BadgerDB Design Report

Authors:

- Zihan Zhao (zzhao383@wisc.edu)
- Haishuo Chen (hchen727@wisc.edu)
- Qinhang Wu (qinhang.wu@wisc.edu)

Implementation

Constructor `BTreeIndex::BTreeIndex`

Generally, the constructor initializes some private data members of the bTree class and constructs headerPage and rootPage separately for either existed index file or brand new relation file. For existed index file, the buffer manager will first read the headerPage from file, and then unpin it after retrieving the information from it. For new relation file, the buffer manager will allocate new pages for the headerPage and rootPage separately and unpin them right after the corresponding information have been inserted.

inserter `BtreeIndex::insertEntry`

Logic: The insertion can be split into two phases. Firstly, find the bucket of corresponding leaf node to insert an entry. Then, insert the entry and update the leaf node and possibly its parent. The process of search, insertion and update is implemented by recursively call the helper method: insertUnderNode.

The function of insertUnderNode is to search in the current node for where the entry should be inserted (in leaf node) or which child should the search traverse down for further search. For internal node, the method will call another insertUnderNode to search on appropriate child node.

If the leaf node should be split after insertion, the insertUnderNode will split the node half-half, and return a variable of PageKeyPair type to tell the program that its parent node has a new child and the return value is the new child's key and page number. If the internal node should be split as well, we do the same to its parent node.
The boundary condition of splitting is root node split, which is implemented in the body of insertEntry. For example, suppose the current root node is denoted as n1, and insertUnderNode return a node called n2, which should be n1's sibling, we will allocate a

new root node (denoted as n3) as n1 and n2's parent. Then we will update the metadata, rootPageNum and all other related attributes.

Some design details:

1. We add a field called height in metadata to record the height of the tree. We also use it to tell whether the root node could be a leaf node.
2. When we split the node, if the number of entries is odd, we will give the left sibling one more entry than its right sibling.
3. We set an entry to be boundary in the nodes at the end of its array attributes to help us clarify how many entries does a node have. For internal nodes, it is {key = INT_MAX, pageNo = Page::INVALID_NUMBER} at the end of KeyArray and PageNoarray. For leaf nodes, it is {key = INT_MAX, rid.PageNo = Page::INVALID_NUMBER}.
4. Each time we read or allocate a page (node), we unpin them when returning back to its parent node.
5. Another new helper function called copyArray is defined to help insertion and split. Analysis: As we set an entry to be boundary in every node, the maximum number of entries a page store is one less than its actual size. But for now, it simplifies the program logic. In general, insertEntry requires an average of logarithmic time complexity.

Scanner

The scanning of this project is a range search, so the process can be split into two phases. The first phase is used to locate the minimum valid value, and then the second phase is to linearly read values onwards until reaching the higher boundary. After all these, the memory needs to be cleaned up. The said two phases and the final cleanup just match the given functions: `startScan`, `scanNext`, and `endScan`.

`BTreeIndex::startScan`

This is the most complex part of the entire scan process, which involves a lot of branching and searching. Importantly, given the design of insertion, the root node can be a leaf node when the height of the tree is 1, and otherwise, it is always a non-leaf node. Because of the variability of the type of the root node, I first look up the meta page of the index file to nail down the type of the root node, where a page is read into buffer. As soon as I get the height of the tree to determine whether the root node is a leaf node, I unpin the meta page. Next, I read the root page. If the root page is a leaf node, then I skip all the tree searching and start linear searching directly inside the root node from the beginning to find the first valid record within the given range. However, if not a leaf node, then I traverse down the tree until the first leaf node that **may** contain the desired minimum key is found. Importantly, by our design, there would always be a leaf that **may** contain the desired minimum key, and

any leaf node's parent would have its level attribute be 1, i.e., all other non-leaf nodes have their level attributes be 0. Since every access to a node requires an access to a page in buffer, and it is no longer in use in the next loop as the program has traversed to its child node, I always unpin the currently reading page before loading the page of its child. Having found the leaf node that may contain the desired minimum key, I'll do the same linear searching as described above. If no such key is found in the leaf node, then throw the corresponding exception. Otherwise, set up scan parameters properly and then finish the function. In general, to find the minimum key requires an average of logarithmic time complexity, and pages are unpinned as soon as they are not used.

`BTreeIndex::scanNext`

This is very simple as it doesn't involve many page accesses. Because we keep track of the next entry to scan, I only need to update the page to read when all of the records in the current page have been fetched; otherwise, I simply take out the record at the correct entry of the current page. This is done in constant time. I also unpin the page when needed as soon as possible.

`BTreeIndex::endScan`

This is also very simple. This function resets all the necessary scan parameters and unpins the currently reading page.

Additional Tester

We implemented additional tester functions in `main.cpp` to test our design extensively. `test5()` searches for key from -1000 to 6000 as a boundary check of the scanning functionality. `test6()` reopens an index file after it was saved to test the construction of a b+ tree based on an existed one. `test7()` constructs a b+ tree based on 50 thousand records to force the non-leaf node to split. Our program passes all the given tests as well as the self-implemented ones. We also conduct a local test to make sure that the b+ tree splits its nodes properly.