CS 387 : Databases and Information Systems Lab

Page Replacement Strategies
Detailed Design Report

Shudhatma Jain (130050024)   Aditi Laddha(130050026)   Anmol Arora(130050027)

Objective
To implement various **page replacement algorithms** and compare their **performance** for database operations such as '**Join**' and '**Select**'.

Introduction

Why do we require page replacement?

Virtual memory systems use a two-level memory hierarchy consisting of main memory and secondary memory.
Pages are brought to the main memory, only when the program demands them (known as Demand Paging). If a page is not present in the main memory (known as page fault) , then it needs to be brought from the secondary memory.
As physical memory is much smaller than virtual memory, it fills up quite fast. So now if there's a page fault with physical memory being full, we need to replace already existing page(s) to accommodate the new one(s) to keep the program going. Hence page replacement is necessary.

Why developing a good page replacement algorithm important?

While we may replace any page at random, it would be much better if we replaced a lesser used page. This has a significant effect over the performance as accesses from secondary memory are very costly as compared to main memory accesses.

Design Strategies for Various Techniques

We discussed LRU and MRU in class. It was discussed that LRU is a good strategy for OS, however for DBMS better strategies are possible. For example, in the case of Join, MRU might be a good strategy (as discussed in class). The implementation methodology of various replacement strategies (changes made in buf.c mostly) and the comparison operations are as follows :

Least Recently Used (LRU) :
The page unused for the longest time is thrown first.
Implementation : Already implemented in Toydb.
How :

1) Whenever a new buffer is required, this function is called : PFbufInternalAlloc(bpage,writefcn). The call either:
    a) uses the existing free buffer pages
    b) if not then creates more space for buffer page if buffer memory is not full
    c) else replaces existing pages in the memory.
2) PFbufInternalAlloc is called in PFbufGet and PFbufAlloc . Page Replacement Strategies play role in only the third case (c) of PFbufInternalAlloc.
3) Whenever a buffer-page is used, it is inserted into the list using PFbufLinkHead(PFbpage * bpage) which links the buffer-page to the head of the list.
4) PFbufLinkHead is used in PFbufInternalAlloc (the page is newly created) , or after PFbufUnlink(..) in PFbufUnfix(..) and PFbufUsed(..)

Most Recently Used (MRU) :
The most recently used page is thrown first.
Implementation (Different from what we had proposed earlier to simplify code):

1) While choosing a buffer page to replace from the buffer, rather than looping from the last page in the list, we find the first unfixed page from the front of the list. As most recent pages are added to the front of the list, we achieve our target of removing most recent page.

First-in first-out (FIFO) :
The first page to enter the list is thrown the first.
Implementation :

1) In the Least Recently Used strategy, we avoid unlinking and relinking i.e. PFbufUnlink(..) followed by PFbufLinkHead(..) . The corresponding codes are put in an if condition in PFbufUnfix(..) and PFbufUsed(..).
2) Hence the list ordering will remaing same as done during PFbufInternAlloc() and will give us FIFO replacement strategy.

Least Frequently Used (LFU) :
The least frequently used page is thrown first.
Implementation :

1) An extra variable corresponding to the frequency of the use of the page is added to PFbpage. This variable is incremented every time a call to PFbufGet(..)/PFbufAllocate(..) is made.
2) While choosing the page to be replaced in PFbufInternAlloc , we traverse through the complete list and find out the page least frequently used and replace it.

Random Strategy :
A page is selected randomly for throwing.
Implementation :

1) While choosing the page to be replaced in PFbufInternAlloc we choose one at random and then replace it.

Belady's Algorithm (Optimal Algorithm) :
This is the best possible page replacement algorithm however almost impossible to implement. The pages are sorted in the order of their requirement in future, and the one required last is thrown first.
Implementation : The algorithm is unrealizable. At the time of page fault, the future list of page accesses is a must making it an offline algorithm. However, as the query calls are dynamic, this future list of pages is not possible.

What we did : We implemented it assuming the queries are known before hand and for only page accesses (pflayer). Assuming this, a separate function : mygetpage(..) was written in mytestpf.c . This function finds out whether page replacement is necessary. If so the farthest page in the page buffer is found out. All the pages other than this are fixed using PFbufFix(..) and then this page is replaced by the function PFbufGet(..) as it is the only unfixed page. After this all the pages are unfixed again using PFbufUnfixed(..) .

Select Operation Implementation

Method 1 : Scanning through the complete pages

We did this using just the pflayer.

1) A file was created.
2) A certain number of pages were written in the file
3) In this we assumed that one tuple == one page. This is a valid assumption as for accessing a tuple, a complete page needs to be brought into the buffer. Also two tuples access from the same page can be simulated as accessing the same page again.

Method 2 : Using B+ tree index to access tuples

A separate "dblayer" was created using the methods of amlayer and pflayer. We implemented functions which can write records into a file on the drive. The writing of records on the file was done using functions which initialise page header and the pages, add records to the file and access records from the file , given the record number.

1) We assume that we know the schema of the relation beforehand. In the project , we used a table with 5 attributes.
2) The tuples were read from a file and stored in the database. Simultaneously a B+ tree was formed on one of the attribute say "A".
3) The select and join operations scan the so formed B+ tree for a key-value(of attribute "A") to get the record ids in successive manner. The getRecord(..)

function implemented in the db layer then returns the char buffer containing the attribute values for that record id, which is processed (by the knowledge of attributes' properties from before) to get the appropriate values.

Data Structure:

For Method 1: We directly access the pages from file. The data structure is same as already implemented in pf layer.

For Method 2: The structure of files, used to hold the records, follow the standard unix format. We have implemented functions to modify and access the contents of data to hold the records.

Join Operation Implementation:

The records for each table are accessed in the same way as the Select operation. However instead of one, we will be accessing two relations simultaneously in a nested loop fashion:
When following method 1 of data access, for each page of table 1 , all pages of table 2 are accessed sequentially.
When following method 2 of data access, for each tuple in table 1, we access all tuples in table 2 sequentially. For each pair, join conditions are applied and output relation is obtained.

Inputs/Outputs:
Join :
<u>Accessing by pages</u> : We created two files each having 100 pages each. A scan is executed over the first file with another loop for the second file being executed for every page of first file.
<u>Using B+ tree index</u> :

<u>Access by tuples:</u> We created two database files, each having 1000 entries which are generated by us. Each record has 5 attributes: 2 varchar(50) and 3 varchar(10), giving a size of 130B for each record. The schema for both database files is same.

For each tuple in first table, a search in B+ tree is done, which returns recId. recId lookup is then done from the database file. For each of these tuple, a similar lookup is done for records in the second table.

Select :
Accessing by pages : The performance of various strategies highly dependent on access pattern. Belady obviously performs significantly better than others !
TODO : I guess we can remove this and instead give file names.
Using B+ tree index :
Same access procedure is done as for join operation, except that this time records are accessed from only one table.

Performance Measures

Count of Miss and Access:
Count of Misses was done by incrementing a Global MISS counter every time a page was replaced successfully in PFbufInternAlloc(..).
Count of Accesses was done by incrementing a Global ACCESS counter every time a page was asked i.e. every time PFbufGet(..) or PFbufAlloc(..) is called.

Time Measure:
We measured time taken to perform select/join operations by taking an average over a large (100 in our case) no. of observations.

Observations:

Accessing By Pages :
Join :

Algo	ACCESS	MISS	MISS RATE	TIME TAKEN
LRU	20000	10100	0.505	18.65695958

MRU	20000	8300	0.415	12.96839236
LFU	20000	9929	0.496450	18.76869916
FIFO	20000	10100	0.505	16.2733364
BELADY	20000	8299	0.414950	133.23863749
RANDOM	20000	~9500	~0.4750	18.93926625

Select : We analysed this in the following two ways :

a. We found out access patterns where a strategy performs better than every thing else. These files are lru_input.txt having lru as optimal strategy mru_input.txt has optimal strategy for mru and so on ….
b. We generated randomly 100 inputs for select and took an average over them for MISS and time taken.

Algo	ACCESS	MISS	TIME TAKEN(ms)
LRU	100	82.99	0.11297944
MRU	100	82.57	0.18378263
LFU	100	81.87	0.1658869
FIFO	100	82.16	0.11084317
BELADY	100	64.36	0.32624959
RANDOM	100	82.35	0.14534716

Accessing By Tuples :
Join:

Algo	ACCESS	MISS	MISS RATE
MRU	482003	145143	0.301125
LRU	482003	32	0.000066
LFU	482003	488	0.001012
FIFO	482003	82	0.000170
RANDOM	482003	~115	0.000239

Select:

Algo	ACCESS	MISS	MISS RATE
MRU	1203	17	0.014131
LRU	1203	17	0.014131
LFU	1203	17	0.014131
FIFO	1203	17	0.014131
RANDOM	1203	17	0.014131

Conclusion:
Join :

1) Scanning through both the tables : In this case MRU performs the best and gives same performance as belady algorithm.
2) Using B+ tree index to retrieve pages : In this case MRU doesn't perform well. Instead, LRU turns out to be a much better strategy.
   Analysis:

The reason MRU doesn't perform well in this case is because there is an intermediate access to the pages of B+ tree.
Suppose at some instant the buffer is full. The most recently used page in the buffer is of the page which was accessed for the record of second table. To get the next record id, the page of B+ tree(leaf node) needs to be accessed, thus it replaces the above mentioned page. The next record access from the file replaces this B+ page again. This keeps on happening for each record access.

In truth , the B+ page needs to be accessed for a large number of records(corresponding to the key-values stored in this page). Thus it needs to be loaded onto buffer only once. The MRU replacement strategy forces this page to get unloaded from buffer on every record access , causing a huge number of buffer misses.

In case of LRU, when a B+ page is loaded, it becomes the most recently used, and it is ensured that it won't be replaced. Hence the problems of MRU strategy are not faced here.

Select :

1) Scanning through both the tables : No clear winner. Different strategies perform better in different access patterns.
2) Using B+ tree index to retrieve pages : All strategy perform equally well.

References :

1) http://www.ijetch.org/papers/218-T697.pdf
2) https://en.wikipedia.org/wiki/Main_Page
3) http://www.cs.utexas.edu/users/witchel/372/lectures/16.PageReplacementAlgos.pdf
4) http://www.jonahharris.com/osdb/ToyDB/