CS 564
Group 55

Project 3 Design Decisions

Distinguishing between leaf and non-leaf nodes

We use the *level* field in the NonLeafNodeInt and LeafNodeInt structs to distinguish leaf node and non leaf node. Leaf node has level -1 and non leaf node has level = 1 if its children are leaf nodes and otherwise 0.

Splitting nodes

When leaf node is full, we split it by half and copy the smallest key of the right leaf node up to its parent node. When non leaf node is full, we also split it by half but push the smallest key of right non leaf node up to its parent. We implemented two methods for split: one for non leaf nodes and one for leaf nodes. This helped us modularize our code for easier testing and debugging.

Inserting

To insert, we used an extra helper method "insertHelper". The insertHelper method is called recursively to implement the insertion. If the node corresponding to the current pageNo is non leaf node, we'll find the correct subtree to insert into by comparing the input key with keyArray. After finding the subtree, which is also a node, we'll recursively call the insertHelper method to find its child until leaf node is reached. In leaf node, call insertIntoLeaf, which will insert into the leaf node. If the leaf node is full, then split it by half. Shift the right half into a new node and return the page number of new Node. The new node page No. is passed through newChildPageNo, which is initialized to zero. If newChildPageNo = 0, it means there's no split. If newChildPageNo contains a nonzero number, it indicates that split happened and newChildPageNo is the page number of new node after split. We keep the newChildPage number by returning it from last recursive call and assign it to the current newChildPageNo. The split can propogate up when trying to insert into parent. This can be accomplished by our recursion since we're checking whether newChildPageNo not equal to zero when in the non leaf node case. Putting the recursion operation into a new method allows us to add extra parameters and operate with return values.

Pinning and unpinning pages

We use pageNo as input in insertHelper, which means we have to pin the page for every node we visit when we're searching for the leaf node to insert into. The pages are unpinned after after insert. the dirty bit = 0 for visit and dirty bit = 1 for insert. When split happens, we choose to return pageNo of the new node instead of pointer to node or node itself. This means we need to pin the new node every time after split in order to add the smallest key in the node into its parent to preserve Btree structure. The page is unpinned after insert key into parent. The dirty bit for both original node and new node are 1. In the startScan and scanNext methods, a page is unpinned soon after it is read. Once the page is read, we set currentPageNum and currentPageData to the values obtained from the page. This helps us keep track of the current node we're at when traversing. Once that information is read, we no longer need the page and

thus unpin it. We kept track of pages read and unpin them on the go – this was a lot easier for us then unpinning all pages at the end. We were resourceful in that once a page is read and we get what we want from it, we unpin it since it's no longer needed. Additionally, this frees up unused frames.

Range search

The startScan method begins by checking for a valid range and opcodes. Then we check whether the root page is a leaf or not. If it is, that is the node we will search, and it is the only node in the tree. If the root page is not a leaf, we begin searching from the root. We keep track of the current page we are on by updating this->currentPageNum every node we visit. It is set to the root page number at the beginning of the traversal. Then, we run a while loop as long as the current page we are on is not a leaf node. Once we hit a leaf node, we know that we're at the bottom of the tree, and we break out of the while loop. Within the while loop, we run a for loop through every key value in the current page we are on. In this loop, we look for the appropriate slot, find the appropriate child node, and update this->currentPageNum to the page number of that child node. The while loop then starts searching in the child node. Once we exit the while loop, we will have found a leaf node to begin traversing. The data is stored in this->currentPageData. We use this->nextEntry to indicate the slot in the current page (this->currentPageData) that we will evaluate next to see if it satisfies our range criteria. It initialized to 0 to search at the beginning of the leaf node.

The scanNext method returns the next record that matches the range criteria through outRid. We will first grab the page that is saved in this->currentPageData, which is the leaf node to begin searching. We implemented a while loop to continuously traversing through the records stored in the leaf nodes to find a record that matches the range criteria. We broke the traversal into different cases, depending on the op codes. We then incremented this->nextEntry if there are still records left in the node. Otherwise, we move to the right sibling page and set this->nextEntry to the first record of the right sibling.

This design allows us to perform range search efficiently. We do not unnecessarily traverse up and down the tree multiple times. Rather, we traverse down the tree just one time, finding the leaf node that contains, potentially, the first key that satisfies the range conditions. We then move to the right each time, and between nodes by using rightSiblingNo, instead of searching the whole tree again.

Testing for large relations

We implemented three tests in main.cpp to ensure that our implementation works for larger values. We added a relation size parameter in the create relation methods to allow for differing relation sizes, instead of just 5000.

IndexMetaInfo

We create a reference of IndexMetaInfo called metaInfo. It keeps track of the root page no.

Additional helper methods

We implemented numerous additional helper methods to make our debugging easier, and to modularize our code for better reusability. The nonLeafNodeRecNo and leafRecNo methods in BTreeIndex calculates and returns the number of records in the non leaf and leaf node, respectively. This allows us to avoid scanning the entire record array in the nodes, in the case that the record array is sparse. Looping through potentially large number of invalid record numbers is wasteful. The isLeaf method in BTreeIndex returns a Boolean value depending on whether the current node is a leaf. We do this to make our code clearer, others reading the code may not know what our level variable represents. The isLeafFull and isNonLeafFull methods allow us to quickly determine whether nodes are full. This helps with splitting full nodes and additional debugging that we performed to test the implementation of insert.