CS564 Programming Project 3: B+ tree Design Report

Group 8 : Chien-Ming Chen (id:9081715170)
Hao-Lun Hsu (id:9081439458)
Reng-Hung Shih (id:9075297623)

1.B+ tree Indexing

Insert Entry

Call searchEntry to find the leaf node the entry should be inserted to and this leaf node would be unpinned at the end of insertion. After finding the leaf node, the next thing is to insert the entry into the node. If the node is not full (i.e. the keyArray is not full. We create a member "**length**" to record this.), just insert the entry and the insertion completes. If not, we call the function "splitLeafNode" to split the leaf node. After splitting the leaf node, insert the key returned by function splitLeafNode into the parent node. If the parent's node is full, call the function "splitNonLeafNode" to split the node. We repeat this procedure recursively until a non-leaf node need not to be splitted or the root node is splitted. After splitting the root node, we create a new root node to be the parent node. Then, open the meta data page and update the root node's pageId.

The time complexity of this function is O(*h*), where *h* is the height of the current tree. This is because the search step contributes to *h* I/Os, the insert and split step result in at most another *h* I/Os (some of the page might still be in the buffer so typically the number of I/Os is less than *h*).

Search Entry

We create a function searchEntry to traverse the tree and find the target leaf node. Some implementation details are described as follows.

1. In function searchEntry, we use "level" of struct NonLeafNodeInt to decide if the next node is a leaf node.

2. During the traversal, we unpin a page whenever we get the page's child pageId.

3. We do not unpin the target leaf page because insertEntry would insert a key to the node.

4. The root node might be a leaf node. To know if the root node is a leaf node, we add a new data member "**numNonLeafNode**" to record the total number of

non-leaf nodes in this tree. Thus, if numNonLeafNode is equal to zero, the root node is a leaf node.

Split leaf node

Here we use terms “left node” and “right node” to represent the two nodes after splitting.

1. Create a new page for the **right** node and use the original leaf node as the left node. This is because the right node is the left node’s right sibling node and the right node’s sibling node is the original leaf node’s. If we create a new page for the left node, we need to find the original leaf node’s left sibling node and update its right sibling node, which is much more complex to implement.

2. After splitting, we return the right node’s first key and insert the key to its parent node.

3. Finally, unpin the two nodes.

Split non-leaf node

This function does the same things as what function splitLeafNode does.

2.Scan

---

scanStart

To locate the entry based on the input range, **searchEntry** will first be called to traverse the tree to the leaf node which contains the expected key. During searchEntry, each traversed node page is pinned to check the next node and is unpinned once traversing to the next node. After the leaf node is located, the **binary search function lower_bound** (complexity O(logN)) will be used to locate the specific entry corresponding to the key. During the Scan, the node page keeps being pinned so that scanNext can get rids efficiently and the page is unpinned by endScan.

scanNext

With the node and entry located by scanStart, scanNext will continue to pick up the RecordID on the current node. If scanNext is called continuously to read rids and reaches the end of the key array, the current page will be unpinned and the next node page will be read and pinned for the further scan.

3.Tests

To test B+ Tree Index
test_tree

- test case 1 :
  1.createRelationForward and build B+ tree index
  2.Check if the results of both pre order post order traversals are correct

- test case 2 :
  1.createRelationBackward and build B+ tree index
  2.Check if the results of both pre order post order traversals are correct

For pre order post order traversals, the time complexity is O(*n*), where *n* is the number of nodes in this tree.

To test error conditions
errorTests

- test case 1 : endScan before startScan
- test case 2 : scanNext before startScan
- test case 3 : Scan with bad lowOp
- test case 4 : Scan with bad highOp
- test case 5: Scan with bad range
- test case 6 : starScan twice, then endScan twice
- test case 7 : scanNext after endScan

Basic tests
test1,tes2,test3

To test large scalability
test4: relationSize 50000, createRelationForward
test5: relationSize 50000, createRelationBackward
test6: relationSize 50000, createRelationRandom

- 12 scan ranges are tested
  1. (-300,-200) should return “No Key Found Exception”
  2. (-1,0) should return “No Key Found Exception”

3. (-1,0] should return 1 result
4. [0,1) should return 1 result
5. (49700,50100) should return 299 results
6. (-10000,1000) should 1000 results
7. (30000,30087] should 87 results
8. [0,3000] should have 3001 results
9. [0,relationSize] should have relationSize results
10. [relationSize-2,relationSize-1] should have 2 results
11. [relationSize-1,relationSize] should have 1 result
12. [relationSize,relationSize+1] should return “No Key Found Exception”

To test irregular scan behavior

test7:

- test case 1 :

  startScan twice, and then scanNext. scanNest should return the rid in the range set by the first scanNext.

- test case 2 :

  startScan twice, and the second startScan is with invalid arguments. The second startScan should end without exception and the original startScan should keep working normally.