CS44800 Project 2
The RDBMS Lower-Level Layers
Spring 2016

Due: Tuesday 16 February, 2016, 11:59PM
(There will be a 10% penalty for each late day up to four days. The assignment will not be accepted afterwards.)

Notes:

- This project should be carried out in groups of two. Choose a partner as soon as possible.
- This project will be based on Minibase, a small relational DBMS, structured into several layers. In this project, you are required to implement parts of the lower-level layers: BM - the buffer manager layer and HFPage - the heap file layer. Layers should be tested and turned in separately.

PART 1: Buffer Management

1. Introduction

You are required to implement a simplified version of the Buffer Manager layer without support for concurrency control or recovery. You will be given the code for the lower layer: the Disk Space Manager.
You should begin by reading the chapter on Disks and Files to get an overview of buffer management. In addition, HTML documentation is available for Minibase. There is a link to the Minibase home page **here**. In particular, you should read the description of the DB class, which you will call extensively in this assignment. The Java documentation for the *diskmgr* package can be found **here**. You should also read the code under diskmgr/ carefully to learn how the package is declared and how exceptions are handled in Minibase.

2. The Buffer Manager Interface

The simplified Buffer Manager interface that you will implement in this assignment allows a client (a higher level program that calls the Buffer Manager) to allocate/de-allocate pages on disk, to bring a disk page into the buffer pool and pin it, and to unpin a page in the buffer pool. The methods that you have to implement are described below:

```
public class BufMgr {
/**
* Create the BufMgr object.
* Allocate pages (frames) for the buffer pool in main memory and
```

```
* make the buffer manage aware that the replacement policy is
* specified by replacerArg (e.g., LH, Clock, LRU, MRU, LFU, etc.).
*
* @param numbufs number of buffers in the buffer pool
* @param lookAheadSize: Please ignore this parameter
* @param replacementPolicy Name of the replacement policy, that parameter will be set to "LFU" (you
can safely ignore this parameter as you will implement only one policy)
*/
public BufMgr(int numbufs, int lookAheadSize, String replacementPolicy) {};

/**
* Pin a page.
* First check if this page is already in the buffer pool.
* If it is, increment the pin_count and return a pointer to this
* page.
* If the pin_count was 0 before the call, the page was a
* replacement candidate, but is no longer a candidate.
* If the page is not in the pool, choose a frame (from the
* set of replacement candidates) to hold this page, read the
* page (using the appropriate method from {\em diskmgr} package) and pin it.
* Also, must write out the old page in chosen frame if it is dirty
* before reading new page.__ (You can assume that emptyPage==false for
* this assignment.)
*
* @param pageno page number in the Minibase.
* @param page the pointer point to the page.
* @param emptyPage true (empty page); false (non-empty page)
*/
public void pinPage(PageId pageno, Page page, boolean emptyPage) {};

/**
* Unpin a page specified by a pageId.
* This method should be called with dirty==true if the client has
* modified the page.
* If so, this call should set the dirty bit
* for this frame.
* Further, if pin_count>0, this method should
* decrement it.
*If pin_count=0 before this call, throw an exception
* to report error.
*(For testing purposes, we ask you to throw
* an exception named PageUnpinnedException in case of error.)
*
* @param pageno page number in the Minibase.
* @param dirty the dirty bit of the frame
*/
public void unpinPage(PageId pageno, boolean dirty) {};
```

```
/**
* Allocate new pages.
* Call DB object to allocate a run of new pages and
* find a frame in the buffer pool for the first page
* and pin it. (This call allows a client of the Buffer Manager
* to allocate pages on disk.) If buffer is full, i.e., you
* can't find a frame for the first page, ask DB to deallocate
* all these pages, and return null.
*
* @param firstpage the address of the first page.
* @param howmany total number of allocated new pages.
*
* @return the first page id of the new pages.__ null, if error.
*/
public PageId newPage(Page firstpage, int howmany) {};

/**
* This method should be called to delete a page that is on disk.
* This routine must call the method in diskmgr package to
* deallocate the page.
*
* @param globalPageId the page number in the data base.
*/
public void freePage(PageId globalPageId) {};

/**
* Used to flush a particular page of the buffer pool to disk.
* This method calls the write_page method of the diskmgr package.
*
* @param pageid the page number in the database.
*/
public void flushPage(PageId pageid) {};

/**
* Used to flush all dirty pages in the buffer pool to disk
*
*/
public void flushAllPages() {};

/**
* Returns the total number of buffer frames.
*/
public int getNumBuffers() {}

/**
* Returns the total number of unpinned buffer frames.
*/
public int getNumUnpinned() {}
```

};

3. Internal Design

The *buffer pool* is a collection of *frames* (page-sized sequence of main memory bytes) that is managed by the Buffer Manager. Conceptually, it should be stored as an array bufPool[numbuf] of Page objects. However, due to the limitation of the Java language, it is not feasible to declare an array of Page objects and later on writing strings (or other primitive values) to the defined Page. To get around the problem, we have defined our Page as an array of bytes and deal with the buffer pool at the byte array level. Note that the size of the Minibase pages is defined in the interface *GlobalConst* of the *global* package. Before jumping into coding, please make sure that you understand how the Page object is defined and manipulated in Java Minibase.
In addition, you should maintain an array bufDescr[numbuf] of *descriptors*, one per frame. Each descriptor is a record with the following fields:

page number, pin_count, dirtybit.

The *pin_count* field is an integer, *page number* is a PageId object, and *dirtybit* is a boolean. This describes the page that is stored in the corresponding frame. A page is identified by a *page number* that is generated by the DB class when the page is allocated, and is unique over all pages in the database. The PageId type is defined as an integer type in *global* package.
A simple *hash table* should be used to figure out what frame a given disk page occupies. You should implement your own hash table class and not use existing HashTable Java classes. The hash table should be implemented (entirely in main memory) by using an array of pointers to lists of *<page number, frame number >* pairs. The array is called the *directory* and each list of pairs is called a *bucket*. Given a *page number*, you should apply a *hash function* to find the directory entry pointing to the bucket that contains the frame number for this page, if the page is in the buffer pool. If you search the bucket and do not find a pair containing this page number, the page is not in the pool. If you find such a pair, it will tell you the frame in which the page resides.

The hash function must distribute values in the domain of the search field uniformly over the collection of buckets. If we have HTSIZE buckets, numbered 0 through M-1, a hash function *h* of the form *h*(*value*) = (*a*value+b*) *mod HTSIZE* works well in practice. HTSIZE should be chosen to be a prime number.

When a page is requested, the buffer manager should do the following:
1. Check the buffer pool (by using the hash table) to see if it contains the requested page. If the page is not in the pool, it should be brought in as follows:

    (a) Choose an unused frame. If all frames are occupied (i.e., the buffer pool is full), choose a frame for replacement using the Least-Frequently-Used policy as described below.
    (b) If the frame chosen for replacement is dirty, *flush* it (i.e., write out the page that it contains to disk, using the appropriate DB class method).

(c) Read the requested page (again, by calling the DB class) into the frame chosen for replacement; the *pin_count* and *dirtybit* for the frame should be initialized to 0 and FALSE, respectively.
(d) Delete the entry for the old page from the Buffer Manager's hash table and insert an entry for the new page. Also, update the entry for this frame in the bufDescr array to reflect these changes.

2. *Pin* the requested page by incrementing the *pin_count* in the descriptor for this frame and return a pointer to the page to the requestor.

The Least Frequently Used (LFU) Replacement Policy

The LFU replacement policy keeps track of the frequency of past references of the buffered pages. In particular, LFU keeps track of the number of times a page is referenced in memory. When the buffer pool is full and requires more room to host an unbuffered page, LFU will select the page with the lowest reference frequency as a victim (i.e., will be evicted from the buffer pool to allow buffering a new page). Notice that when a page, say P, is victimized at Time t and brought later into the buffer pool at Time t + i, its reference frequency does not capture the reference frequency before Time t.

4. Error protocol

Though the Throwable class in Java contains a snapshot of the execution stack of its thread at the time it was created and also a message string that gives more information about the error (exception), we have decided to maintain a copy of our own stack to have more control over the error handling.
We provide the *chainexception* package to handle the Minibase exception stack. Every exception created in your *bufmgr* package should extend the ChainException class. The exceptions are thrown according to the following rule:
Error caused by an exception caused by another layer:

```
For example: (when try to pin page from diskmgr layer)
try {
   Minibase.BufferManager.pinPage(pageId, page, true);
}
catch (Exception e) {
        throw new DiskMgrException(e, "DB.java: pinPage() failed");
}
```

Error not caused by exceptions of others, but needs to be acknowledged:

```
For example: (when try to unpin a page that is not pinned)
if (pin\_count == 0) {
        throw new PageUnpinnedException (null, "BUFMGR: PAGE_NOT_PINNED.");
```

}

Basically, the ChainException class keeps track of all the exceptions thrown across the different layers. Any exceptions that you decide to throw in your *bufmgr* package should extend the ChainExceptionclass.
For testing purposes, we ask you to throw the following exceptions in case of error (use the exact same name, please):

BufferPoolExceededException
Throw a BufferPoolExceededException when an attempt is made to pin a page to the buffer pool with no unpinned frame left.

PagePinnedException
Throw a PagePinnedException when an attempt is made to free a page that is still pinned.

HashEntryNotFoundException
Throw a HashEntryNotFoundException when the page specified by PageId is not found in the buffer pool.

Feel free to throw other new exceptions as you see fit. But make sure that you follow the error protocol when you catch an exception. Also, think carefully about what else you need to do in the *catch* phase. Sometimes you do need to unroll the previous operations when failure happens.

5. Where to Find Makefiles, Code, etc.

Please copy this file (proj21.zip) into your own Unix local directory. The directory lib contains the jar file that implements the disk manager, you will use this file to compile your code. The content of the directory src is:

under *bufmgr* directory
You should make all your code for *bufmgr* package implemented under this directory.

under *diskmgr* directory
You should study the code for *diskmgr* package carefully before you implement the *bufmgr* package. Please refer to the java documentation of the packages.

under *tests* directory
TestDriver.java, *BMTest.java*: Buffer manager test driver program. Note also that you may need to change the test file *BMTest.java* to select the replacement policy you will implement.

We provide a sample *Makefile* to compile your project. You will have to set up any dependencies by editing this file. You may need to design your own Makefile. Whatever you do, please make sure that the classpath is correct.

You can find other useful information by reading the java documentation on other packages, such as the *global* and *diskmgr* package.

6. What to Turn in

You should turn in copies of your code together with the Makefile and a readme file. All files need to be zipped in a file named: **your_career_login1_your_career_login2_bufmgr.zip.**
In the readme file, put the name of your group members, and the feature you would like the TAs to know. The TAs should be able to compile/run your program using make. You do not need to include the library file and other files provided.
The directory structure of your zip file should be identical to the directory structure of the provided zip file (i.e., having the directory src, the Makefile, ...), except the top-level name (should be your career login above). Your grade may be deduced 5% off if you do not follow this. In the readme file, make sure to state the roles of each member in the group (i.e., who did what). Only one member should submit the project on behalf of the other member (please indicate in the read me file who is the one submitting).

PART 2: Heap Files

1. Introduction

In this part, you will implement the Heap file layer. You will be given the documentation for the lower layers (Buffer Manager and Disk Space Manager), as well as the documents for managing records on a Heap file page. You can find the package index for the above here.

This assignment has three parts. You have to do the following:

1.Familiarize yourself with the Heap file, HFPage, Buffer Manager and Disk Space Manager interfaces.

2.Implement the Heap file class. You can ignore concurrency control and recovery issues, and concentrate on implementing the interface routines. You should deal with free space intelligently, using either a linked list or page directory to identify pages with room for records. When a record is deleted, you must update your information about available free space, and when a record is inserted, you must try to use free space on existing pages before allocating a new page.
3.Run the tests provided.

The major methods of HeapFile.java that you will implement include:

public **HeapFile**(String name)

public RID **insertRecord**(byte[] record)

```
public Tuple getRecord(RID rid)
public void updateRecord(RID rid, Tuple newRecord)
public void deleteRecord(RID rid)

public int getRecCnt() //get number of records in the file
public HeapScan openScan()
```

2. Available Documentation

You should begin by reading the chapter on Disks and Files, to get an overview of the HF layer and buffer management.

3. Complexity Requirements

1. **insertRecord**

insertRecord needs to be done in at most O(log(n)) time, which means using a linked list without any additional data structures to manage the pages of free space is not enough. You can assume the size of a record will not exceed the size of a page. Different data structures and different inserting algorithms may have different pros and cons. You will be asked your design and advantages/disadvantages of your design when grading your project.

2. **updateRecord, deleteRecord**

these two operations need to be done in at most O(log(n)) time as well. You need to consider the case when updating a record, the updated record may not be stored in its original page.

3. **getRecCnt** need to be done in O(1).

4. Classes to Familiarize Yourself with First

There are three main packages with which you should familiarize yourself: *heap, bufmgr, diskmgr*. Note that part of the *heap* package contains implementation for HFPage. The java documentation of HFPageis provided to you. A Heap file is seen as a collection of records. Internally, records are stored on a collection of HFPage objects.

5. Compiling Your Code and Running the Tests

Copy this file (proj22.zip) to your own local directory and study them carefully. The files are:

- In directory src/heap: Again, the java documentation for packpage *bufmgr, diskmgr* and *heap* are online.

Note that you DO NOT have to throw the same exceptions as documented for the heap package. However, for testing purposes, we DO ask you to name one of your exceptions InvalidUpdateExceptionto signal any illegal operations on the record. In other error situations, you should throw exceptions as you see fit following the error protocol introduced in the Buffer Manager Assignment.

- In directory src/tests: This directory contains the source code of the test. Make the required changes.

- In directory lib:*heapAssign.jar:* this is a library that has the implementation of the Disk Manager and Buffer Manager layers. As you know, you will develop the Buffer Manager layer in

the first part of this project. We are providing this library to help you start working with the second part as soon as possible and also to help you test your code comparing the behavior of your code with the one obtained used the library. The final goal is to make the system work without using this library and using your code instead.

6. What to Turn in

Part 2 & 3 should be compiled together. See Part 3 to know what to turn in.

PART 3: Heap Scans

After completing the buffer manager and heap file layers, you will be able to implement a basic scan of a given file. A heap scan is simply an iterator that traverses the file's directory and data pages to return all records. The major methods of **HeapScan.java** that you will implement include the following (please see the javadoc for detailed descriptions):

protected **HeapScan**(HeapFile hf)
protected void **finalize**() throws Throwable
public void **close**()

public boolean **hasNext**()
public Tuple **getNext**(RID rid)

Internally, each heap scan should consist of (at least) the following:

Directory Position

The current directory page and/or entry in the scan.

Record Position

The current data page and/or RID in the scan.

The Tuple class is the wrapper of an array of data. Internally it should contain a declaration byte[] data and other methods called by the test driver.

Your implementation should have at most one directory page and/or at most one data page pinned at any given time.

What to Turn in

Part 2 & 3 should be compiled together. You should turn in copies of your code together with the Makefile and a readme file. All need to be zipped in a file named:**your_career_login1_your_career_login2_heapfile.zip.**

In the readme file, put the name of your group members, and the feature you would like me to know. I should be able to compile/run your program using make. You don't need to include the library file and other files provided. In the readme file, you should also include the roles of each group member in the project, i.e., who implemented what.

The directory structure of your zip file should be identical to the directory structure of the provided zip file (i.e., having the directory src, the Makefile, ...), except the top-level name (should be your pucclogin above). Your grade may be deduced 5% off if you don’t follow this. Only one member is allowed to submit Part 2 & 3 (the same person who submitted Part 1).

You should upload all your zip files using Blackboard.