UW-Madison CS564 Database Management Systems

Dec. 13 2015

Project 4 Design report

Yexin (Bruce) Wu, Buzun Chen

This project asks us to implement a single-attribute B+ index tree with basic functions include create/read index files, insert entries, and range searches. We are provided a simulation of pages and buffer manager classes. This document is a basic explanation of the design behind our final implementation.

Templates: We use templates to reduce the amount of duplicate code. We use L\_T and NL\_T, which refer to leaf or non-leaf nodes; P\_T and RID\_T, which refer to pageKeyPair<T> and RIDKeyPair<T>. In this way we can pass information upwards from lower level in an efficient manner. We also have type T to indicate the data type in tuples, which is passed all the way down from the top level. T can be either int, double or char\*.

Constructor:

The constructor of the BTreeIndex requires us to read from a previously created index file if existed, or create a new B+tree index file by recursively inserting all tuples in the relation.

We wrote two methods to accomplish this task: createIndexFile() and openIndexFile().

In createIndexFile(), we flush all contents corresponding to the new index file in the buffer manager to disk after all insertions are completed. We paid extra attention to properly unpinning any pages read or allocated. It would be too much if we list all individual cases here, but the general principle is that we pass in the page number of the nodes we need to modify/extract info from, and make sure each recursive level does not increase the pin number of pages outside of that level.

insertEntry() function:

Once we read in a tuple, we need to insert it into our B+tree. We consider two cases at this recursive level:

case(a): The current rootnode is a leaf node. This is the case during the first few insertions when the root node is the only node in the tree. In this special case we call the insertRootLeaf() function.

case(b):  
If the current rootnode is a non-leaf node, it means that we need to traverse down the tree to find the correct leaf node to insert. We do this by calling the traverse() method, which will not only insert the entry into leaf node at the deepest recursive call, but also return to its previous recursive level a generated new non-leaf entry (if exists).

We then check if this returned entry has valid page number in it. This returned entry was generated in the form of a pageKeyPair<T>, which contains both the page number of the newly generated leafnode/nonleafnode, and the smallest key in the newly splitted node. After the recursive call returns to the top level, we check if the current root node is full. If it is we split the old root node and create a new root.

insertRootLeaf() function:  
This function is called if the current root node is still a leaf node (i.e, only one node in the entire tree), and a new entry need to be inserted into this node. Insertion will happen as usual if the root leaf is not full. If it is full, we will split the current root leaf node and call createNewRoot() method.

createNewRoot() function:  
We call this function when a new root needs to be created. Note there are two cases here, whether the current root node is a leaf or not. If it's not, the newly created root node will have level 0. If it is, the newly created root node will have level 1.

traverse() function:  
In order to insert an entry/tuple into our B+tree, we need to first traverse to the right leaf node of the tree. To do this we wrote a traverse() method, which recursively goes down the tree if the level of current node is not 1. The implicit assumption here is that in each recursive level the node being traversed currently is a non-leaf node.

We ensure this assumption is satisfied by first checking if the level of current node is 1:

If it is, we find the right leaf node to extract, then try to put our insert entry into that leaf node. We then check if such action caused the leaf node to split, if so we need to insert a non-leafnode entry into the current non-leaf node. After we are done inserting entry at the current level, we pass the newly generated non-leaf entry for the previous level (if such entry exists) to the previous level.

If the current level is not 1, that means we need to traverse down the tree further. We pass in the appropriate parameters, and check if any new entries are returned from the level below. If so we need to insert entry into current non leaf node, and the rest of the procedure are similar with the level 1 case.

splitLeaf()/splitNonleaf() function:

For our leaf/nonleaf nodes, if the next level generated a non-leaf entry for the current level and the current node is already full, we need to call the splitLeaf/splitNonLeaf method. We split the current node by half, and create a new leaf/non-leaf node to store the right half of the old node. From this method we also obtain a non-leaf node entry to return to the level above.

putEntryLeaf/putEntryNonLeaf method:

If we need to put a leaf/nonleaf entry into a node, we call this method. Note that before calling this method we check if the current leaf/nonleaf node is full. We only call this method if they are not full.

We find the right position to insert our entry by looping through the keyArray of the node linearly, check if the element we are looking at is greater than our inserting entry. If not we go to the next element in the array. Note that for some cases our calculated insert position may be off by one, in those cases we would have to subtract one from that position before use.

Once we found the appropriate insert position, we shift every element right to that position by one, then insert our entry at position. Since our key type is char\* for strings, we cannot simply use the = operator to shift elements. We wrote an assign() method with specialized template to solve this problem.

assign/assignPrime() method:

We have assign() method to change values for char\*, and assignPrime() to change values for int/double types. These two methods are necessary since we cannot cast variables inside methods using templates.

compare method:

Same reason as assign() method. Takes in two variables, return 1 if the first element is bigger, 0 if equal, -1 if smaller.

startScan method:

We use this method to scan our data stored in the B+tree. After checking whether the inputs are valid, we need to traverse down the tree to find the correct position for the lowerbound value. We do this by calling scan() method.

scan method:  
The ideas for this method is similar to traverse(), but we do not return any variables. After we found the appropriate leafnode page and nextEntry, we set these two global variables accordingly.

findPos method:

In order to find the right position for the next entry for the scan, we create this method to traverse through the leaf/non-leaf node appropriately. This method is used by scan() to find the initial position for the scan.

scanNext method:

After user called this method, we first check if the nextEntry value is still valid under current constraints. Also we check if the current leaf node