A3 Design Report

Jirayu Burapacheep, Yuan Ling Hew, Sui Jiet Tay
March 7, 2022

Design choices

Overview

- Insert
  - We use recursion to implement the insertion. If the node is being split, we create a new node and pass a new key to add to the parent.
- Scan
  - When start scanning, we find the first leaf to start with and the corresponding starting index. For each step, we increment the entry index and/or move it to the next sibling node if needed.

Pin and unpin

- Insert
  - Each node on the path to the correct leaf is pinned as we traverse through and unpinned when we are done with the insertion in their subtree. When the split both on leaf or non-leaf nodes happen, a new page (node) is being created, pinned, modified their entries and unpinned right after.
- Scan
  - Pin and unpin operations are used during traversal to the leaf node. The *lowerBound()* function is used to find the index of pageId in pageNoArray to ride to the next level. As the new pageId index in the new level is determined and read, the pageId that was previously read and pinned in the previous level will be unpinned immediately.
  - Pin and unpin operations are also used during the scanning process. The page is read and pinned once when the leaf node is first encountered after the traversal, or when it rides the sibling pointer to the sibling node. The page is unpinned immediately once a new leaf node is encountered and read.

Insert algorithm

- Normal insertion
  - During a normal insertion, where the size of the node being inserted is of order < 2d, the BTree is traversed to the leaf node, where the keys in the node are linearly scanned and inserted into the correct slot in the page accordingly. Since the number of keys in a node is constant, insertion will account for running for traversal which is $O(\log_f n)$, where n is the number of leaf pages, and f is the fan out of the BTree.

- Split
    - During the case of a split, the node that is bound for an insertion will have a size that matches its capacity, 2d. A split creates a new array, and moves the right half, d elements to the new array. The inserted element is compared with the middle value where the split occurred. If the capacity of a node, c is odd, the middle element of index c/2 is chosen to determine if the element is inserted to the original or the new node. If the capacity of a node, c is even, the element of index c-1 and c are compared to determine if the element is inserted to the original or the new node. An insertion which causes a split may occur in both the leaf or the non-leaf node. This issue is addressed using two different implementations - *splitLeafNode()* and *splitNonLeafNode()*. If the split occurs in the leaf node, the inserted element will be copied and inserted to the appropriate position in the parent node. If the split occurs in the non-leaf node, the inserted element will be moved to the appropriate position in the parent node and removed from the new node.
    - The time complexity of split is O(nodeOccupancy) because we are moving half of the keys into the new node.
- The total runtime complexity of one insertion is O(Height of the tree * nodeOccupancy) = O(Height of the tree) = O(log n) where n is the number of leaf nodes.

Scan algorithm

- When startScan is called, member variables for scanning are set before a helper function is used to traverse the BTree to the leaf node, guided by the keys that are smaller or equal to the searched key. When the leaf node is reached, lowerBound() function is called to determine the index of the search key that falls in the bound of the search range. This takes a total of $O(\log_f n)$ to traverse to the leaf nodes, where n is the number of leaf pages, and f is the fan out of the BTree.
- Once the leaf node is reached, the scan is commenced by incrementing the entry index. If the last element of the leaf node is reached, determined by its size, it rides the singly-linked list pointer to the sibling node, where the process is repeated until the valid range is reached. The function then returns. This iteration takes a total of O(n) if the smallest key in the BTree is used as the starting element of the scan.
- This means the worst case time complexity of a scan given an adversarial case is O(n+logn) = O(n)
- The design choices made while completing this group project closely resembles a traditional B+ Tree index structure. While scanning through the B+ Tree, the nodes which resemble pages are pinned one at a time to prevent buffer overflows. This is done by recursively unpinning the previously visited page while pinning the current page. Upon reaching a leaf node, the page is scanned similarly to an array till it reaches the end where a pointer is followed to scan the sibling page. This is implemented to prevent the unnecessary process of going up the B+ tree node and going back down to visit the sibling node which could negatively affect the time complexity.

Test cases

Included Test Cases

test1()
- Tests the correctness of the B+ Tree implementation when inserting tuples in a relation in a forward order by calling the createRelationForward() method

test2()
- Tests the correctness of the B+ Tree implementation when inserting tuples in a relation in a backward order by calling the createRelationBackward() method

test3()
- Tests the correctness of the B+ Tree implementation when inserting tuples in a relation in random order by calling the createRelationRandom() method

errorTests()
- Tests the correctness of the B+ Tree implementation in terms of throwing the relevant exceptions: ScanNotInitializedException(), BadOpcodesException(), BadScanrangeException()

Additional Test Cases

test4_CreateBigRelationForward()
- Inserts 470,000 tuples into a relation in a forward order
- Tests the correctness of the B+ Tree when the non-leaf node is forced to be split

test5_CreateBigRelationBackward()
- Inserts 470,000 tuples into a relation in a backward order
- Tests the correctness of the B+ Tree when the non-leaf node is forced to be split

test6_CreateBigRelationRandom()
- Inserts 470,000 tuples into a relation in a random order
- Tests the correctness of the B+ Tree when the non-leaf node is forced to be split

test7_BoundTest_Forward()
- Inserts 5,000 tuples into a relation in a forward order
- Test the correctness of the B+ Tree when scanning out of range keys

test8_BoundTest_Backward()
- Inserts 5,000 tuples into a relation in a backward order
- Test the correctness of the B+ Tree when scanning out of range keys

test9_BoundTest_Random()

- Inserts 5,000 tuples into a relation in a random order
- Test the correctness of the B+ Tree when scanning out of range keys

test10_3000_Sparse()

- Inserts 0, 1, 4, 9, 16, 25, …, 3000^2 into a relation in increasing order
- Test the correctness of the B+ Tree when the keys are sparse instead of consecutive numbers

test11_ReopenIndex()

- Inserts 5,000 tuples into a relation in a random order
- Reopens and tests the B+ Tree multiple times
- Test the correctness of the B+ Tree when the same index has been reopened multiple times