Implementing a DBMS Bufferpool Manager with Varied Page Replacement Policies

Samir Farhat Dominguez
Boston University
Boston, MA
safarhat@bu.edu

Yuxin Li
Boston University
Boston, MA
yuxinli@umich.edu

Stephany Yipchoy
Boston University
Boston, MA
syipchoy@bu.edu

ABSTRACT

Within the context of Database Management Systems(DBMS), it is often impossible for a bufferpool to fit the entirety of a workload's prospective page requests, let alone be populated exactly with those pages. As such, page replacement and eviction becomes a necessary part of any DBMS systems employing bufferpools. There are several different page replacement policies that bring their own benefits and drawbacks in terms of performance. The selection of a page eviction policy is a tremendously nuanced issue, as the degree of these benefits and drawbacks are influenced by factors such as the DBMS system being used, the workload of an application, the machine a DBMS system is running on, the physical structure of the memory hierarchy members, and many other aspects. This work will seek to focus on page replacement policies, performing a comparative analysis under a diversity of relevant parameters; including read/write ratios, bufferpool sizes, randomized skewing, disk sizes etc.

1 INTRODUCTION

One of the central paradigms in computer architecture is the notion of memory hierarchies with decreasing capacities and increasing access times. Within this paradigm, a critical subset of computing problems arise. These are; the selection of data structures maintained in smaller but faster memory, the algorithms maintaining these data structures, and the policies deciding how to discard data from this higher value memory. One such proposition to address the former two, is a bufferpool and its respective insert, delete and read algorithms. Bufferpools are constituted by a series of frames, that can be either occupied by a page from the disk, or not.[1]

1.1 Motivation

A bufferpool is often used in DBMS systems, and occupies a portion of main memory. The bufferpool acts as a cache for table and index data on the disk, which is quite large but tremendously slow in terms of read/write times.[6]

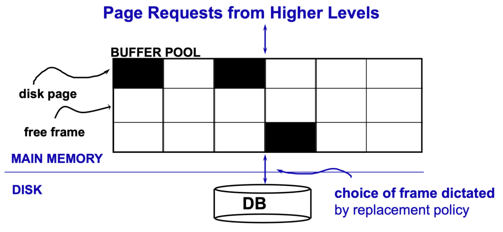
Page Requests from Higher Levels
BUFFER POOL
disk page
free frame
MAIN MEMORY
DISK
DB
choice of frame dictated by replacement policy

Figure 1: Depiction of a bufferpool

Therefore, optimal performance with a bufferpool can be achieved such that when an application requests a page from the disk, the bufferpool contains this page, a page hit takes place, and the page is read with the faster main memory access time. If a page miss does occur, then the DBMS is forced to go down in the hierarchy, and find the requested page from the disk, which is far slower to access. This page is then written to the bufferpool, either by populating an empty frame, or replacing a page contained in a populated frame. If a replacement does occur, bufferpools maintain a 'dirty' bit, indicating if the page has been modified since it was brought in from disk. If it was the disk must be written to with the updated value, requiring another very slow access and write procedure. If it is not dirty, then it is simply discarded from the bufferpool.[3]

When working with a DBMS, there exists a persistent goal to make your workload faster and more efficient. Examining the bufferpool and its page replacement policy is an essential place to begin on this journey because it determines if you will be able to immediately access a page or have to begin the slow journey down the memory hierarchy.

1.2 Problem Statement

Analyzing the performance of a general workload under four different bufferpool page replacement policies (FIFO, LRU, CFLRU, and LRU-WSR) will give us an idea of which to choose when speedily trying to increase the efficiency of a DBMS, and give a bouncing off point for more research on how changing the bufferpool page replacement policy leads to performance gains. We have implemented LRU, CFLRU, LRU-WSE and FIFO replacement polices. Our program simulates disk management (read / write pages in disk) along with the bufferpool in order to accurately test and compare the performances of implemented replacement policies.

1.3 Contributions

The work has been divided into two sections: implementation of the bufferpool and four replacement policies (LRU, CFLRU, LRU-WSR and FIFO), and testing and comparing the performance of the DBMS bufferpool manager with different parameters.

We began by implementing a basic bufferpool, which holds a queue of vectors; the vectors very simply just hold a page id and a dirty bit. We proceeded to set up the basic functionality of the bufferpool: creating functions to search for a page id, processing read requests, and processing write requests. Next, we implemented the actual page replacement policies that will be used to measure performance. These are elaborated on in the next two sections of the work, but just to enumerate these we have LRU, FIFO, CFLRU, and LRU-WSR. With these page replacement policies we include tracking of relevant statistics, most notably timing and I/O procedures. We also implement a simulation for disk behavior, by performing reads and writes on a file throughout the program.

Having implemented the above, we sought to consolidate our testing to perform a comparative analysis of these page replacement policies. As such, we developed a testing suite, where we are able to conduct a full barage of tests within parameters of interest. We also add functionality to isolate page replacement policies and test them under the other permutations of parameters, furthering our understanding of the strengths and weaknesses of these. Finally, in our testing suite, we implement a graphs and plotting engine to eventually present relevant results in Section 5.

2 BACKGROUND

The physical structure and logic of the bufferpool is quite straightforward. This work focuses on how the choice of page replacement policy drives the performance of the bufferpool, which is why implementing different policies is a major component of this project. We will be implementing the LRU, FIFO, CFLRU, and LRU-WSR policies.

LRU is the most frequently used replacement policy- it is based on the assumption that a page that was more frequently referenced is more likely to be referenced again than a page that was referenced longer ago. Ergo, temporal locality is its principal ethos. For the implementation of this page replacement policy, a list is maintained which is sorted by how recently a page was accessed, and chooses the page that was accessed the longest time ago to be removed from the cache to make room for another page.[7]

FIFO, or First In First Out, is known as a more naive example of bufferpool page replacement policies, where a list is maintained of pages accessed, and each time a new page is accessed it is simply appended to the back of the list. When it is time to remove a page from the bufferpool, the first page in the list is removed.[2]

CFLRU is a page replacement policy that has been optimized for flash based storage, which is employed for storage devices like SSDs that use nonvolatile memory to read and write data. It is an extension of the LRU policy in that it divides the list maintained in LRU into two parts: working and clean first. The working part contains pages that have been recently accessed, while the clean first part contains clean, or unaltered, pages. CFLRU works by first removing pages from the clean first part, and once all those pages have been removed, it begins using the basic LRU algorithm to remove dirty, or updated pages. The purpose of this order is to minimize the number of writes that will be done to the pages.[8]

LRU-WSR, or LRU with Write Sequence Reordering, is similar to CFLRU in that it is also optimized for flash based storage, and is also an extension of the LRU page replacement policy. It works by prioritizing replacing all dirty pages last, meaning that an altered page that has not been accessed recently will remain in the bufferpool for longer than a clean page that was recently accessed. This works by maintaining a cold flag on the list of pages, which is reset when a page is accessed. If a dirty page's cold flag is off, it is not a candidate for page replacement, while all clean pages are evicted.[5]

3 ARCHITECTURE

We implement a bufferpool simulation system consisting of the following header and corresponding `.cc` files.

(1) `buffmanager.cc`: This is where our main is found. We implement the parsing of arguments, progression of the simulation, building the workload, and running the bufferpool simulation while tracking statistics.
(2) `parameter.cc and parameter.h`: This defines and implements the `Simulation_Environment` class. Where the parameters are managed and shared across the program.
(3) `workload_generator.cc` and `workload_generator.h`: The workload generator files define and implement the workload generation function, where pageIDs and operation types(Read or Write) are defined.
(4) `executor.cc and executor.h`: The executor files represent the bulk of the implementation. Defining both the `Buffer` and `WorkloadExecutor` classes, which handle everything related to instruction handling for the disk and Bufferpool.

All of the above are elaborated upon in this section.

3.1 Program Flow and Externals

There are separate execution scripts that we have written in `Python`, nominally to run tests, collect data, and process it. However, when it comes to the core bufferpool simulation this has been fully implemented in `C++`. We begin with the initial command with non-specific parameters to correctly run the program. This is `./buffermanager -b -n -x -e -a -s -d -k`. `buffermanager` is the resulting executable from the `make` command defined in the `Makefile`. The parameters are defined in the `parameter.h` and defaults set in `parameter.cc` as part of the `Simulation_Environment` class. These parameters are as follows:

(1) `-b`: The size of the bufferpool in regards to page count.
(2) `-n`: The size of the disk in terms of pages, which will always be larger than the size of the bufferpool.
(3) `-x`: The total number of operations, which will be either reads or writes.
(4) `-e`: The read and complement of write percentage distribution of operations.
(5) `-a`: The page replacement replacement algorithm for the simulation run.
(6) `-s`: The skewed distribution of operations on data.
(7) `-d`: The skewed data percentage which defines the distribution of operations on data, which defines the concentration of page collisions.
(8) `-k`: This is our own implemented parameter relevant to disk handling, and is the number of bytes per page. This parameter alongside the `-n` define the size of the disk.

Once we issue the command we we can elaborate on program flow by discussing `buffermanager.cc`, where our main function is found. We begin by instantiating the simulation enviornment with the relevant parameters, and perform some print statements as a sanity check. We then load our `WorkloadGenerator` object and generate the respective workload, which is a series of instruction types and page IDs. Finally we run the workload, where we read the workload generated. We open and populate the disks, and then run the read and write operations until its terminated. Throughout the logic implemented in the executor files is running to do the appropriate sequence of actions. Finally we write statistics to the appropriate files and standard out as a sanity check and overview of the run.

3.2 Bufferpool Structure

A buffer pool is an area of main memory that has been allocated by the database manager for the purpose of caching data as it is read from disk. Therefore, the database bufferpool holds blocks of data retrieved from the disk. The bufferpool structure is a vector. Each member of the vector represents a page. Pages in the bufferpool can be either in-use or not, and they can be dirty or clean. While the data structure might be different based on different replacement policies, the essential data stored for each page in the bufferpool contains a page id; and a dirty bit, a value to denote if the page is dirty or not. Bufferpool has a fixed size and therefore needs different replacement policies to maintain fewer I/O operations result, which means faster access to stored data.

3.3 Search

The search function uses the page id to perform a linear search in the bufferpool. Since every page id is unique, the function will return the position index of this page in the bufferpool if it found a match and increase buffer hit index by 1. Otherwise it returns -1 and increases buffer miss index by 1.

3.4 Bufferpool update: Read

If the instruction is "R", The read function will be called with the page id and the chosen replacement function. We will first search if the page id is in the bufferpool and return it's position index. If the bufferpool is empty or no match id can be found return -1.

(1) If the page id is found in the bufferpool, the data in that page will be directly accessed from the bufferpool. And therefore the function will only be updating the LRU list by using an iterator to search the page id in the LRU list, delete the matching element from the LRU list and append a new one with the same page id at the front.

(2) If the page id is not found and the bufferpool is not full yet, we read the page from disk into the bufferpool, mark the page clean and increase the read index by 1. We then update the LRU list by simply append the new page id the front of the list which is the position of MRU.

(3) If the page id is not found and the bufferpool is full, we proceed to call replacement functions based on chosen policies (LRU, CFLRU, LRU-WSE and FIFO) to get the position of eviction target in the bufferpool. After we found the target page to be evicted, we also need to check the dirtyBit of the target page. If the target page has been marked dirty, we write this page into the disk and increase write index by 1. We then replace its page id with new page id we read in from the disk, mark the page clean and increase the read index by 1. Finally we update the LRU list by simply append the new page id the front of the list.

3.5 Bufferpool update: Write

If the instruction is "W", The write function will be called with the page id and the chosen replacement function.

First, we search for the specified page id that will be updated in the bufferpool. If it is in the bufferpool, we return its position, and then we use an iterator to find the page in the LRU list, delete it from the list, and them append a new one with the same page id and a dirty bit to the front of the LRU list.

If the page id cannot be found in the bufferpool, we first check to see if the bufferpool is full or not. If the bufferpool is not yet full, we first read the page from the disk, and turn on the page's dirty bit to indicate that it has been updated. We then append the page to the front of the LRU list, and increment the read index.

If the page id is not in the bufferpool, and the bufferpool is full, we first use the chosen replacement function to get the position of the least recently accessed page in the LRU list. If the dirty bit of the page has been turned on, indicating that it has been updated, we write this page back to the list and increment the write index. We then remove this page from the bufferpool, and read in the new page, and increment the read index. We finally add this page to the front of the LRU list, and turn on its dirty bit to indicate that it has been updated.

3.6 Disk Management

The disk management aspect of the codebase is implemented in the `WorkloadExecutor` class. In the run workload we create our disk object as an `fstream` pointing to `disk.txt`. The stream itself allows input and output and is part of the instance of the buffer in the workload. The disked is worked through in 2 different functions.

3.6.1 `writeDisk`.

The `writeDisk` function takes an instance of the buffer and is utilized as an initial population of the Disk. We simply use a pseudo-random generator with a seed generated from the timestamp in order to populate a string with a certain amount of characters. In C++ the standard `char` type is 1-byte. As such we populate the string with the amount of characters set by the parameter, exception being the last character which is simply and endline character. As such, each page is divided by the rows in the disk text file.

3.6.2 `diskOp`.

The `diskOp` function takes an instance of a buffer, an integer indicating if its a read or a write, and a page ID. The page ID and `pageSize` taken from the buffer instance are used to calculate the amount of bytes prior to the starting point of the page of ineterest. The `seekg` function is used in both cases to move the pointer to this computed byte. If the operation is a read we simply retrieve byte by byte into a temporary string. Since we aren't actually interested in what is contained in that string that terminates the read, as we are only simulating the action of loading a page byte-by-byte into memory. If the operation is a write we select a character from a sequence, and replace they bytes of the page byte-by-byte. This simulates writing into disk and thus will have a more realistic impact on the runtime of our program, therby facilitating our comparative analysis.

3.7 Replacement Policies Implemented

In this section, we will briefly talk about the structure of bufferpool and LRU list including how we managed to correctly update and maintain them. And we will mainly focus on our design and implementation details about implemented replacement policies. [4]

3.7.1 LRU. LRU (Least Recently Used) function will only be called when the bufferpool is full but needs to read in fresh data. The LRU function returns the evict position in bufferpool. When a new page needs to be stored and the bufferpool is full, the LRU function will go to the last element in the LRU list and get the least recently used page id. Delete this element in the LRU list as this page will be replaced. Using the target page id to perform a linear search in the bufferpool and return the position of that page id in the bufferpool.

We also have the LRU list, which is a list of pointers. There is a most recently used end, and a least recently used end. Each pointer in the list refers to a page in the database buffer cache. The way we implement is maintaining a deque storing page ids as the pointer to that page in the bufferpool since every page id is unique. This is because deque is faster for adding/removing elements to the end or beginning. The front of the deque represents the most used end and the back of the deque represents the least used. Generally, every time a block in the bufferpool is read or written as the result of a query, the pointer to that block (here is page id) is moved to the most end of the LRU list (MRU). Therefore, over time, this natural sorting mechanism results in the pointers to the least recently used blocks ending up at the least end of the list. These are the least likely to be required by any subsequent queries, so they are always the first ones to be replaced when the bufferpool needs to read in fresh data.

3.7.2 FIFO. FIFO, which stands for "First In First Out", is a straightforward page replacement algorithm that is applicable for small systems. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal. The difference between LRU and FIFO is that when a page hits, LRU will move this page to the MRU position to update the LRU list while FIFO will make no changes to the queue. The main advantage of the FIFO page replacement algorithm is its simplicity. However, when the number of incoming pages is large, it might not provide excellent performance because every frame needs to be taken account off.

3.7.3 CFLRU. CFLRU, also known as Clean First LRU, has a similar algorithm to LRUs eviction policy, and also returns the evict position in the bufferpool. CFLRU divides the bufferpool into two sections: the working section, and the clean first region. When looking for a page to evict in the bufferpool, the algorithm first looks in the clean first region, which is composed of the N least recently accessed pages, where N is the floor of a third of the size of the bufferpool, following the standard measure, which ensures that there are fewer misses. If there is any clean page in the clean first region, then the first one will be the page that is evicted. If there is no clean page in the bufferpool's clean first region, then the least recently accessed page will be the one evicted.

3.7.4 LRU-WSR. LRU-WSR, also called Write Sequence Reordering, is another replacement function that is similar to LRU policy. The only difference between those two policies is that LRU-WSR assign each page with a bit flag called "cold-flag". And this cold flag is the key to make the LRU-WSR algorithm a "second chance algorithm". The LRU-WSR function also returns the targeted evict page position in bufferpool. When the bufferpool is full and a new is coming, we check the last page in the LRU list (LRU position). If this page is clean, this page is the target page to be evicted and its position in the bufferpool will be returned. Otherwise, if the page is dirty, we check its cold flag. If the cold flag is set, this means the page has been the evicted target before, then this page is the target page to be evicted and its position in the bufferpool will be returned. If the cold flag is not set, the function move this page to the front (MRU position) and set its cold flag. Then the function will point at the current LRU position page and do the above process again until it find a page that is qualified to be evicted.

4 EXPERIMENTATION

4.1 Parameter Selection

4.2 Testing Suite

For our testing, we implement a separate set of python modules in the `workload_suite`; which implements testing automation, execution, data processing, and plotting.[4]

4.2.1 Automation and Execution. In the `full_barrage.py` script, we enable testing for all permutations of parameters as well as policy focused test. We do so by taking in a `-algo` argument at the command line. Here, we use some logic to enter the right sequence of tests. From there, we use the helper functions in `testing_helpers` to build the commands with the intended parameters and run them as bash terminal commands. Results are saved in the `raw_results` directory.

4.2.2 Processing and Plotting. *We intend to discuss our data processing and plotting scripts.

* We intend to add the desired testing system details here as well as our measurement metrics and parameters. While we have conducted preliminary testing for LRU, it was done more so to test functionality. we've added some snippets just to show functionality. From "buffersize_150_disksize_1500 _numops_7500_rw_60_ skewedperct_90_skeweddataperct 10_algorithm_1.txt"

```
WRITE LRU List  143 7 124 148 58 125 59 103 47 116 100 38 90 36 105
READ LRU List  103 143 7 124 148 58 125 59 47 116 100 38 90 36 105
READ LRU List  57 103 143 7 124 148 58 125 59 47 116 100 38 90 36
WRITE LRU List  120 57 103 143 7 124 148 58 125 59 47 116 100 38 90
READ LRU List  46 120 57 103 143 7 124 148 58 125 59 47 116 100 38
...
****************************************************
Printing Stats...
Number of operations: 7500
Buffer Hit: 583
Buffer Miss: 6916
Read IO: 6917
Write IO: 2917
Global Clock:
```

****

5 RESULTS

Graphs are basically a requirement here! * We have managed to run about 2000 runs with varied parameters at each step. That being said we still are yet to make a final decision on the parameter ranges we think will be productive. As such this section, the following and the previous are scant. That being said we have essentially finalized our implementation besides code clean up, and just need to add the plotting scripts once we decide what independendent variables we wish to focus on.

6 CONCLUSION

Final thoughts here.

REFERENCES

[1] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. 1994. The uniform memory hierarchy model of computation. *Algorithmica* 12, 2 (1994), 72–109.

[2] Clifford E Cummings. 2002. Simulation and synthesis techniques for asynchronous FIFO design. In *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*.

[3] Jaeyoung Do, Donghui Zhang, Jignesh M Patel, David J DeWitt, Jeffrey F Naughton, and Alan Halverson. 2011. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1113–1124.

[4] Samir Farhat Dominguez, Yuxin Li, and Stephany Yipchoy. 2022. *Bufferpool Manager, https://github.com/SamirFarhat17/bufferpool-manager*.

[5] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. 2008. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *IEEE Transactions on Consumer Electronics* 54, 3 (2008), 1215–1223.

[6] Papon Manos Athanassoulis. 2022. The uniform memory hierarchy model of computation. *Data-intensive Systems and Computing Lab Department of Computer Science College of Arts and Sciences, Boston University* 12, 2 (2022), https://bu–disc.github.io/CS561/projects/CS561–S22–SysProj–Bufferpool.pdf.

[7] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[8] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. 2006. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. 234–241.