CS4432 - Project 1 Implementing a Simplified Version of DBMS Buffer Manager

Release Date: March 27, 2025
Due Date: April 3, 2025
Total Points: 100

Project to be done individually.
Programming Language: Java (Better use java 8 or later) Or C

Important: Read the entire document to understand all the requirements before you start your design and implementation.

Generative AI Usage Instruction

You are permitted to use generative artificial intelligence tools (such as ChatGPT, etc.) to assist in completing the assignment. However, when using these tools, you must clearly state the following:

- How the generative artificial intelligence tool was used
- How the generated answers were verified to be correct

Section 1: Overview

In this project, you will learn and implement the key concepts of buffer management as done in database management systems. Although the project will not be done within a real DBMS, you will simulate the actions taken by the buffer manager as we learnt in class.

In traditional DBMS a table consists of multiple data blocks. In our simplified project, a "table" will correspond to a "directory" in the file system, and a "data block" will correspond to a small "file" under this directory. Then, each DB record will correspond to a line in one of these files. In our context, all files have the exact same size. Each file stores 100 records (i.e., 100 lines), and each record (line) is 40 bytes. You can observe that each file will be of size roughly "4KB", which mimics a disk block.

Note: each file is now the unit of processing. That is, an entire file is read from disk to the buffer or taken out from the buffer and written back to disk.

- Every record is exactly 40 bytes (the format is shown below). There is NO end-of-line character at the end of each record. This means all records are just concatenated after each other.
- Conceptually, the format and content of a record will not make a difference. However, for ease of testing and readability, let every record follow this format. A record #j in file #i will have (for *i* use two digits like 01, and for *j* use three digits like 001):
  Fi-Recj, Namej, addressj, agej.

- Our numbering system will start from 1 for both files (index *i*) and records (index *j*). Index *j* resets and start with 001 in each file.
- Example: the 23rd record in the 3rd file will have the format of (exactly 40 bytes)
  F03-Rec023, Name023, address023, age023.

Example 1:
Assume a "Student" table that consists of 99 disk blocks and each block contains 100 records. Each record has 40 bytes containing the student ID, Name, address, phone number, etc. Now, this design in our project will correspond to a file system directory with name "Student", under which there are 99 small files (names F1, F2, F3, …F99), each file holds the content of one disk block.

Example 2:
Let's say we need to read record number *k (say k =250)*, we will need to do steps like the following:

- You need to do the calculations to figure out this record exists in which block (in our terminology, which file). For example, if k =250 and since each file holds 100 records, then the record we need is in file #3. Then, calculate the record number with this file. In this example (k = 250), it will be the 50th record in F3.
- You need to check whether or not file #3 is in memory (in the buffer). Let's say, it is in memory, then no I/O is needed. You need to find where in the buffer file #3 exists and read from this buffer record #50 (which corresponds to record #250 in the entire directory (table)).
- If file #3 does not exist in the buffer, then let's assume the easy case where there is an empty space in the buffer. In this case, you need to read the content of file #3 (all the content of this file), put it in a certain frame in the buffer, and then read the record we need from that buffer.

Section 2: Design Guidelines

In this section, you are given the high-level design ideas and guidelines to build the operations of a buffer manager. These are just guidelines. It is your responsibility to come up with the complete end-to-end design to have a working system.

Single Buffer (Memory Frame): "Frame" class

- This is an object in your Java program to hold one file (one block).
- You should design a "Frame" class, which contains some key (private) elements such as:
    - "*content*": array of bytes of size 4K //to hold the file content
    - "*dirty*": Boolean flag //set to True if the content of this block has changed and //need to be written to disk when this frame is taken out
    - "pinned": Boolean flag // True if there is a request to keep this block in memory and not // take it out. False, means it can be taken out.
    - "blockId": integer // It should be the Id of the block stored in this frame. E.g., if we //need to read file #3 as in Example 2, then "blockId = 3". // You can use "-1" to indicate that the fame is empty and there is // no block in this frame
    - *… any other variables you think useful and you need it in your design*
    - You should also have a set of "public" methods to **<u>set</u>** and **<u>get</u>** the values of the variables in this class.
    - You should also have methods to for example return a specific record in this block, e.g., record number *i* (remember that all records have the same size of 40 bytes). This method should take as input the record number (*i*) and return the content of this record (string of 40 bytes).
    - Another method you can think of is to update a specific record. This method should take the record number and the new content (40 bytes). Its job is to set that record to the new content.
        - ✓ Remember, if the content changes, the "dirty" flag should be set.
    - …You may think of other methods, e.g., initialize() method to initialize all of the variables above.

Buffer Pool: "BufferPool" class

- This is the object in your Java program which represents the entire available buffer
- You should design a "BufferPool" class, which contains some key (private) elements including:
    - "buffers": array of "Frame" objects. // The size of this array is decided at run time. // the program should take an input argument that decides // the size of this array
    - Any other elements and variable you think are useful
    - One public method is "initialize" that should (1) build the array given the input argument, (2) go over each frame and initialize this frame, e.g., by calling the *initialize()* method of each frame.
    - One method to search if a certain block (file) is available in the buffer pool. The method takes as input the block Id (file Id), and should return the buffer number (slot number in the array) holding this block (or -1 if not available).
    - Another method to possibly return the content of a given block Id. This method should also take as input the block Id (file Id). Call the method in the previous bullet to know the buffer number (if the block is present), and then it can read the content.
    - Another method to be used if the needed blockId is not in the buffer pool. In this case, this method should read the block (file) from disk, and bring it to the buffer pool (in an empty frame)!!!
    - You need a method to search and give you back a number in the array for an empty frame (if any)
    - If there are no empty frames in the buffer pool, then you may need to take one out and return it back to disk (if possible). This method will differ in how to select this to-be-evicted frame depending on the placement policy.
    - Any other methods that you need for your design

Section 3: Project Requirements

You should design a Java program following the guidelines highlighted above. The program should support the following functionalities (Make sure to read Section 4 before jumping to implementation):

1) **[10 Points] Calling the program** and passing one input parameter (*n*) representing the buffer pool size
    a. The program should create the bufferPool class with "n" slots in the "buffers" array
    b. Do all proper initializations. Initially, all frames are empty. Also initialize any metadata you have.
    c. After that, write on the screen message "The program is ready for the next command", and wait for the user to enter the next command, which will be any of the following commands.

    //For consistency and ease of grading, all commands should be UPPER CASE (GET, SET, PIN, UNPIN)

    For the input data, assume the directory is named "**Project1**" in the same place from which the java program will run. Under this directory, there should be files F1, F2, …

    Note: TAs will not test for out-of-range record number or file number. Whatever is requested in the following commands is expected to be present under directory "**Project1**".

2) **[40 Points] Command #1 (GET command) : "*`GET k`*".** In this command, the user needs to print the content of record #k from the file.
    a. The program should call a GET() function in the buffer pool class. The function should calculate which block (file) contains this record (refer to Example 2).
    b. The function should scan the "buffers" array to figure out whether or not the desired file is in memory. You should call the corresponding method in the "BufferPool" class to do so. There are four possible cases:

Table 1

CASE	What Needs to be done
CASE 1 **[10 Points]:** The block (file) is in memory	This is an easy case, find the desired record (40 bytes only) and return it to be printed on the screen.
CASE 2 **[10 Points]**: The block is not in memory, but there are empty buffers in the buffer pool array	You need to read the right file from disk. Copy its content to an empty buffer frame. Update the metadata properly. Once the file is in memory, you should call the functions used in CASE #1 to do the rest. //For this part, choose the 1st empty frame from the beginning of the array (do not choose randomly). That is, if the array has the 1st 5 frames full and the rest are empty, then the file should go to frame #6.

CASE #3 **[10 Points]**: The block is not in memory, the buffer pool array is full (no empty frames), but some frames can be taken out.	The content of a non-empty frame can be taken out ONLY IF its "pinned" flag is "False". Otherwise, this frame is not a candidate to be taken out. You need to search for a frame that can be taken out. If you find one, then, there are two cases: Case 1: The "dirty" flag is False. This means it can be taken without the need to write back to disk. Just overwrite the content. Case 2: The "dirty" flag is True. This means that the content must be written back to disk, otherwise the changes will be lost. <u>Remember that the entire file content (4KB) is one unit, even a single byte changes the entire file needs to be written back to disk and overwrite the previous content.</u>
CASE #4 **[10 Points]**: The block is not in memory, the buffer pool array is full (no empty frames), no frames can be taken out	This may happen if all frames have content and all of them are "pinned". In this case, print out a message "The corresponding block #<and write the number> cannot be accessed from disk because the memory buffers are full".

<u>**Output From "GET" Command:**</u> The required output is:
(1) Print the record content (the 40 bytes) for CASEs #1, 2, 3 above, or the message indicated in CASE #4
(2) Print whether or not an I/O is done (i.e., whether the block was already in memory or brought from disk)
(3) Print the frame# (the entry number in the buffers array) that contains the block (for CASES #1,2,3)

3) **[10 Points] Command #2 (SET command) : "*`SET k <string of 40 bytes>`*".** In this command, the user needs set the content of record # k to the given string.
   a. All of the work in Command #1 applies to have the desired block (file) in memory.
   b. In addition, you need to change the content of the record to the new string. You will be given exactly 40 characters, so you do not need to do any error checking for that.
   c. Make sure to set the "dirty" flag to "true"

<u>**Output From "SET" Command:**</u> The required output is:
(1) Print whether or not the write is successful
(2) Print whether or not an I/O is done (i.e., whether the block was already in memory or brought from disk)
(3) Print the frame# (the slot number in the buffers array) that contains the block (for CASES #1,2,3)

NOTE: In the SET command, DO NOT write the file back to disk. It will remain "dirty" in memory until the buffer manager decides to take out, at that time it should be written to disk.

4) **[10 Points] Command #3 (PIN command) : "*`PIN BID`*".** In this command, the user wants to pin specific block (BID) in memory.
   a. Notice that BID is a block number (file number) not record number. Example "PIN 3" means pin block #3 (F3) in memory.
   b. There are few cases

Table 2

CASE	What Needs to be done
CASE 1: The block is already in the buffer pool	Set the "pinned" flag to True. If it is already set, then do nothing.
CASE 2: The block is not in memory, and the buffer pool is either have empty slots or you can take out a block (CASEs 2 &3 in Table 1)	You should bring the BID to memory, and set the "pinned" flag to True
CASE 3: The block is not in memory, the buffer pool is full, and no blocks can be taken out (they are all pinned) (CASE #4 in Table 1)	In this case, print out a message "The corresponding block BID cannot be pinned because the memory buffers are full".

Output From "Pin" Command: The required output is:
(1) Print the frame # in the buffer pool array that is pinned.
(2) Print whether or not the "pinned" flag was already true.

5) **[10 Points] Command #4 (UNPIN command) : "UNPIN *`BID`*".** In this command, the user wants to unpin specific block (BID) in memory.
   a. Notice that BID is a block number (file number) not record number. Example "UNPIN 3" means unpin block #3 (F3) in memory.
   b. There are few cases

Table 3

CASE	What Needs to be done
CASE 1: The block is in the buffer pool	Set the "pinned" flag to False. If it is already False, then do nothing.
CASE 2: The block is not in memory	In this case, print out a message "The corresponding block <BID> cannot be unpinned because it is not in memory".

Output From "Unpin" Command: The required output is:
(1) Print the frame # in the buffer pool array that is unpinned.
(2) Print whether of not the "pinned" flag was already false.

6) **[10 Points] Code Commenting**
You are required to provide good comments on your code. Every function needs to be properly commented such that others can understand your code. Lack of comments will cause you to lose to points.

Section 4: Some strategy to follow

- ***How to locate an empty frame in the buffer pool?***
  - The easy and straightforward way is to search all entries in the array and check "blockId" value. If the value is -1, then this frame is empty and can be used. This is an acceptable strategy to use in this project.
  - As covered in class, this scan strategy is slow if the number of the frames (the buffer pool size) is large, and thus there are other more efficient ways that DBMSs actually use, e.g., the use of bitmaps. It is up to you if you would like to implement and use the bitmap strategy. Think of where this bitmap should be stored and design the manipulation methods that manage the bitmap.

  - Either of the scan approach or the bitmap approach should lead to the exact same selection, which is covered in next bullet.

- ***In what order an empty frame is selected or chosen?***
  - You must select the 1st empty frame that you find starting from the top of the buffer pool. For example, if the pool has 10 frames, and the first 3 are full, then the next one to choose MUST be frame #4, and then #5, and so on.
  - When grading, the TAs will expect that order, otherwise your number will not match, and you will lose points.

- ***In what order to evict a full frame to make space for a new block to bring to memory?***
  - Remember that any "pinned" block cannot be taken out (it is not candidate for eviction) until it is "unpinned"
  - The order of selection must follow the following strategy. Select the 1st candidate frame following the last frame being evicted. For example, if the last evicted frame is #1, and frames #2 & #3 are pinned, then the next frame to be evicted should be frame #4
  - Think of this strategy in a circular fashion. That is if you reach the end of the buffer pool, then you start from the beginning again.
  - When grading, the TAs will expect that order, otherwise your number will not match, and you will lose points.

- ***How to test your code?***
  - A test case is provided to you (dataset, set of commands, the expected output). Use it to test your code.
  - Also, create your own test cases to further validate your code.

What to Deliver

- The entire source code package
- Readme.txt, in which you must include:
  - Your name and student ID
  - Section I: section on how to compile and execute your code. Include clear easy-to-follow step by step that TAs can follow
  - Section II: section on test results. A test case will be provided along with the expected output. This test case will help you testing your code. In the Readme.txt file state which of the test case commands is successfully working and which ones are failing. This will help TAs to better test your code and give you fair grades
  - Section III: section describes any design decisions that you do beyond the design guidelines given in this document. For example, any additional variables or methods or classes that you add.

What and Where to Submit

A single file .zip to be submitted in the Canvas system