

CS 564, Spring 2017

BadgerDB Assignment #2: B+-Tree Index Manager

Due Date: March 27, by 2:00PM. No late days.

Project Grade Weight: 13% of the total grade

Introduction

As 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 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. In this assignment, you implement a tree-based index, namely the ubiquitous B+-Tree index.

To help get you started, we will provide you with an implementation of the following 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 to suit the particular needs of the B+-tree index. 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 determine their page numbers.

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()**
Shutsdown 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 a pointer to the “current” record. The record is the one in 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 the following five ways:

1. 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).
2. Second, the B+-tree only needs to support single-attribute indexing; i.e., not a *composite* attribute where the key has more than one attribute, such as a pair (age, zipcode).
3. Third, the indexed attribute may be one of three data types: integer, double, or string.
4. Fourth, in the case of a string, you can assume that all string attributes and search keys in BadgerDB are larger than 10 characters, and that the first 10 characters of each entry that is inserted into the index is unique. In other words, for strings, you can use the first 10 characters as the key in the B+-tree.
5. Finally, you can assume that we will never insert two data entries into the index with the same key value. This last part simplifies the B+-tree implementation (think about why, and put that in your report).

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 meta-data for that file, you will dedicate a header page for the B+-tree file too for storing meta-data 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. An index file name is constructed as: 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 “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, then the file is opened. Else, a new index file is created.

The inputs to this constructor function are:

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: <pre>struct RECORD { int i; double d; char s[64]; };</pre> 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 "offsetof".
const Datatype attrType	The data type of the attribute we are indexing. Note that the Datatype enumeration {INTEGER, DOUBLE, STRING} is defined in the file 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 the function 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>.

The inputs to this function are:

const void * key	A pointer to the value (integer/double/string) 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 ("a", GT, "d", LTE), then the scan should seek all entries greater than "a" and less than or equal to "d".

The inputs to this function are:

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 <code>BadOpcodesException</code> . Note that the <code>Operator</code> enumeration is defined in <code>btree.h</code> .
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 <code>BadOpcodesException</code> .

Both the high and the low values are in a binary form, i.e. for integer (double) keys, these point to the address of an integer (double), and for character strings they point to a character string (char array) that can be of any length greater than 10 characters. You only need to use the first 10 characters in your key for comparison.

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 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 the 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 to the next leaf which holds successive key values for the scan.

The input to this function is:

RecordId & outRid	An output value; this is the record id of the next entry that matches the scan filter set in <code>startScan</code> .
----------------------	---

- **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

- When you implement these methods, you will need to call upon the buffer pool to read/write pages. Make sure that you do not keep the pages pinned in the buffer pool unless you need to. If you keep some pages pinned, make sure that you have a good reason and justify that reason in your design report.
- For the scan methods, you will need to remember the “state” of the scan specified during the `startScan` call. Use the appropriate member variables in the `BTreeIndex` class to remember this state. Make sure that you reset these state variables in the `endScan` and the destructor.
- 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).
- 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.

- e) The constructor and destructor should not throw any exceptions.
- f) 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 a leaf page and are propagating the split upwards, and then encounter a buffer manager error, exiting the method without cleaning up can corrupt the B+-tree structure. To keep the assignment simple, don't worry about this type of cleanup, simply return the error code. However, make sure that you do not 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.

Your Assignment

Start by copying <http://pages.cs.wisc.edu/~jignesh/cs564/projects/BadgerDB/Btree.tar.gz> After extracting this directory, 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 BTreeIndex 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 btree 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.

Submitting Your Assignment

To hand in your work, please go to the *Canvas: Assignment 3 (BadgerDB, B+-Tree)* to upload your files. Your submission should include the source code, new test cases and the design report. 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). Your files must be uploaded by the deadline stated on the first page.

Please follow these instructions to submit the project:

- 1) Name the project directory: <lastname>_<firstname>_P3 (e.g. Murray_Bill_P3)
- 2) Run: `make clean` from inside the directory (so that the submitted file size is small)
- 3) Run: `tar -czvf <lastname>_<firstname>_P3.tar.gz <path>?/lastname_firstname_P3`
- 4) Submit the tar file
- 5) To check, you can uncompress the tar file (run: `tar -xzvf <lastname>_<firstname>_P3.tar.gz`)
You should see the *Makefile*, design docs and all the source code files inside the untarred directory. **If you do not adhere to this standard, you risk losing all the test points, as our test driver will fail.** Since we are supposed to be able to test your code with any valid driver, it is important to be faithful to the exact definitions of the interfaces that are specified here.

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.
 - Finally, your design report must also explain how your design/implementation would change if you were to allow duplicate keys in the B+-tree, i.e. allow multiple data entries with the same key value.

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 is just a week or two before the deadline.

Adventerous? Need Extra Credits?

If you are adventerous, try implementing the delete function for the B+-Tree (by the deadline for this assignment). If you do this, and can demonstrate this functionality works, then you can earn upto **2% extra credits for the course**. This is about the credits for one quiz (2.5%). If you need to make up for a exam/quiz or another assignment, then here is your chance. If you are already doing well in the course, here is a chance to put yourself over the top.

Turn in the delete functionality with the regular assignment, and clearly note that you have implemented the delete function in your report. Also, explain your design in the report, and before the deadline, send an email your instructor with the Subject line: **[CS 564: Spring 2017] B+-Tree delete functionality submitted.**