CS 564 Project 3 B+ Tree Design Report

Built by: Zeren Li (zli868@wisc.edu), Zhaoyi Ren (zren45@wisc.edu), Matthew Silveus (msilveus@wisc.edu)

Introduction:

The purpose of this project is to create B+ tree indexing for a Database Management System. This specifically is using the background of the BadgerDB framework. This will allow for faster scans for ranged searches. The implementation is written in C++ and is compiled using the C++ 14 standards. The report will outline the different functions that were implemented and the design decisions that went into the code.

The project mainly focuses on the implementation of the B+ tree index manager which the index will store data entries in the form <key, rid> pair in a separate index file that "points to" the data file for the efficiency when run the queries.

Finally we had an error that consumed two days of debugging problems that the professor did not know how to help us with. This error took much time and caused us to not be able to debug effectively the rest of our code. If we were to reflect on this more time should have been given. The error has to do with the dereferencing of the file iterator that would cause a segmentation fault that we are unable to solve. And in this case there is the error left as the HashAlreadyPresentException which caused by the insertEntry part but we didn't have enough time to debug for this one. All of us learned a lesson from the project that for later works and projects, other than starting early, we need to do testing and debugging when we try to write the code, so that it will save our time so that we will not be trapped by some weird error just before the deadline.

B+ Tree Index Constructor and Deconstruction:

B+ Tree Index Constructor: In the constructor, the first part was using a try-catch block to check to see if the corresponding index file exists. If so, open the file, if not, it will throw the FileNotFoundException and create the index file and insert entries for every tuple in the base relation using FileScan class. If the index file already exists for the corresponding attribute, but values in metapage(relationName, attribute byte offset, attribute type etc.) do not match with values received through constructor parameters, the constructor will throw the BadIndexInfoException. Other than that,

EndOfFileException will be thrown when use the FileScan to scan the file and reaches the end of the file.

There is a helper method allocateLeafNode as the private helper function for initializing the root node as it casts the page into the LeafNode and initializes the keyArray so that all of keyArray[i] were initialized to zero.

B+ Tree Index Deconstructor: The deconstructor will not throw any exceptions, any initialized scan,flush index file, after unpinning any pinned pages, from the buffer manager will be ended here.

Insert Entry function and Related helper functions:

Insert Entry Function: This function has 4 different cases that it handles 1 Generic case and 3 edge cases. 2 of the edge cases consist of when the height of the tree is one and the root node must be cast to a Leaf Node. Thus comes into play the first design decision. We have decided as a group to track the height of the tree. This is done initially to help distinguish whether the Root Node should cast to a leaf node or to a non leaf node. Height then is utilized secondarily later on while iterating into the tree. Additionally the height is assumed to be one if the root node is empty. We assume the root node in this case is just an empty node. The two sub cases that exist in this realm are when that root node is full and when it is not full. A private checkOccupancy helper function is used to check whether the node is full. See the check Occupancy description for a more detailed explanation of usage. The node is split in the full case using the split function. Again for a more detailed explanation see the split function see the split function description. In the second case we find the index and insert the entry.

Now in the third case is the generic where the root node is assumed to be a non-leaf node and that some general tree structure is addressed. Next the algorithm descends through the tree to get to the tree level. This is where the next design choice is implemented. This could have been done via a recursive function. However after consideration this would be a memory inefficient. Additionally there is a concern of whether that design of function may cause an infinite loop in an edge case. Thus it was decided to simulate recursion using stacks. The important data points are stored in stack nodes. After the algorithm gets to the leaf node it checks whether a split is needed. If a split is needed then an entrance traveling back up the tree is initiated at each step correction. The correction then in an edge case is propagated to the root node where a new level is

created. At the end if the root node is split then a corrective measure is taken to correct the level in each non-node level. This is due to the fact that the root split creates a new level and the height is corrected. See the description of correctiveHeight function for more details.

Finally there is the edge case where the key that is inserted is a duplicate. We have one final design decision that is here. If a duplicate key is found in the find index function a duplicate key exception is thrown. The exception is caught. It is thrown and caught to avoid having to do additional error checking. It leaves and does nothing with the tree. It leaves it in its original state.

Split Function: This function is meant to be a generic function to be used for the split of a node. The current node and its id is inserted. Additionally a newId reference variable is inserted. This function encapsulates much but it has two base cases. It has the Leaf Node case and it has the Non Leaf Node case. These are handled with several helper functions. But the key is that the function returns the key that is split on and the currentId and the newId which are the ids of the new nodes. Additionally as far as design goes currentId is associated with the left child and the newId is associated with the right node. This is for when the case of a leaf node that the the leaf to the left of the original node still has a good reference and won't require any additional correction.
The key idea is that the caller of the split function then has all it needs for any additional Corrections for the B+ tree.

Check Occupancy Function: This function looks at the last 2 nodes of the key array of a node and checks whether they communicate a full node. The easiest case is that the last two keys are zeros. We assumed the key array is originally initialized to zero. We have also the additional case where the the last key is zero and the second to last is also equal to zero. In addition, the final keys are equal to zero but the last pageIDs or the last rids are not equal to zero. These outline all the cases that the occupancy is not full. Finally every inverse of the case is a full case. Finally we have one additional case where both the last two nodes are not equal to zero. This is a full case. This function returns a boolean, true if full and false if empty.

Find Index Function: This function is responsible for finding the index where to insert the key during a split. It assumes that the array that is inserted is at least 1 larger than the size that is inserted. It takes the first index at which the key is larger then the key inserted for comparison. If a duplicate key is found it throws a duplicate key

exception that is caught by the split function and then thrown again and in the split function. And then caught again in the main insert function.

Move Item Class of Functions: These functions are helper functions that move data items in arrays starting at the inserted index and then leaves open the space in the array for something to be inserted at the index by the split function.

Scan Functions:

Two helper functions:

Is_key_in_range: helper function checks if the key we are searching is in the range given. Return false if not, and return true if yes.

Find_next_nonleaf_node: helper function is a handy method which returns the pageID of the child node which should contain the value of key that we termed.

startScan: The start scan method's function is to perform a filtered scan of the index. We seek the entries with value between lowValParm, and highValParm. Be careful with the boundary. The operator indicates whether we are including the boundary or not. We start from the root page in the structure. First, compare the key values in the current page (which is the root page at this point) with the lowValParm and go down to the appropriate child node which should contain the value of lowValParm if the root has a child. Then, we call the helper function: find_next_nonleaf_node. we do this recursively until we arrive at just one level above the leaf node, so that the next iteration will make sure we arrive at the right leaf node which should have the value of lowValParm. Ultimately, we use the rightSibPageNo given in the header file to check if the index we are looking for exists in some of the sibling nodes. Throw BadScanrangeException if the search range is not valid. Throw BadOpcodesException if lowOp and highOp do not contain one of their expected values. Throw NoSuchKeyFoundException if in the process of searching, there is no corresponding key index found in the node.

scanNext: The scan next method used the same idea as we found the first index entry in the start scan method, except at this time we just simply return the rid of the next index entry that matches the scan. We go to the current node's sibling if there is no corresponding key index found. Throw the ScanNotInitializedException if the scan is not initialized, Throw IndexScanCompletedException if we reach the end of the leaf or if we arrive at a key value of the entry in the current node that's higher than what's expected.

endScan: it terminates the current scan and upin all pages whichever have been pinned during the process of scan. Then, set the page currently being scanned to null.Throws ScanNotInitializedException when called before a successful startScan call.