CS44800 Project 2
Buffer Manager
Spring 2019

Due: **February 25, 2019, 11:59PM**

Total Points: **9 points**

Learning Objectives

1. Understand the design and architecture of lower-level database systems components.
2. Implement the buffer manager component in a simplified database system.
3. Implement a replacement policy for the buffer management layer.

Project Description

The goal of this project is to implement a simplified version of a Buffer Manager layer, without support of concurrency control or recovery. For this assignment you will be given the code for the lower layer.

This project is based on Minibase, a small relational DBMS, structured in several layers in order to allow a modular, abstract approach to implementing a DBMS. This approach allows for ease of implementation, as no layer relies on any specific implementation of any lower layer – they only have to make the appropriate calls to functions defined in an API.

In this project, you will be implementing one of the lower levels of a DBMS: the Buffer Manager level that is responsible for bringing pages into and out of main memory from the disk. The actual disk access functionality is already implemented for you in the Disk Manager layer of Minibase - the source files for the Disk Manager are included in the *diskmgr* package if you would like to investigate them, but this is not necessary in order to complete this project. Do not make any changes to the *diskmgr* package.

You should begin by reading **Chapter 16 (Disk Storage, Basic File Structure, Hashing, and Modern Storage Architecture)** on the Textbook (**Fundamentals of Database Systems by Elmasri & Navathe, 7th Edition**) to get an overview of the buffer manager. This material will be cover in class and during the PSO sessions. In addition, a conceptual overview of Minibase is available here.

This handout first provides a description of the Minibase components you will be working with. The actual requirements you need to complete are described in the “Buffer Manager Interface” section of this handout.

Buffer Manager Components

The Buffer Manager consists of several data structures that are used to manage and track pages in-memory:

Buffer Pool: The Frame Pool is an array of Page objects, defined in the skeleton code as *bufFrames*. This array consists of *n* elements, where *n* is a parameter defined at the creation of the database (in this case, 100). This is where the contents of pages are actually stored while they are resident in memory. A Page object is essentially an array of bytes. Minibase provides methods to read and write datatypes to the pages – you should not have to change the actual contents of any Page within your Buffer Manager implementation, but you can see these methods used in the test cases.

It's important to note about Pages that the Page object in-memory is not the same as a Page stored on disk. It is only a container for the byte data stored in that page. All pages are identified by a PageId – a globally unique value generated by Minibase when pages are allocated. This PageId is what Minibase uses to actually access pages from the disk – a call to the Disk Manager is made to read or write the page with the given PageId, and then the DiskManager will either read or write data to the provided Page object.

Frame Descriptors: An array of FrameDescriptor objects, defined in the skeleton code as *bufDescr*. This is also an array consisting of *n* elements, where each element in *bufDescr* corresponds to an element in *bufFrames*. A FrameDescriptor tracks information about the contents of each frame: the ID of the Page stored in the frame (-1 if no Page is stored in that frame), the PinCount of a frame, and the dirty bit (true if the page has been written to since being brought into memory, false otherwise).

Hash Table: A Java HashTable with key=PageID and value=FrameNumber, defined in the skeleton code as *hashTable*. This hash table associates a Page with the Frame that the page is stored in. If the page is not present in memory, then the hash table should not contain an entry for the Page. This HashTable allows the Buffer Manager to quickly determine whether a page needs to be brought into memory or which frame that page is stored in otherwise.

Replacement Policy: The Buffer Manager will need to implement a replacement policy in order to manage bringing pages into and out of memory. If the Buffer Pool is full when a page needs to be brought into memory, the Replacement Policy will be used to select the appropriate unpinned page to evict. In this project, we ask you to implement a FIFO replacement policy, and you may use the *fifo* Queue object to manage this.

The Disk Manager

The Disk Manager provides methods that handle allocation of new pages on disk, reading pages, and writing pages. The Disk Manager stores certain data – such as the map of allocated/unallocated space on disk – in pages itself. It will use the Buffer Manager to manage these pages and bring them into and out of memory as necessary.

The different modules of the DBMS are implemented in Minibase as static instances. In order to call the Disk Manager, you need to access this instance as follows:

Minibase.DiskMgr.<method_to_call>()

where <method_to_call> is whichever method you are trying to execute.

Several methods are provided to enable you to use the Disk Manager. The methods you will need to use are as follows:

void read_page(PageId pageno, Page apage)
Reads the page denoted by the *pageno* PageId from disk, and stores the contents of the page in the *apage* parameter.

void write_page(PageId pageno, Page apage)
Writes the contents of the Page *apage* to the page denoted by the *pageno* PageId

PageId allocate_page(int run_size)
Allocates a contiguous run of pages of size *run_size* on disk and returns the PageId of the first page in that run

void deallocate_page(PageId start_page, int run_size)
Deallocates a run of pages on disk of size *run_size,* starting with the PageId *start_page*

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.

Some methods of the Buffer Manager are already provided to you. You must implement the following methods:

void pinPage(PageId pageno, Page page, boolean emptyPage)

The pinPage() method attempts to pin the requested page. The *emptyPage* parameter is not used for this assignment and can be ignored. This procedure follows several steps:

- **Determine if the page is already present in memory.**
  If it is already in memory, all that needs to be done is to increase the pin count. Otherwise, we need to bring the page into memory and proceed to the next step.
- **Determine an appropriate frame to store the new page in.**
  If there is an unoccupied frame in the Buffer Pool, choose that to store the page in. Use the disk manager to read the page into the frame and update the FrameDescriptor for that frame with the appropriate PageID and PinCount.
  If no unoccupied frame exists, use the page replacement policy to select a victim frame to evict from the Buffer Pool. If the page has been modified (its dirty bit is set to true), use the Disk Manager to write the page to disk. Then use the Disk Manager to read in the new page into the victim frame, and update the Frame Descriptor accordingly.
- **If no appropriate frame can be found (i.e. all pages in the Buffer Pool are pinned)** then throw a **BufferPoolExceededException**

- **After reading the page into the frame, set the *page* argument to the frame the requested page was stored in**

void unpinPage(PageId pageno, boolean dirty)

The unpinPage() method attempts to unpin the requested page, and sets the dirty bit if necessary. This procedure follows several steps:

- **Determine which frame the page is stored in.**
  Use the Hash Table to look up which frame the requested page is stored in. If the page is not found, throw a **HashEntryNotFoundException**.
- **Decrement the PinCount.**
  If the PinCount is already 0 before this method is called, you should instead throw a **PageUnpinnedException**
- **Set the dirty bit for the page**

PageId newPage(Page firstPage, int howMany)

The newPage() method attempts to allocate *howMany* pages in a consecutive run of pages on disk by calling the appropriate DiskManager method. It will then attempt to pin the first page of the run into the Page specified by the *firstPage* parameter.
If the run of pages is successfully allocated but is unable to be pinned (due to the buffer pool being full) then the pages should be deallocated using the DiskManager before throwing the exception.

Replacement Policy

You are asked to implement a FIFO replacement policy in order to determine which pages should be evicted when bringing new pages into memory if the Buffer Pool is full. A Queue datastructure *fifo* is defined for you to use to implement this. Pages should be inserted into the FIFO queue in the order that they become unpinned

Exception Handling

All exceptions mentioned in the project requirements should be thrown according to the specifications.

Calls to the Disk Manager methods will also generate exceptions if errors occur. Some of these exceptions may be generated as a result of failures from the calls the Disk Manager makes to the Buffer Manager to manage its data pages, and will result in a **BufMgrException**. Any such BufMgrException generated by the Disk Manager should be caught and thrown as a **DiskMgrException**. All other exceptions generated by the Disk Manager should be able to be thrown directly.

Testing/Running the Program

Due to the nature of the Minibase implementation of the Buffer Manager and Disk Manager, all required Buffer Manager methods must be implemented in order to properly test your implementation.

A JUnit Test Suite has been provided to enable you to test the functionality of your program. These tests will be used during grading. In order to run the tests, we also provide a Makefile.

Input the command *make* from the terminal on the university machines in the top-level directory of your program in order to compile and run your Minibase implementation and the tests.

We can only offer support for compiling and running this from the terminal on the university machines. You may use your personal machine or an IDE to help with development for this project, but please make sure your implementation can run on the university machines with the provided Makefile.

We cannot help with setting up an IDE to run the program properly – you will have to be familiar with setting the paths to the necessary libraries from within your IDE. Further, this code is not directly compatible with Windows systems – if you wish to run this on a Windows system, you will have to edit some commands in the test cases that are used to clean up the database files after running.

What to Turn in

1. The BufMgr.java file, as well as any other Java files you may have created for your implementation. Any additional files you create should be in the *bufmgr* package – do not create or change any files in any other package of the implementation.

These files should be submitted on Blackboard.