Milestone 2: Implementing a Sorted File (Part 1)

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 parts, but only one submission & demo after you finish both parts. The first part involves implementing the `BigQ` class, which is 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! Try your best to eliminate any bugs.

The second part involves using the `BigQ` class to actually implement the sorted version of the `DBFile` class. Details will be released soon.

Part 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. You can use `pthreads` or `C++ standard threads (std::thread)` for our multi-threading. I have talked about std::thread in class, and if you want to use `pthreads` but are not familiar with it, 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 included in the archive P2-1/Pipe.tar. The functionality of the `Pipe` class is quite simple: it follows the producer & consumer patten as talked in class, and allows producer 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!

Test:

A test program is included in the archive P2-1/a2test.zip. You will need to follow the instructions in README file of a2test to run test.cc, then double check the output to make sure the records are sorted as you expected. Your demo will also be based on the test program.

Specifically, the test program includes a producer that reads records from a DBFile, a BigQ that sorts all the records, and a consumer that receives the sorted records then displays or saves them. The producer inserts records to the input pipe, BigQ reads records from the input pipe and inserts sorted records to the output pipe, while the consumer reads records from the output pipe. See the figure below:

Note: if you run test.out correctly, it might get stuck on large tables (e.g., lineitem) like below, as the BigQ has not been implemented, therefore the insertion to InputPipe will be blocked. It will not be stuck on small tables like nation.

```
[Info] In DBFile::Open (const char *f_path): Length of file ../../data/bin/lineitem.bin : 100
 producer: opened DBFile ../../data/bin/lineitem.bin
 consumer: removed 0 recs from the pipe
 consumer: 0 recs out of 0 recs in sorted order
```

Part Two: To Be Released Soon

What You Need To Demonstrate

To be released together with Part 2.

What You Need To Turn In

To be released together with Part 2.