Assignment 5: B+ Trees

In this assignment you will get to explore an implementation of the B+ tree data structure. The tasks to complete are as follows:

- Complete the B+ tree tuple-file implementation. *(70 points)*
- Do some basic analysis of the B+ tree data structure. *(30 points)*

Overview

In the `edu.caltech.nanodb.storage.btreefile` package you will find a basic implementation of a B+ tree tuple file. This implementation follows the description given in class, with a few important differences. Possibly the most important one is that the B+ tree implementation can be used to store tables as well as indexes. This is an important design choice, because it makes it very easy to perform file-scans over indexes without having to make any substantial changes to the file-scan code paths. More importantly, it allows us to exercise our B+ tree implementation purely through table operations that we already support.

Here are some additional notes on NanoDB's B+ tree implementation:

- "Fullness" of a node is not determined by the number of pointers or entries in the node, but rather by the number of bytes used in the node. Leaf and inner nodes can store different numbers of entries and/or pointers, since their structures are slightly different. When a node is split, the implementation simply divides the number of pointers or entries in half, and moves half of the entries to a sibling node. This means that we will generally satisfy the "at least half-full" rule, but there may be nodes here and there that do not satisfy this constraint. (But, they will be close.)

- When relocating entries or pointers from a node to a sibling, only enough entries are relocated to allow the new entry to be added to either node. There is no attempt to "even out" the number of entries between the pair of nodes. (We try to make space for the new entry to be added to either node because when relocating or splitting, we don't necessarily know which sibling the new entry will end up in.)

Other than that, the implementation follows the description in class almost exactly.

- Each leaf node references the next leaf in the B+ tree, forming a linear sequence of leaves.

- Inner nodes only reference other nodes in the tree (thus, they use an unsigned short for these page-pointers).

- No additional structure is maintained beyond that described in the lecture slides. Nodes do not reference their parents in the tree structure. Leaves do not reference their previous sibling, only their next sibling. Inner nodes only reference nodes deeper in the structure. (The reasons for this will be explored in the design document.)

Important Implementation Classes

Here are the major components in this B+ tree implementation. You will notice that it is mostly similar to the heap file implementation, with a few obvious differences due to the implementation details.

The `BTreeTupleFile` class provides most of the operations for accessing or modifying tuples in a B+ tree file. It delegates many file-manipulation tasks to two classes, `InnerPageOperations` and `LeafPageOperations`, but it does perform some of the most basic operations such as looking up a leaf-entry in the file based on a search-key, or finding new empty pages in the file when more data needs to be stored.

The `InnerPageOperations` and `LeafPageOperations` classes handle larger-scale tasks like inserting entries into B+ tree nodes, splitting nodes, and relocating entries between nodes. If you review this code, you will note that the implementations are very similar, but *just* different enough to force two separate implementations. (Oh well.)

These two classes also use the `InnerPage` and `LeafPage` wrapper-classes to manipulate individual B+ tree pages. Each of these classes is used to wrap a `DBPage` object, allowing the contents of the node to be manipulated more easily.

Finally, the `BTreeTupleFileManager` class provides file-level operations such as creating a new B+ tree file, storing the metadata, and so forth.

Important Final Notes!!!

The B+ tree code in NanoDB is still a bit buggy in very specific scenarios. You should be able to get all required tests to pass for the assignment, but don't be surprised if you run into a little difficulty.

This assignment may not be particularly easy to break down among teammates, so a team-programming style may be the best approach, where teammates work together at one computer to understand what needs to be done, and to make sure the implementation is correct. Different teammates can tackle different steps in the implementation, but you must make sure you understand the parts you are not implementing, before you dive in.

Part 1: Complete Missing B+ Tree Operations

The B+ tree implementation you have been given is missing several important pieces, which you must implement. Those pieces are outlined in this section.

To test your implementation, you can create tables using the "`btree`" storage format, which corresponds to the B+ tree implementation. For example, you can do this:

```
CREATE TABLE bt (
    a INTEGER
) PROPERTIES (storage = 'btree');

INSERT INTO bt VALUES (53);
INSERT INTO bt VALUES (21);
INSERT INTO bt VALUES (65);
...
```

As you get your implementation working, you should be able to "`SELECT * FROM bt`" and see the tuples always produced in order. Note that the tuples are sorted by all columns.

You can also use the `VERIFY` command to check your table for structural issues.

```
VERIFY bt;
```

This command will check the table for any issues, along with any indexes built against the table. All problems that are encountered will be printed out to the console.

1. All operations – adding a tuple, removing a tuple, or searching for tuples – require the B+ tree structure to be navigated from root to leaf. This operation is partially implemented in the `navigateToLeafPage()` method of the `BTreeTupleFile`. **You will need to complete this implementation.**

   Note that this method only navigates the inner-page structure of the index until it reaches a leaf, and then the leaf page is returned to the caller. What happens after that depends on the specific operation being performed.

   Also, all key-comparisons should be performed with the `comparePartialTuples()` method of the `TupleComparator` class (`edu.caltech.nanodb.expressions` package). This method allows tuples of different lengths to be compared, which allows us to search on any prefix of the tuple file's columns, not just the full set of columns. (It will also allow us to find tuples in indexes without specifying the file-pointer at the end of the search-key.)

   Once this function is finished, you should be able to create a table like the one above, insert records into it, and see that the contents of the table always appear in order. However, if your table gets large enough to require two leaf pages, the implementation will fail. The reason is that NanoDB doesn't yet know how to split a leaf page into two leaves. Continue to the next step…

2. To support B+ tree files larger than one leaf page, the implementation must be able to split a leaf into two leaves, and then update the parent of the leaf with the new leaf-pointer. This operation is handled by the `splitLeafAndAddTuple()` method of the `LeafPageOperations` class. **You will need to complete this implementation.**

   As always, there are many helper functions to help you with the implementation, on both the `LeafPage` and `InnerPage` classes. Probably the most complicated part will be updating the parent of the leaf properly, but you can use the `InnerPageOperations` class to help you with this task.

   Note that the `pagePath` argument must always be the path to the specific page being manipulated by a given function. Thus, when calling `InnerPageOperations` functions, you must remove the last element from the `pagePath` list. This is simple to do, and fast too, even though we are using an `ArrayList` for the collection: since we are removing the last element in the array-list, this will be a constant-time operation.

   Once you are done with this task, you should be able to create B+ tree files with many leaf pages. There is one more problem, though – the index implementation still can't support multiple inner pages. To fix this issue, continue on to the final step.

3. The last functionality to complete for this index implementation is the code that allows inner-page pointers to be moved to a left- or right-sibling page. This is required for splitting an inner page into two, and also for relocating pointers between two sibling inner pages. This functionality is provided by the `movePointersLeft()` and `movePointersRight()` methods of the `InnerPage` class.

   These methods are a bit tricky to implement, mainly because of the requirement that every tuple in an inner page must be sandwiched between two pointers. Given an inner page containing N pointers and N-1 tuples, if you move M pointers (and the M-1 tuples between these pointers) from the node to its right sibling ($M < N$), this will expose a tuple in the node without a

pointer on its right. Similarly, if you move *M* pointers from the node to its left sibling, this will expose a tuple without a pointer on its left.

Additionally, the sibling node receiving the *M* pointers and *M*-1 tuples will already have pointers on both sides of all its tuples.

This is where you must figure out how the parent node's tuple fits into the puzzle. In the slides we discussed what happens when a single pointer is moved to a sibling inner-node, but in this implementation it is possible to move *M* pointers, not just one. You will have to figure out where to store the parent's old tuple, if provided, and what to return as the parent's new tuple.

(You will always return a new key in your implementation. You may not receive an old tuple if the top-level inner page is being split, since there will not yet be a parent of the node being split. The tuple you return will be used in initializing the new top-level inner page.)

The other complexity is that when moving *M* pointers to the left sibling, these pointers are taken from the start of the node's sequence, whereas when moving the pointers to the right sibling, they are taken from the end of the node's sequence. When moving pointers right, the implementation must make room in the target node for the new entries. When moving pointers left, the implementation must slide the remaining entries in the source node left. For these kinds of operations, the `DBPage.moveDataRange()` method will be very helpful.

You must <u>never</u> write to the `DBPage`'s internal byte-array directly! Doing this will break the `DBPage`'s ability to track whether the page is dirty. Always use the operations provided on the `DBPage` to write to its data. (You may find it helpful to *read* from the underlying byte-array, however, when moving data back and forth.)

Once you have successfully completed this task, your B+ tree should be complete.

Part 2: Analysis of Implementation

The design document for this assignment has a number of questions for your team to answer about the implementation of B+ trees. The questions are in section B (of course). Complete all questions in this section. You will need to be familiar with the details of NanoDB's B+ tree implementation for several of these questions.

Part 3: Extra Credit

This assignment is hard, and the extra credit is also hard, so you probably won't get to this.

- The `TestBTreeFile.testBTreeTableMultiLevelInsertDelete()` test currently fails. Figure out why, and fix it. *(+10 points for good explanation + fix; definitely not worth it)*

Submitting Your Assignment

When you are finished with the coding part of the assignment, tag it with a `hw5` tag as usual, and then push all of your changes to your repository on the CMS cluster.