Assignment 2: Implementing a Sorted File

Milestone One is due Monday, February 12th at 8PM
Milestone Two is due Friday, February 23rd at 8PM

Overview

In this assignment, your task is to extend the `DBFile` class so that it implements both a sorted file and a heap. You did most of the work for the heap file in the last assignment, so the vast majority of the work in this assignment involves implementing the sorted variation of the `DBFile` class.

This particular assignment is rather involved, and to help everyone finish it successfully, there are two separate milestones, each with a different due date. The first milestone involves implementing the `BigQ` class, which a disk-based priority queue implements the TPMMS algorithm. The `BigQ` class will be used by the sorted file to actually do its sorting. Be aware that since databases are so fundamentally based on sorting, you will also make extensive use of the `BigQ` class elsewhere in your database system: the sorted file is not the only place where it will be used!

The second milestone involves using the `BigQ` class to actually implement the sorted version of the `DBFile` class. Most of the complexity with the second milestone involves implementing the ability to query the sorted file in such a way that the file makes use of the sort order in a binary search.

Milestone One: The `BigQ` Class

Your first task is implementing the `BigQ` class. The `BigQ` class encapsulates the process of taking a stream of inserts, breaking the stream of inserts into runs, and then using an in-memory priority queue to organize the head of each run and give the records to the caller in sorted order, just like we talked about in class.

One thing that complicates the `BigQ` class just a little bit is that it needs to support multi-threading. Multi-threading will actually make our database engine simpler and easier to implement later on. We will use `pthreads` for our multi-threading; if you are not familiar with `pthreads`, there is a very nice description at https://computing.llnl.gov/tutorials/pthreads/. Fortunately, our multi-threading is not too hard to figure out. All of the concurrency and synchronization in our system will be managed using the `Pipe` class, which I've implemented and can be downloaded from http://www.cise.ufl.edu/class/cop6726sp08/A2/Pipe.tar. The functionality of the `Pipe` class is quite simple: it allows produce and consumer threads to communicate via a pipe of records from one thread to another. When you want to add a record into a pipe, you call `Pipe.Insert`. When you want to get the next record from a pipe, you call

`Pipe.Remove`. Records come out of a pipe in the same order that they go in. If someone adds to a pipe but the pipe is full, then the call to `Pipe.Insert` blocks until the pipe has room for the record. When someone tries to take from a pipe but the pipe is empty, then `Pipe.Remove` blocks until there is a record to remove (or until the pipe shuts down). In this way, the `Pipe` class allows for synchronization among producers of records and consumers of records.

Given the `Pipe` class, the interface for the `BigQ` class is very simple, and consists only of a constructor and a destructor. The constructor for the `BigQ` class is as follows:

```
BigQ (Pipe &inputPipe, Pipe &outputPipe, OrderMaker
&sortOrder, int runLength);
```

When the constructor is called, the `BigQ` class sets up its internal data structures, spawns its own (and only) worker thread, and then returns from the constructor. Right after it is born, the worker thread repeatedly calls `inputPipe.Remove` to get all of records from the input pipe and uses the TPPMS algorithm to sort them. The sort order used is given via an instance of the `OrderMaker` class. The `BigQ` class starts out by sorting all of the records piped through `inputPipe` into runs, each of which consists of (approximately) `runLength` pages of records. That is, you keep on accepting records as long as they fit into no more than `runLength` pages, and then you sort them and write them out as a run. The worker thread keeps reading in records, sorting them, and then writing them out into sequences of pages until the input pipe is exhausted. This completes phase one of the TPMMS algorithm. Note that for several reasons (most notably problems with having too many open files), all of the runs that you produce should be put into the same instance of the `File` class, and not into multiple files.

After that, the worker thread runs phase two of the TPMMS algorithm. It builds an in-memory priority queue over the head of each run, and then uses the queue to merge the records from all of the runs. It shoves all of the records into `outputPipe` in sorted order, and when it has processed all of the records that it has been given, the worker thread shuts down the output pipe, and dies. That's it!

Milestone Two: Extending the `DBFile` Class

Your next job in this assignment is to add a bunch of code to the `DBFile` class that allows a `DBFile` to be *either* a heap or a sorted file. Since you already did most of the heap stuff, most of the hard work involves adding the sorted file capability.

First, let me say a few words about the basic design of the `DBFile` class here. The way I've designed this is that both the sorted file and the heap file are encapsulated within the same `DBFile` class. One way to do this would have been to make `DBFile` a base class, and then derive both the heap and the sorted file classes from the base `DBFile`. But in designing our system, I decided not to do this for an important reason. When an instance

of the DBFile class is created, the main program does not always know whether the DBFile is a sorted file or a heap file at creation time, and the object instance *itself* might need to figure this out! This makes it somewhat inconvenient to use subclassing. Consider the case where a database file is created as a heap, then closed, the database is stopped, then started again, and then the DBFile needs to be opened up once again. Your higher-level database code should just have to create an instance of a DBFile class and then call Open; the DBFile itself should then figure out that it's a heap. The database itself should not need to figure out that the file is a heap and then create an object of the appropriate type – this would be quite annoying and require the database to store this information externally, separate from the DBFile. That is why DBFile is not subclassed to handle the various file types.

What you are certainly free to do (and what I recommend that you do) is to create a virtual base class that is used *internally* by the DBFile, to implement either sorted or heap file functionality. Say you call this class GenericDBFile, and it has two subclasses: Heap and Sorted. When you fire up the DBFile (that is, when you call DBFile.Create or DBFile.Open), your DBFile creates an instance of either Heap or Sorted as appropriate, and then stores this instance in an internal class variable of type GenericDBFile (assume that we call this myInernalVar). Whenever someone external to the DBFile calls one of the DBFile functions like Insert, your DBFile in turn calls myInternalVar.Insert, which will automatically invoke the correct (sorted or heap) functionality. I would especially recommend this if you plan on also implementing a B+-Tree, since this will add a third variation on the DBFile functionality that you will have to manage and things will get quite messy if you are not careful.

Keeping this in mind, we'll now consider each of the DBFile member functions that you were asked to implement last time, and how they should be extended/changed.

The first function is Open:

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

Of course, Open assumes that the file in question has already been created previously and is sitting on disk at the location indicated by name. Since a DBFile may be either sorted or an unordered heap in Assignment Two, for this assignment you should definitely associate a meta-data file with each and every instance of the DBFile class. While you didn't necessarily need this meta-data file in Assignment One, you certainly do this time around so that you can store important information about the DBFile when the database is shut down, like the file's type (heap or sorted). In the Assignment Two version of the DBFile class, Open should first go to the metadata file associated with the name parameter, and then figure out the type of the file (either sorted file or heap; you might just want to write the word "sorted" or "heap" in the first line of the meta-data file to remember this). If the file is a heap, then for the remainder of its life (until it is closed or destroyed) this instance of the DBFile will implement exactly the

functionality described in the previous assignment. If the file is a sorted file, then your meta-data file should also contain information about the sort order for the file. In practice, this means that you should write a text version of the `OrderMaker` instance that is passed to the sorted file on creation to the `DBFile`'s meta-data file(see `Create` below for information on how you are actually passed this `OrderMaker` in the first place). Then, on a call to `Open`, you "re-constitute" this `OrderMaker` instance from the text in the meta-data file, you are ready to go.

Now, we are ready to describe the updated, assignment two version of `Create`:

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

When your updated `Create` is called, you should first check whether the parameter `myType` indicates that the new file will be a heap or a sorted file. If it is a heap, then you will run all of the code that you wrote for assignment one for the rest of the file's existence. In this case, you can ignore the parameter `startup` (just like in the last assignment).

If the file is a sorted file, then `startup` is actually a pointer to an instance of the following `struct`:

```
struct SortInfo {
     OrderMaker *myOrder;
     int runLength;
};
```

The sort order for the sorted file is specified in `myOrder`, and the length of the runs (in pages) used for sorting is specified in the parameter `runLength` (this will be used by an internal instance of the `BigQ` class).

Where the sorted file really gets interesting is with the `Add` and `Load` functions:

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

If the `DBFile` is a sorted file, then internally, your `DBFile` should have two modes that it can be in: "reading" or "writing". If the current mode is "writing", then the sorted file should internally maintain an instance of the `BigQ` class in addition to its normal, sorted data file that is encapsulated within an instance of the `File` class; effectively, this instance of the `BigQ` class acts as a differential file. The run size for this `BigQ` is given in the call to `DBFile.Create`. When `Add` or `Load` are called for a sorted `DBFile`, then the `DBFile` simply inserts the new record (or records) into its instance of the `BigQ` class. If the sorted `DBFile`'s current mode is "reading", then its `BigQ` is empty (or uninitialized). If someone calls `Add` or `Load` while the sorted `DBFile` is in "reading" mode, then the `DBFile` sets up its internal `BigQ`, adds the new records to it one-at-a-

time via the queue's input pipe, and changes its mode to "writing". If the current mode is "writing" and someone calls any other valid function besides `Add` or `Load` (that is, they call `MoveFirst`, `Close`, or `GetNext`), then the first thing that the sorted `DBFile` needs to do is to merge its internal `BigQ` with its other sorted data (this empties out the `BigQ`) and then change its mode to "reading". This merge process is actually not too hard. All you do is to shut down the `BigQ`'s input pipe, and then remove the records from the `BigQ`'s output pipe one-at-a-time to get the records from the `BigQ` in sorted order. At the same time that you are removing the records from the internal `BigQ`, you scan the sorted file's data in sorted order. Using a standard, two-way merge algorithm (like the one used in a classic merge sort), you merge the two sets of data and write the result out to a new instance of the `File` class. When you are done, this instance of the `File` class now contains all of the `DBFile`'s data, and you are ready to begin reading or shut down the file.

Assuming that your `BigQ` class is working well, processing data insertions is the easy part. Answering queries is more interesting and far more work – especially the version of `GetNext` that accepts a `CNF` instance.

Just like in Assignment One, the following function forces the pointer to correspond to the first record in the file:

```
void MoveFirst ();
```

The first version of `GetNext` is exactly the same in both the sorted and heap files. It just scans the file in sequential order:

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

Next, we have the version of `GetNext` that accepts a selection predicate. Just like before, it returns the next record in the file that is accepted by the selection predicate:

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

However, in the case of a sorted `DBFile`, this is an interesting and rather complicated function! What this function does is to look at the `CNF` `applyMe` that it is passed in order to see if it matches (in some way) the sort order used by the file. If the sort order matches the input `CNF` instance, then a binary search is used to make the search for matching records as fast as possible.

To do this, you need to write code to examine the `CNF` that you are passed, to try to build up an instance of the `OrderMaker` class that you can use to run a binary search on the sorted file. This search should consider all records that come *after* the current record in the file. I'll call the instance of the `OrderMaker` class that you build by looking at `applyMe` your "query" `OrderMaker`. This "query" `OrderMaker` can be used to see

whether a record in the sorted file is less than, greater than, or equal to the literal record passed into the `GetNext` function, with respect to both the sort order of the file, and the `CNF` that has been passed to `GetNext`.

The process for building up the "query" `OrderMaker` is a bit involved, so think about it carefully before you begin implementing it! At a high level, when building the "query" `OrderMaker`, you are trying to figure out whether or not your sorted file's sort order will help you find the first record in the file that is accepted by the `CNF` that you have been asked to evaluate. To do this, you loop through all of the ordering attributes in the `OrderMaker` that is used to sort your file, from first to last – the order is very important! For each sorting `OrderMaker` attribute, you look to see if the attribute is present in any of the subexpressions (disjunctions) in the `CNF` instance that you are asked to search on. If this attribute is in the `CNF` instance, *and* it is the only attribute present in its subexpression, *and* the `CNF` instance says that it is comparing that attribute with a literal value with an equality check, then you add it to the end of the "query" `OrderMaker` that you are constructing. As soon as you find any attribute in your file's sorting `OrderMaker` that is not present in the `CNF` instance that you are trying to evaluate, you have to stop building up your "query" `OrderMaker` because past that point, you can't make use of the sort order that your file provides. For example, say that the query CNF asks for an equality comparison with literal values for attributes 1, 2, and 5, and your file's sorting `OrderMaker` specifies a sort first on attribute 2, then on attribute 1, then on attribute 4, and then on attribute 5. You can build a "query" `OrderMaker` that specifies a comparison first on attribute 2, then on attribute 1, but past that, you can't use the file's sorted order. Why? If two records have the same value for attribute 2 and 1, your sorted file next orders them on attribute 4. But your input `CNF` does not care about attribute 4, so you can't use the sort order past that point!

Once you have constructed your "query" `OrderMaker`, use this object instance in conjunction with the appropriate comparison function to run a binary search on the file. This search should find the first record in the sorted file past the current record that is "equal to" the literal record that has been passed into the `GetNext` function, at least in terms of the attributes used to sort the file that are also useful for evaluating the `CNF` instance (note that if the "query" `OrderMaker` comes up empty – that is, it has no useful sorting attributes – then by definition, the first records that is "equal" to the literal record is the first record in the file and there is no reason to even do a binary search!).

Once you complete your binary search, there are two possibilities. First, you might have found no record that "equals" the literal record, as far as your "query" `OrderMaker` is concerned. In that case, `GetNext` returns a zero indicating that it was unable to fund a record that satisfied the `CNF` instance that was passed in. Or, your binary search might have found a matching record. In this case, you start scanning the file starting with that possible record, one record at-a-time. For each record, first evaluate the "query" `OrderMaker`, and then evaluate the `CNF` instance. If the "query" `OrderMaker` does not accept the record, then return a zero. If the "query" `OrderMaker` says that the file's current record matches the literal, but the `CNF` does not accept the record, then try the

next record. If the `CNF` also accepts the record, then return it to the caller. Return a zero as soon as you run off of the end of the file, or you find a record that is not accepted by the "query" `OrderMaker`.

One final point with respect to this version of `GetNext` over a sorted file. If this version of `GetNext` is called more than one time in succession without an intervening call to any other function, you can just assume that the caller has passed you the same parameters once again (in practice, it will never be the case that the caller switches parameters on you across calls without an intervening call to some other function; once they call `GetNext` with a given `CNF` instance and literal value, they will keep calling with the same parameters until they call `MoveFirst` or they perform some kind of write). In the case of repeated calls to `GetNext`, there is no reason to re-construct your "query" `OrderMaker`, nor is there is reason to re-do the binary search. You can just remember the "query" `OrderMaker` and pick up the search for matching records right from where you left off, using a sequential scan to find the next record in the file that is accepted by the `CNF` parameter.

All of this might seem quite complicated, but it really isn't if you keep in mind the final goal. A user of the `DBFile` class should be able to call `GetNext` repeatedly, and have all of the records matching the parameter `applyMe` returned in an efficient manner until they are exhausted. Your job as you try to implement the sorted version of the `DBFile` class is to facilitate this efficient search by using the file's sort order to help you, if it is at all possible.

Finally, we have the `Close` function, `Close` simply closes the file. The return value is a 1 on success and a zero on failure. `Close` needs to write out any meta-data that it will need to re-open the file (such as the sort order for a sorted file and the type of the file), and if the file is a sorted file, it should make sure that the internal `BigQ` instance has been merged with the data in the file.

```
int Close ();
```

What You Need To Turn In

Just like for assignment one, test code and turning instructions will be posted shortly!