B+ Tree Implementation Report

Steven Kan (netID: pkan2)

Shengwen Yang (netID: syang382)

Zelin Lv (netID: zlv7)

April 25th, 2020

1 Assumptions

- All valid page numbers are nonzero.
- All valid records are nonzero.
- All records in a file have the same length (for a given attribute its offset in the record is always the same).
- The B+ Tree only supports single-attribute indexing.
- Each record corresponds to a distinct key. No two records share the same key value.
- This implementation supports INTEGER attributes type.
- The page size is assumed to be 8KB.

2 Implementation Choices

- Overall implementation idea is based on recursively searching to the leaf node from the root node, and then from the leaf node to do insertion or scanning. During the insertion process, we have to specially handle the cases of:
  - inserting into the leaf node
  - splitting the leaf node due to the fact that this leaf node is totally filled up
  - pushing up and inserting a key into the upper level of non-leaf node
  - splitting the non-leaf node due to the fullness of the non-leaf node
  - creating a new root node, if the old root node is full
- During the searching process with the *searchLeafPageWithKey* function, we use a vector to keep track of the search path along the tree, i.e. the pages that we have been passing by. The data structure of *std::vector* enables us to add in and remove pages object freely, without needing the concern for initial allocation of space. The usage of such data structure benefits us to keep track of the parent non-leaf node of the node we are currently at, which can help us to find the upper-level non-leaf node we should go to in one path, instead of recursively searching from the root again. Therefore, the back up insertion process will only have **linear time complexity** $O(H \cdot NodeOccupancy) = O(H)$, where H is the height of current tree and $NodeOccupancy$ is a constant to represent the max amount of elements in each non-leaf node.
  Also, during the whole insertion process, we have to guarantee that the node we are currently at is not in the vector for searching path. In this way, when we check the size of the vector and find the size of 0, it means that we are at the **root node**. If needing to keep on inserting into upper level, we will create a new root node first and update all the private variables and the meta info page.

- During the splitting process of the leaf node, we generate the new leaf node as the one with larger key value, because there are other leaf nodes pointing toward the current leaf node already. In order to keep such pointer from other leaf nodes and the sorting order of the leaf nodes, we will remain the smaller key values in this original leaf node and copy the larger key values into the new node. Then, we set the *rightSibPageNo* pointer of the original leaf node towards this new leaf node instead. Such process will only take **constant time complexity**, which is $O(LeafOccupancy)$, where $LeafOccupancy$ is a constant to represents the max amount of pairs of key and rid in the leaf node. After copying the pairs of key and rid into the new leaf node, we will use the $insertNonLeafNode$ to push the first key in the new leaf node and the pageId of this new leaf node into original leaf node's parent non-leaf node, to maintain the structure of the B+ Tree.

- It is clear that we have to specially handle the cases when the root node is a leaf node and the root node is a non-leaf node for both insertion and scanning methods, since the structure of leaf node and the non-leaf node are totally different. Therefore, we add an extra private boolean variable *rootIsLeaf* for the *BtreeIndex* class.

- In order to keep track of the amount of valid keys and the pairs of key and pageId or the pairs of key and rid in each node, we add a private variable $slotTaken$ in the structure of leaf node and non-leaf node. With variable $slotTaken$, during the insertion process, we can see whether the node has been filled up already, whether we need to split this node as new insertion key, and where we should insert the new key into. During the scanning process, we can see whether we are reaching the end of the current leaf node and whether the *nextEntry* should be at the next nearby leaf node.

3 Buffer Management

- During each round of the recursively insertion process, we use the $BufMgr :: readPage$ function to read in the current page we are working with and allocate a new page from the buffer manager, if we need to split the current page. After inserting the new key and splitting the original page, we will unpin both the original page and also the new page we allocated, before recursively calling the start of next round.

- During starting scanning, we recursively try to find the page which contains the given lower bound. At the non-leaf node, the page is retrieved to find the page number we want to move to the next and is unpinned before moving to the next level. Once we reach page storing a leaf node, we will not unpin this page. If none satisfied page exists, we unpin the current page and call *endScan* before throwing.

- During scanning, we continue to read the pinned leaf page until we have returned the last element stored in this page or the current entry is greater than the upper bound provided. If we reach the end of this pinned leaf page, we unpin the current page and pin the next pointed page. If we reach an element with a key greater than the upper bound, then the scan is completed and we unpin the current page.

- During the *endScan* called, the currently pinned page is unpinned.

4 Efficiency Analysis:

- Searching target leaf page based on key:
  In this phase, the IO cost will just be twice the number of pages we read in and write out through the searching path from root node to the leaf node, which is exact the height of the current B+ Tree. Thus, the IO cost is $2 \cdot H$, where H is the height of the current B+ Tree.
  And the time complexity should be **linear time complexity**, which is $O(H)$, since we are traversing only one node in each level of the tree and iterating through each slots in these nodes, where the maximum amount of slots in each node is a constant number.

- Inserting a key into a node:
  In this phase, the IO cost will just be 2, since we only need to read in and write out the node page that we are working on.
  And the time complexity should be **constant**, which is $O(NodeOccupancy) = O(LeafOccupancy) =$

$O(1)$, where $NodeOccupancy$ is the max amount of slots for a non-leaf node and $LeafOccupancy$ is the max amount of slots for a leaf node, both of which are constant number. The reason for such time complexity is that we only need to iterating through the current node and find the slot that we want to insert the key in, where the max amount of slots we will iterate through is a constant number.

- Splitting a node:
  In this phase, the IO cost will be 4, since we only need to read in and write out the node page that we are working on and the new page we created.
  And the time complexity should be **constant**, which is $O(NodeOccupancy) = O(LeafOccupancy) = O(1)$, where $NodeOccupancy$ is the max amount of slots for a non-leaf node and $LeafOccupancy$ is the max amount of slots for a leaf node, both of which are constant number. The reason for such constant time complexity is that we only need to iterating through the current node and copy some of the slots into the new node we create, and the amount of slots we need to copy is constant.
- Overall recursively inserting back to parent levels:
  In this whole process, the worst case is we have to insert back all the way to the root and we need to split node in each level. Therefore, the IO cost for the overall inserting back up process in the worst case will be $6 \cdot H$, where H is the height of the tree.
  And the time complexity should be **linear** in H, i.e. $O(H)$, since it takes constant time for the insertion and splitting for the node in each level. And for the worst case, we have to such process for each level.
- Scanning process: In this process, the worst case is when we have to scan out all the leaf nodes. Therefore, the IO cost in the worst case will be twice the total amount of leaf nodes, where we have to read in and write out all the leaf nodes.
  The time complexity will be **Linear Time Complexity**, which is depending on the number of keys falling in the range of the scan.

5 Tests

- All tests are included in main.cpp, and all test cases are called by default.
- *test1()*
  - Provided test which creates key-value pairs from 0 to *relationSize* which is defaulted to be 5000 in the forward order.
  - Test if the output amount matches the input amount when scanning.
- *test2()*
  - Provided test which creates key-value pairs from 0 to *relationSize* which is defaulted to be 5000 in the backward order.
  - Test if the output amount matches the input amount when scanning.
- *test3()*
  - Provided test which creates key-value pairs from 0 to *relationSize* which is defaulted to be 5000 in the random order.
  - Test if the output amount matches the input amount when scanning.
- *test4_boundTest_Random()*
  - Added test which creates key-value pairs from 0 to *relationSize* which is defaulted to be 5000 in the random order.
  - Test if the output amount matches the input amount when scanning.
  - This test finds keys that are out of the range of input.
- *test5_boundTest_Forward()*

  - Added test which creates key-value pairs from 0 to *relationSize* which is defaulted to be 5000 in the Forward order.
  - Test if the output amount matches the input amount when scanning.
  - This test finds keys that are out of the range of input.

- *test6_boundTest_Backward()*
  - Added test which creates key-value pairs from 0 to *relationSize* which is defaulted to be 5000 in the Backward order.
  - Test if the output amount matches the input amount when scanning.
  - This test finds keys that are out of the range of input.

- *test7_int_CreateMoreRelation_Forward()*
  - Added test which creates key-value pairs from 0 to a larger *relationSize* in the Forward order.
  - The *relationSize* is set to be 200000 to test splitting on non-leaf node.
  - Test if the output amount matches the input amount when scanning.

- *test8_int_CreateMoreRelation_Backward()*
  - Added test which creates key-value pairs from 0 to a larger *relationSize* in the Backward order.
  - The *relationSize* is set to be 200000 to test splitting on non-leaf node.
  - Test if the output amount matches the input amount when scanning.

- *test9_int_CreateMoreRelation_Random()*
  - Added test which creates key-value pairs from 0 to a larger *relationSize* in the Random order.
  - The *relationSize* is set to be 200000 to test splitting on non-leaf node.
  - Test if the output amount matches the input amount when scanning.

- *errorTests()*
  - Test whether *ScanNotInitializedException* will be thrown if *endScan* is called before *starScan*.
  - Test whether *ScanNotInitializedException* will be thrown if *scanNext* is called before *starScan*.
  - Test whether *BadOpcodesException* will be thrown if scan with bad *lowOp*.
  - Test whether *BadOpcodesException* will be thrown if scan with bad *highOp*.
  - Test whether *BadOpcodesException* will be thrown if scan with bad range.