B+ Tree Index Design Report

Project team members:

Shihan Cheng StudentID: 9079915733

Yimiao Cao StudentID: 9080624373

Jiayuan Huang StudentID: 9075663824

Implementation choice:

In this project, we construct a B+ Tree Index. In order to improve the performance when we handle the relation, we need to efficiently use the Buffer Manager. When we create the index file at the beginning, we select to initialize the root as a leaf node. In this way, the leaf page will not be fully taken by the data-entries. Therefore, we can make less I/O cost at every search. Meanwhile, saving the memory space leads a better efficiency.

When we need to insert data-entries(records), a recursive function can be simple and clear. Once the recursion reaches the correct position i.e. the leaf page, the corresponding pages will be unpinned as soon as possible. In the whole insert process, the height of the tree is also the maximum number of pages that need to be pinned. Then, we chose to traverse the tree in the scan part. Here, for improving the performance, we want to traverse the tree only at the same level instead of traversing cross different levels(such as up and down), to avoid the situation of Buffer overflowing.

Thus, we immediately unpin the pages that we passed. Once we find the page which might contain the smallest ket we are searching, we use pointer does the search from left to right. If we reach the last entry on that page, we go to its sibling-leaf page. In the end, we have to remember to unpin the pages.

Through the above implementation ideas, the B+ Tree Index will be efficient and simple when we need to pass the tests.

Functions-Implementation:

InsertEntry:

In our implementation, data records are inserted one at a time via using the <key, rid> pairs. We start from the root page and call the helper method insertion immediately after that.

Insertion:

For the first time, we assume that the root of the B + tree is initialized as a leaf node so that we can save I/O cost and disk space. It will recursively call itself until we find the parent of the leaf node for which the new entry is to be added. That is to say, the B+ tree will be traversed in this manner until it reaches the level 1 non-leaf node and then insert the entry in the leaf node. If it still has some space, we will insert the key and record id to the node and unpin it as soon as we can. If not, the leaf node will be split using the function splitLeaf and first element will be copied up and insert to the parent non-leaf node. If necessary, we will do a split using function splitNonleaf. It will be repeated done until all the node satisfied the occupancy limit.

splitLeaf:

Create a new leaf node, initialize it with the values in the right half part of original one and clear them from the current original one. The new entry is inserted into the left(original) or right(new) side of the node depend on the value of the element we add to. In the end, a new root is created and value in the middle is copied to the new root.

splitNonLeaf:

Allocate a new non-leaf node and initialize it with the values in the right half part of original one and clear them from the current original one as what we did in the function splitLeaf. Different from the SplitLeaf, the second half is copied to the new leaf node while excluding just the middle key value. The middle key is then copied up to the parent and the page is written back from the buffer manager.

insertToNonLeaf:

If the current non-leaf node has enough space, we will call this function to find the correct position to Insert (by iterating keyArray) and shift the rest part the keyArray and pageNoArray by 1 (by calling the memmove function). After finishing all the preparation work, we can save the key and record id to the leaf node sound and safe.

insertToLeaf:

If the leaf page is empty, we can set the key and record id directly without consider complicate situation. If not, we will shift the rest elements in the keyArray and ridArray by 1 and make room for the new entry to enter.

startScan:

This method first checks if all arguments are valid. Then make sure no scanning is running, end the scan otherwise. Then traverse the whole tree. Once we scanned the page, we unpin it then. In this way, we only have one page pined at the buffer-pool every time. When we reach the page which might contain the lowest key in the range that we need, the searching starts from left to right in all leaves at the same level, that is, we go to the siblings if nothing found.

Therefore, we do not need to traverse across different levels. Moreover, we unpin the pages immediately.

scanNext:

This method fetches the record ID for the next entry that matches the scan. Once we go through all entries in the current page, we move to this page's right sibling i.e. rightSibPageNo field. Then we reach the next leaf node instead of moving their upper levels of the tree. Then continue scanning all entries to satisfy the scanning conditions.

endScan:

This method stops the current scanning, reset the scanning status i.e. scanExecuting to false, then unpin the current page from the buffer-pool.

Additional Tests:

Since the relationSize is fixed, we designed following three additional test:

1: When the tree is empty and data is also empty.

2: When the tree only contains one leaf node such that the data number is less than 682.

Here we chose to use 600 as the data number.

4: When the relation has negative values.