CS 44800: Introduction to Relational Database Systems

Project 1: Buffer Manager

Fall 2021

Due: 5 October 11:59pmEDT

This Project is based on the SimpleDB database system. SimpleDB is a multi-user transactional database server written in Java. SimpleDB was written by Professor Dr. Edward Sciore and is widely used as a teaching tool for database systems. For this project, you will be working with the Buffer Manager that comes with SimpleDB.

According to Dr. Edward Sciore,

"*The SimpleDB buffer manager is grossly inefficient in two ways:*

- *When looking for a buffer to replace, it uses the first unpinned buffer it finds, instead of doing something intelligent like LRU.*
- *When checking to see if a block is already in a buffer, it does a sequential scan of the buffers".*

To overcome the inefficiency of the existing buffer manager of SimpleDB, one can do any of the following-

1. Implement LRU (Least Recently Used) as a replacement policy. You can keep a list of unpinned buffers. When a replacement buffer needs to be chosen, remove the buffer at the head of the list and use it. When a buffer's pin count becomes 0, add it to the end of the list. This implements LRU replacement.

   OR

2. Implement MRU (Most Recently Used) as a replacement policy. Unlike LRU, here, you will have to replace the buffer which has been used most recently. In other words, this MRU policy is the opposite of the LRU.

3. Now, after choosing one of the above policies to implement, you will also have to fulfill the following requirements for your implemented policy:
   - Regardless of your chosen buffer replacement policy (i.e., LRU or MRU), in the list of unpinned buffers, you will have to distinguish between modified and unmodified buffer when looking for an available buffer.
   - Let us assume that the unpinned buffers which have not been modified before are called unpinned_unmodified buffers. Moreover, the unpinned buffers which have been modified before are called unpinned_modified. Now, augment your chosen policy (i.e., LRU/MRU) by keeping track of the unpinned_unmodified and unpinned_modified buffers.

- Finally, when choosing a buffer to replace, follow the LRU/MRU requirements. After that you select a buffer from the list of unpinned_unmodified first. If none found, then select a buffer from the list of unpinned_modified. In other words, you will give priority to the unpinned_unmodified buffer over unpinned_modified ones while maintaining the LRU/MRU requirements.

Project Description:

Now, the tasks for you for this project are-

Task 1-(Decide): Which one of these two approaches (LRU or MRU) do you think will perform best? Will it perform better to use unmodified buffers first, or to use the policy (LRU or MRU) strictly, without worrying if a buffer has been modified? Give reasoning for your answer. You may want to give some examples.

Task 2-(Design and implement): Choose one of the two approaches (LRU or MRU) to implement. If the approach you are going to implement is different than the approach you picked in Task 1, give a brief description of why you decided to try this approach instead. After implementing one of the policies (e.g., LRU/MRU), you must implement the second version of your implemented policy following the requirements mentioned in ***3.***

Task 3-(Measurement and Evaluation): Explain how you can test to determine if performance improves. You should discuss the modifications that you have made to the existing buffer manager, how are you measuring performance and collecting performance data, etc. Note that there is likely to be file system caching that make impact wall-clock times, you'll have to think about ways to get a robust measurement. After that evaluate your implementation by comparing the performance of the existing buffer manager and your version of improved buffer manager. You need to use graphs or charts or some analysis of your collected performance data along with your discussion to justify your evaluation.

Task 4-(Comparison): Now, what if you want to know how your implementation evaluates against other approaches. That is why, for the last task, we ask you to collaborate with at least ***one*** other member from your PSO who has implemented a different approach. Write some test cases and queries as a team and compare the performances for implementations. You may have up to ***three*** people in a team, in which case two of you will have implemented the same approach – in this case, you should discuss if the two that are the same approach behave similarly (and if not, why not?)

Project Deliverables:

1. The modified code of buffer manager and test files, submitted using turnin (turnin -c cs448 -p project1 <submission folder> , on CS department linux machines). While you can develop in the environment of your choice, these should run on the CS department

linux machines (e.g., the ones in HAAS G050, or data.cs.purdue.edu). You can access CS department machines remotely. Please submit only modified source code and data files needed to run your code (last semester, most people turned in files that were under 500KB compressed, but some people managed to turn in much larger – over 100MB – and it exhausted our quota.)

2. A project report that covers all the points that are mentioned in the project description, submitted in gradescope.

Tasks 1, 2, and 3 must be done individually, although you can discuss your ideas with others. You will need to coordinate with your team members to ensure that you pick different approaches (even if you both feel the same approach will give the biggest improvement.) Only Task 4 will be done collaboratively. The report, except Task 4, should be written only by yourself! You may choose to write Task 4 independently, but it is acceptable to write it collaboratively and turn in the same writeup for Task 4.

Late work: We realize that some team members may need to use late days that others may not need. If your teammate needs more time than you do, please turn in your code and the report for Tasks 1-3. In the writeup for Task 4, state "See <team member name> for task 4 writeup."

Additional Instructions:

- Submit your team (2-3 members per team) by email to your PSO instructor by September 24.
- If you cannot find a teammate, submit that to your PSO instructor who will assign you teammate(s). Note that you may end up having someone added to your team, even if you already have two.
- You can pick your own teammate, but they need to be from the same PSO.
- Yes, you can post in Piazza asking for teammate/s.
- Some description of the Buffer Manager of SimpleDB is added for your reference at the end of this document.

SimpleDB Buffer Manager

The Package that you need to work on for this project is called "*simpledb.buffer".* There are lots of classes in "*simpledb.buffer"*, you are not going to modify all of them. You need to modify only

the classes "*Buffer*" and "*BufferMgr*" in order to achieve the goals of this project. the Followings are the list of useful classes and their functions that will help you to proceed with the project.

Buffer : The Buffer class represents an individual buffer. A databuffer wraps a page and stores information about its status, such as the associated disk block, the number of times the buffer has been pinned, whether its contents have been modified, and if so, the id and other information of the modifying transaction.

Some useful functions-

***block()*:** Returns a reference to the disk block allocated to the buffer.

***isPinned()*:** Return true if the buffer is currently pinned.

***flush()*:** Write the buffer to its disk block if it is dirty.

assignToBlock(BlockId b): Reads the contents of the specified block into the contents of the buffer.

pin(): Increase the buffer's pin count.

unpin(): Decrease the buffer's pin count.

***BufferMgr* :** The BufferMgr class is responsible for managing the pinning and unpinning of buffers to blocks. This is the place where you will be implementing most of your codes for this project.

Some useful functions-

BufferMgr(FileMgr fm, LogMgr lm, int numbuffs) *:* Creates a buffer manager having the specified number of buffer slots. This constructor depends on a FileMgr and LogMgr object. numbuffs is the number of buffer slots to allocate. FileMgr is responsible for synchronized read and write of pages to and from blocks and LogMgr is responsible for writing log records into a log file.

available() : Returns the number of available (i.e. unpinned) buffers.

flushAll(int txnum) : Flushes the dirty buffers modified by the specified transaction. "*txnum"* the transaction's id number.

unpin(Buffer buff): Unpins the specified data buffer. If its pin count goes to zero, then notify any waiting threads. "*buff"* is the buffer to be unpinned.

pin(BlockId blk): Pins a buffer to the specified block, potentially waiting until a buffer becomes available. If no buffer becomes available within a fixed time period, then a *BufferAbortException* is thrown. "*blk"* a reference to a disk block. Returns the buffer pinned to that block.

tryToPin(BlockId blk): Tries to pin a buffer to the specified block. If there is already a buffer assigned to that block then that buffer is used; otherwise, an unpinned buffer from the pool is chosen. Returns a null value if there are no available buffers; otherwise, returns the pinned buffer. "*blk"* a reference to a disk block.