CS 564 Programming Project #3

Do-Men Su, Ping-Han Chuang, Hung-Ting Lee

Implementation Choices

Format modification
For the tree, we add a member variable named "**order**" to refer to "d". A node requires splitting when it's number of entries exceeds 2d.

For the node, we add a member variable named "**numValidKeys**" for the node to record how many keys the node owns. This variable would be used when the node requires splitting.

For the scan, we keep a member variable named "**scanExecuting**" to check if scan is ongoing. This variable will be updated during the scan.

Constructor
To initialize a BTreeIndex constructor, we first assign value to its member and create an empty file. If the given name maps to an existing file we just open it, otherwise we start creating the index file by newing a BlobFile, a file that has only name but nothing else.

Then, we create the header page and write metainfo into it. This is completed by a separate function writeMetaInfo(). An object of IndexMetaInfo is created and metainfo is assigned to this object.

Finally, we create a root node, initialized as a LeafNodeInt. The root node is an empty page at this point. When the tree grows and the root has children, it'll be updated as a NonLeafNodeInt.

traversing
Traversing from root to target leaf node is used in insert and scan. To make the code reusable, we define a function named "**searchLaef**" to address the traversing and return the target page number. Inside the searchLaef function, currPageNum will be updated in a loop and the loop will not stop until the key in the page meets the searching criteria. We unpin the page which is not required every round of the loop.
We also keep a vector named "**parents**" to store currPageNum at each level for addressing the changes propagating back. We could recover the parent node easily by retrieving the stored page number from the parents vector and fetching the page number into a buffer frame. Since the page number is stored according to the level in the parents vector, if we need to get upper level nodes, we just fetch the subsequent page number in the parents vector. This approach waives the inefficiency to traverse from down to top.

insertEntry
To insert an entry into a node, we first need to locate it. A node can only be inserted if it is a LeafNodeInt, so we locate the node by traversing from the root in a loop until the current

node resides at the lowest level of the tree. Since the read api of buffer manager requires page number, we keep a variable named "**currPageNum**" initialized as rootPageNum and update it when traversing new pages. We call searchLaef to do the traversing. By the end of traversing, currPageNum will reflect the page number of the target node.

After we locate the node, we insert entries (key-rid pairs) into it and check if splitting is needed. By comparing "numValidKeys" with "order", we can judge if splitting is needed.
If the node needs not to split, we only need to unpin the page and insert is completed.
Otherwise, we split the node. Splitting first occurs at leaf level but there will be further work in non-leaf level as the middle entry needs to be copied up. We address things in leaf level and non-leaf level in sequence and discuss them separately.

In leaf level, the first step is creating a new LeafNodeInt as the right sibling node of current node to accommodate splitted entries. We allocate a new, empty page as this new node, assign [d~2d+1] entries from the current node to the new node and update the numValidKeys of both nodes. Also, we record a variable named "**rightSibPageNo**" to track page number of the right sibling so the linked list in the leaf level can include the new lode. Finally, we copy the middle entry (new node's first entry) to node's parent. We assign the value of this entry's key to a variable named "**splitKey**" for using it later. We unpin page number not required and work at leaf level is completed.

In non-leaf level, since the changes may propagate to the root node, we keep a variable named "**needNewRoot**", check if further propagation is necessary and update this variable in each level. We recover the parent node by retrieving the page number in the parents vector. Then, we traverse the entry array, compare the value of key to locate where the middle entry should be inserted, make the insertion and increment the numValidKeys. The final step is to address the check-if-null problem like what we did in the leaf level. Here we push up the middle entry rather than copy the entry up. The propagation will not stop until no splitting is needed or root node has been reached.

If the last parent page in the "parents" vactor has been fetched into the buffer, we create a new root node. The key of the new root node equals the splitKey pushed up from the child. We then assign other necessary info for the root node and write the metainfo. The insert accomplishes when creating the root is done.

<u>startScan</u>
This method is used to traverse from root, find the page that meets the criteria passed in and locate the target entry. We first check if the passed in parameter is legal and throw an exception if not. Then, we call the searchLaef function to find the page. Inside the page, we use a linear approach to locate the entry that conforms to the criteria.

<u>scanNext</u>
This method is usually called in a loop to continue scanning the next rid. outRid reflects the target RecordId when the method ends.

<u>endScan</u>
This method is used to terminate the current scan. We set scanExecuting to false and unpin currentPageNum.

Policy of Unpin

Pages read into the buffer frame are unpinned once not required. By doing so, the buffer manager has enough space to read new pages and the performance could be improved.

destructor

We call endScan(), flush the file in buffer frame and delete the file.

Testing

In the self defined test, we make the program run in a loop and throw some random value for scan to make sure the program will not crash if the page number reaches the boundary of pages.