CS 564 2017 Fall
Stage 4
Group 6
Jack Chen, Conard Chen, Yudong Huang

Design report

Struct:

We think the original struct for non-leaf nodes and leaf nodes are very insufficient, therefore we add more variables for each struct.

For non-leaf nodes, we add "size", indicating how many current keys are in the node; "pid", indicating current node's page id. "size" allows us to use binary search for keys (see kb_search() below, kb_search stands for key-binary_search).

For leaf nodes, we also add "size", and a bool "sorted". The idea for the bool value is that, leaf nodes do not need to be sorted when they are added while creating a new index file, it is sufficient to sort them before they need to be splitted or be all records are inserted. Leaf nodes will be used in two conditions: (1) when splitting, the smallest key of second node (the node that is splitted out) need to be returned to previous recursion, to find the node, we need to sort current leaf node; (2) when scanning, leaf nodes needs to be sorted to allow fast range search. Therefore, we only sort leaf nodes before splitting and before scanning (or after all insertions are completed). We wrote our own quick sort to sort keyArray and ridArray in parallel. Since sorting a sorted list is not necessary given that insertion might not happen to a leaf node after splitting, we use the bool "sorted" to indicate whether this page is recently sorted or not.

The idea of sorting the leaf nodes lazily (done by function sortAllLeaf()) can improve complexity of creating new index files. If we insert an entry and maintain order of leaf node by shifting other entries, the average overall complexity will be O(nd), where n is total number of entries, and d is the entry capacity of a leaf node. *However, if we sort leaf nodes lazily, every leaf node will be sorted at most twice (once before splitting, once after completing insertion, and new node is responsible for the sorting happens before splitting), so the average overall complexity will be O(n/d * dlog(d)) = O(nlog(d)).*

The trade off is, the capacity each leaf node will decrease due to the additional variable "size" and "sorted", but just a few.

sortAllLeaf():

Reaches the first leaf, then use right sibling to go through all other leaf. Sort the leaf only when sorted is true.

Constructor:

Two additional methods are designed for constructor: createIndexFile() and openIndexFile();
createIndexFile():

Scan throw the entire given data file linearly and insert into the btree;

Besides creating metapage and rootpage for btree file, we also create two empty sibling leaf pages to avoid complications while implementing insertEntry(); two empty leaf nodes ' page id is stored in first and second entry of root page's pageNoArray.

Use a class member, bool createExecuting, as an indicator for insertEntry. During the creation of a new index file, leaf nodes will be sorted lazily. But during normal insertion (if there is any), new entries will be inserted while order is also maintained (by shifting other

entries). We choose to do that because scanning may happen after any insertion, the order needs to be maintained to allow basic scanning functionality.

openIndexFile():

Just open existing file and assign class members.

insertEntry():

We follow pseudo codes on the cow book, and write a overload helper method, which is responsible for recursively inserting and splitting nodes.

All pages read or allocated by current recursion will be unpinned before exiting current stack frame. Keeping any page pinned has insignificant improvement, because recently read pages are not going to be swapped out (due to the clock algorithm in buffer.cpp), only two or three pages are pinned each insertion, and hash table look up has little overhead. However, keeping some page pinned over complicates the design. Therefore, we choose to unpin any page when finished using it.

We also agree that if a non-leaf node's level is zero, then next node must be a leaf node.

One special case, if the non-leaf node has size == 0, which can only happen in the beginning, the first key will be stored at index 0 of root page's keyArray.

We created many overload utility functions to simplify insertEntry(), they are the followings:

kb_search():

Do a binary search on node's keyArray, return the PageId that is Ki <= key < Ki+1 for non-leaf page; binary search has a better performance than linear search, that is the reason why we chose it. We have to implement our version of binary search, because standard library only provide searching for a specific element in a given array, rather then searching a key within some range and return values in another array. The overload kb_search() for leaf nodes will be explained in startScan();

Add_key():

Add a key pageId pair into non-leaf node while maintaining its order. Because non-leaf node is used during every insertion, and every insertion needs to search for the correct page id, so it has to be sorted. Otherwise, kb_search() cannot function. *This may result very bad performance during splitting. We considered using linked list as an alternative, which has good insertion complexity. However, splitting happens occasionally, the most often case is using binary search to find the page Id. Therefore, we chose array, which allows binary search, and thus has better performance than linked list.* The overload function just add a key recordId pair to the end of leaf node, which will be sorted when needed as explained previously.

Insert_key():

Insert a key recordId pair into leaf node. This is used only when insertion does not happen during creating a new index file (e.g. when createExecuting is false)

newPage():

A wrapper function for allocating a new page, avoid typing bufMgr and this->file over and over.....and over.....

qsort():

Quick sort, which sorts leaf node's two arrays in parallel. We write this function because std::sort does not support sorting parallel arrays (or we don't know how).

split():

Split non-leaf node and leaf node into two new nodes. After splitting, size of both nodes will be reset; and for leaf nodes, new node will be connected to the old node.

is_full():

Indicate whether the node is full. Full is defined as one smaller than INTARRAYNONLEAFSIZE or one smaller than INTARRAYLEAFSIZE, so that when is_full returns true, we can still insert one more entry to the node, which allows sorting and splitting the node easier (done in one page).

startScan():

Place a starting probe to leaf node.

If the operator is GT or LT, the given low value will +1, given high value will -1.

Use existKeys() to check whether there are entries between low value and high value.

Use scanSearch() to find the location of low value and high value in leaf node.

existKeys():

If low and high value are in the same leaf page, use kb_search() to obtain their index on the keyArray. They have the same index, then key does not exist.

This kb_search() for leaf node will still find the index where Ki <= key < Ki+1.

If low and high value are in different leaf page, use right siblings of low value's leaf node to find high value's leaf node. If cannot be found (reach the end), then key does not exist.

scanSearch():

Use kb_search() to dive into the index tree from root to level 0 non-leaf node.

scanNext():

Return current RecordId, and advance nextEntry by one. Switches leaf page when reaching the end of current leaf page. Use right siblings to swtich page, no touching any non-leaf pages. Only pins currently reading page.

endScan():

Write root page to file (protective approach, just in case), flush the file, and delete the file.

If there are duplicates in keys, our implementation for kb_search() need to changed, in order to tolerate duplicate keys. Our implementation for qsort() also need to be changed, which not only sort entire nodes, but also sort groups that have same key value within nodes. Our implementation for leaf node structure need to change; depending on how many duplicate keys there are. If duplicates are not many, we should consider adding pointers to each entry, which points to next entry that has same key value. Of course, if duplicates do not need any order, there is no need to change qsort() or the leaf node struct.