Software Programming for Performance

In-Memory Key Value storage

18.02.2020

4 Horsemen:

>> Mohsin Mamoon Hafiz - 2018101029

>> Srivathsan Baskaran - 2018101049

>> Pushkar Talwalkar - 2018101010

>> Bhavesh Shukla - 2018101036

Overview

The project requires us to build an In-Memory Key-Value Storage data structure in C++.

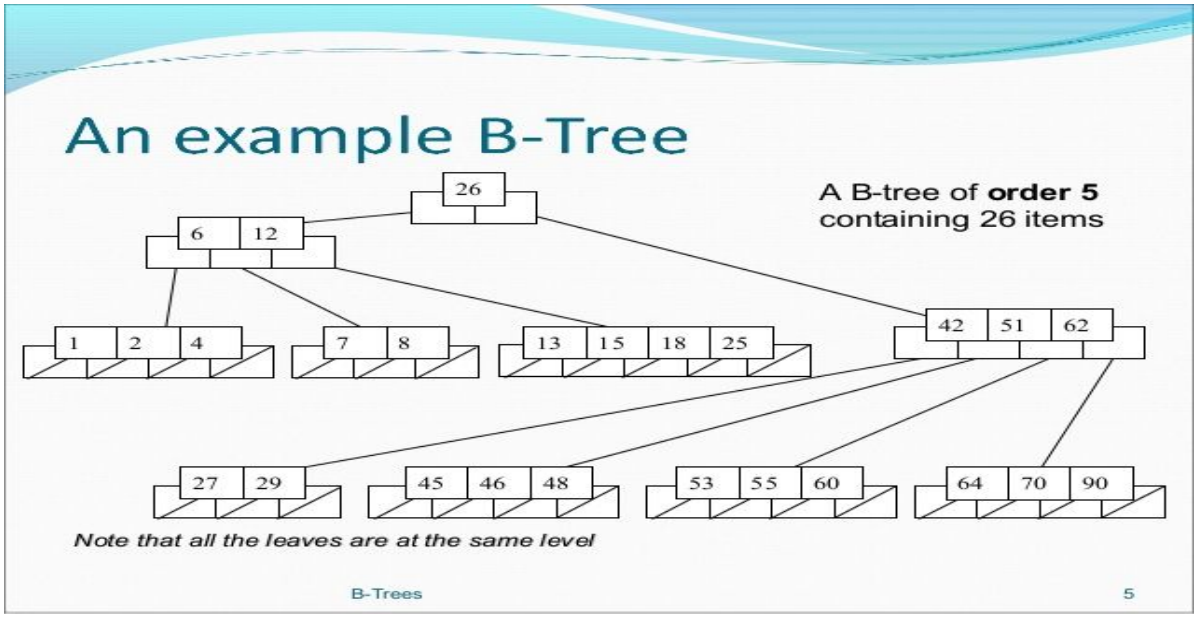
A data structure needs to be built which stores these Key-Value pairs and facilitates the following operations:

1. **get(key):** **Return value for the key**
2. **put(key, value):** **Insert key-value, overwrite value if it already exists.**
3. **delete(key) :** **Delete the key-value from the data structure.**
4. **get(int N):** **Return the Nth key-value pair**
5. **delete(int N):** **Delete the Nth key-value pair from the data structure.**

The project demands these operations on the data structure to be fast, efficient and consuming less memory and CPU while having a high transaction rate.

Data Structure used

In the project we used B-Trees. We had a discussion between the 2 subteams - to implement B-Tree or Red-Black tree but in the end it was decided that B-Tree would be better overall.



Why we chose B-Trees over Red-Black Trees

B-Trees have better cache locality than red-black trees.

This can be explained by the following reasoning: B-Trees have arrays at each level of the tree. As we know, this implies excellent cache locality as the nodes are stored in a contiguous block of memory**(spatial locality)**. On the other hand we have red-black trees which have pointers strewn all over the place, which implies bad cache locality.

Another reason behind this decision was simplicity. We found B-Trees much simpler to implement than red-black trees.

Advantages of B-Trees

1) As more nodes can be stored at the same level, the height of the B-tree is much lesser.
2) Better than any ordinary tree due to better cache locality as nodes at a level are allocated contiguous blocks of memory, as they form an array.
3) The tree is balanced, thus ordinary search, deletion and insertion operations take roughly uniform time, O(height of tree) = O($\log_b n$). This is in stark contrast with data structures such as BST which are unbalanced.
4) They accommodate random insertions and deletions at a low cost with relatively less cost whilst remaining balanced.
5) Sometimes it is possible to find the search key-value before reaching the leaf nodes.

Disadvantages of B-Trees

1) Program complexity increases.
2) The operations executed at each level of the node are more complex than the ones executed in a simple BST.
3) Only a small fraction of key-value pairs are found before reaching the leaf nodes.

Operations Analysis : Time and Space Complexity

n = net total no. of nodes

Operation	Time Complexity	Space Complexity
get(key)	$O(\log_b n)$	O(1)
put(key,value)	$O(\log_b n)$	O(1)
delete(key)	$O(\log_b n)$	O(1)
get(int N)	$O(\log_b n)$	O(1)
delete(int N)	$O(\log_b n)$	O(1)

Usage of Hashmaps

Using a hashmap in the traditional sense does not work well due to the large number of collisions. To handle the collisions, we had decided to implement a hashtable where each node of the hashtable would be a B-Tree.
This way, we would get to a B-Tree in O(1) using the hash value and then, any operation on the B-Tree would take O(log h) where h is the no. of elements in the found B-Tree.
On average, for n = 10^7, h = 10^7/size_of_hastable which is very less and would improve performance overall.
But, the main issue that we faced is get(int N) and del(int N) operation was getting significantly slower and hence we decided not to go with hashing

Details of the analysis

We were considering implementing a hash table. In the hash table there would be 345437 B-trees, in which the key value pairs would be organized in ascending order.
NOTE : ALL log() are of base 2 unless stated otherwise.

First , we mapped the values of

A->1, B->2,...,Z->26

a->27,b->28,...,z->52

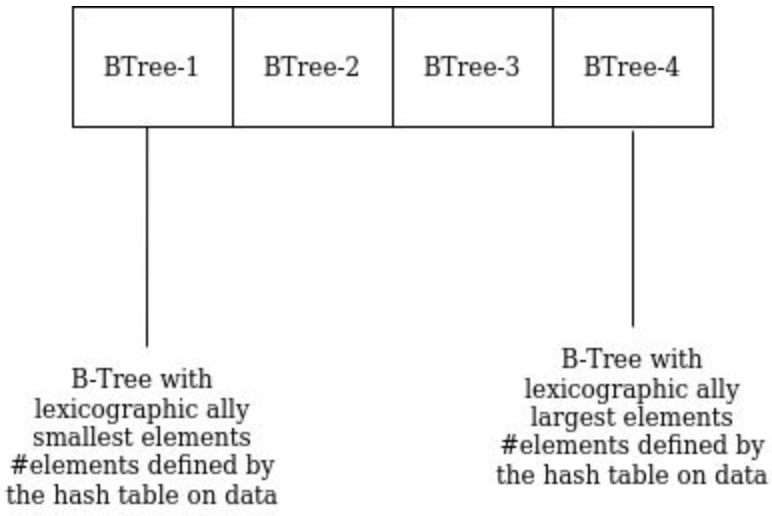
The hash function had to be selected such that for two strings a, b:

If (strcmp(a,b) < 0) then hash(a) < hash(b)

This would ensure the correctness of get(int N) and del(int N).

An optimal hash function that we decided upon is defined as follows:
3^8* key[0] + 3^4*key[1] + 3*key[2]

Fig : Hashmap in which element is a BTree.



The hash function is chosen such that the smallest element from a later B-Tree is greater than the greatest element of an earlier B-Tree. This is intended to reduce the height of each tree whilst ensuring that all the operations occur in a reasonable amount of time.

The algorithm involves hashing a given key to be searched/deleted/inserted and performing the apt operation in log(H) of that particular tree. For the get(n) and delete(n) operations, we store in each B-tree(in the hashmap), the number of elements in that particular B-tree. We then sequentially go through each of the B-trees (worst case 345437, average case 345437/2) , searching for the apt B-tree and upon finding it, we do the necessary operation in log(H).

Now, let us assume that there are 10^7 entries in the database. The number of B-Trees is of order 10^5 .
Then the average number of elements per B-tree is 100.
Thus, with the following table, we encapsulate the differences in the 2 approaches.

Note : We are keeping the #operations in integers.
ops: number of operations
We see that a B-tree without hashmap performs better than multiple trees indexed by a hashmap , hence we do not use hashmaps.

OPERATION	**WITHOUT HASHMAP**	**WITH HASHMAP**
get(key,value)	O(log n) = 23 ops	O(log H) + O(1) = O(log (n/10^5)) = 6.64 = 7 ops
put(key,value)	O(log n) = 23 ops	O(log H) + O(1) = O(log (n/10^5)) = 6.64 = 7 ops
delete(key, value)	O(log n) = 23 ops	O(log H) + O(1) = O(log (n/10^5)) = 6.64 = 7 ops
get(int n)	O(log n) = 23 ops	O(#B_Trees)+O(log H) = 10^5 + 7 ops

delete(int n)	O(log n) = 23 ops	O(#B_Trees)+O(log H) = 10^5 +7 ops

This analysis can be extended to varying numbers of BTrees in the hash table. Say, if the #B_Trees is in order of 10^4, then the operations will have similar trends, just the values will be less skewed.

Operation	#B_Trees = 10^4	#B_Trees = 10^3	#B_Trees = 10^2
get(key,value)	O(log H) = O(log(n/10^4)) = 9.96 = 10 ops	O(log H) = O(log(n/10^3)) = 13.287 = 13	O(log H) = O(log (n/100)) = 16.609 = 17
put(key,value)	O(log H) = 9.96 = 10 ops	O(log H) = 13.287 = 13	O(log H) = 16.609 = 17
delete(key, value)	O(log H) = 9.96 = 10 ops	O(log H) = 13.287 = 13	O(log H) = 16.609 = 17
get(int n)	O(#B_Trees) + O(log H) = 10^4 + 10	O(#B_Trees) + O(log H) = 10^3 + 13	O(#B_Trees) + O(log H) = 100 + 17 = 117
delete(int n)	O(#B_Trees) + O(log H) = 10^4 +10	O(#B_Trees) + O(log H) = 10^3 + 13	O(#B_Trees) + O(log H) = 100 + 17 = 117

These values are the worst case values, but even if it is the average case, all the get(n) and delete(n) operations reduced by half, they are still significantly higher than the no hashmap implemented function. Thus, the no hashmap implementation is better.

NOTE : WE HAD TRIED HASHMAP AND WE REALIZED IT IS SIGNIFICANTLY SLOWER. WE HAVE ATTACHED THE HASHMAP VERSION CODE SNIPPET AT THE END OF THIS REPORT.

Optimizations

We found the following strategies very useful:

- Avoiding loop invariant code in the loop.
- Using temporary variables wherever possible. Even in the simplest of things, such as " X = Y - 1 ".
- Using cache locality to our advantage. When possible declaring variables near the first time they are operated on.
- Minimizing conditionals
- Using Short-circuiting where possible without errors. [Error Eg: segmentation fault in case of invalidity of checking a particular character] We do this while balancing 2 central factors, as it often involves a tradeoff:
    - The likelihood of truthness of the statement (most likely **false** statement in case of a "**and"** and most likely **true** statement in case of an "**or"**)
    - Computational comparison of operation. Eg: strncmp for a 64 length string is heavier than the comparison of 2 integer values.
- Using register_uint8 variables where possible : Thus, if and where possible the variable is stored in a register.
- Avoiding dereferencing pointers - They are very slow. Using a temporary variable instead is much faster in almost any case.
- Loop unrolling provides success in certain cases:
    - In small loops , total loop unrolling can help quite a lot.
    - In larger loops, partial loop unrolling is best.
- Inlining functions - Inlining small functions helps in speeding up the operations. Especially useful in avoiding overhead of the function calls.
- Depending on the size of the function and the instruction cache size, shortening the function size can help. Eg :Consider a system with 32

bytes instruction cache. A small function , occupying just 35 bytes of memory , if possible, must be reduced to 32 bytes to fit in 1 instruction cache block. This can be done by removing newlines, tab spaces and spaces if and when possible. Eg: removing new lines and continuing the same line (by virtue of C++ ';'), the length of the function can be reduced.

- ➢ Avoiding global variables in the code. Reason : Global variables are not assigned a register, while local variables are assigned registers wherever possible.

Sources of error

- ➢ DVFS
- ➢ Optimization on my one computer need not imply optimization on another system.

Hashtable Implementation Snippets:

Class Definition and Constructor:

```
class kvStore {
    BTree **tr;
    int max_ent;
    public:
        int pres_ent;

        kvStore(uint64_t max_entries){
            tr = new BTree* [345437] {0};
            max_ent = max_entries;
            pres_ent=0;
        }
```

get(Slice &key, Slice &value)

```
bool get(Slice &key, Slice &value) {
    int hh = hash(key.data, key.size);
    if(tr[hh] != NULL){
        char * returned = tr[hh]->search(&key,&value);
        if(returned == NULL) return false;
        else {
            return true;
        }
    }
    return false;
```

put(Slice &key, Slice &value)

```
bool put(Slice &key, Slice &value){
    int hh = hash(key.data, key.size);
    if(tr[hh] == NULL) tr[hh] = new BTree(3);
    bool updated = false;
    tr[hh]->insert(&key, &value, &updated);
    if(!updated){
        pres_ent++;
        tr[hh]->elems++;
    }
    return updated;
}
```

del(Slice &key)

```
bool del(Slice &key){
    int hh = hash(key.data, key.size);
    if(tr[hh] != NULL){
        bool removed = true;
        tr[hh]->remove(&key, &removed);
        if(removed){
            pres_ent--;
            tr[hh]->elems--;
        }
        return removed;
    }
    return false;
}
```

get(int N, Slice &key, Slice &value)

```
bool get(int N, Slice &key, Slice &value) {
    N++;
    for(int i=0; i<345437; i++) {
        if(tr[i] != NULL) {
            int elems = tr[i]->elems;
            if(elems >= N) {
                return tr[i]->lognnth(N, &key, &value);
            }
            else N -= elems;
        }
    }
    return false;
}
```

del(int N)

```
bool del(int N) {
    Slice key, value;
    if(get(N, key, value)) {
        char* todel = new char[key.size];
        strncpy(todel, key.data, key.size);
        key.data = todel;
        return del(key);
    }
    else return false;
}
```

References

Geeks For Geeks
Wikipedia
Youtube