Comp Sci 564 Database Management System

Programming Project 3: B+ Tree Index Manager

April 30, 2021

Members:

1. Minseok Gang (net ID: gang3)
2. Jiwon Song(net ID: jsong99)
3. Jeijun Lee(net ID: lee783)

B+ Tree Design Report

1. **Implementation Choice**

1) Overall implementation: Our implementation is based on recursively searching the leaf node from the root node and then from the leaf node to insertion. There are three major cases to consider during the insertion process.

2) Searching Process

(1) We used a stack to keep track of the search path along the tree. More specifically, we can easily add and retrieve certain paths that we want to reach (Last In, First Out). To be specific, with the data structure, it is possible to compare nodes' levels and go to a particular node without recursively going back to the root node.

3) Insertion Process:

(1) Inserting to the leaf node

- When inserting to the leaf node, we have two cases where the leaf node is the root node itself or the leaf node at the bottom of the tree.
- When it's a root node, we just insert it by comparing the key of the entry tuple that is about to be added.
- When it's a leaf node at the bottom of the tree, we use the searching process to keep track of the path (a collection of parent nodes leading to

the leaf node). We insert it by comparing the key of the entry tuple that is about to be added.

(2) Inserting to the non-leaf node

- The situation of the leaf node determines the case for inserting it into the non-leaf node. When the leaf node is filled with the maximum number of keys, the splitting process occurs, pushing up a particular key value. Therefore, the pushed key is inserted into the non-leaf node until it is filled with the maximum number of keys.
- When the non-leaf node is root node and it needs to split, we have created a function that creates and updates the new root. The new root would be created and will have a pushed up key from its child. The metadata page will be updated accordingly for the root page number field that it contains.

4) Splitting Process: When the node is filled with the maximum number of keys, our implementation spits the node into three pieces of nodes.

(1) When we are splitting the leaf node, we have to push up the key that is located in the middle. In addition to this, we have to keep the key to the right node of the splitted nodes. When pushing up to the parent node, according to the key inserted (the specific index), we have inserted the corresponding page numbers, one for the left node and one for the right node.

(2) When we are splitting the non-leaf node, we have to push up the key that is located in the middle. In this case, we don't need to store the middle key anywhere but just have to push up to the parent node. While doing so, we kept track of the page ids that have to be modified accordingly to keep the B+ tree in order with the ascending keys.

2. Efficiency

- For efficiency, we have tried our best to avoid using nested for loop to search for the appropriate place to insert the key, its corresponding page numbers or rid records.
- For searching for the leaf node where the new entry should be inserted, the time complexity would be linear because it will check the key for key array of each node for each level. In total there will be an amount of "level of the B+ tree" * the for loop for executing the key check. Since the for loop is constant and the level of the B+ tree is linear, the overall time complexity would be linear too.

- For inserting the entry, it will be constant complexity. It will go through the for loop for the key array for the appropriate node. However, since the size of the key array is fixed, it is considered as a constant time complexity.
- For splitting the node, it will be constant as well. It pops out the middle value and push that up to the parent node that is going to be in a constant time complexity. Since the most of the algorithm used is a for loop, looping through the fixed size array, it is considered constant complexity.
- There is a worst case scenario where the node splits and the pushed up key splits the upper level until the root (even the root may split and create a new root). In this case, the recursive call to split and insert to non-leaf node may be a linear time complexity because it is going to be determined by the height of the B+ tree, which serves as a variable in our system.

3. Tests

1) test1(): Given test that creates a relation with tuples valued 0 to relationSize and perform index tests on attributes of all three types (int, double, string)

2) test2(): Given test that creates a relation with tuples valued 0 to relationSize in reverse order and perform index tests on attributes of all three types (int, double, string)

3) test3(): Given test that creates a relation with tuples valued 0 to relationSize in random order and perform index tests on attributes of all three types (int, double, string)

4. **Buffer Management Methods** to keep track of pinned pages

1) Our implementation mainly uses three functions in BufMgr: readPage, allocPage, and unPinPage. The purpose of using the function is to read, make, and clean the pages used in our implementation.
2) readPage: Our implementation uses this function to read the current page to search, insert, and split for a target key. If the current node is a root node, our implementation uses the function to read rootPageNumber and rootPage from the current file. However, if the current node is not a root node, our implementation reads the target page and target page number after finding a leaf node.
3) allocPage: Our implementation uses this function to allocate a new page. More specifically, at the very beginning, our implementation uses allocPage to make a beginning page. When we split the nodes, then our implementation uses this function to make corresponding pages. Finally, when splitting the pages and

nodes, our implementation uses this function to update the corresponding parent nodes.

4) unPinPage: Our implementation uses this function when we try to clear the modified pages or none satisfied page exists. When we split the existing pages, our implementation unpins the existing pages to clear and pin the newly created pages. In addition, if there is no satisfying page, then our implementation calls endScan and unpin before throwing an Exception.