Assignment 1: Implementing a Heap File

Due Saturday, Feb 3th by 12PM

Overview

In this assignment, your task is to implement the `DBFile` class. The job of the `DBFile` class within your database system is simply to store and retrieve records from the disk. By the time you are done with the next assignment (assignment two), there will be three "flavors" of `DBFiles` that you will implement: a heap (an unordered file of records, where new records simply go at the end of the file), a sorted file, and a B+-Tree file. To start with in assignment one, you only need to implement the heap.

The Code Base You'll Start With

You will see that along with this assignment, I have provided a couple of thousand lines of code in an archive. This is the basic code that your database system will be constructed on top of. The interface to this code is provided in five header files: `Record.h`,

`File.h`, `Comparison.h`, `ComparisonEngine.h`, and `Schema.h`. You should probably not change the class definitions provided in these files, except to add to them if you feel that not enough functionality has been provided. If you feel that you do need to substantially change these header files, I'd strong recommend that you talk with me first!

The first header file (`Record.h`) contains the basic definition of the `Record` class. This class implements the actual objects that your database will store, and so your `DBFile` will contain objects that are instances of this class. This class stores all of the data in each record as a flat bit string.

The second header file (`File.h`) contains the basic definitions of the `Page` and `File` classes. The `Page` class is the in-memory realization of a database page; a page is essentially a collection of database records. The `File` class is a disk-based container class that essentially holds an array of pages. Your `DBFile` will be built on top of the `Page` and `File` classes, so that when someone inserts a bunch of records into your `DBFile`, your `DBFile` in turn will group those records into instances of the `Page` class, and then store those instances of the `Page` class into an instance of the `File` class. The details of how this grouping and storage are done will vary based upon the file type, but in the case of a simple heap file, you'll want to have a single instance of the `Page` class stored within your `DBFile` object to act as a one-page buffer. During a series of writes, after that instance of the `Page` class fills, you write its contents to disk using an instance of the `File` class that you'll use to actually store the `DBFile`. During reads, you use that instance of the `Page` class to read records from disk, one page at a time, and then return the records to the caller as they are requested. Of course, you'll have to deal with

the case when someone switches back and forth between reads and writes, in which case you may have to write this page to disk before it is full, assuming that it is dirty.

The third header file (`Comparison.h`) implements many of the standard operations that must be provided by the database record manager; that is, they will allow your database to semantically interpret the records that it stores. There is one class called `CNF`, which is constructed from the parse tree for a conjunctive normal form predicate. This class tells the database system how to apply a user-supplied conjunctive normal form expression to a given records. There is another class called `OrderMaker` (which you won't use in the first assignment) that encodes a less-than/greater-than comparison across two records; this class is used for sorting operations that you'll write starting with assignment two.

The fourth header file (`ComparisonEngine.h`) contains code that actually uses the classes that are provided in `Comparison.h` to perform comparisons. For example, the `ComparisonEngine` class will allow you to actually use a `CNF` object to see whether or not a given record has been accepted by the underlying conjunctive normal form predicate.

Finally, the fifth header file (`Schema.h`) encodes a few functions that load up a relation schema from the database catalog. An example database catalog is given in the file `catalog`. In fact, this catalog gives the relation schemas for several of the tables from the "TPC-H" databases schema, which you will become very familiar with over the semester. The TPC-H is a very well-known standard database benchmark that we will make extensive use of in this class. One of the first things that you should do when you start on this assignment is to check out http://www.tpc.org/tpch, and at least look over the PDF document describing this schema.

Note that there is also a simple `lex`/`yacc` parser included in the archive that will parse CNF statements (see http://en.wikipedia.org/wiki/Lex_programming_tool and http://en.wikipedia.org/wiki/Yacc if you have no idea what these tools are). This parder makes it possible for you to easily type in CNF statements using the keyboard; later on in the semester, you will considerably extend this parser so that you can actually parse full SQL queries.

Getting Started

It is always quite difficult to become familiar with an existing code base, and before you actually write any code of your own, you will need to try out and get familiar with all of the files that I have given you.

After you have downloaded my code and carefully read this assignment, the first thing that you should do is to compile my code and the small, sample main program I have provided (`main.cc`). You can compile this program by running the `Makefile` that I have included inside of the archive. Make sure that you are logged on to a CISE machine or that you are

running linux (raw or in a VM on your laptop, Ubuntu Subsystem for Windows might work as well). If you work on your laptop, make sure you have gcc (or clang) installed together with "bison" and "flex". "bison" provides the "yacc" command and "flex" the "lex" command needed to implement the parser.

At the Unix/Linux command prompt, type `make` to make the program, and then type `./main` to actually run the program. The program will ask you to enter in a CNF expression. For starters, you can type in `(l_orderkey > 27) AND (l_orderkey < 45)` and then press <control-D>. See what happens. Take a look at `main.cc`. The program will read in and parse the CNF text that you gave it, and then use the text along with the system catalog to build a `CNF` structure that will accept/reject each of the records from the TPC-H `lineitem` relation (check out the `lineitem` relation's schema by looking at the text file `catalog`). After you have typed in your CNF expression, the program then goes off to a text file located in `/cise/tmp/chris`, and converts those lines of text into `lineitem` database records one at a time. For each database record that it finds, it applies the instance of the `CNF` class that it built, and then prints out the record to the screen if the `CNF` evaluates to true over the record.

After you have compiled and tried out `main.cc`, I would advise you to play with `main.cc` a bit before you really get started on this assignment, to familiarize yourself with the code base that you will start from. Try to alter `main.cc` so that it loads a page up full of records, and then writes that page to disk. Then try to retrieve the page from disk and print the records to the screen. Once you feel fairly comfortable with the code base and how to use it, then you are ready to implement assignment one.

What You Need to To Do In This Assignment

In this assignment, your task is to partially define and implement the `DBFile` class. Your version of the `DBFile` class can contain whatever private data structures that you wish, though it needs to contain the following eight public member functions (in addition to a standard constructor and destructor).

As described above, your `DBFile` will be build on top of my `File` and `Page` classes. So the actual records that are stored inside of a `DBFile` will be stored in a single instance of the `File` class (at least in the case of the simple heap file). You can also have any auxiliary text or meta-data files that you need in order to implement your file in order to make it persistent. Just read/write any metadata to a text file using the standard `iostream` or `stdio` functions when you open or close your `DBFile`.

Now, we are finally ready to describe the interface to the `DBFile` class that you need to implement.

Each `DBFile` instance has a "pointer" to the current record in the file. By default, this pointer is at the first record in the file, but it can move in response to record retrievals. The

following function forces the pointer to correspond to the first record in the file:

```
void MoveFirst ()
```

In order to add records to the file, the function `Add` is used. In the case of the unordered heap file that you are implementing in this assignment, this function simply adds the new record to the end of the file:

```
void Add (Record &addMe)
```

Note that this function should actually consume `addMe`, so that after `addMe` has been put into the file, it cannot be used again.

There are then two functions that allow for record retrieval from a `DBFile` instance; all are called `GetNext`. The first version of `GetNext` simply gets the next record from the file and returns it to the user, where "next" is defined to be relative to the current l ocation of the pointer. After the function call returns, the pointer into the file is incremented, so a subsequent call to `GetNext` won't return the same record twice. The return value is an integer whose value is zero if and only if there is not a valid record returned from the function call (which will be the case, for example, if the last record in the file has already been returned).

```
int GetNext (Record &fetchMe);
```

The next version of `GetNext` also accepts a selection predicate (this is a conjunctive normal form expression). It returns the next record in the file that is accepted by the selection predicate. The literal record is used to check the selection predicate, and is created when the parse tree for the CNF is processed.

```
int GetNext (Record &fetchMe, CNF &applyMe, Record &literal);
```

Next, there is a function that is used to actually create the file, called `Create`. The first parameter to this function is a text string that tells you where the binary data is physically to be located – you should store the actual database data using the `File` class from `File.h`. If you need to store any meta-data so that you can open it up again in the future (such as the type of the file when you re-open it) as I indicated above, you can store this in an associated text file – just take name and append some extension to it, such as `.header`, and write your meta-data to that file.

The second parameter to the `Create` function tells you the type of the file. In `DBFile.h`, you should define an enumeration called `myType` with three possible values: `heap`, `sorted`, and `tree`. When the `DBFile` is created, one of these tree values is passed to `Create` to tell the file what type it will be. In this assignment, you obviously only have to deal with the `heap` case. Finally, the last parameter to `Create` is

a dummy parameter that you won't use for this assignment, but you will use for assignment two. The return value from `Create` is a 1 on success and a zero on failure.

```
int Create (char *name, fType myType, void *startup);
```

Next, we have `Open`. This function assumes that the `DBFile` already exists and has previously been created and then closed. The one parameter to this function is simply the physical location of the file. If your `DBFile` needs to know anything else about itself, it should have written this to an auxiliary text file that it will also open at startup. The return value is a 1 on success and a zero on failure.

```
int Open (char *name);
```

Next, `Close` simply closes the file. The return value is a 1 on success and a zero on failure.

```
int Close ();
```

Finally, the `Load` function bulk loads the `DBFile` instance from a text file, appending new data to it using the `SuckNextRecord` function from `Record.h`. The character string passed to `Load` is the name of the data file to bulk load.

```
void Load (Schema &mySchema, char *loadMe);
```

What You Need To Turn In

The TAs for the class will shortly post a set of test cases that you need to run using a small main program that you will implement to test your `DBFile` class. You will turn in your code as well as the results of running your program on the specified test cases. In the meantime, you should begin implementing and testing your `DBFile`. You can start with the `lineitem` table. I would also encourage you to download, compile, and run the `tpchgen` program available from the TPC-H website to generate your own test data instances, which can be larger than the data that I've supplied. In fact, this is what you will need to do to actually run the TAs' test cases.

A Few Final Words

One last thing. Since your `DBFile` instances can store quite a lot of data, you won't be able to store the data using your CISE account. You can do one of two things. Either use your own hardware to test and implement this project, or else use the CISE hardware and actually store the data in the scratch space in `/cise/tmp`. However, if you choose the latter option, *BE POLITE TO OTHERS AND DON'T LEAVE YOUR TRASH AROUND* . In other words, delete any data files that you create right after you are done with them, so that you don't end up taking up a bunch of space in `/cise/tmp` for no reason.

Finally, good luck and have fun!