Chapter 5

Search Trees Using Libxac

In this chapter I describe how memory-mapped transactions can be used in a practical application, specifically in search trees that support concurrent searches and insertions. I also present experimental results comparing the performance of search trees written using LIBXAC to the B-tree of Berkeley DB [44], a high-quality transaction system.

For data that resides on disk, B-trees are the canonical data structure for supporting dictionary operations (search, insertion, deletion, and range queries). In Section 5.1, I describe the Disk Access Machine Model, the performance model primarily used to analyze B-trees. In this model, computation is free, but moving a block between main memory and disk has unit cost. I then describe the concept of a cache-oblivious algorithm, an algorithm that tries to minimize the number of memory transfers per operation without knowing the actual block size. Finally, I briefly describe how a cache-oblivious B-tree (CO B-tree) supports dictionary operations with an asymptotically optimal number of memory transfers.

In Section 5.2, I investigate the practical differences between two different B-tree variants by presenting experimental results comparing the performance of a serial B^+-tree and a serial CO B-tree, both written using memory mapping, but without LIBXAC.[1] The data demonstrates that a CO B-tree can simultaneously support efficient searches, insertions, and range queries in practice. Random searches on the CO B-tree ran only 3% slower than on a tuned B^+-tree on one machine and ran 4% faster on a newer machine.

In Section 5.3, I describe the ease of using memory-mapped transactions to convert the serial implementations of the B^+-tree and CO B-tree into parallel versions.

I present experimental results in Section 5.4 that suggest that small, durable memory-mapped transactions using LIBXAC are efficient. In an experiment where a single process performed random insertions, each as a durable transaction, the LIBXAC B^+-tree and CO B-tree are both competitive with Berkeley DB. On the three newer machines, the performance of the B^+-tree and CO B-tree ranged from being 4% slower than Berkeley DB to actually being 67 % faster. This result is quite surprising, especially in light of the fact that I am comparing an unoptimized prototype with a sophisticated, commercial transaction system.

Finally, in Section 5.5, I conclude by describing possible future experiments for evaluating the performance of LIBXAC. I also discuss potential improvements to the implementation that are motivated by the experimental results.

5.1 Introduction

The DAM Model

In today's computer systems, there is significant disparity between the time required to access different memory locations at different levels of the memory hierarchy. In Chapter 4, we saw examples

[1]Sections 5.1 and 5.2 describes joint work with Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul.

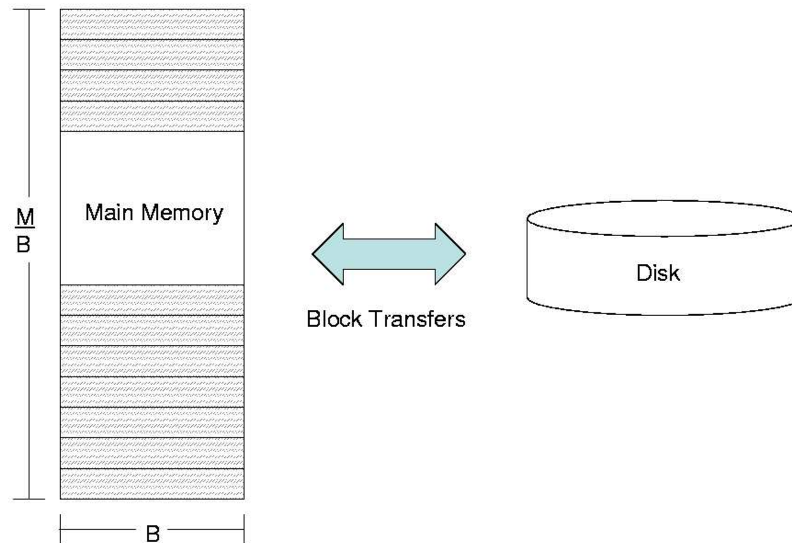


Figure 5-1: An illustration of the Disk Access Machine (DAM) model.

of this phenomenon. A single clock cycle on the newer test systems is less than 1 nanosecond. A `memcpy` between two arrays in memory takes a few microseconds, while using `fsync` to force data out to disk typically requires several milliseconds. The rotational latency of a 10,000 rpm disk is 6 ms, creating a lower bound on the worst-case time to read data from disk. Because the time to access disk is at least 6 orders of magnitude larger than the time for a single clock cycle, the cost of actual computation for a program that performs many disk accesses can often be ignored. Instead, the performance model traditionally used to analyze programs that access large data sets on disk is the *Disk-Access Machine* (DAM) model [3]. The DAM model assumes a system has two levels of memory, as illustrated in Figure 5-1. Main memory has size M, disk has infinite capacity. In this model, computation on data in main memory is free, but each transfer of a size-B block between main memory and disk has unit cost.

Using the DAM model we can analyze the cost of doing a single query on a B$^+$-tree. A B-tree can be thought of as a normal binary-search tree, except with a branching factor of $\Theta(B)$ instead of 2. A search on a B-tree storing N keys requires $O(\log_B N)$ block transfers: a constant number of transfers at every level of the tree. An information-theoretic argument proves a lower bound on the worst-case time for a dictionary operation of $\Omega(\log_B N)$. Thus, searches on B-trees use an asymptotically optimal number of memory transfers. A B$^+$-tree is similar to a B-tree, except that the data is stored only at the leaves of the tree, minimizing the number of block transfers by putting as many keys in a single block as possible. The fact that B-trees or variants of B-trees are widely used in practice corroborates the validity of the DAM model.

Cache-Oblivious B-Trees

The optimality of the B$^+$-tree in the DAM model requires that the implementation know the value of B. Unfortunately, in a real system, it is not always clear what the exact value of B is. For example, on a disk, the cost of accessing data in a block near the current position of the disk head

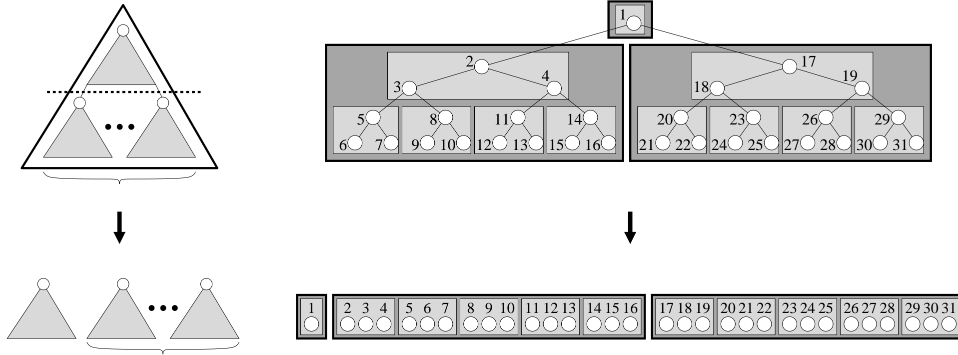


Figure 5-2: The van Emde Boas layout (left) in general and (right) of a tree of height 5.

is cheaper than accessing a block on a different track. There are multiple levels of data locality: two memory locations may be on the same cache line in L1 cache, the same line in L2 cache, the same page in memory, the same sector on disk, or the same track on disk. In a real system, there may not be a single "correct" block size B.

An alternative to the DAM model is the cache-oblivious model of computation [16, 39]. An algorithm is said to be *cache-oblivious* if it is designed to minimize the number of memory block transfers *without knowing the values of B or M*. A fundamental result for cache-oblivious algorithms is that any algorithm that performs a nearly optimal number of block transfers in a two-level memory model without knowing B and M also performs a nearly optimal number of memory transfers on any unknown, multilevel memory hierarchy [39].

A cache-oblivious B-tree (CO B-tree) [4] is a search tree structure that supports dictionary operations efficiently. The CO B-tree guarantees the following bounds on dictionary operations without needing to know the exact value of B:

1. Search: $O(\log_B N)$ memory transfers.

2. Range queries of R elements: $O(\log_B N + R/B)$ memory transfers.

3. Insertions and deletions: $O(\log_B N + \log^2 N/B)$ memory transfers.

The bounds for searches and range queries are asymptotically optimal. A CO B-tree achieves these bounds by organizing the tree in a *van Emde Boas layout* [39]. This layout is a binary tree recursively laid out in memory. A tree with N nodes and height h can be divided into one root tree with $\Theta(\sqrt{N})$ nodes and height approximately $h/2$, and $\Theta(\sqrt{N})$ child subtrees, each also with $\Theta(\sqrt{N})$ nodes and height $h/2$. In the van Emde Boas layout, each of these height $h/2$ subtrees is stored contiguously in memory, with the layout recursively repeated for each height $h/2$ tree. Figure 5-2 illustrates this layout for trees of height 5.

Intuitively, this layout is cache-oblivious because for any block size B, we can recurse until our layout eventually gets to a tree of height approximately $\Theta(\lg B)$. This tree fits entirely into one block, so any query from root to leaf in the original tree visits $O(\lg N)/\Theta(\lg B)$, or $O(\log_B N)$ blocks.

The van Emde Boas layout is sufficient for a static dictionary that does not support insertions or deletions. One method for creating a dynamic tree is to use this static tree as an index into a packed memory array [4]. A packed memory array stores N elements in sorted order in $O(N)$ space. The array leaves carefully spaced gaps in between elements and carefully maintains these gaps to satisfy certain density thresholds. These threshold ensure that large rearrangements of the array are be amortized over many insert or delete operations.

I have only sketched the details the the CO B-tree here. For a more thorough presentation of this data structure, I refer the reader to [4].

Data structure	Average time per search	
	Machine 4: small	Machine 3: big
CO B-tree	12.3ms	13.8ms
Btree: 4KB Blocks:	17.2ms	22.4ms
16KB blocks:	13.9ms	22.1ms
32KB blocks:	11.9ms	17.4ms
64KB blocks:	12.9ms	17.6ms
128KB blocks:	13.2ms	16.5ms
256KB blocks:	18.5ms	14.4ms
512KB blocks:		16.7ms

Table 5.1: Performance measurements of 1000 random searches on static trees. Both trees use 128-byte keys. In both cases, we chose enough data so that each machine would have to swap. On the small machine, the CO B-tree had 2^{23} (8M) keys for a total of 1GB. On the large machine, the CO B-tree had 2^{29} (512M) keys for a total of 64GB.

5.2 Serial B$^+$-trees and CO B-trees

In this section,[2] I present several experimental results that show the CO B-tree is competitive with the B$^+$-tree for dictionary operations. When doing random searches on a static tree, the CO B-tree ran 3% slower than the B$^+$-tree with the best block size on one machine and 4% faster than the B$^+$-tree on another machine. For dynamic trees, we observe that as the block size increases, the time to do random insertions in the B$^+$-tree increases, but the time for range queries and searches decreases. The CO B-tree is able to efficiently support all three operations simultaneously.

These experiments were conducted on Machine 3, which has 16 GB of RAM, and on Machine 4, which has only 128 MB of RAM.[3]

Random Searches on Static Trees

The first experiment performed 1000 random searches on a B$^+$-tree and a CO B-tree. On Machines 3 and 4, these static trees had 2^{29} and 2^{23} keys, respectively. These sizes were chosen to be large enough to require the machine to swap. For the B$^+$-tree, we tested block sizes ranging from 4 KB to 512 KB. In this test, we flushed the filesystem cache by unmounting and then remounting the file system before the first search.

From the results in Table 5.1, we see that the CO B-tree is competitive on Machine 4: the B$^+$-tree with the best block size only outperformed the CO B-tree by 3 %. For Machine 3, the CO B-tree was 4% faster than the B$^+$-tree with the best block size. This data also hints at the slight difficulty in finding the right block size B for the B$^+$-tree. On Machine 3, the best block size was 256 KB, while on Machine 4 it was 32 KB. Both values are significantly larger than the default operating system page size of 4 KB. For each machine, the B$^+$-tree needed to be tuned to find the optimal block size, while the CO B-tree was efficient without tuning.

Dynamic Trees

The next experiment tested dynamic trees on the smaller machine, Machine 4. We compared the time to insert 440,000 and 450,000 random elements for the CO B-tree, respectively. We chose these data points because 450,000 is the point right after the CO B-tree must reorganize the entire data structure. We also compared this data to B$^+$-trees with different block sizes. For the B$^+$-tree experiments, allocation of new blocks was done sequentially to improve locality on disk. This choice represents the best possible behavior for the B$^+$-tree. In a real system, as the data structure ages, the blocks become dispersed on disk, possibly hurting performance. Finally, we compared this data

[2]This section describes joint work with Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul

[3]These machines are described in Section 4.1. At the time of this test, however, Machine 3 was running a 2.4 kernel.

	Block Size	insert random values	range query of all data	1000 random searches
CO B-tree 440,000 inserts		15.8s	4.6s	5.9s
CO B-tree 450,000 inserts		54.8s	9.3s	7.1s
B^+-tree				
Sequential	2K	19.2s	24.8s	12.6s
block	4K	19.1s	23.1s	10.5s
allocation:	8K	26.4s	22.3s	8.4s
(450,000	16K	41.5s	22.2s	7.7s
inserts)	32K	71.5s	21.4s	7.3s
	64K	128.0s	11.5s	6.5s
	128K	234.8s	7.3s	6.2s
	256K	444.5s	6.5s	5.3s
Random block allocation:	2K	3928.0s	460.3s	24.3s
Berkeley DB (default parameters):		1201.1s		
Berkeley DB (64 MB pool):		76.6s		

Table 5.2: Timings for memory-mapped dynamic trees. The keys are 128 bytes long. The range query is a scan of the entire data set after the insert. Berkeley DB was run with the default buffer pool size (256KB), and with a customized loader that uses 64MB of buffer pool. These experiments were performed on the small machine.

Time to insert a sorted sequence of 450,000 keys	
Dynamic CO B-tree	61.2s
4KB Btree	17.1s
Berkeley DB (64MB)	37.4s

Table 5.3: The time to insert a sorted sequence 450,000 keys. Inserting sorted sequence is the most expensive operation on the packed memory array used in the dynamic CO B-tree.

to random insertions done into a Berkeley DB database using the `db_load` command. The buffer pool size for Berkeley DB was set to 64 MB.

The data in Table 5.2 demonstrates that the CO B-tree performs well, even at the pessimal point, just after reorganizing the entire array. For the B^+-tree, there is a tradeoff between small and large block sizes. Small block sizes imply that insertions are faster, but only at the cost of more expensive range queries and searches. The CO B-tree is able to efficiently support all three operations simultaneously. We did not observe any block size B where the B^+-tree was strictly better than the CO B-tree for all three operations.

Using Berkeley DB with the default tuning parameters, it took 20 minutes to load the data. Building a customized version of `db_load` with the buffer pool size set to 64 MB, however, we managed to improve Berkeley DB to run only 40% slower than the CO B-tree for insertions. Perhaps by tuning additional parameters, Berkeley DB could be sped up even further. Unfortunately, needing to optimize a large number of tuning parameters represents a disadvantage in practice.

In-Order Insertions

Finally, we ran an experiment testing the worst-case for the CO B-tree, when the data is inserted in order. Table 5.3 shows the time required to insert 450,000 elements in order into each search tree. In this case, the CO B-tree is about 65% slower than Berkeley DB. This behavior is reasonable considering we are testing the CO B-tree at a worst possible point.

In summary, these empirical results show that the performance of a serial CO B-tree for dictionary operations is competitive, and in some situations, actually faster than an optimally tuned B^+-tree or Berkeley DB.

Code	Serial Version	Using LIBXAC	LIBXAC Function Calls Added
B^+-tree	1122 lines	1527 lines	23
CO B-tree	1929 lines	2026 lines	17

Table 5.4: Changes in Code Length Converting B^+-tree and CO B-tree to Use LIBXAC.

5.3 Search Trees Using Libxac

This section explains how I parallelized the serial implementations of the B^+-tree and the CO B-tree tested in the previous section. This process was relatively painless and involved few changes to the existing code.

Parallelizing Search Trees Using Libxac

The serial implementations of a B^+-tree and a CO B-tree that I started with both stored data in a single file. Each tree opened and closed the database using `mmap` and `munmap`, respectively. I modified the open and close methods to use LIBXAC's `xMmap` and `xMunmap` instead. I supported concurrent searches and insertions by enclosing the search and insert methods between `xbegin` and `xend` function calls. In the implementation, no backoff method was specified; every transaction immediately retries after an abort until it succeeds.

Parallelizing these codes required only these few, simple changes. Table 5.4 gives a rough estimate of the size of the source code before and after modification with LIBXAC. Although counting the number of lines of code is, at best, an imprecise way to estimate code complexity, these numbers reflect the total programmer effort required to use LIBXAC. Less that two dozen LIBXAC function calls were required for each data structure. For the B^+-tree, 8 out of the 23 calls were actually optimizations, i.e., calls to the advisory function, `setPageAccess`. Also, most of the additional code for the B^+-tree was for testing concurrent insertions on the tree, not for supporting the data structure operations.

Because the conversion process was simple, I was able to successfully modify the CO B-tree structure in only a few hours, i.e., overnight. Since the CO B-tree was previously implemented by another student [29], most of this time was spent actually understand the existing code. The serial B^+-tree was also coded by someone else.

My experience provides anecdotal evidence as to the ease of programming with LIBXAC. Using this library, it was possible to modify a complex serial data structure to support concurrent updates, knowing only a high-level description of the update algorithm. Unlike a program that uses fine-grained locks, the concurrency structure of the program with transactions is independent of the underlying implementation.

5.4 Durable Transactions on Search Trees

In this section, I describe experiments performing insertions on LIBXAC search trees, with each insertion as a durable transaction. On newer machines, I found that the search trees coded with LIBXAC were actually competitive with Berkeley DB's B-tree, running anywhere from 4% slower to 67% faster. Tables 5.5 and 5.6 summarize the results from this experiment. For more details on the experimental setup, see Section C.6.

On a single process, on an `ext3` filesystem (Machines 1 and 3(b)), the average time per insertion on the LIBXAC search trees was over 60% faster than on Berkeley DB. On the Reiser FS filesystem (Machine 3(a)), the LIBXAC search trees ran only 4% slower than Berkeley DB. These results demonstrate that durable memory-mapped transactions with LIBXAC can be efficient.

It is unclear exactly why Berkeley DB takes so long to perform random insertions as durable transactions. It is possible that I have not tuned Berkeley DB properly, or that I have not taken

Machine	Search Tree	Avg. Time/Insert (ms) 1 Proc.	2 Proc.	% Speedup	Speedup
1	B$^+$-tree, w. adv.	18.3	14.6	20.2%	1.25
1	CO B-tree, no adv.	13.5	15.7	-16.3%	0.86
1	Berkeley DB	45.9	44.2	3.7%	1.04
3(a)	B$^+$-tree, w. adv.	7.7	7.7	0 %	1.00
3(a)	CO B-tree, no adv.	7.5	7.8	-4.0 %	0.96
3(a)	Berkeley DB	7.4	5.1	31.1 %	1.45
3(b)	B$^+$-tree, w. adv.	7.4	7.2	2.7%	1.03
3(b)	CO B-tree, no adv.	7.2	7.3	-1.4%	0.99
3(b)	Berkeley DB	22.4	17.7	21.0%	1.28
4	B$^+$-tree, w. adv.	82.0	–	–	–
4	CO B-tree, no adv.	66.5	–	–	–
4	Berkeley DB	57.7	–	–	–

Table 5.5: Time for 250,000 durable insertions into LIBXAC search trees. All times are in ms. Percent speedup is calculated as $\frac{100(t_1-t_2)}{t_2}$, where t_1 and t_2 are the running times on 1 and 2 processors, respectively.

Machine	B$^+$-tree vs. BDB 1 Proc.	2 Proc.	CO B-tree vs. BDB 1 Proc.	2 Proc.
1	60%	67%	71%	65%
3(a)	-4%	-51%	-1%	-53 %
3(b)	67%	59%	68%	59%
4	-42%	–	-15%	–

Table 5.6: The % speedup of LIBXAC search trees over Berkeley DB. Percent speedup is calculated as $\frac{100(t_L-t_B)}{t_B}$, where t_L and t_B are the running times on the LIBXAC and the Berkeley DB tree, respectively. Speedup is t_1/t_2.

full advantage of its functionality. The fact that I cannot simply use Berkeley DB with default parameters is another argument in favor of simpler interfaces like the one provided by LIBXAC.

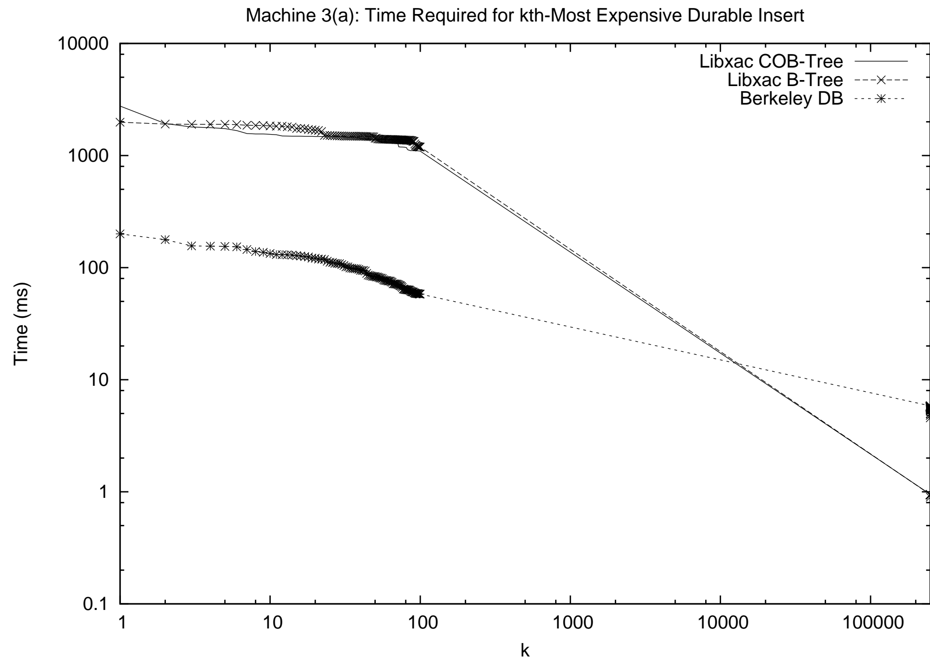
The LIBXAC search trees on Machine 3 achieve almost no speedup or slight slowdown going from one to two processes. These results are consistent with the previous data from the concurrency tests on durable transactions in Table 4.10: the simple transactions in concurrency Test A take about 6 ms on average, while the search tree inserts take about 8 ms. It is interesting that the B$^+$-tree achieves speedup about 20% speedup on Machine 1. One observation is that concurrency test A takes about 8 or 9 ms on Machine 1, while the B$^+$-tree inserts take about 18 ms. Thus, there may be more potential for speedup compared to Machine 3.

We can look a little more closely at the time required for individual inserts. Figure 5-3 plots the time required for the kth most expensive insert on Machine 3(a) and 3(b).

For all the search trees, only about 100 insertions require more than 100 ms. There is a sharp contrast between the LIBXAC search trees and Berkeley DB; the most expensive inserts for LIBXAC trees take over a second, while the most expensive inserts for Berkeley DB take on the order of a tenth of a second. The fastest inserts for LIBXAC tend to be faster than Berkeley DB however, taking on the order of a millisecond. The conclusion is that the Berkeley DB B-tree exhibits more consistent behavior than LIBXAC search trees, but on average the two systems are competitive.

Since LIBXAC relies more heavily on the operating system than Berkeley DB, the fact that some insertions with LIBXAC are expensive is not surprising. Also, since all results are real-time measurements, it is possible that some of these 100 expensive insertions include times when the program was swapped out for a system process.

Finally, although I do not present the detailed results here, I have observed that even when the



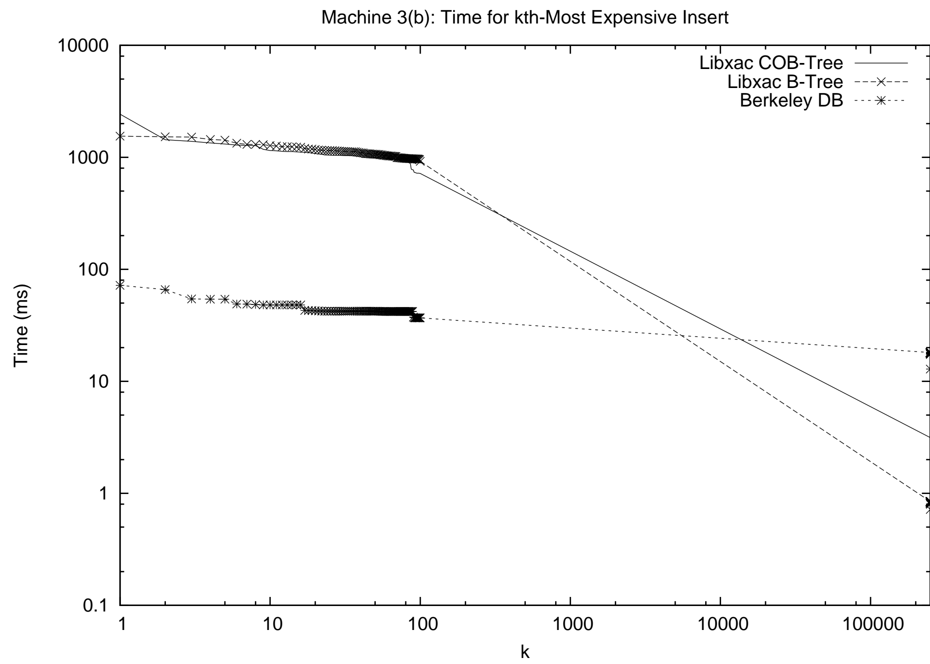


Figure 5-3: Machine 3: Time for kth most expensive insert operation.

write-cache on the harddrives are enabled, Berkeley DB and the LIBXAC search trees are comparable (see Appendix C, Table C.12). Although these transactions are not strictly recoverable, these results suggest that memory-mapped transactions using LIBXAC may still be efficient in other situations, (if our systems had harddrive caches with battery-backup, for example).

5.5 Summary of Experimental Results

In this chapter, I have presented experimental results testing the performance of search trees implemented with and without LIBXAC. The results in Section 5.2 show that a CO B-tree can simultaneously support efficient searches, insertions, and range-queries in practice. The CO B-tree is even competitive with a B-tree whose block size B has been carefully tuned. Section 5.4 shows that the LIBXAC B^+-tree and CO B-tree can support insertions as durable transactions efficiently. In the experiments I conducted, insertions using LIBXAC search trees ranged from being only 4% slower to about 67% faster than Berkeley DB. This last result is quite surprising, considering the fact that I am comparing an unoptimized prototype of LIBXAC with several significant flaws to a high-quality transaction system such as Berkeley DB.

Although this result is quite promising, I believe there is still significant work that needs to be done:

1. LIBXAC needs to be modified to fully support recovery on multiple processes, and a recovery process needs to be implemented and fully tested. Separating the log data and meta-data into separate files should accomplish this goal and not hurt performance if LIBXAC uses multiple disks, but it is impossible to know for sure without actual tests.

   In particular, one shortcut I took during implementation was to not calculate the checksum for each page during transaction commit. Appendix C, Table C.8 shows that calculating an MD5 checksum on a single page takes about 36,000 clock cycles on Machines 1 and 2 (about 12 to 15 μs). For small transactions that touch only a few pages, this cost seems fairly reasonable. For larger transactions, however, performing two calls to `fsync` to ensure that the commit record is written to disk after all the data pages is probably more efficient than computing a checksum. In some cases, I have noticed that the time for a second `fsync` is fairly small if it occurs soon after the first, possibly because the disk head does not move between the two writes. It would be interesting to more rigorously test whether performing two `fsync`'s during a transaction commit substantially impacts the performance of LIBXAC.

2. Berkeley DB supports group commits, i.e., allowing transactions on different threads or processes commit together with the same synchronous disk write. Modifying LIBXAC to support group commits may improve concurrency when the system uses a single disk.

3. The prototype currently limits a transaction's maximum size to around 64 or 128 MB.[4] Unfortunately, on the three newer machines, this constraint allows us to test only search trees that can fit into main memory. It would be interesting to test LIBXAC on large databases that do not fit into memory.

   Another interesting experiment would be to test LIBXAC in a memory-competitive environment, with other applications running simultaneously.

4. Currently, LIBXAC maintains its logs at page granularity; the runtime saves a copy of every page that a transaction writes, even if the transaction modifies only a few bytes on a page. A single run of the experiment doing 250,000 insertions to a B^+-tree or CO B-tree, LIBXAC generates approximately 5 GB of log files. In contrast, Berkeley DB only creates 185 MB of log files. There is significant room for improvement in the way LIBXAC maintains its logs.

In addition to questions related to the LIBXAC implementation, there are also several theoretical questions that these experiments raise:

[4]In Linux, a process is allowed to have at most 2^{16} different `mmap`ed segments. The LIBXAC runtime ends up creating 1 or 2 segments for every page a transaction touches.

1. In all these experiments, I have used the oldest-wins abort policy with no backoff when transactions conflict. A backoff loop may improve performance for concurrent insertions in practice. It would be interesting to experiment with other policies for contention resolution, especially if the Libxac runtime is modified to be more decentralized.

2. Although I managed to "parallelize" the CO B-tree, it is unclear whether this data structure still performs an optimal number of memory-transfers per operation. For example, some CO B-tree insert operations must rebalance the entire packed memory array, leading to a transaction that conflicts with any other transaction that modifies the tree. Appropriate backoff in this situation may improve the performance of a concurrent version of the CO B-tree.

3. The serial version of the CO B-tree written without Libxac is cache-oblivious by construction. Since Libxac supports multiversion concurrency by memory-mapping multiple copies of pages in complex ways, it is unclear whether the property of cache-obliviousness still holds. For example, in Libxac, it is possible for two adjacent pages in the user file to end up being mapped to two nonadjacent pages in Libxac's log file, and vice-versa. The behavior is even more complicated when operations are being done on multiple processors. One interesting research question to explore is whether a serial, cache-oblivious B-tree can be converted into a parallel cache-oblivious structure while still supporting multiversion concurrency.

In conclusion, I consider Libxac not as a finished product, but as work in progress. The Libxac prototype has some interesting features in its implementation, but there is much room for improvement. The fact that Libxac manages to support durable search-tree insertions as efficiently as the Berkeley DB B-tree in our experiments is a strong indication that memory-mapped transactions can be practical.

I have spent the majority of this chapter discussing performance, but arguably the most important result is the one I have spent the least time discussing. Libxac is intended to be a library that is easy to program with. For concurrent and persistent programs, the hope is that the ease of parallelizing serial data structures is the rule rather than the exception.