DATABASE MANAGEMENT SYSTEMS LAB

Assignment - 5 (Group Project)

Database Buffer Manager with Different Buffer Replacement Policies

Team members

Soni Aditya Bharatbhai (20CS10060)
Pranav Mehrotra (20CS10085)
Saransh Sharma (20CS30065)
Pranav Nyati (20CS30037)
Anand Manojkumar Parikh (20CS10007)

Overview of operation of the Buffer Manager

The buffer manager is the layer of the Database Management System that fetches data from the disk into memory. It also decides which parts of data to evict (write back into the disk) to create space for new data using various algorithms.
We have simulated memory access by in-memory arrays and disk access by file read/writes.

Our Buffer Manager operates under certain assumptions and provides functionalities as follows:

- The upper layer of the DBMS demands pages, uniquely identified by file pointer (FILE *) and page number (no. of page inside the file, 0-indexed).
- One out of 3 replacement algorithms can be set as per choice (again, by the upper layer), which are
    - Clock Replacement
    - LRU (Least Recently Used)
    - MRU (Most Recently Used)
- The page pointers returned when a page is accessed (may have to be read into memory) are always pinned by default. Once the upper layer is done with it, it is their responsibility to unpin the page. Only then, will the buffer manager be allowed to evict the page out of memory (write it back to the disk).

Page structure:

```
class Page{
        private:
                FILE *fptr;
                int page_num;
                bool ref_bit;
                bool pin;
                Page();
                ~Page();
        public:
                void *disk_block;
                friend class clock_buffer_manager;
                friend class lru_buffer_manager;
                friend class mru_buffer_manager;
};
```

Brief explanation:

- FILE * fptr : Pointer to actual physical file it resides in
- int page_num : The page number of the page within the file
- bool ref_bit : Used by Clock Replacement Algorithm
- bool pin : Marks whether the page is pinned or unpinned
- void * disk_block : pointer to the in-memory buffer used to store the page
- The constructor and destructor are made private and the buffer manager classes are made friend classes for security reasons, so that no other classes (belonging to higher system layers / user defined) can create/destroy a Page

LRU (Least Recently Used) Scheme:

The LRU scheme is implemented by the class lru_buffer_manager–

```
class lru_buffer_manager{
        private:
                FILE *log_ptr;
                unsigned int num_bufs;
                int buf_cnt;
                list <Page*> buf_pool;
                unordered_map <pair<FILE*,int>,list<Page*>::iterator,hash_struct> buf_map;
        public:
                int accesses, disk_reads;
                lru_buffer_manager(unsigned int n);
                ~lru_buffer_manager();
                Page *read_page(FILE *f,int page_number);
                void unpin_page(Page* p);
};
```

The attributes and methods are described below:

Attributes:

1. FILE * log_ptr : Pointer to the log file for printing system logs
2. uint num_bufs : The maximum number of pages the buffer can hold
3. int buf_cnt : The current number of pages stored in the buffer.
4. list <Page *> buf_pool : An STL std::list<>. Basically a doubly-linked list of Page pointers, always maintained in sorted order by time it was last accessed.

5. unordered_map <pair<FILE*,int>,list<Page*>::iterator,hash_struct> buf_map : An STL std::unordered_map<>. It is used for efficiently locating a page residing in the buf_pool, by mapping the key (file pointer , page number) to the iterator.
6. int accesses : The number of times a requested page was found already residing in memory (number of page-hits, used for statistical maintenance)
7. int disk_reads : The number of times a requested page had to be fetched from the disk (number of page-misses, used for statistical maintenance)
8. void unpin_page(Page * p) : Mark the given page as unpinned

The reasons for the above choice become clear when the methods are explained

Methods:

1. lru_buffer_manager::read_page :
   Responds to page requests. If the page is found in memory by checking in buf_pool, then a pointer to the Page object is returned.
   Otherwise, check if there is a free frame available to load the page into and load the page by accessing the given file and page number and return the pointer to the page object. Put this page in the front of the list.
   If buf_pool is full and the page is not found, then the LRU replacement algorithm is applied:
   - Start iterating over the list of pages from oldest to newest, until an unpinned page is found. If all pages are pinned, return NULL, indicating that the buffer is full (failure).
   - Else, remove the oldest unpinned page (efficiently by unlinking from the linked list and resetting the pointers) and insert it at the front. The data inside the page is written back to the old file it was pointing to. Data from the requested file and page number is now written in this page and the pointer to it is returned.

<u>MRU (Most Recently Used) Scheme</u>:

This is a simple modification to the LRU scheme.
All data structures and algorithms are nearly identical, with the only difference being that when the buffer is full and a page-miss occurs, then the Most Recently Used (newest) unpinned page is evicted from memory and written back into the file, and this frame is now available as a fresh page. (similar class definition)

Clock Replacement Scheme:

The Clock Replacement scheme is implemented by the class clock_buffer_manager–

```
class clock_buffer_manager{
        private:
                FILE *log_ptr;
                int clock_hand;
                unsigned int num_bufs;
                int buf_cnt;
                unordered_map <pair<FILE*,int>,int,hash_struct> buf_map;
                Page *buf_pool;
                int replace_page(FILE *f,int page_number);
        public:
                int accesses,disk_reads;
                clock_buffer_manager(unsigned int n);
                ~clock_buffer_manager();
                Page *read_page(FILE *f, int page_number);
                void unpin_page(Page *p);
};
```

The attributes and methods are described below:

Attributes:

1. FILE * log_ptr : Pointer to the log file for printing system logs
2. int clock_hand: The current position of the clock in the buf_pool.
3. uint num_bufs : The maximum number of pages the buffer can hold
4. int buf_cnt : The current number of pages in the buffer.
5. unordered_map <pair<FILE*,int>,int,hash_struct> buf_map : An STL std::unordered_map<>. It is used for efficiently locating a page residing in the buf_pool, by mapping the key (file pointer , page number) to the index in the buf_pool array.
6. Page *buf_pool : An array of Pages, basically the circular buffer in which the clock replacement algorithm will be implemented.
7. int accesses : The number of times a requested page was found already residing in memory (number of page-hits, used for statistical maintenance)
8. int disk_reads : The number of times a requested page had to be fetched from the disk (number of page-misses, used for statistical maintenance)

Methods:

1. clock_buffer_manager::replace_page :
   It is called when the read_page wants to replace some other page from the buffer, because there is no space left for the requested page. It applies the Clock Replacement Algorithm, by incrementing the clock_hand and checking the reference bits and if it is set, then resetting it and continuing; else checking if the block is pinned or not. If the block is pinned, continue; else Overwrite the Page, by loading the requested page into the memory of this index in the buffer. It returns the clock_hand index after successful loading, else on error it returns -1.
2. clock_buffer_manager::read_page :
   Responds to page requests. If the page is found in memory by checking in buf_pool, then a pointer to the Page object is returned.
   Otherwise, check if there is a free frame available to load the page into and load the page by accessing the given file and page number and return the pointer to the page object.
   If buf_pool is full and the page is not found, then the Clock Replacement algorithm is applied:
   - Calls replace_page, and gets the index, at which the Page is loaded or -1 if error.
   - Erases the previous page entry from buf_map, and inserts new page entry into buf_map
   - Updates all the meta data of the page entry, like pin and reference bits.
3. clock_buffer_manager::unpin_page :
   Mark the given page as unpinned.

Main Function to test:

- To simulate the working of the buffer, we use files as a disc containing the database. File read would logically imply disc I/O in this case.

- To test the working of LRU, clock, MRU buffer managers, the main.cpp file has been written. The main function first asks the user which replacement technique he wants to check and then asks for the nature of the query.

- The function supports 2 types of query i.e. selection of name and join on roll number. The two databases file used contains data about DBMS marks i.e. dbms.txt (64 byte tuples, used for selection) and networks marks i.e. networks.txt (32 byte tuple, used as second parameter for join).

- Dbms database schema includes: sr number, roll number, name, class test marks and mid-sem marks.

- Networks database schema includes: sr number, roll number, mid sem marks and class test marks.

- For a select query, every page of the DBMS database is loaded into the buffer, every tuple's name is matched to the input name given by the user and the output of this comparison is printed on the terminal.

- For join operation, block nested join is used. One page of DBMS database and one page of Networks database is loaded into the buffer, every tuple of DBMS page is then compared against every tuple of network's page and the compatible tuples are printed in output.txt.

- The number of disc accesses and number of buffer hits are then printed.

- **Page Hit and Disk I/O Statistics for the different Page Replacement Algorithms:**

(A) Keeping Number of Pages in Buffer Fixed and varying the Page Size:

(1) SELECT Query:

- The three different page replacement algorithms (LRU, MRU and Clock Replacement algorithms, give **same number of Page Hit and Disk I/O on a SELECT query** for a particular student's name in the dataset, as it involves fetching only 1 page (the page containing the tuple) and is same for any tuple in the relation.
- The number of page hits and miss for the different page sizes are given below along with the plot:

PAGE SIZE:	64	128	192	256	320	384	448	512
BUFF_HITS:	0	0	0	0	0	0	0	0
DISK_I/O :	96	48	32	24	19	16	13	12

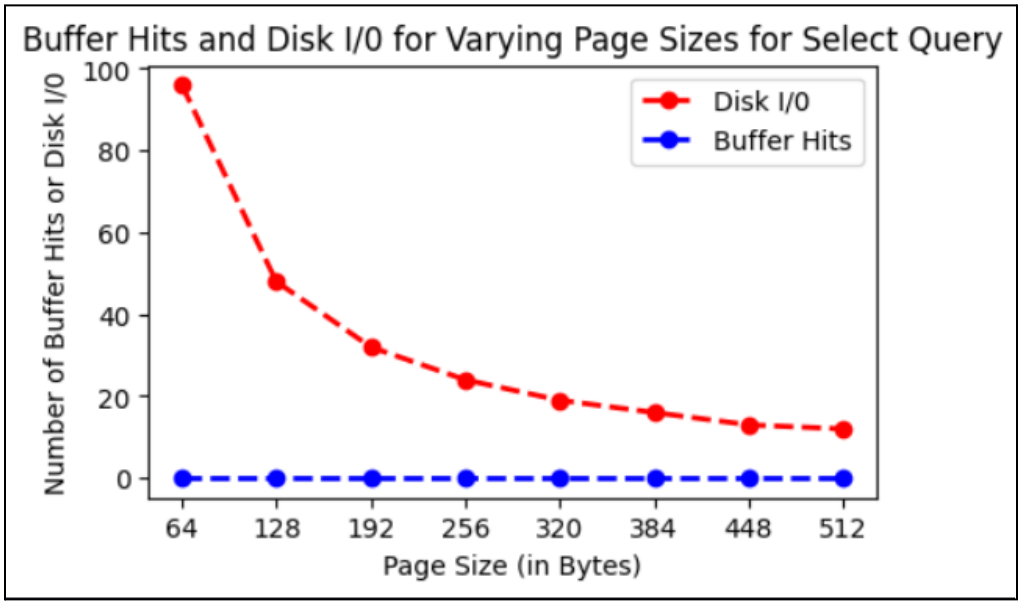

Fig1. Plot showing Buffer Hits and Disk I/0 for Varying Page Sizes for Select Query

- The no of buffer hits are 0 always as the first time a query is run, there is no page from the relation already in the database buffer, and has to be fetched from the file

(2) JOIN Query:

- The three different page replacement algorithms (LRU, MRU and Clock Replacement algorithms, give **different of Page Hit and Disk I/O on a JOIN query** on the two relations as it involves comparing each tuple of first relation with each tuple of the 2nd relation to check if both have the same **Roll_No attribute (**inner join on Roll_no attribute**)** . So multiple pages are fetched one after the other into the buffer and there are high chances of page hits as some of the pages required at a later stage in the two while loops for comparing each tuple of 1st relation with each tuple of the 2nd relation may already exists in the buffer from some previous stage .

- **Join Query Statistics for Clock Replacement Algorithm:**
  The number of page hits and miss for the different page sizes for JOIN operation in CLOCK REPLACEMENT are given below along with the plot:

PAGE SIZE:	128	256	384	512	640	768	896	1024	1152	1280
BUFF_HITS:	0	0	0	10	21	18	9	9	6	4
DISK_I/O :	1200	312	144	74	24	22	15	15	9	8

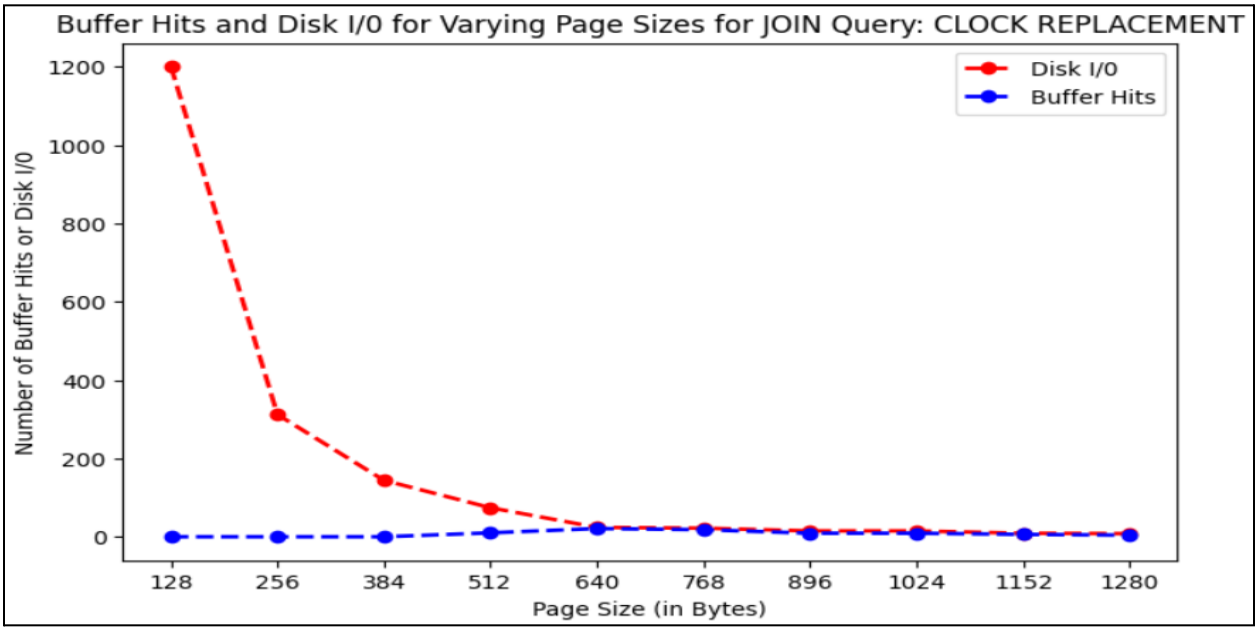

Fig2. Plot showing Buffer Hits and Disk I/0 for Varying Page Sizes for Join Query for Clock Replacement Algorithm

- **Join Query Statistics for LRU Replacement Algorithm:**
  The number of page hits and miss for the different page sizes for JOIN operation in LRU REPLACEMENT are given below along with the plot:

PAGE SIZE:	128	256	384	512	640	768	896	1024	1152	1280
BUFF_HITS:	0	0	0	0	32	28	15	15	8	6
DISK_I/O :	1200	312	144	84	13	12	9	9	7	6

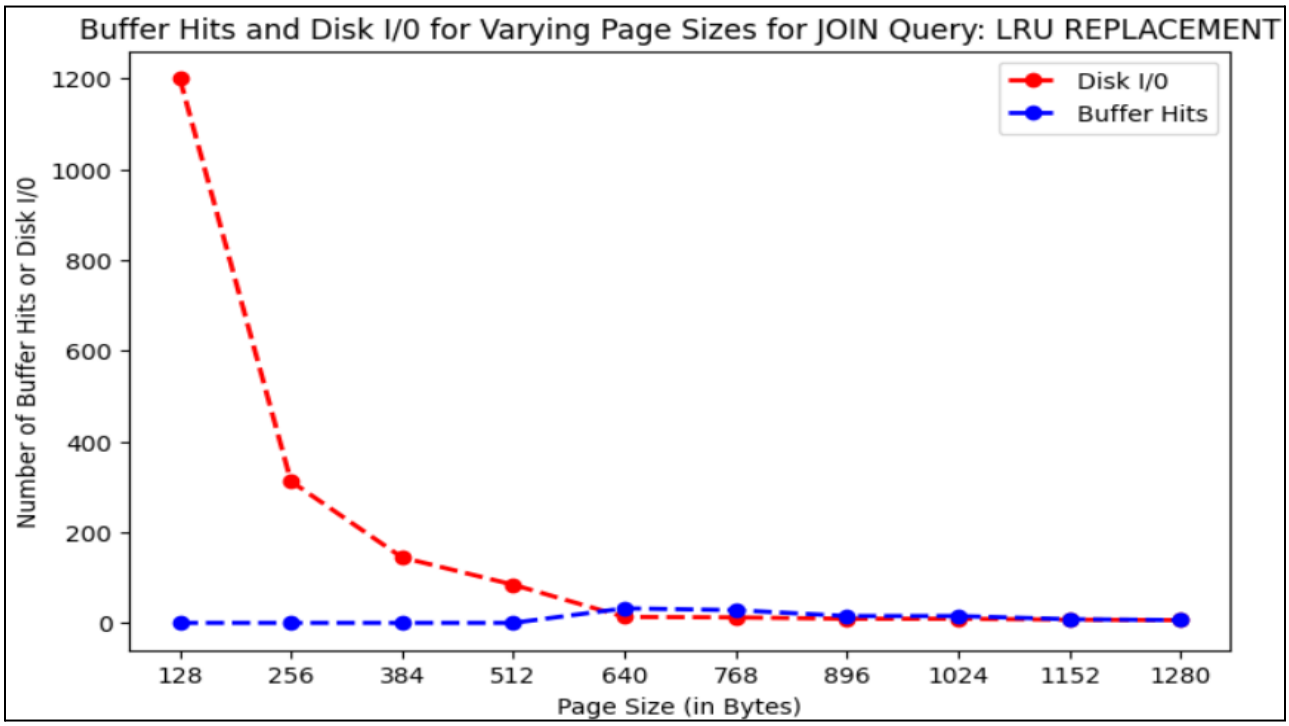

Fig3. Plot showing Buffer Hits and Disk I/0 for Varying Page Sizes for Join Query for LRU Replacement Algorithm

- **Join Query Statistics for MRU Replacement Algorithm:**
  The number of page hits and miss for the different page sizes for JOIN operation in MRU REPLACEMENT are given below along with the plot:

PAGE SIZE:	128	256	384	512	640	768	896	1024	1152	1280
BUFF_HITS:	6	6	6	6	6	6	6	6	5	5
DISK_I/O :	1194	306	138	78	39	34	18	18	10	7

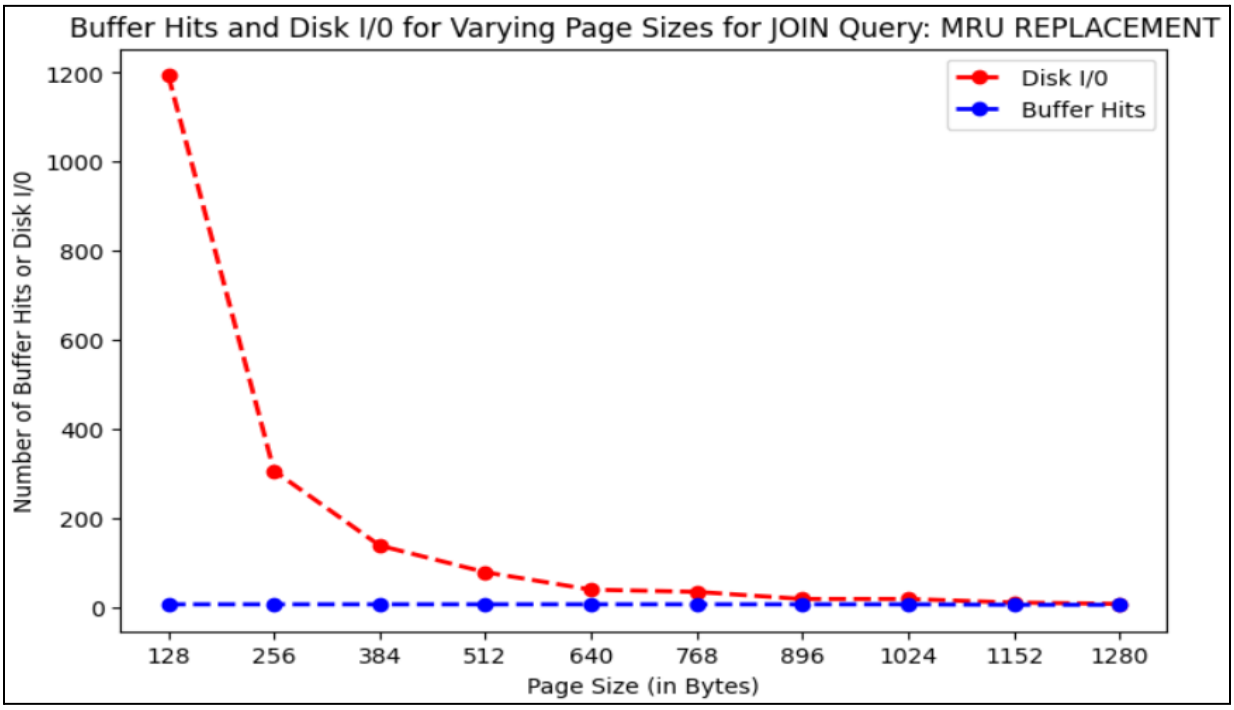

Fig4. Plot showing Buffer Hits and Disk I/0 for Varying Page Sizes for Join Query for MRU Replacement Algorithm

(B) Keeping Page Size Fixed and varying the Number of Pages in Buffer (Buffer Size):

(1) SELECT Query:

- The three different page replacement algorithms (LRU, MRU and Clock Replacement algorithms, give **same number of Page Hit and Disk I/O on a SELECT query** for a particular student's name in the dataset, as it involves fetching only 1 page (the page containing the tuple) and is same for any tuple in the relation.

- The number of page hits and miss for the different buffer size (different no of pages in buffer) with **fixed page size = 512 bytes** are given below along with the plot:

BUFF SIZE:	4	5	6	7	8	9
BUFF_HITS:	0	0	0	0	0	0
DISK_I/O :	12	12	12	12	12	12

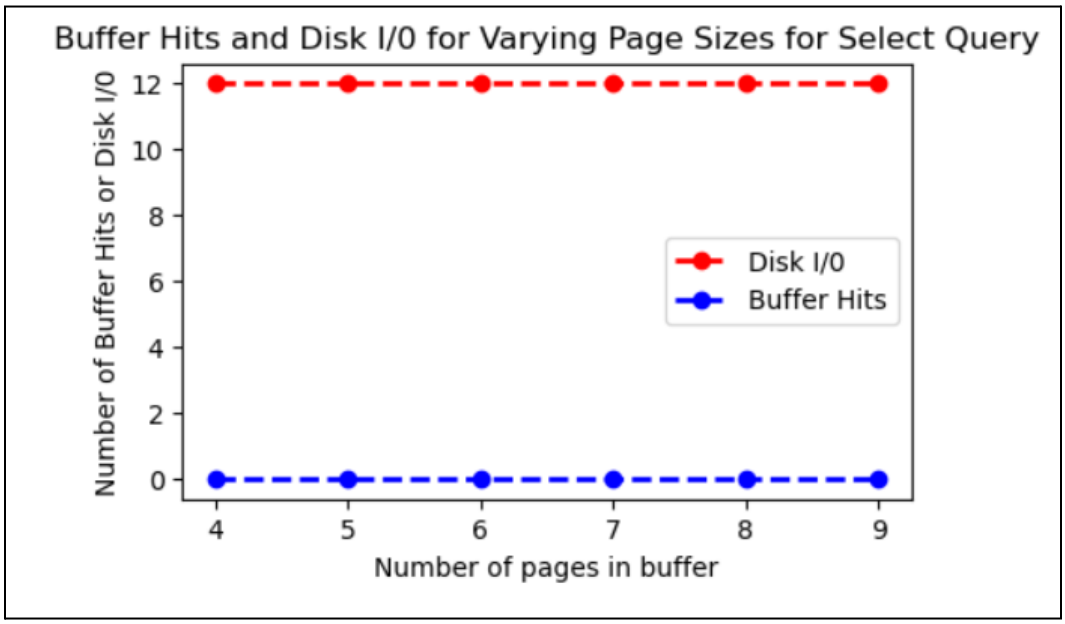

Fig5. Plot showing Buffer Hits and Disk I/0 for Varying Buffer Size for Select Query

(2) JOIN Query:

- **Join Query Statistics for Clock Replacement Algorithm:**
  The number of page hits and miss for the different buffer size (different no of pages in buffer) with **fixed page size = 512 bytes** for JOIN operation in CLOCK REPLACEMENT are given below along with the plot:

BUFF SIZE:	4	5	6	7	8	9
BUFF_HITS:	0	10	11	40	42	60
DISK_I/O :	84	74	73	44	42	24

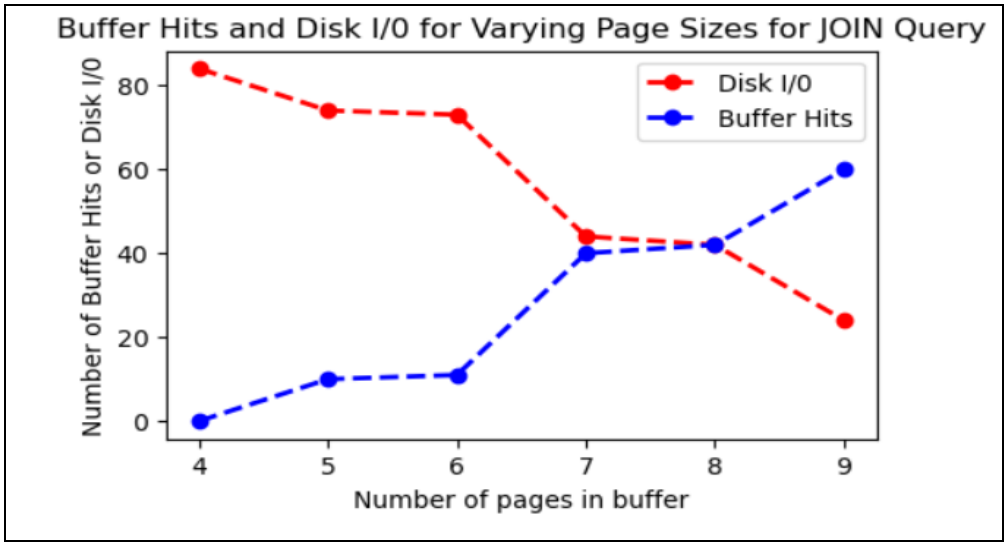

Fig6. Plot showing Buffer Hits and Disk I/0 for Varying Buffer Sizes for Join Query for Clock Replacement Algorithm

- **Join Query Statistics for LRU Replacement Algorithm:**
  The number of page hits and miss for the different buffer size (different no of pages in buffer) with **fixed page size = 512 bytes** sizes for JOIN operation in LRU REPLACEMENT are given below along with the plot:

BUFF_SIZE:	4	5	6	7	8	9
BUFF_HITS:	0	0	0	66	66	66
DISK_I/O :	84	84	84	18	18	18

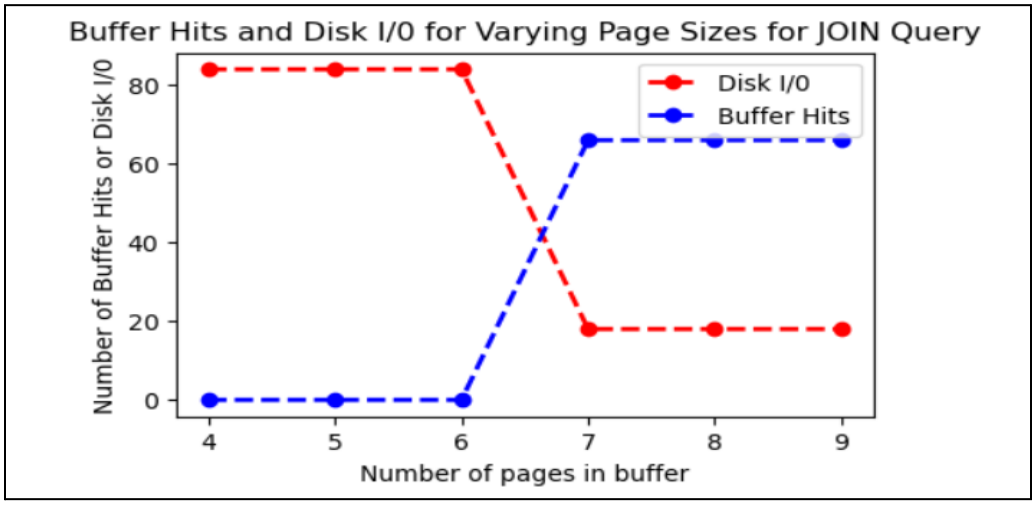

Fig7. Plot showing Buffer Hits and Disk I/0 for Varying Buffer Sizes for Join Query for LRU Replacement Algorithm

- **Join Query Statistics for MRU Replacement Algorithm:**
  The number of page hits and miss for the different buffer size (different no of pages in buffer) with **fixed page size = 512 bytes** sizes for JOIN operation in MRU REPLACEMENT are given below along with the plot:

BUFF_SIZE:	4	5	6	7	8	9
BUFF_HITS:	3	6	10	15	21	27
DISK_I/O :	81	78	74	69	63	57

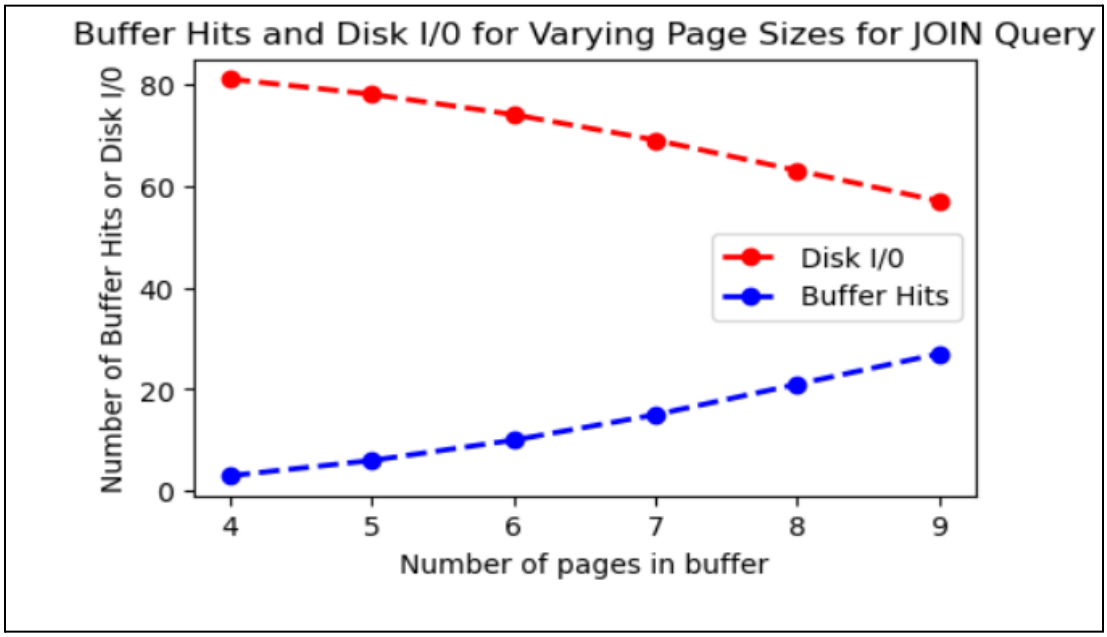

Fig8. Plot showing Buffer Hits and Disk I/0 for Varying Buffer Sizes for Join Query for MRU Replacement Algorithm