B+ Tree Index Manager Design Report

Group Member: Tianyi Zhao, tzhao86@wisc.edu

Bowen Yao, byao8@wisc.edu

Xuxiang Sun, xsun272@wisc.edu

# Prerequisites:

According to the instructions document, below are some of the specifications that draws limit upon the use case of our B+ Tree Index Manager:

1. The index will store data entries in the form *<key, rid>* pair.
2. All keys are of integer type.
3. Never insert two data entries into the index with the same key value.
4. All records in a file have the same length.
5. The B+ Tree only needs to support single-attribute indexing.
6. The *level* member of each of the non-leaf structure is set to 1 if the nodes at this level are just above the leaf nodes. Otherwise, *level* is set to 0.

# Implementation and Design Choices:

This section is divided into the following parts: Insertion documentations and Scanning documentations. In each of the part, specific implementation plans are described. Any given function or added helper function is documented as well. More detailed documentations can be found in the comments in the source code.

**Insertion**: This part documents the insertion procedure followed in this B+ Tree Index Manager, as well as any helper function or added private variables. A flow chart diagram showcasing the overview of insertion procedure is also attached.

*insertEntry()*: This is a provided function that serves as the wrapper for the other helper functions below. During insertion, we first search for the node to insert into by calling *searchEntry()*, then, we insert the entry by calling *insertEntryLeaf()*. Note that a vector, *std::vector<PageId>* *visitedNodes*, is initialized in this function. C++ Standard Template Library is used here for efficient initialization of the vector data structure. *visitedNodes* stores and tracks a list of visited pages as we traverse down the B+ Tree. This vector is used to track the parent node of any node during splitting. Without it, each split would require the program to search for a parent node starting from the root node again.

*searchEntry()*: This is an added private helper method. It searches for the node in B+ Tree where the wanted key value belongs recursively. It loops through all keys in the node and stops once it finds a key that is strictly greater than the given key value. When there is only one root in the tree, it defaults to return the *rootPageNum*.

*insertEntryLeaf()*: This is an added private helper method. It inserts the provided data entry <key, rid> pair into the leaf node specified by *pageNo*. If the leaf node has enough space, the entry is inserted into keyArray and ridArray of current leaf node by shifting up elements in keyArray and ridArray until the right slot to insert into is found. Otherwise, splitting of the leaf node is performed by calling *splitLeaf()*. Note that the variable, *numOccupied*, is introduced in struct *LeafNodeInt*. This variable tracks the number of filled key slots in a leaf node.

*splitLeaf()*: This is an added private helper method. It splits a leaf node after an overflow in leaf node. The size, or capacity, of the new node is calculated based on *leafOccupancy* using the formula *ceil((leafOccupancy+1)/2*. It loops through every key slot in a leaf node and chooses to either insert in the new node or insert in the original node by comparing the value of the key with each element in node. A new root node is created if the split results in the creation of one.

*insertEntryInternal()*: This is an added private helper method. It inserts the key that was propagated up into the internal node specified by *pageNo*. If the internal node has enough space, the entry is inserted into keyArray and pageNoArray of current leaf node by shifting up elements in keyArray and pageNoArray until the right slot to insert into is found. Otherwise, we split by calling *splitLeaf()*. Note that two variables are introduced. The variable *numOccupied* is introduced in struct *NonLeafNodeInt*. This variable tracks the number of filled slots in an internal node. The variable *splotFromLeaf* is a boolean variable that tracks whether the splitting happens at an internal node 1 level above leaf node or more levels above leaf node. This variable is need because there are two cases when inserting in an internal node: if the splitting happens at an internal node 1 level above leaf node, we shift *pageNoArray[currInternalNode->numOccupied-i+1]* down by one index and put *newPageNo* into that index. If the key to be inserted is not coming from a leaf node, this means that the key has already been propagated up.

*splitInternal()*: This is an added private helper method. It splits an internal node after an overflow in internal node. The size, or capacity, of the new node is calculated based on *nodeOccupancy* using the formula *ceil((nodeOccupancy+1)/2*. It loops through every key slot in a leaf node and chooses to either insert in the new node or insert in the original node by comparing the value of the key with each element in node. A new root node is created if the split results in the creation of one.

Diagram

Description automatically generated

**Scanning**: This part documents the scanning procedure followed in this B+ Tree Index Manager, as well as how often a page is pinned or unpinned in the buffer manager.

*StartScan()*:

End the scanning if it is called in the middle of the last scanning.

We first need to check whether the given arguments are valid: Is the lower boundary less than the higher boundary? Is the given operator valid and in the correct position? After that, we need to get the leaf node that contains our first record to scan by searching through each level in B+ tree, starting from the root node. Then, we get the first record to scan by searching in the current leaf node. Noticed that we use the linear search instead of the binary search in this case. *NoSuchKeyFoundException* will be throwed if all records in the current leaf node are outside the lower or higher boundary. Otherwise, we update the nextEntry pointer to the first record to scan, and unpinned the page.

*ScanNext()*:

We use the value of nextEntry to determine if the current record is valid. If the value of nextEntry is -1, means the current record is invalid, we should unpin the page and throw the *IndexScanCompletedException*. To reach the next record, we need to check if the next record is in the current leaf node. If the next record is in the right node of our current leaf node, we need to point our currentNode to the right node of our current node. Otherwise, we don’t need to change the current page.

Before we update our nextEntry pointer, we need to check if the next record is valid and inside the given range. We can do that by comparing the value of our current record with the higher boundary to see if the value is in the range given by the higher value and the higher operator.

*EndScan()*:

We end the executing scan and throw *ScanNotInitializedException* if there is no executing scan. No need to unpin the page since we already unpinned the page in *startScan* and *scanNext*.

# Tests and Correctness:

The provided tests already test the correctness of ascending, descending and random ordering relation of size 5000 and all the possible errors that may be thrown. To further validating our program, we added some more tests in *intTest()* that focuses on testing edge cases and out of bound cases to ensure correctness of our program. We also added three additional test functions in main.cpp: *test4()*, *test5()* and *test6()*. Each of them tests the correctness of ascending, descending and random ordering relation of size 300000 to make sure the B+ Tree also works for large relation inputs. (We modified given three relation-creation functions to accept input to modify the relation size, and modified *intTest()* to accept input to separate the test cases added for large relations in order to realize our tests.)

# Efficiency analysis:

Insertion analysis: (*N* = total number of elements in the Btree)

1. Case: There is only a root in Btree, then root is also a leaf node, the time complexity of insertion is *O(leafOccupancy) = O(1)* since leafOccupancy is a constant and our implementation is just looping on the leaf node.
2. Case: There is more than one level in Btree, we need to search the leaf for given key and do the insertion. Search a leaf in the Btree’s cost is *O(logN)* because our implementation is doing recursion call on only one node of each level, and traverse a node to decide which node on the below level to do the recursion is a constant cost *O(nodeOccupancy) = O(1)*. Insertion cost is *O(leafOccupancy) = O(1)* similar to the analysis in Case 1. Thus, the runtime in this case is *O(logN)*
3. Case: When split is needed: Since both leafOccupany and nodeOccupancy is constant and we just traverse the node or leaf once, the split itself cost *O(1)*, and propagating a key to the above node cost *O(logN)* in the worst case when we need to split again and again after we propagate up until root is reached. Our implementation propagates up a key to and traverse though only one node on each level, so the runtime of the worst case can be guaranteed.

Scanning analysis: (*N* = total number of elements in the Btree)

1. Case: When we start scanning, we need to find the first record to be scanned. To do that, we first need to search for the leaf node that might contains that record. This step uses the same algorithm as search a leaf node for insertion, and costs *O(logN)*. Then, we will use the linear search for the first record that satisfied in the given range. The time complexity is *O(leafOccupancy)* in the worst case since we are using linear search for our implementation.
2. Case: During the scanning, we repeatedly access the next record and examine if the next record is valid or inside the given range. The time complexity is *O(1)* since we can use index number to access the next element. In the case we need to reach the record in the next page, the time complexity is still *O(1)* since we have a pointer to the next page.

Therefore, out implementation of Btree does not have unnecessary traversals and is not inefficient.