Group 32
Nicholas LePar
Wei Li
Xiaofei Liu

Project 3: B+ Tree Index Manager

This project focused on implementing a B+ Tree index on an search key of integer type in a typical relation. The idea behind implementing an index is to improve performance when accessing elements in the relation. This is accomplished by storing <key,rid> pairs into the B+ Tree index. Efficient use of the B+ Tree and Buffer Manager are important to the structure and must be taken into consideration.

In our implementation, data records from a relation are inserted one at a time. When the index file is created for the first time, the root of the B+ tree initialized as a type of leaf node instead of a nonleaf node. The reason to make the initial root as a leaf node is to accommodate small relation files whose data entries will not fill up a leaf page. In this case, we can save one I/O cost per search and also save disk space. In the insert process, “insert” function is recursively called on itself until a correct leaf page is found and corresponding pages are unpinned immediately. The maximum number of pages pinned in the insert process is the height of the B+ tree. During scan ranges, we traverse the tree and unpin corresponding pages immediately. This leads to only one paged pinned at a time. When the first page that might contain the lowest search key is found, we search from left to right in the leaf level by utilizing the pointers to sibling leaf pages. This avoids any unnecessary upward and downward traversal in the tree. Finally we make sure again to unpin pages which we do not need as soon a possible in order to avoid buffer overflows. We believe that these design choice follow the idea of a simple yet efficient B+ Tree index, especially when we consider the requirements for testing.

As for duplicate keys, we could use overflow pages to store duplicate values. More specifically, we can turn the leaf <key,rid> node into a pointer to a page that contains a data structure, while still retaining the sibling properties as stated by our standard B+ Tree. This structure could also focus on being a more efficient design than just a typical array depending on how we differentiate between entries. For example at this Key value we could implement a hash table or a B+ tree across other attributes. This would maximize performance and scalability of our Index, especially for instances where our Index contains many duplicates.

As for the testing, addition to the three tests provided, we add more tests specific for these situation:

1: When the data volume is large so we will counter the situation that non leaf page would also split.

2: When the data is empty and we have a empty tree.

3: When there is only one leaf node for the entire tree (data number < 682).

4: When the relation contains negative values

We add these situation into the test with some combination of create relation forward and random to five extra tests with one more addition test for the error which is provided in the main.cpp file. All of these tests passed. The only issue is that when the data volume is too large, it took a while for constructing a B+ tree.