Jake Dalton, Jacob Panackal, Rohan Mendiratta, Jaskaran Bakshi

CS 564 PP3: BTree Index

April 30, 2021

**Group Members:**

* Jake Dalton
  + CS login: jdalton
  + Net ID: 9077351758
* Jacob Panackal
  + CS login: panackal
  + Net ID: 9078227452
* Rohan Mendiratta
  + CS login: rmendiratta
  + Net ID:907 987 4104
* Jaskaran Bakshi
  + CS login: jsbakshi
  + Net ID: 9078312387

**How to run:**

* make && ./src/badgerdb\_main

**Assumptions**

* All valid page numbers are nonzero.
* All valid records are nonzero.
* duplicate keys can be inserted.
* All records in a file have the same length.
* B+ Tree only supports single-attribute indexing
* The indexed attributes are all integer data type
* The page size is assumed to be 8192

**Implementation Choices**

* Created rootIsLeaf private variable to keep track of if root is a leaf node = 1 if root is leaf, = 0 otherwise
* Set rightSiblingPageNo to 0 if there was no right sibling
* Unpinned any page that wasn't the current page of the search
* Used NonLeafNode int variable level to check if a non-leaf node was the parent of a leaf level = 1 if node was parent of leaf, 0 otherwise
* The way we check whether or not the index file exists is by creating a BlobFile in a try catch block. If we catch a FileExistsException, we will write the code corresponding to a file existing in the catch block. Then the code for a file not existing is in the try block.
* We end any uninitialized scan by calling endScan in the destructor
* In scanNext, we create boolean variables to keep track of what the high and low operator range values are, and use them to check that the scan criteria is satisfied. We can tell that we've reached the final leaf node if its rightSibPageNo variable is equal to 0.
* In scanNext, we traverse to the right sibling of the current leaf by setting the currentPageNum to the rightSibPageNo of the current leaf, then we use readPage to read in the right sibling leaf of the current leaf into the currentPageData variable.
* this method is completed.
* In btree.h, we added a variable to NonLeafNodeInt and LeafNodeInt structs that somehow protects the variables underneath it from being filled with a suspiciously mysterious number "535822336".
* The variable, int bodyguard, is then filled with that number for some reason.
* For the insertion method we decided to make the implementation so that we have a cursor that runs along the B+ tree until we find a place for the new recordID and key to be place within a specified page. The cursor will be a NonLeafNode and we will continue traversing until we reach a node where we can insert the rid,key pair. While the cursor is changing every time we are also keeping track of the previous cursor in a parent variable.
* Once we have found the correct node to place the record into we can start checking to make sure if we can place the current record.
* For this we have decided to break it up into cases, one where the size of the current array is not overflowing with rid’s and the other is the case where we need to handle to splitting of the tree.
* In order to handle the first case we decided to simply insert the key and the record id into the current node which was found, we do this by going through all the different record ID’s in the cursor node we found, until we find a pageID that is less than the inserted pageID. Once we know the index, we want to move all other entries in the list back by one so that we can insert the new rid pair.
* To handle the other case where we are inserting into an overflowing node, we must first create a virtual node and fill in the non-leaf node with the cursor information. Then we have a recursive helper method which is able to take the new leaf and the old cursor to make sure that they are split and one of the gets any new leaves, this recursively happens so that if there is more splitting required we always have the parent to child relationship to figure out which keys need to be distributed.

**Implementation for Duplicate Keys**

* Currently our code when inserting to the B+ tree a duplicate key, we will handle it by storing each entry through its RID and pageID in order to make sure that all the keys can be inserted even duplicates so under different names.

**Buffer Management (When to Pin & Unpin)**

* In endScan, we teerminate the current scan by setting the scanExecuting variable to false, and setting all specific variables to their defaults. We also unpin the current page of the scan. Because we keep all pages except for the current scan page unpinned, unpinning the current scan page should make it so that all pages in the bufMgr are unpinned by the time
* We keep all pages unpinned except for the current page in the scan, so by calling endScan in the destructor, we successfully unpin any pinned pages prior to flushing file and ending scan.
* In startScan, we traverse down the tree until we find a leaf node(level = 1), then we make sure that the search criteria is satisfied by some key. The page that corresponds to the first leaf with a key that satisfies the scan criteria will be kept pinned for the scanNext method.
* In startScan, we check if the root is a leaf or not. If the root is a leaf, we just set the current page of the scan as the root, and keep it pinned in the bufMgr.

**Runtime Analysis**

We were not able to resolve many of the issues in our code base with the assumptions laid out from above.

**Tests**

* All tests are included in main.cpp, and all test cases are called by default
* Test 1
  + Creates a relation of size 5000 in ascending order
  + Then we check if the output amount is the same as the input
* Test 2
  + Creates a relation of size 5000 in backwards order
  + Then we check if the output amount is the same as the input
* Test 3
  + Creates a relation of size 5000 in random order
  + Then we check if the output amount is the same as the input
* Test 4
  + This test creates key-value pairs from 0 to 5000 in random order by shuffling the input vector
  + Test if the output amount matches the input amount when scanning
  + This test finds keys that are not all in the range of input
* Test 5
  + Inserters entries at random with no duplicates,
  + Test if the output amount matches the input amount when scanning
  + Test if outputs match the sorted input vector
* Test 6
  + Creates key-value pairs from 0 to a huge relationSize, 350000, in ascending order
  + Test if the output amount matches the input amount when scanning
* Test 7
  + Creates key-value pairs from 0 to a huge relationSize, 350000, in backwards order
  + Test if the output amount matches the input amount when scanning
* Test 8
  + Creates key-value pairs from 0 to a huge relationSize, 350000, in random order
  + Test if the output amount matches the input amount when scanning