PERFORMANCE REPORT of

Data Structure Engineering: on memory BTree with variable data size
Ismail Safa Toy
Technical University of Munich
School of Computation, Information and Technology

Review

License: CC-BY
Review report received on: 02.06.2023

Summary

B+Tree, from now on B Tree, is an efficient data structure for storing variable data. It has broad applications, mostly in file systems and relational data base systems. The standard C++ implementation stdMap, is implemented with Red-Black Tree underneath, has logarithmic runtime for important operations. The self implementation uses the slotted layout and optimizations introduced in the lecture. In almost all operations the self implementaions is twice as fast as the standard implementation, the only slow operations being the scan.

Overall rating

The overall rating for this implementation is 4 out of 5.

Positive aspects

Approach: *It implements textbook slotted BTree, this allows for less memory allocations and therefore less interruptions to make syscalls. Overall, optimizations like fence keys and head-rest show a neglicible performance gain over the simple slotted BTree, as the roughly 13% of the execution is used up by byte swaps to ensure correct endianness for head-rest optimization.*
Performance: *It is quantitatively faster than the stdMap of the standard library in insert, remove and lookup.*
Structure: *A better structure would be to seperate the BTreeNode and the BTree to make the files smaller.*
Style & Content: *Code style is well kept with auto formatting. Names could be more verbose to help the reader, but it does not bother too much.*

Aspects to improve

Performance: *Scan function has a linear runtime. Next pointers are omitted for an upper pointer, or the ŕightest_most_child́ in the template. This was a headache to make it work. It was working for Integers and LONG setups, but personal random file test was not passing. There was a one byte off error with the lookup in the scan, even though lookup was working. So I gave up and used in order traversal to implement scan. This could be much improved to have also log time.*
Double Free and Memory Leaks: *During the destroy, there is a double free. Upper is deleted double! Also, the pointers allocated during split are not deleted even though all the references can be used for in order traversal (see in order scan implementation). This puzzles me. Another problem is the btree_lookup. The returned char pointer should be freed and to make the messy new u8[**1024**] a little less wasteful, I propose to get the size beforehand. This introduces massive overhead as it is 6% of the total execution. Also the allocated buffer is caller freed, which is not optimal for drop in replacements.*
Memory Bottleneck: *The Perf data shows, the implementation is mostly compromised of memcmp (40%) , as ex-*

pected, and also of bswap (13%). Copying data is relatively inexpensive as memcpy is only 4-5% of the execution time. march=native could be added to
Unaligned instructions: *In the perf overview there are a lot of instructions that are unaligned, like memcpy_avx_unaligned. Using álignasḿight reduce the cost of unaligned execution.*
Heap-Buffer-Overflows: *The padding was not propagated properly in the BTreeNode. This is now fixed by setting it to 0.*
Lack of quality tests: *The proposed implementation is tested via randomised text input of varying sizes (excluding the value-key size over 1Kib). The test suite is written in python and uses standard libraries. Generates files with desired row and column lenght. It is tested between 1-512 Byte sizes. Even though random testing is a good start but static and dynamic testing with more robust testing suites should be added, like doctest.*
Cache Misses: *I could not use ´-e cache-misses ´ in perf as my kernel does not support this. I tried in rechnerhalle but the perf is also not reliable there. This is an open question till I fix the perf issue locally or try it on the remote server.*

Additional Tables

The stdMap has insane fast scan, which does not make sense as it should have log time also. Other than that the speed difference is clear between the custom implementation and the stdMap. This is because of the simplicity of the custom implementation and the optimizations, mostly the slotted layout and the fences. The perf run of the stdMap also shows that the 65% of the run is in iterator and not in the memory operations like in our implementation. This is probably the reason for the speed difference. As comparing memory in AVX registers is much faster than the Iterators that use a lot of references to next and hasNext. The data is harvested on a local computer with nothing running in the background. AMD 5950hs, 32 gb 3200 MT/s, Clang ver. 14.0.0-1ubuntu1.1.
The stdMap has insane fast scan, which does not make sense as it should have log time also. Other than that the speed difference is clear between the custom implementation and the stdMap. This is because of the simplicity of the custom implementation and the optimizations, mostly the slotted layout and the fences. The perf run of the stdMap also shows that the 65% of the run is in iterator and not in the memory operations like in our implementation. This is probably the reason for the speed difference. As comparing memory in AVX registers is much faster than the Iterators that use a lot of references to next and hasNext.

Operation	Time
Insert	0.32
Scan	43.26
Lookup	0.37
Remove	0.29

Table 1: BTree: 1e6

Operation	Time
Insert	0.96
Scan	0.44
Lookup	0.93
Remove	0.97

Table 2: stdMap: 1e6

Operation	Time
Insert	5.96
Scan	568.31
Lookup	5.63
Remove	4.30

Table 3: BTree: 1e7

Operation	Time
Insert	16.55
Scan	6.03
Lookup	17.05
Remove	16.94

Table 4: stdMap: 1e7

Operation	Time
Insert	39.24
Scan	3182.53
Lookup	40.22
Remove	34.79

Table 5: BTree: 5e7

Operation	Time
Insert	100.42
Scan	39.43
Lookup	103.21
Remove	101.73

Table 6: stdMap: 5e7