Project 4 Design report

UW-Madison CS564 Database Management Systems
Dec. 13 2015

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 functions 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()` function, 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()` function.

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()` function, 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 function. 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 function we also obtain a non-leaf node entry to return to the level above.

putEntryLeaf()/putEntryNonLeaf() function:

If we need to put a leaf/nonleaf entry into a node, we call this function. Note that before calling this function we check if the current leaf/nonleaf node is full. We only call this function 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() function with specialized template to solve this problem.

assign()/assignPrime() function:

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

compare() function:

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

startScan() function:

We use this function 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()` function.

scan() function:

The ideas for this function 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() function:

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

scanNext() function:

After user called this function, we first check if the nextEntry value is still valid under current constraints (if the scan is still active, if nextEntry's corresonding key value is above the upper bound). Also we check if the current page number is 0. If so, then we throw `IndexScanCompletedException`.
We do this because in the following code, we traverse the entries in leaf node linearly, and change current page number if the entires on one page have been iterated through. If the next page obtained by `currLeaf->rightSibPageNo` has a page number of zero, that means we have already gone through all entires in B+tree. We set the current page number to zero and return. Then when scanNext is called one iteration later,
`IndexScanCompleteException` will be thrown and the scan will exit.

Test cases:

In addition to the provided tests and `errorTests()` function, we created two additonal tests in our main.cpp file. One is for reading existing B+tree Index from previously created files. To test this function, user need to first call `badgerdb_main` and enter command line option 6, this will create three B+Tree Indexs for consecutive numbers from 0 to 5000. Then user can call normal tests to see if the tree behaves normally when the indexes are read from disk. We added another test to see if our tree can handle range from (-1000,6000), and it behaved as expected. The additional code can be seen in individual type tests like `intTests()` function.

We also tested cases where non-leaf splits occurred. To shorten our runtime, we shrinked the nonleaf node capacity to 10 to create a three-level B+Tree. We also tested the case where inserted keys are sparse, we did this by generating 3000 numbers with range from 0 to 100000. However these two test cases are not compatible with other cases, and for time purposes we did not include them in our final main.cpp file. An code snippet for sparse test is included in our handin directory.

Duplicate key scenario:

If duplicate keys are allowed,for nonleaf nodes, the mechanism will be the same, however for leafnode entries we can create an additional layer beneath the original leafnode. Each leafnode entry will now point to a position in the additonal layer, where the entries after the position will store values greater or equal to the key value in leafnode entry. By creating a list of duplicate keys underneath the original leafnode, the shape property above the addtional level is maintained. In the lowest level, key values are stored in sorted order, and we can do range search similar to the original design with similar speed.