Comp 115 - Spring 2017 - Class Project Part B

Due May 10th, 9:00pm

Overview: The Heap File Layer

In Part A of the class project, you implemented one component of a DBMS: the buffer manager. The buffer manager creates and deletes files, opens and closes files, and allocates new blocks when needed.

In Part B, you will implement another DBMS component: the heap file layer. The heap file layer enables an application to scan a file sequentially, or to request a specific block from a file. It supports these operations by making calls to the buffer manager, so that it doesn't need to worry about whether the blocks are stored in memory or on disk.

As a reminder, Part C of the class project is optional and can be completed for extra credit. It will involve adding an index file layer to the DBMS.

Timeline

Part B of the class project is due on May 10th at 9:00pm. This is the time at which the course's final exam block ends. You will not take a final exam during this block.

If you need to modify your Part A implementation between the Part A and Part B due dates, the modified Part A implementation is also due on May 10th at 9:00pm. The changes should be documented in the accompanying short project description document of Part B.

Late Policy

No late submissions at this point, because it is the end of the semester! A *few hours of lateness is ok*, so you make sure that *your submission is indeed your best work*.

What to Submit

Your submission should include the following components:

- An implementation of the Heap File Layer API (described below)
- A suite of unit tests for the API functions
- A README that contains a short description of your design choices and implementation strategy

How To Submit

Submit your assignment with this Provide command:

```
provide comp115 115_projectB <code archive>.tar <report document>.pdf
```

Key Concepts and Necessary Background

Most of the information that you need to complete the class project is found chapters 8 and 9 of the course textbook, *Database Management Systems* (Ramakrishnan & Gehrke). For Part B, pay particular attention to sections 8.2, 9.5, 9.6, and 9.7.

We have also covered this information in class. Slides that are relevant to Part B can be found in the lecture slides for Class 9 (File Organization & Introduction to Indexing) and Class 11 (Storage Layer).

Like Part A, Part B requires basic familiarity with the C programming language. The programming techniques that you used in Part A should be sufficient to handle Part B. There are many free online resources that you can use to brush up on these techniques; here are just a few:

W3 Schools C Tutorial (look at the sections on pointers and file I/O)
http://www.w3schools.in/c-tutorial

Fresh 2 Refresh C Tutorial (look at the sections on dynamic memory allocation and file handling)
http://fresh2refresh.com/c-programming/c-file-handling/

The Heap File Layer API

To help you get started, we will provide you with a high-level API for the heap file layer. The API includes a signature for each heap file layer function, along with a description of what the function should do. Your job is to fill in the bodies of these functions, and then write a test suite to demonstrate that these functions work as expected.

While we are providing you with an initial framework, you will have to make several choices about the design of the heap file layer. Here are some aspects of the implementation to consider:

- **Page Organization**: The page organization ideas in Part B are mainly the same as in Part A. The only difference is that the buffer manager doesn't care how pages are organized internally, whereas the heap file layer needs to know how pages are organized so that it can retrieve individual records from them.
  As a reminder, the buffer manager moves one fixed-size page of a file at a time between an external storage device and main memory. You can think of a page as a collection of slots, each of which is either empty or contains a record. There are several different ways to organize slots on a page; section 9.6 of the textbook describes some of them, and we discussed a standard approach called *N-ary storage model* (NSM) in class. For extra credit, you can implement a more advanced strategy called *Partition Attributes Across* (PAX) that we also discussed in class.

- **Tuple Organization:** The heap file layer API provides access to one tuple or record at a time (see the "HFL_get_first_rec", "HFL_get_next_rec", and "HFL_get_this_rec" functions below). Each tuple is a fixed-width record, and each of its fields also has a fixed width. You will need to choose a strategy for storing information about the widths of tuples in a given file, and for using

this information to retrieve a tuple with a given ID. The first few lecture slides from Class 9 should be helpful in this regard.

Here is the heap file layer API that you will implement:

```
/* Example of how to represent a fixed-width record as a C struct. */
typedef struct _record
{
int attribute1;
int attribute2;
int attribute3;
char attribute4;
char attribute5;
char attribute6;
char attribute7;
} record;

/* Representation of a record ID. */
typedef int recordID;

/* Representation of a scan operation for a given file (details below). */
typedef int scanDesc;

/* The operations below call the Buffer Manager API to retrieve pages of a file.                    */
/* You can use the same fileDesc representation as the Buffer Manager or a different one. */

// Initialize state used by the Heap File Layer.
void HFL_init();

// Create a file with the given filename.
errCode HFL_create_file(char* filename);

// Open an existing file with the given filename and return a file descriptor.
fileDesc HFL_open_file(char* filename);

// Close an open file with the given filename.
errCode HFL_close_file(fileDesc fd);

// Insert a record into the file identified by fd.
recordID HFL_insert_rec(fileDesc fd, record* rec);


// Delete the record with the given ID from the file identified by fd.
errCode HFL_delete_rec(fileDesc fd, recordID rid);

// Make rec point to the first record in file fd.
```

```
errCode HFL_get_first_rec(fileDesc fd, record** rec);

// Advance rec to the next record in fd.
errCode HFL_get_next_rec(fileDesc fd, record** rec);

// Make rec point to the record in fd with ID rec_id.
errCode HFL_get_this_rec(fileDesc fd, recordID rid, record** rec);

// Return a scan_id for file fd. This is the primary operation of the Heap File Layer. At any given point in
// time, there could be one or more scans of the same relation in progress, so you need to have multiple
// scan_ids to keep track of multiple scan iterators.
scanDesc HFL_open_file_scan(fileDesc fd);

// Make rec point to the next record in the scan identified by scan_id.
errCode HFL_find_next_rec(scanDesc scan_id, record** rec);

// Close the file scan abstraction identified by scan_id.
errCode HFL_close_file_scan(scanDesc scan_id);

// Print an error message.
errCode HFL_print_error();
```

We have created a header file that contains these signatures; you can use it as the starting point for your implementation. The header file will be available on the project website.