/Os needed to insert or retrieve a record.

Once DBMS applications moved to multi-threading, it became necessary to modify B-Trees to support concurrent access. Several different locking schemes have been used in traditional B-Trees. The naive approach is to lock any node that could split on an insert. It is impossible to know ahead of time which nodes will be accessed and which can potentially split, since, in a tree where each node is at full capacity, a split could propagate all the way up the tree to the root node. The simplest approach, then, is to acquire an exclusive lock on the root node before inserting, thereby locking the entire tree. This is, in effect, no different from a single-threaded application, as no concurrency can be supported by this locking mechanics.

A slightly more refined locking mechanism involves acquiring a shared lock on the root node at the beginning of the operation. As the operation traverses the tree, it checks each node to see if it may be split by this operation. If it is determined that the child node cannot split, the shared lock is maintained. Otherwise, an exclusive lock must be acquired on both the child and parent nodes. This allows an increased degree of concurrency, as read operations can still proceed in a large portion of the tree while the insert takes place. It should be noted, however, that this locking scheme can still lead to contention for a lock on the root node, especially in trees with small fanouts where nodes are split with a higher frequency.

3. OUR APPROACH

We began by implementing a sequential access B-Tree with no locking mechanism and no support for concurrent access. This tree served as our baseline for measuring the performance of the concurrent access trees that we will discuss in Section 4. The sequential tree, as well as all other implementations, support the `insert(key, value)` and `get(key)` operations. We felt that this set of operations was sufficient to test our B-Tree implementations and locking mechanisms, as they provide the read and write functionality that will be needed to test our granular locking schemes.

We designed each B-Tree implementation in a manner that allows the fanout to be adjusted by setting a parameter. We were particularly interested in effect of node fanout on the performance of our implementations, given that the index and database are hosted entirely in main memory. Tree traversal is expensive for databases that are too large to fit in main memory, as a disk I/O is incurred every time a node is read in. This cost is not nearly as severe for an in-memory database, so we hypothesized that we would see less of a decrease in performance as fanout increased than would be observed in older databases.

We used the `std::rand()` function in C++ to generate key-value pairs for insertion and to determine whether an operation would be an insert or a retrieval. The percentage of reads and writes in the workload can be adjusted by means of an input parameter given to the program at runtime. We tested our implementations on insert-heavy workloads in order to simulate the workload of a modern OLTP database.

Given that our B-Tree implementations are intended for use in an in-memory database subject to insert-heavy workloads, we made several changes to the traditional B-Tree structure. First, we do not we do not require the tree to be height-balanced. Balanced trees are essential in DBMS applications for which the index and data are not hosted in memory, as a disk I/O is incurred for every level of the tree that is traversed. Therefore, balancing was required to limit disk I/O and provide reliable response times. For an in-memory database, the cost of reading node is merely the cost of following a pointer, so operation latencies are unlikely to vary significantly in an unbalanced tree. ??? 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-Trees 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. B-TREE IMPLEMENTATIONS

The goal of our experiment is to observe the performance impact of varying each of the following parameters

- Input Size
- Number of Threads
- Read Write Percent Workload
- Fan out on inner nodes
- Data slots on leaf nodes

In order to examine the performance of B-Trees in in-memory databases, we built four different tree implementations.

4.1 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.

4.2 ReaderWriter Trees (Array and List-based)

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

4.3 B-Link Tree

The B-Link tree is based on the work of CITE STUFF HERE and employs a bottom-up locking mechanism. The structure of the B-Link tree differs from that of a traditional tree in two main ways.

First, each node maintains a "link pointer" to the next node to the right at the same level. The pointer for the node furthest to the right is simply assigned a null value. These pointers appear as the horizontal arrows connecting nodes at the same level in Figure 1.

Second, each node maintains a "high key". The high key is the value of the highest key found in the subtree rooted at the node pointed to by the last non-link pointer in the node.

These structural modifications allow for the implementation of a minimal and efficient locking scheme. On an insertion, each node accessed as the tree is traversed is pushed onto a stack, but is not locked. The first node to be locked is the leaf into which the new value is being inserted. If the insertion causes the node to exceed its capacity, the node is split. A link pointer is added from the original node to the new node, and the link pointer of the new node is set equal to the old link pointer of the original node.

Because the leaf's parent node does not know of the newly created leaf node, the two leaves can be considered one leaf, as the new node is essentially an extension of the original that can be reached by following the link pointer. The parent node is popped from the stack and locked. A pointer to the newly formed node is inserted, atomically adding the new node to the tree. At this point, the leaf node may be unlocked. If the parent node is forced to split, the same split procedure is followed. This continues up the tree until the split cannot propagate any further, at which point the last lock is released.

The search algorithm for the B-link tree is very similar to that of a traditional B-tree, but it acquires no locks. As the search travels down the tree, it scans each node to find the appropriate child to insert into. If the high key of the current node is less than the key being inserted, then this node has been split, and the search simply follows link pointers until it finds the node that contains a pointer to the proper child node.

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. We were unable to update the compiler on `bigdata.eecs.umich.edu`, so we 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.