B+Tree Design Report

btree.h

Changes made to btree.h:

- Added bool ‘isRootLeaf’ to IndexMetaInfo and BtreeIndex. It keeps track of whether the root is a leaf or not.
- Added int entries to LeafNodeInt to keep track of number of entries filled.
- Added int entries to NonLeafNodeInt to keep track of number of entries filled.

btree.cpp

Changes made to btree.cpp:

Constructor, Destructor, endScan

These methods perform exactly what is expected from the instruction and do not involve any significant design choices.

insertEntry

This method initializes a vector of PageIds called **path**. This vector is used to keep track of all parent nodes, which will be used during insertions into the tree. If the root is a leaf, we insert a value into the root using the **insertIntoLeaf** method. If it is not a leaf we insert a value into the tree using the **searchNodes** method. This method does not pin any pages and, due to the lack of a loop it has a time complexity of O(1).

insertIntoLeaf

This method is used to insert a value into a leaf node. It first checks if there is atleast one empty slot in the leaf to insert. If a slot exists, the key is inserted in sorted order into the leaf. Once a key is inserted the earlier, larger inserted values are shifted to the right. If no slot exists, the leaf must be split and the **splitLeaf** method is called. This method pins a page at the start, and unpins it before exiting/ calling splitLeaf. Inserting the key into its correct position has a time complexity of O(n).

splitLeaf

This method is used to split the leaf node into two nodes. A new sibling leaf for the existing leaf is created. Based on whether the number of leaf slots is even/odd, and the position of the key to be inserted, a threshold is determined. This threshold is determined in such a way that both nodes are guaranteed to be 50% full when the number of leaf slots is even. When the number of leaf slots is odd, the right sibling has one entry more than the left sibling. Once the threshold is determined, it is used to split the existing leaf node. All entries from the threshold's index to the end of slots are copied onto the new node. Based on the position of the key to be inserted, the **insertIntoLeaf** method is called on the appropriate leaf node. This method call is not problematic since the number of available slots in both nodes is half the original number, and it won't result in calling splitLeaf again. The smallest value from the right sibling is now sent up to the higher nodes. If the path vector is empty, a new root node is created and the value is inserted. If the vector is not empty the **insertIntoNode** method is called to insert the value, as well as a pointer to the newly created node. This method pins the leaf node, as well as the right sibling node at the start, and unpins them prior to calling insertIntoLeaf. It then pins the right sibling, reads the value to be pushed up, and unpins the page. If path is empty a root node is pinned and unpinned before the method exits. Finding the threshold has a time complexity of O(1), copying the existing keys onto the new leaf has a time complexity of O(n), and generating a new root in case the path is empty has a time complexity of O(1).

searchNodes

This method is used to find the right leaf to insert a new key value into. It starts from the root node and recursively searches its children, until a leaf node is found. The next node to be searched is found by iterating through all the existing values and choosing the left child of the first value larger than the key to be inserted. If the key is larger than all values, it chooses the rightmost child. It adds the PageId of the node being searched to the path, and calls insertIntoLeaf if the next node is a leaf. It pins the current node being searched and unpins it before recursively calling searchNodes, or calling insertIntoLeaf. Finding the next node has a time complexity of O(N). **insertIntoNode** - This method is called by insertIntoLeaf when a value is being pushed up to parent nodes. Using path, the parent node is found and it receives the value to be inserted, as well as the pageId of the newly created node. If a slot exists, the key is inserted in sorted order into the leaf. Once a key is inserted the earlier, larger inserted values are shifted to the right. If no slot exists, the node must be split and the **splitNode** method is called. This method pins the parent node page once, and unpins it before exiting/calling splitNode. Finding and inserting the key into the node takes O(N) time complexity.

splitNode

This method is used to split a non leaf node and push up the smallest value from the right sibling. A new sibling leaf for the current node is created. Based on whether the number of node slots is even/odd, and the position of the key to be inserted, a threshold is determined. This threshold is determined in such a way that the right sibling will either have an extra value, or the same number of values as the left sibling, after pushing up the smallest value from the right - without storing it in the node unlike in leaf nodes. All entries from the threshold's index to the end of slots are copied onto the new node. Based on the position of the key to be inserted, the **insertIntoNode** method is called on the appropriate node. This method call is not problematic since the number of available slots in both nodes is half the original number, and it won't result in calling splitNode again. The smallest value from the right sibling is popped and sent to the higher nodes. All remaining values are shifted to the left. If the path vector is empty, a new root node is created and the value is inserted. If the vector is not empty the **insertIntoNode** method is called to insert the value, as well as a pointer to the newly created node. This method pins the current node, as well as the right sibling node at the start, and unpins them prior to calling insertIntoNode. It then pins the right sibling, reads the value to be pushed up, and unpins the page. If the path is empty a root node is pinned and unpinned before the method exits. Finding the threshold has a time complexity of O(1), copying the existing keys onto the new node has a time complexity of O(n), popping the smallest value from the right sibling has a time complexity of O(n), and generating a new root in case the path is empty has a time complexity of O(1).

startScan

This method checks the operators and ranges and throws exceptions if they do not match the requirements. It then calls the **searchKey** method to find the leaf node that contains the key value, or the closest largest value. To ensure that there are valid keys present it searches this leaf to find a value greater than or equal to the low int value. If it does not find a valid key, it searches the right sibling node since it could contain the values just greater than low val int. If no values are found it throws a NoSuchKeyFoundException. This method pins the leaf node page and unpins it only in case no key is found. Otherwise it leaves the page pinned for **scanNext** to use. It has a time complexity of O(N) to search for a valid key.

searchKey

This method is used to find a leaf node having values close to key in order to be used by startScan. It recursively calls searchKey on the left child of the first value in the node that is greater than key. If the child is a leaf, the method returns its value through the cid variable. If the key is larger than all values the rightmost child is searched next. This method pins the current page and unpins it prior to returning. Finding the next child has a time complexity of O(n).

scanNext

This method checks if the scan is executing, and if the scan is completed and throws the respective exceptions. It then returns the current value from the pinned leaf page if it satisfies the upper bound requirement of the scan. It then advances to the next value. If the end of the leaf is reached, it pins the right sibling of the leaf if it satisfies the upper bound condition. This method unpins the leaf if it reaches the end, and pins its right sibling. It has a time complexity of O(1).

main.cpp

Added additional tests to main.cpp:

Search tests

Added method **rigorousIntTests** for performing more rigorous key search operations over various ranges that include smaller ranges, maximum possible ranges, no key found cases, boundary cases etc.

Large relations tests

Added methods **createRelationForwardStressTest**, **createRelationBackwardStressTest** and **createRelationRandomStressTest** that create relations of size 400000 to force a non-leaf node split. However, for the purpose of testing by the TAs, we have commented these tests out since they take a lot of time to create the relations.

Sparse relations tests

We use a modified **createRelationForwardSparse** that creates a sparse tree by random shuffling of a large array of size 10000 and taking the first 3000 elements. We then perform the new index tests described in **sparseindexTests** where we expect the number of matching key entries to be close to one-third of the range size.