COMP 530 Assignment 3: Sorting A File Using TPMMS

Due Thursday, Tuesday, February 23rd at 11:55 PM.

1. The Task

In this assignment, your task is to use all of the infrastructure that we have built so far in order to develop an external sort for a file. Basically, this means that you need to fill out the body of the `sort()` function in the file `Sorting.cc`. This function takes as input:

1. A run size that is the size, in number of pages, of each run that is written as the result of the sort phase of the TPMMS.
2. The `MyDB_TableReaderWriter` whose contents are to be sorted.
3. The `MyDB_TableReaderWriter` that the results of the sort are to be written into. The result of the sort should be appended to the end of this file.
4. The comparator function (a boolean lambda) that is to be used to power the sorting.
5. The two `MyDB_Record` objects that the comparator function operates over. The idea is that to compare two records, one loads the records into the two provided `MyDB_Record` objects, and then invokes the lambda to see if the contents of the first record are "less than" the contents second.

While sorting a file using TPMMS is a complicated thing to do, one of the fun things about this assignment is that all of the infrastructure we've built up will make it relatively easy. Nearly all of the code needed to run the TPMMS is in `Sorting.cc`—my `Sorting.cc` is only 203 lines long.

2. Details, Details, Details

I'm actually providing you with a lot of infrastructure that you can use to make this task easier, including my implementation of Assignment 2 and much of my `Sorting.cc` (the one thing that is missing is the contents of the `sort ()` function).

2.1 Sorting the Contents of Individual Pages

One of the things that I've provided you with is my my implementation of `MyDB_PageReaderWriter`, which provides a sort function that sorts an individual page. This function takes as input a comparator function along with two `MyDB_Record` objects. It then sorts the contents of the page using the comparator function and the two `MyDB_Record` objects, and then outputs a smart pointer to a new `MyDB_PageReaderWriter` whose contents are identical to the original one, except that they are sorted. The data in the page referred to by this `MyDB_PageReaderWriter` are backed by an anonymous page.

2.2 Additional Iterators

The provided `MyDB_PageReaderWriter` and `MyDB_TableReaderWriter` classes have a number of additional iterators (beyond those from Assignment 2) that can make the task of sorting a file easier. First, there are the various "alternative" iterators. Unlike the iterators you built for Assignment 2, the "alternative" iterators are not built over specific record objects. Instead, these iterators have an `advance ()`/`getCurrent ()` interface, where `getCurrent ()` accepts the record object to iterate into. This style of iterator is very useful for the current assignment because the current contents of the iterator can be loaded into a specific records object using `getCurrent ()`, and then a comparator built over the that record object can be used to compare with another record.

There is also an alternative iterator available that accepts a vector of `MyDB_PageReaderWriters` as input, and then iterates over all of the pages in the vector. This is very useful for sorting a large file because you can create a bunch of anonymous pages and then easily iterate over all of the records in all of those pages.

2.3 `mergeIntoList`

I've also provided you with an implementation of a function called `mergeIntoList ()` which accepts two alternative iterators as well as a comparator, and then merges the contents of the two iterators into a list of anonymous pages. This list is then returned as a vector of `MyDB_PageReaderWriter` objects. This is very useful because in the sort pass of a TPMMS, you can perform a merge sort by first sorting single pages (using the `sort ()` method described in 2.1), and then merging those pages into larger and larger lists of pages whose contents are all sorted using the `mergeIntoList ()` function.

2.4 `mergeIntoFile`

Finally, I've provided you with an implementation of a function called `mergeIntoFile ()` which accepts a list of alliterative iterators, and then merges the contents of all of those iterators into a single file. This is very useful because in a TPMMS, you can first sort the input file into runs (where each run is represented as a list of anonymous pages), and then merge all of those runs into a single file using `mergeIntoFile ()`.

2.5 Outline of the Algorithm

You are free to implement the `sort ()` function however you want (as long as you implement it using a TPMMS whose run size is as indicated), but it is possible to write it in just 50 to 100 lines of code if you follow the approach outlined above and use the code that I've provided for you. First, go though the input file and sort the input file into runs. This is done as outlined in 2.3, where you repeatedly pass through the subset of the pages in the input file that you want to sort, merging into sorted lists of pages of larger and larger size, until you have a single list of pages (stored in a vector) that

corresponds to the sorted run.

After the file is sorted into runs, then you use `mergeIntoFile ()` to merge each of those runs into the single output file.

2.6 My Code, Your Code, and the Honor Code

As in A2 (where I gave you the solution to A1) in A3 I am giving out my solution to A2 and A1. Simply stated, you are not allowed to distribute my code to **anyone** who is not taking the class right now… **ever**! Doing this constitutes a violation of the honor code. I'm particularly concerned about my code falling into the hands of students who might take the class in the future. Thus, a hard and fast prohibition on distributing my source code.

2.7 Testing And Grading

Again, we're using `Qunit`. Again, I have written a few `Qunit` tests. When you build your project using the `SCons` build tool these tests will be complied and an executable will be produced.

Again, if you can make it through my test cases, it is likely that you've worked through the majority of the bugs in your buffer manager, and you've got most of the functionality done. However, you'll probably want to create a few additional test cases of your own—some simpler ones that you can use early on as you develop your code, and some additional, nasty ones just to make sure that everything is working.

When you turn in your code, and it's time for us to grade, we'll run the test cases I've suppled, as well as several others that won't be made public until after the turnin. You'll be graded on your code's success in passing all of the test cases, though we revere the right to browse through your code and take off additional points if it appears you are missing some functionality or have somehow hacked something in a sketchy sort of way. You won't be graded on style and comments. However, I strongly encourage you to take this opportunity to put your best software engineering practices to use.

2.8 Project Difficulty

All of the code that you need to write can be put into the single function sort (). I'd expect that you will write only 50 to 100 lines of code to complete this assignment.

2.9 Notes Regarding Performance

My implementation takes around 10 seconds to run the provided test cases on my Mac, and around 7 or 8 seconds to run the test cases using an Amazon AWS machine. That is actually pretty slow, considering that the most expensive task is sorting only 320,000 records (around 50MB of records). It actually turns out that the majority of this cost is associated with parsing records using `fromBinary ()` and the methods that

`fromBinary ()` calls. That is, every time that an iterator needs to read a record from a page, it is parsed, and it is this CPU cost that dominates the overall sorting time. If we really wanted to speed this up, there are things that we could do. For example, the current design requires re-parsing a record every time it is processed, rather than loading the record into an object once, and keeping it in the object, re-using the object as long as possible. However, such performance tuning is beyond the scope of our project, and would change our design. Still, it is interesting (and instructive) that the most expensive thing the database is doing is performing the relatively mundane task of parsing records.

3. Getting Started

The instructions this time around for getting started are much the same as the instructions last time around, so I won't reproduce them here.

4. Turnin

Simply zip up all of your source code and then turn it in on OwlSpace (make sure to archive into the zip format, and not some other archiving format. If you choose to use something else, we'll take off a few points!). Please name your archive A3.zip. Please do not change the original directory structure, except for perhaps adding some new files. The root should be a directory called A3, with two subdirectories Build and Main. And so on.

And remember, **to get *any* credit on A3, your code must compile and run on Clear**. That way, we have a common environment for grading and we don't have to spend time getting your code to compile.

Finally, and this is important: **include a README file in the root of your project with any important information**, including the names of the one or two people who worked on the project.

And also: if you work with a partner, **only turn in one copy of your source**. Otherwise, we'll possibly grade your submission twice, and end up getting quite annoyed.