Programming Project 3: B+ Tree Index Manager

INTRODUCTION

As we discussed in class, relations are stored in files. Each file has a particular organization. Each organization lends itself to efficient evaluation of some (not all) of the following operations: scan, equality search, range search, insertion, and deletion. When it is important to access a relation quickly in more than one way, a good solution is to use an index. For this assignment, the index will store data entries in the form **<key, rid> pair**. These data entries in the index will be stored in a file that is separate from the data file. In other words, the index file "points to" the data file where the actual records are stored. Two primary kinds of indexes are hash-based and tree-based, and the most commonly implemented tree-based index is the B+ Tree.

For the second part of the BadgerDB project, you need to implement a B+ Tree index. To help get you started, we will provide you with an implementation of few new classes: **`PageFile`**, **`BlobFile`**, and **`FileScan`**.

The `PageFile` and `BlobFile` classes are derived from the `File` class. These classes implement a file interface in two different ways. The `PageFile` class implements the file interface for the `File` class, as was done in your last buffer manager assignment. Hence, we use the `PageFile` class to store all the relations as we did in the buffer manager assignment.

The `BlobFile` class implements the file interface for a file organization in which the pages in the file are not linked by `prevPage/nextPage` links, as they are in the case of the `PageFile` class. When reading/writing pages, the `BlobFile` class treats the pages as blobs of `8KB` size and hence does not require these pages to be valid objects of the `Page` class. We will use the `BlobFile` class to store the B+ index file, where every page in the file is a node from the B+ Tree. Since no other class requires `BlobFile` pages to be valid objects of the `Page` class, we can modify these pages as we wish without worrying that these pages will not be valid after their arbitrary modification. Inside the file `btree.cpp` you will treat the pages from a `BlobFile` as your B+ Tree index nodes, and the `BlobFile` class will read/write pages for you from disk without modifying/using them in any way. `BufMgr` class has also been changed so that it does not use page objects to find out their page numbers.

THE FILESCAN CLASS

The `FileScan` class is used to scan records in a file. We will use this class for the base relation, and not for the index file. The file `main.cpp` file contains code which shows how to use this class. The public member functions of this class are described below.

- **`FileScan (const std::string &relationName, BufMgr *bufMgr);`**
  The constructor takes the relationName and buffer manager instance as parameters. The methods described below are then used to scan the relation.

- **`~FileScan()`**
  Shuts down the scan and unpins any pinned pages.

- **`void scanNext(RecordId& outRid);`**
  Returns (via the `outRid` parameter) the `RecordId` of the next record from the relation being scanned. It throws `EndOfFileException()` when the end of relation is reached.

- **`std::string getRecord();`**
  Returns the record identified by `rid`. The `rid` is obtained by a preceding `scanNext()` call.

- **`void markDirty()`**
  Marks the current page being scanned as dirty, in case the page was being modified. (You don't need this for this assignment, but the method is here for completeness).

B+ TREE INDEX

Your assignment is to implement a B+ Tree index. This B+ Tree will be simplified in a couple of ways. First, you can assume that all records in a file have the same length (so for a given attribute its offset in the record is always the same). Second, the B+ Tree only needs to support single-attribute indexing (not composite attribute). Third, the indexed attribute may be only one data type: **integer**. Finally, you may assume that we never insert two data entries into the index with the same key value.

The index will be built directly on top of the I/O Layer (the `BlobFile` and the `Page` classes). An index will need to store its data in a file on disk, and the file will need a name (so that the DB class can identify it). The convention for naming an index file is specified below. To create a disk image of the index file, you simply use the `BlobFile` constructor with the name of the index file. The file that you create is a "raw" file, i.e., it has no page structure on top of it. You will need to implement a structure on top of the pages that you get from the I/O Layer to implement the nodes of the B+ Tree. Note the `PageFile` class that we provide superimposes a page structure on the "raw" page. Just as the File class uses the first page as a header page to store the metadata for that file, you will dedicate a header page for the B+ Tree file too for storing metadata of the index.

We'll start you off with an interface for a class, `BTreeIndex`. You will need to implement the methods of this interface as described below. You may add new public methods to this class if required, but you should not modify the interfaces that are described here:

- **`BTreeIndex`**
  The constructor first checks if the specified index file exists. And index file name is constructed by concatenating the relational name with the offset of the attribute over which the index is built. The general form of the index file name is as follows: `relName.attrOffset`. The code for constructing an index name is shown below:

```
std::ostringstream idxStr;
idxStr << relationName << '.' << attrByteOffset;
std::string indexName = idxStr.str(); // indexName is the name of the
    index file
```

  If the index file exists, the file is opened. Else, a new index file is created.

  Input to this constructor function:

`const string& relationName`	The name of the relation on which to build the index. The constructor should scan this relation (using `FileScan`) and insert entries for all the tuples in this relation into the index. You can insert an entry one-by-one, i.e., don't worry about implementing a bottom-up bulkloading index construction mechanism.
`String& outIndexName`	The name of the index file; determine this name in the constructor as shown above, and return the name.
`BufMgr *bufMgrIn`	The instance of the global buffer manager.
`const int attrByteOffset`	The byte offset of the attribute in the tuple on which to build the index. For instance, if we are storing the following structure as a record in the original relation: `struct RECORD { int i; double d; char s[64]; };` And, we are building the index over the `double d`, then the `attrByteOffset` value is **`0 + offsetof (RECORD, i)`**, where `offsetof` is the offset position provided by the standard C++ library "`offsetoff`".
`const Datatype attrType`	The data type of the attribute we are indexing. Note that the `Datatype` enumeration `INTEGER, DOUBLE, STRING` is defined in `btree.h`.

- **`~BTreeIndex`**
  The destructor. Perform any cleanup that may be necessary, including clearing up any state variables, unpinning any B+ Tree pages that are pinned, and flushing the index file (by calling `bufMgr->flushFile()`). Note that this method does not delete the index file! But, deletion of the file object is required, which will call the destructor of `File` class causing the index file to be closed.

- **`insertEntry`**
  This method inserts a new entry into the index using the pair `<key, rid>`.

  Input to this function:

`const void* key`	A pointer to the value (integer) we want to insert.
`const RecordId& rid`	The corresponding record id of the tuple in the base relation.

- **`startScan`**
  This method is used to begin a "filtered scan" of the index. For example, if the method is called using arguments `(1,GT,100,LTE)`, then the scan should seek all entries greater than `1` and less than or equal to `100`.

  Input to this function:

`const void* lowValue`	The low value to be tested.
`const Operator lowOp`	The operation to be used in testing the low range. You should only support `GT` and `GTE` here; anything else should throw `BadOpcodesException`. Note that the `Operator` enumeration is defined in `btree.h`.
`const void* highValue`	The high value to be tested.
`const Operator highOp`	The operation to be used in testing the high range. You should only support `LT` and `LTE` here; anything else should throw `BadOpcodesException`.

  Both the high and low values are in a binary form, i.e., for integer keys, these point to the address of an integer.

  If `lowValue > highValue`, throw the exception `BadScanrangeException`.

- **`scanNext`**
  This method fetches the record id of the next tuple that matches the scan criteria. If the scan has reached the end, then it should throw the following exception: `IndexScanCompletedException`. For instance, if there are two data entries that need to be returned in a scan, then the third call to `scanNext` must throw `IndexScanCompletedException`. A leaf page that has been read into the buffer pool for the purpose of scanning, should not be unpinned from buffer pool unless all records from it are read or the scan has reached its end. Use the right sibling page number value from the current leaf to move on to the next leaf which holds successive key values for the scan.

  Input to this function:

`RecordId& outRid`	An output value; this is the record id of the next entry that matches the scan filter set in `startScan`.

- **`endScan`**
  This method terminates the current scan and unpins all the pages that have been pinned for the purpose of the scan. It throws `ScanNotInitializedException` when called before a successful `startScan` call.

ADDITIONAL NOTES

1. When you implement these methods, you will need to call upon the buffer pool to read/write pages. Make sure you don't keep the pages pinned in the buffer pool unless you need to. If you keep some pages pinned, make sure you have a good reason that you justify in your design report.

2. For the scan methods, you will need to remember the "state" of the scan specified during the startScan call. Use appropriate member variables in the `BTreeIndex` class to remember this state. Make sure you reset these state variables in the endScan and the destructor.

3. The insert algorithm does not need to redistribute entries, i.e., always prefer splits over key redistribution during inserts. (It is easier to implement inserts this way too).

4. At the leaf level, you do not need to store pointers to both siblings. The leaf nodes only point to the "next" (the right) sibling.

5. The constructor and destructor should not throw any exceptions.

6. In real B+ Tree implementations, when an error occurs, special care is taken to make sure that the index does not end up in an inconsistent state. As you will quickly realize handling errors can be hard in some cases. For example, if you have split the leaf page and are propagating the split upwards, and then encounter a buffer manager error, exiting the method without cleaning up could corrupt the B+ Tree structure. To keep the assignment simple, don't worry about this type of cleanup, simply return the error code. Make sure you don't artificially create such problems by incorrectly using the other components of BadgerDB. For example, if you keep pages pinned in memory unnecessarily, you will quickly encounter a buffer exceeded exception. We will not test your implementation with very small buffer pool sizes (such as 1 or 2 pages). If it makes your implementation easier, you may assume that you have enough free buffer pages to hold 1-2 pages from each level of the index. But UNPIN THE PAGES as soon as you can.

GETTING STARTED

Start by copying the files from `Btree.zip`. In the zipped folder, you will find the files listed below. Follow these instructions to complete your assignment. This directory contains the following files that are relevant to this part of the project (in addition to other files which were created while developing the lower layers):

- `btree.h`: Add your own methods and structures as you see fit but don't modify the public methods that we have specified.
- `btree.cpp`: Implement the methods we specified and any others you choose to add.
- `file.h(cpp)`: Implements the `PageFile` and `BlobFile` classes.
- `main.cpp`: Use to test your implementation. Add your own tests here or in a separate file. This file has code to show how to use the `FileScan` and BTreeIdnex classes.
- `page.h(cpp)`: Implements the `Page` class.
- `buffer.h(cpp)`,`bufHashTbl.h(cpp)`: Implementation of the buffer manager.
- `Exceptions/*` : Implementation of exception classes that you might need.
- `Makefile` : Makefile for this project.

Please do not create any additional files.

In addition to the B+ Tree source files, you must also turn in

1. New test cases that you wrote to test your B+ Tree index.
2. Design report describing your tests and design choices.

DELIVERABLES

Submit your work on Canvas. Your submission should include the source code, new test cases and the design report. Submitting `btree.cpp` is mandatory. If you have also modified the header file, submit `btree.h` as well. Your new test case can be written in `main.cpp`, or in separate file. Clearly indicate what your new test case and design report files are called, for example by putting in a `outline.txt` file that describes your test and design report file locations (these must be uploaded to Dropbox too). Your files must be uploaded by the deadline stated on the first page.

GRADING

The breakup of the grading for this assignment is as follows:

1. **Correctness: 75%**. The correctness part of the grade will be based on the tests that we have provided, and additional (more rigorous) tests that we will run on your submitted projects.

2. **Programming Style: 5%**. For your style points, we will check your code for readability (how easy is it to read and understand the code), and for the code organization (do you repeat code over and over again, do you use unnecessary globals, etc.).

3. **Test design: 5%**. Designing tests cases that test various code paths rigorously. This will not only get you the test points, but will most likely also get you the correctness points.

4. **Design report: 15%**. Your design report must describe the following design choices that you make:

    - Any implementation choices that you make. How often do you keep pages pinned? How efficient is your implementation? We are not going to run a speed test, but will look at the code to check if you are performing operations that are inefficient, such as unnecessarily traversing the tree up and down multiple times during range searches
    - Please use your report to justify any additional design choices that you make.

A FINAL NOTE OF CAUTION

There are number of design choices that you need to make, and you probably need to reserve a big chunk of time for testing and debugging. So, start working on this assignment early – you are unlikely to finish this project if you start just a week or so before the deadline.