Hongyu Zou/ Shicheng Fan

Arun Kumar

CSE 132C

CSE 132C Design Report

1. Design Choices

a. Using if/else to select type instead of using C++ Template

Since the node struct provided in the .h file are all in C struct format, we decide to use if/else to create index of different type. I acknowledged that this involves copying and pasting a lot of code, but this also provides flexibility for different types of node struct, since sometimes it may not be feasible to incorporate different types of nodes with one template.

The other thing I worried about is that the TA of previous quarter tried to use template but failed. Since we want to keep our project on schedule and do not want to spend lots of time picking up templating for C++, we decide to use if/else to implement different types of node.

b. Add additional field keyArrayLength to the node struct

Since the node struct provided has a C array of constant length, it is hard to keep track of the length of the array if we do not have a length variable. For example, if we have 0 in the array, then we might not know whether there is a 0 in the array (Int array is initialized to all 0s). By adding a keyArrayLength field to the node struct, we achieved more flexibility.

c. Add helper function to help insertion

We use several different helper functions to implement insertion. We first use findPageNoInNonLeaf () to find a path to walk down the tree. When we are at leaf level, we use insertKeyToLeaf() or splitLeafNode() to handle two different cases that are possible when inserting a node into the leaf. Similarly, we use insertKeyToNonLeaf() and splitNonLeafNode() to handle different cases that are possible when inserting a node into non leaf. So, these helper functions are much easier to test than a monolithic function and they also provide modularity to the implementation.

d. Changed highValString and lowValString to char[] type

In startScan() function, the input is char\* if the key of index is String. Since we only compare first ten characters of the string, then it is a lot easier to use char[] instead of std::string. We can just use memcpy() and strncmp() to implement comparison between different string keys. If we use std::string there will be additional casting involved.

e. Fine granularity of unpin()

We unpin pages every time we finished using the page. This will definitely add more code but it will help preventing buffer overflow problems. But we always keep the root node in the buffer memory since it will always be accessed.

f. Efficiency

We basically follow the algorithm taught in the lecture to implement B+ tree index. But there are several aspects that can be improved. First, when we insert/search something into the keyArray, we use linear scan instead of binary search. Second, when we split the node, we add temp array to store key values and a map to store key rid/pageNo mapping, which will cost extra space.

2. Duplicate keys consideration

We did not do anything special to handle duplicate keys. When there are duplicate keys inserted, it will be inserted near to the key with the same value. And both keys will have the pointer pointing to the same pageNo/rid. So walking from top of the tree to leaf level and start scanning will still work fine. Both keys will appear in the leaf level.

3. Test Design

Test designed is in the main() function of main.cpp file as comment format.