Project 3 B+ tree index manager Design

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.

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. The destructor, will not throw any exceptions, any initialized scan, flush index file, after unpinning any pinned pages, from the buffer manager will be ended here.

The two helper function related to the scan method are the is\_key\_in\_range and find\_next\_nonleaf\_node. The first one checks if the key we are searching is in the range given. Return false if not, and return true if yes. The second one is a handy method which returns the pageID of the child node which should contain the value of key that we termed.

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 indicated whether we are including the boundary or not. We start from the rootpage in the structure. First, compare the key values in the current page (which is 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 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 exist in some of the sibling node. Throw BadScanrangeException if the search range is not valid. Throw BadOpcodesException if lowOp and highOp do not contain one of their their expected values. Throw NoSuchKeyFoundException if in the process of searching, there is no corresponding key index found in the node.

The scan next method used the same idea as we find 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 match the scan. We go to current node’s sibling if there is no corresponding key index found. Throw the ScanNotInitializedException if the scan not initialized, Throw IndexScanCompletedException if we reached the end of leaf or if we arrive at a key value of the entry in the current node that’s higher than what’s expected.