UNIVERSITY OF WISCONSIN–MADISON
COMPUTER SCIENCES DEPARTMENT

CS 564: DATABASE MANAGEMENT SYSTEMS

Assignment 2: External Sorting
Due: March 8

Instructor: Raghu Ramakrishnan

1 Introduction

In this assignment, you will implement the general multi-way external sorting algorithm that we discussed in class. You will be given the code for the lower layers. In particular, you will work with the interface to the (supplied versions of the) buffer manager, the heap file class and the HFPage class.

You will carry out this assignment in teams of two, with the same partner as in the previous assignment.

You should begin by reading the chapters on Sorting, and on Disks and Files. HTML documentation is available for Minibase, which you can read using Netscape. There is a link to the Minibase home page in the CS564 home page. Note that the sorting code in Minibase is more sophisticated than this assignment requires; so follow the description of sorting in this handout, rather than the HTML description of sorting in Minibase.

Be sure to follow the error protocol described in:

http://www.cs.wisc.edu/coral/minibase/system/error.html

2 The Interfaces that You Will Call

You will call the buffer manager. Although the supplied code supports additional methods, you will only call the methods that you were asked to implement in the buffer manager assignment.

You will use the heap file class. The interface for this class supports the creation and deletion of files of unordered pages, and allows you to scan the file, page by page.

You will use the HFPage class. After reading a page into a buffer pool frame (using the pinpage call), you should cast it into an object of the HFPage class. This will allow you to operate on the records on the page through the methods of HFPage.

3 What You Have to Implement

You have to implement the following:

```
class Sort {
public:
```

```
Sort(
char*    inFile,  // Name of unsorted heapfile.

char*    outFile,      // Name of sorted heapfile.

int      len_in,     // Number of fields in input records.

AttrType  in[], // Array containing field types of input records.
                // i.e. index of in[] ranges from 0 to (len_in - 1)

short    str_sizes[],   // Array containing field sizes of input records.

int      fld_no,        // The number of the field to sort on.
                              // fld_no ranges from 0 to (len_in - 1).

TupleOrder  sort_order,   // ASCENDING, DESCENDING

int         amt_of_buf,   // Number of buffer pages available for sorting.

Status&     s // Status of call
);

~Sort(); // destructor

};
```

4 Internal Design

You have to sort a file of fixed size records, stored in a heap file on disk, using a given amount of main memory. Let the amount of memory that is given be SORTPGNUM pages.

Pass 1: To simplify matters, we will assume that the first phase of sorting is performed by dynamically allocating this amount of main memory. Read the data from the unsorted heap file, one record at a time, into this 'sorting area.' Do this until the sorting area is full (i.e., you've used up the available memory).

Sort this set of records by calling the **qsort** library function. Your call to **qsort** will take a pointer to your sorting area, the number of records contained therein, the size of each record, and a pointer to your comparison function. (The ordering function that you pass to qsort should call the CompareTupleWithTuple function that we supply in the tuple_utils module.) **qsort** does a generalized quick-sort, calling your comparison function to determine the order of any two records it needs to compare. Your comparison function should take two pointers to Tuples, and return -1, 0, or 1 in the manner of **strcmp**. See the man page on **qsort** for more information.

After calling **qsort**, write the sorted run out to a temporary heap file.

1. Use one temporary file per run.

2. To write the run out, simply create a new HeapFile object with a unique name, and repeatedly call the insertRecord method. The fact that all the records are the same size means that the HeapFile will maintain your sorted order. You will not need to keep track of the pages that are pinned and unpinned, but the idea is that only one page needs to be dirtied at a time.

3. Destroy the temporary HeapFile object when you have finished writing it out. This closes the file, and unpins the file's header page.

4. Remember the name you use for each temporary file. You will need it to re-open the file during the next pass.

Passes 2 and above: In the *merging* passes, you will primarily use frames in the buffer pool, rather than additional main memory, unlike Pass 1. Nonetheless, to reflect the fact that you are allowed to use SORTPGNUM pages of main memory, you should use SORTPGNUM-1 buffer frames as input buffers, and one buffer frame as an output buffer. The simplest way to do this is to open scans on SORTPGNUM-1 (heap files containing) runs from the previous pass. Heap file scans pin only one page at a time; as the scan 'moves off' a page, the page is unpinned by the scan. Thus, as you scan the input runs, you have at most one page pinned per run. As for the output, you should create a new temporary file, and insert records into them from the input runs, 'merging' the input runs. Again, as above, the HeapFile will preserve your order and will only use one page at a time.

The temporary files used to hold input runs should be deleted (with the deleteFile method) after the pass in which they are merged. The sorting is complete when you are left with just one file.

An important note on memory usage: We have deliberately simplified this assignment by providing you the HeapFile class. In the process, we have fudged the issue of memory usage a little.

A HeapFile object keeps one header page pinned for the duration of its existence, in addition to the page that holds the current record you are reading or writing. So the amount of buffer-pool memory you will actually use to read or write using a HeapFile is two pages at a time, rather than one page at a time, as we have assumed. In addition, the interface to read a record from a HeapFile requires you to copy the record to some extra memory—this also slightly increases the actual memory you will use during the merge passes.

The point is that your sorting code would be able to *actually* use only SORTPGNUM pages, simply by replacing HeapFile with a more space-efficient record-file class.

5 Where to Find Makefiles, Code, etc.

Copy all the files from $\sim cs564-1/spring96/assign2/src$ into your own directory. The files are:

1. Makefile: A sample makefile to compile your project. You will have to set up any dependencies by editing this file. You can also design your own Makefile.
2. Sort.h: Specifications for the class Sort. You have to implement these specifications as part of assignment.
3. sort_driver.C: Sort test driver program.

Other useful include files:heapfile.h , scan.h , tuple_utils.h, tuple.h , hfpage.h , buf.h and new_error.h are found in $\sim cs564-1/spring96/assign2/include$.

6 What to Turn In, and When

You should turn in copies of your code together with copies of the output produced by running the tests provided by the TAs. The assignment is due at 5PM on March 8. The solution will be made public after that time, and solutions turned in after that time will not receive any credit. So be sure to turn in whatever you have, even if it is not working fully, at that time.

I emphasize that late submissions will not receive any credit. Computers—and life!—being what they are, expect everything to take longer than you expect, even taking this expectation into account. So start early, and plan on getting things done well before the due date. Nothing short of a nuclear explosion (in the CS building, not the South Pacific) constitutes a valid reason for an extension.