Indian Institute of Technology, Kharagpur
Department of Computer Science and Engineering

Database Management Systems Laboratory

IMPLEMENTATION OF MANUAL MEMORY MANAGEMENT FOR EFFICIENT CODING

16th April, 2023

Shivam Raj
20CS10056

Jatin Gupta
20CS10087

Kushaz Sehgal
20CS30030

Aniket Kumar
20CS10083

Rushil Venkateswar
20CS30045

Contents

1. Project Report

1.1 Objective

The objective of this project is to simulate a small buffer pool for simple Join/Selection queries on few small tables using C/C++ language. Popular buffer manager strategies like LRU, MRU and CLOCK are implemented and compared in terms of the number of disk I/O required.

1.2 Abstract

A buffer pool is a crucial component of database systems, as it allows for the efficient storage and retrieval of data. The buffer pool functions as a cache for frequently accessed data pages, which are temporarily stored in memory, rather than being constantly read and written to and from the disk. This improves system performance, as accessing data from memory is significantly faster than accessing data from the disk.

The buffer manager is responsible for managing the buffer pool and its contents. When a query needs a specific data page, the buffer manager first checks if the page is present in the buffer pool. If the page is found, it can be immediately returned to the query without requiring any disk I/O operations, which can be time-consuming and resource-intensive. However, if the page is not present in the buffer pool, the buffer manager must read the page from the disk and add it to the buffer pool.

As the buffer pool has a finite size, the buffer manager needs to replace an existing page with a new page if the buffer pool becomes full. The process of choosing which page to evict from the buffer pool is known as buffer replacement. There are several buffer replacement policies, each with its unique strengths and weaknesses.

The Least Recently Used (LRU) policy replaces the least recently accessed page in the buffer pool. This policy assumes that pages that have not been accessed for a long time are less likely to be accessed again in the near future. The Most Recently Used (MRU) policy, on the other hand, replaces the most recently accessed page in the buffer pool. This policy assumes that pages that have been recently accessed are more likely to be accessed again in the near future.

The CLOCK policy is a compromise between LRU and MRU. It maintains a circular list of pages in the buffer pool and uses a clock hand to traverse the list. When a page needs to be replaced, the clock hand is moved forward until a page with a reference bit set to zero is found. The reference bit is set to one each time a page is accessed. This policy ensures that pages that have been accessed recently are not immediately evicted from the buffer pool, but also that pages that have not been accessed for a long time are eventually evicted.

In conclusion, selecting the appropriate buffer replacement policy is critical in optimizing database system performance. The choice of policy should depend on the database workload and the system's memory resources.

1.3 Implementation

This project involves the development of a buffer pool simulation using the C++ programming language, alongside the creation of a basic database schema consisting of a Person table with attributes including name, age, and marks, containing a total of 2000 entries. The page size is set at 4096 bytes, and the buffer pool size is configurable at runtime by the user.

The primary objective of this project is to evaluate the performance of different buffer management policies against a workload of join and selection queries. In order to achieve this, each buffer management policy - namely Least Recently Used (LRU), Most Recently Used (MRU), and CLOCK - is implemented, tested, and compared against the query workload.

The project serves as a practical exploration of buffer pool simulation and buffer management

policies within the context of a simple database schema and query workload. Its formal nature stems from its rigorous and structured approach to software development, testing, and evaluation.

1.3.1 Data Structures used

Student

Database table. The structure is as follows:

```
typedef struct{
    char name[20];
    int Grade;
    int roll_no;
}Student;
```

Data Members

- *name:* name of the student
- *Grade:* class of student
- *roll_ no:* roll number of the student

Frame

Frames are used to store the pages in the memory. The Frame class is as follows:

```
class Frame{
    private :
        int numPage ;
        char * dataOfPage ;
        FILE * filePointer ;
        bool isPinned ;
        bool chance_second ;
        void setFrame ( FILE * filePointer , int numPage , char * dataOfPage , bool
            isPinned);
        void unpinFrame () ;
    public :
         Frame () ;
         Frame ( const Frame & f ) ;
         ~Frame () ;
         friend class LRUManage ;
         friend class ClockManager ;
         friend class MRUManager ;
};
```

Data Members

- *numPage:* stores the page number of page
- *dataOfPage:* pointer to character array which stores the daata in page
- *filePointer:* a pointer to a FILE object that represents the file to which this page belongs to
- *isPinned:* a boolean variable that indicates whether the page is isPinned in memory or not.
- *chance_ second:* variable used in clock replacement.

Functions

- *setFrame:* a private member function that sets the data members of the Frame object.
- *unpinFrame:* a private member function that unpins the frame.

- *Frame:* a constructor that initializes the data members of the Frame object.
- *Frame const Frame& f :* a copy constructor that creates a copy of the Frame object.
- *~Frame():* a destructor that deallocates the memory allocated to the dataOfPage pointer.
- *friend class LRUManage:* declares the LRUManage class as a friend of the Frame class.
- *friend class ClockManager:* declares the ClockManager class as a friend of the Frame class.
- *friend class MRUManager:* declares the MRUManager class as a friend of the Frame class.

BufferStatistics
Accounts the number of numAccesss and disk reads. The structure is as follows:

```
class BufferStatistics{
    public :
    int numAccesss ;
    int NumDiskReads ;
    int NumPgHits ;

    BufferStatistics () ;
    void clear () ;
};
```

Data Members

- *numAccesss:* an integer that keeps track of the total number of page numAccesss made
- *NumDiskReads:* an integer that keeps track of the total number of page reads from disk made
- *NumPgHits:* an integer that keeps track of the total number of page hits

Functions

- *BufferStatistics():* a default constructor that initializes all data members to 0.
- *clear():* a function that sets all data members to 0, effectively resetting the buffer statistics.

ReplacementPolicy
It is used to get new pages, replace pages, unpin pages and access BufferStatistics

```
class ReplacementPolicy{
    public :
    virtual ~ ReplacementPolicy () {}
    virtual char * getPage ( FILE * filePointer , int numPage ) = 0;
    virtual void unpinPage ( FILE * filePointer , int numPage ) = 0;
    virtual BufferStatistics getStats () = 0;
    virtual void clearStats () = 0;
};
```

1.4 MRU Policy

The MRU (Most Recently Used) policy is a page replacement algorithm used in the buffer pool of a database management system. This algorithm replaces the page that was accessed the most recently and hasn't been accessed since, in order to make room for new pages to be loaded into memory. The assumption behind this policy is that recently accessed pages are less likely to be accessed again in the near future.

The buffer pool has a fixed size, which is determined by the number of frames that can be fit into it. A frame is a unit of memory that can hold a single page. The MRU policy maintains

a list of frames, called "mru", which is sorted in the order of their usage. The "mp" is a mapping data structure used to keep track of whether a given page number is present in the buffer pool.

When a page is accessed, the MRU policy first checks the "mp" data structure to see if the page is already present in one of the frames. If the page is present, it is removed from its current position in the "mru" list and inserted at the front of the list, indicating that it has been accessed most recently. If the page is not present in any of the frames, the policy removes the first (unpinned) frame in the list and replaces it with the new page. The recently replaced (unpinned) frame is then inserted at the front of the list to indicate that it has been accessed most recently. The "mp" data structure is updated accordingly.

In summary, the MRU policy is a page replacement algorithm that works by replacing the most recently used page in the buffer pool. It maintains a list of frames sorted in the order of their usage, and uses a mapping data structure to keep track of the pages present in the pool. The policy ensures that the most recently used pages are given priority in the buffer pool, with the assumption that they are less likely to be accessed again in the near future.

1.4.1 Data Members:

- *numberOfFrames:* This is an integer variable that represents the number of frames that can be fit in the buffer pool.
- *mru:* This is a linked list of Frame objects that is used to implement the MRU algorithm for managing the buffer pool. It represents the actual buffer pool.
- *mp:* The "mp" data structure is an unordered map that is used to determine whether a given page is present in the buffer pool. It maps a pair of a file pointer and a page number to an iterator that points to the corresponding Frame object in the "mru" list. The "PairHash" class is a hash function object that is used to compute the hash value for the pair.

  In other words, the "mp" data structure is a mapping between a pair of a file pointer and a page number, and the corresponding frame in the buffer pool. The pair serves as a unique identifier for a page in the buffer pool, and the iterator points to the location of the frame that holds the page. This mapping is implemented as an unordered map, which provides fast lookup and insertion of key-value pairs.

  To compute the hash value for the pair, the "PairHash" class defines a hash function that takes in a pair of a file pointer and a page number, and returns a hash value. This hash value is used by the unordered map to determine the location of the corresponding iterator in the hash table. The "PairHash" class may use a combination of the hash values of the file pointer and the page number to compute the final hash value, depending on the specific implementation.
- *stats:* This is an object of the BufferStatistics class that stores statistics about the buffer pool.

1.4.2 Functions:

- *MRUManager(int numberOfFrames):* This is the constructor of the class, which takes an integer argument representing the number of frames in the buffer pool.
- *getPage(FILE* filePointer, int numPage):* The "mp" data structure is an unordered map that is used to determine whether a given page is present in the buffer pool. It maps a pair of a file pointer and a page number to an iterator that points to the corresponding Frame object in the "mru" list. The "PairHash" class is a hash function object that is used to compute the hash value for the pair.

  In other words, the "mp" data structure is a mapping between a pair of a file pointer and a

page number, and the corresponding frame in the buffer pool. The pair serves as a unique identifier for a page in the buffer pool, and the iterator points to the location of the frame that holds the page. This mapping is implemented as an unordered map, which provides fast lookup and insertion of key-value pairs.

To compute the hash value for the pair, the "PairHash" class defines a hash function that takes in a pair of a file pointer and a page number, and returns a hash value. This hash value is used by the unordered map to determine the location of the corresponding iterator in the hash table. The "PairHash" class may use a combination of the hash values of the file pointer and the page number to compute the final hash value, depending on the specific implementation.

- *~MRUManager():* This is the destructor of the class.
- *getStats():* This function returns the statistics about the buffer pool.
- *unpinPage(FILE* filePointer, int numPage):* The "unpinPage" function is used to remove the pin from a page in the buffer pool, which allows the page to be replaced by other pages if necessary. The function takes two arguments: "FILE* filePointer", which represents the file from which the page was fetched, and "int numPage", which represents the page number of the page that needs to be unpinned.

  When a page is isPinned, it means that it is currently in use by a transaction and cannot be replaced by other pages in the buffer pool. When the transaction is finished with the page, it needs to be unpinned so that it can be replaced if necessary. The "unpinPage" function performs this task by finding the frame that corresponds to the given file pointer and page number in the "mp" data structure, and then decrementing the pin count for that frame.

  If the pin count for the frame reaches zero after the decrement, it means that the page is no longer in use by any transactions, and can be replaced by other pages if necessary. At this point, the frame is marked as "unpinned" in the "mru" list, which allows it to be replaced by other pages during page replacement. If the pin count is still greater than zero after the decrement, it means that the page is still in use by one or more transactions, and cannot be replaced by other pages yet. In this case, the function simply returns without doing anything else.

  In summary, the "unpinPage" function is used to remove the pin from a page in the buffer pool, which allows it to be replaced by other pages if necessary. It does this by decrementing the pin count for the corresponding frame in the "mp" data structure, and marking the frame as "unpinned" in the "mru" list if the pin count reaches zero. The function takes two arguments: "FILE* filePointer", which represents the file from which the page was fetched, and "int numPage", which represents the page number of the page that needs to be unpinned.

1.5 LRU Policy

```
class LRUManage : public ReplacementPolicy
{
    private :
    int numberOfFrames ;
    list < Frame > lru ;
    unordered_map < pair < FILE * , int > , list < Frame >:: iterator , PairHash > mp ;
    BufferStatistics stats ;
    public :
    LRUManage (int numberOfFrames ) ;
    char * getPage ( FILE * filePointer , int numPage ) ;
    ~ LRUManage () ;
    BufferStatistics getStats () ;
    void clearStats () ;
```

```
    void unpinPage ( FILE * filePointer , int numPage ) ;
};
```

The LRU policy works by replacing the least recently used page in the buffer pool.

In other words, the page that was accessed the earliest and hasn't been accessed since is replaced. This policy is based on the assumption that pages that haven't been accessed for a long time are less likely to be accessed in the future.

The text describes a memory management strategy called "Least Recently Used (LRU)" which is commonly used in operating systems to manage the memory of computer programs.

In this strategy, a pool of memory frames is maintained, and the frames are allocated to different processes as needed. The "numberOfFrames" variable represents the maximum number of frames that can be allocated in the pool.

The "lru" list is used to keep track of the frames in the pool and their usage history. The frames are sorted in the order of their usage, with the most recently used frames at the beginning of the list and the least recently used frames at the end of the list.

The "mp" variable is used to check whether a specific page of a file is present in a frame or not. If it is present, then the frame is moved to the front of the "lru" list since it was recently used. If the page is not present, then the least recently used frame (i.e., the frame at the end of the "lru" list) is removed from the pool and replaced with the new page. This replacement process is also known as "page replacement."

Once a frame is replaced, it is moved to the front of the "lru" list since it has become the most recently used frame. This ensures that the least recently used frame is always at the end of the list and can be easily identified for replacement.

Overall, the LRU strategy is an effective way to manage memory in operating systems since it optimizes the use of memory frames and ensures that the most frequently used pages are always kept in memory for faster access.

1.5.1 Data Members

- *numberOfFrames* : an integer that represents the number of frames that can be fit in the pool.
- *lru*: a list used to implement LRU (Least Recently Used) page replacement policy.
- *mp* : an unordered map that is used to identify whether a page is present in the buffer or not. It maps a pair of a File pointer and numPage to an iterator in the lru list.
- stats: an object of type BufferStatistics that contains statistics about buffer usage.

1.5.2 Functions

- *LRUManage(int numberOfFrames)* : a constructor that takes an integer argument numberOfFrames and initializes the numberOfFrames member variable.
- *getPage(FILE* filePointer, int numPage)* : This function requires a FILE* pointer called filePointer and an integer numPage as inputs, and returns a pointer to a character buffer. Its purpose is to fetch a page from the buffer. If the requested page is already present in the buffer, it is moved to the front of the least recently used (lru) list, and the corresponding iterator in the map (mp) is updated accordingly. If the page is not currently in the buffer

and there is available space, it is retrieved from disk, added to the front of the lru list, and a new entry is added to the mp map. In case there's no space in the buffer, the least recently used page is replaced with the new page. Additionally, this function updates the statistics in the stats object.

- *LRUManage()* : a destructor that deallocates memory used by the buffer.
- *getStats()* : a function that returns the statistics stored in the stats object.
- *clearStats()*: a function that resets the statistics stored in the stats object to zero.
- *unpinPage(FILE* filePointer, int numPage)*: a function that unpins a page in the buffer. It sets the isPinned flag of the corresponding Frame object to false.

1.6 Clock Policy

```
class ClockManager : public ReplacementPolicy
{
    private :
    int numberOfFrames ;
    Frame * bufferList ;
    int ClockedFrame ;
    BufferStatistics stats ;
    int numPages ;

    public :
    ClockManager (int numberOfFrames ) ;
    char * getPage ( FILE * filePointer , int numPage ) ;
    ~ ClockManager () ;
    void unpinPage ( FILE * filePointer , int numPage ) ;
    BufferStatistics getStats () ;
    void clearStats () ;
};
```

The CLOCK algorithm utilizes a circular buffer to monitor the pages in the buffer pool. Each page is assigned a reference bit that is set to 1 when the page is accessed. The buffer manager starts with a clock hand pointing to the first page in the circular buffer. If the reference bit is 1, it is changed to 0, and the clock hand moves to the next page. If the reference bit is 0, the page is swapped out, and the new page is added to the buffer pool.

To access a page, we search through the buffer pool to see if the page is present. If the number of pages in memory is less than the number of frames, a new page is added at the end. If the frame size is full, we iterate over the buffer pool. We continue the iteration until we find a page that is unpinned and not marked as having a second chance. At that point, we change the second chance status to False and set the second chance of the given frame as False. If we find an unpinned frame with a second chance status of False, we replace it with the new page.

1.6.1 Data Members

- *numberOfFrames*: This is an integer variable that represents the number of frames that can be fit in the buffer pool.
- *bufferList*: This is a dynamic array of Frame objects that is used to implement the clock algorithm for managing the buffer pool. It represents the actual buffer pool.
- *clock hand*: This is an integer variable that represents the clock hand of the clock algorithm. It is used to keep track of the next frame to be replaced.
- *stats*: This is an object of the BufferStatistics class that stores statistics about the buffer pool.
- *numPages*: This is an integer variable that represents the number of pages in the buffer pool.

1.6.2 Functions

- *ClockManager(int numberOfFrames)*: This is the constructor of the class, which takes an integer argument representing the number of frames in the buffer pool.
- *getPage(FILE* filePointer, int numPage)*: This function is used to get a page from the buffer pool. It takes two arguments: FILE* filePointer represents the file from which the page needs to be fetched, and int numPage represents the page number of the page that needs to be fetched. It returns a pointer to the page data.
- $\sim$*ClockManager()*: This is the destructor of the class.
- *unpinPage(FILE* filePointer, int numPage)*: This function is used to unpin a page from the buffer pool. It takes two arguments: FILE* filePointer represents the file from which the page was fetched, and int numPage represents the page number of the page that needs to be unpinned.
- *getStats()*: This function returns the statistics about the buffer pool.
- *clearStats()*: This function is used to clear the statistics of the buffer pool.

1.7 Resuts

A study was conducted to compare the performance of three buffer management strategies: LRU, MRU, and CLOCK. To evaluate the strategies, JOIN and SELECTION queries were executed on a table, and the number of disk I/O operations required to complete each query and page hits were recorded. The number of disk I/O operations was averaged over multiple iterations to measure the performance of each strategy.

For SELECTION queries, all three strategies performed equally well, requiring the least number of disk I/O operations. This can be attributed to the fact that SELECT queries always require new pages, irrespective of the replacement algorithm or buffer pool size. Therefore, the number of page hits remained zero, and the number of disk reads stayed at 7, which is the size of the database.

1.7.1 SELECT QUERY

1.7.2 JOIN QUERY

For JOIN queries, MRU was found to be the best performing strategy, requiring the least number of disk I/O operations. MRU maintains a queue of pages that were most recently used and evicts the most recently used page when the buffer pool is full. This strategy works well for join queries because they access a large amount of data, and pages that were accessed most recently are less likely to be needed again soon.

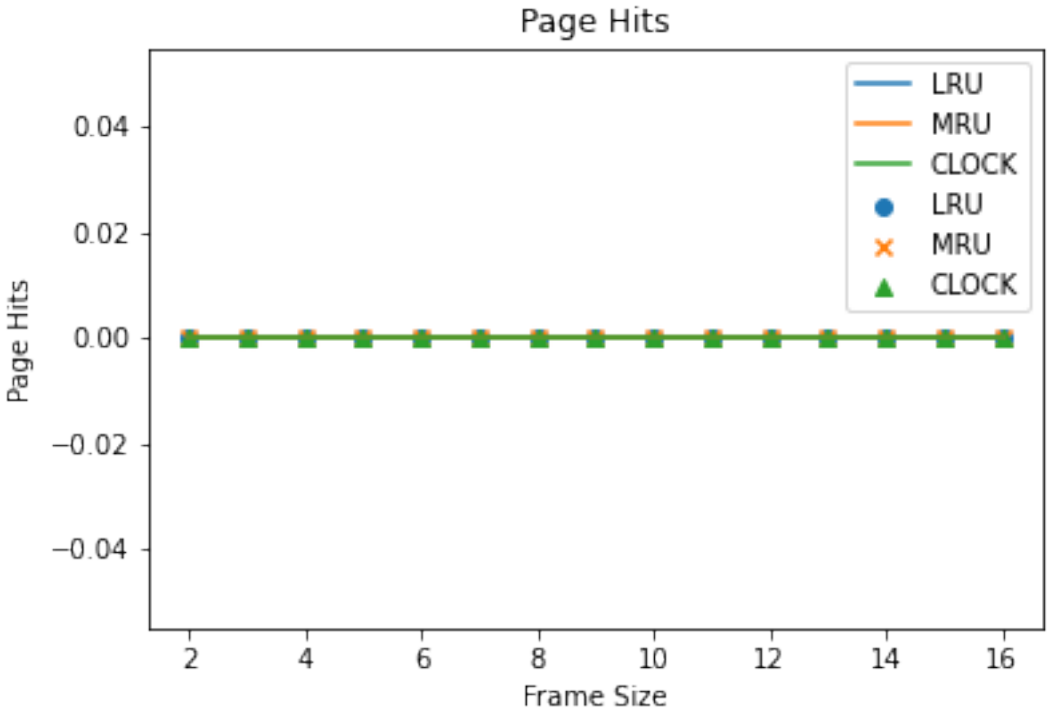

Figure 1: Select - Page Hits

CLOCK performed as well as LRU in most cases, but it did not perform as well on some queries. CLOCK maintains a circular list of pages and evicts the oldest page that has not been referenced recently. This strategy may work well in some cases, but it did not perform as well on join queries as compared to MRU.

In summary, the study suggests that LRU and MRU are good choices for optimizing the performance of selection and join queries, respectively. CLOCK is also a fair choice, but may not perform as well as LRU or MRU in all cases.

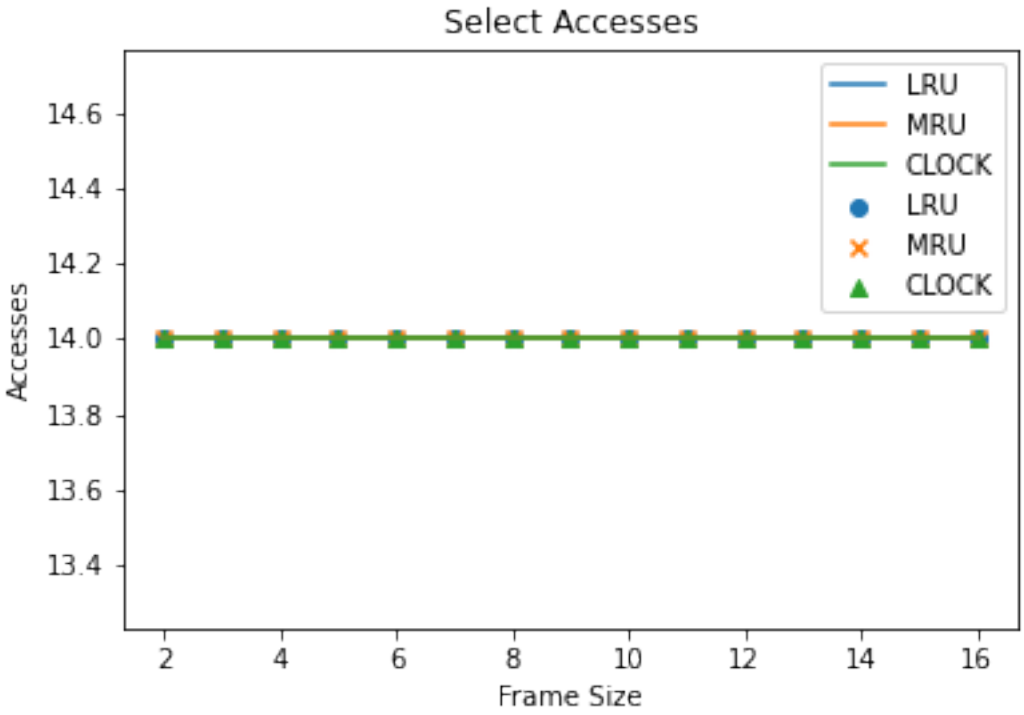

Figure 2: Select - Accesses

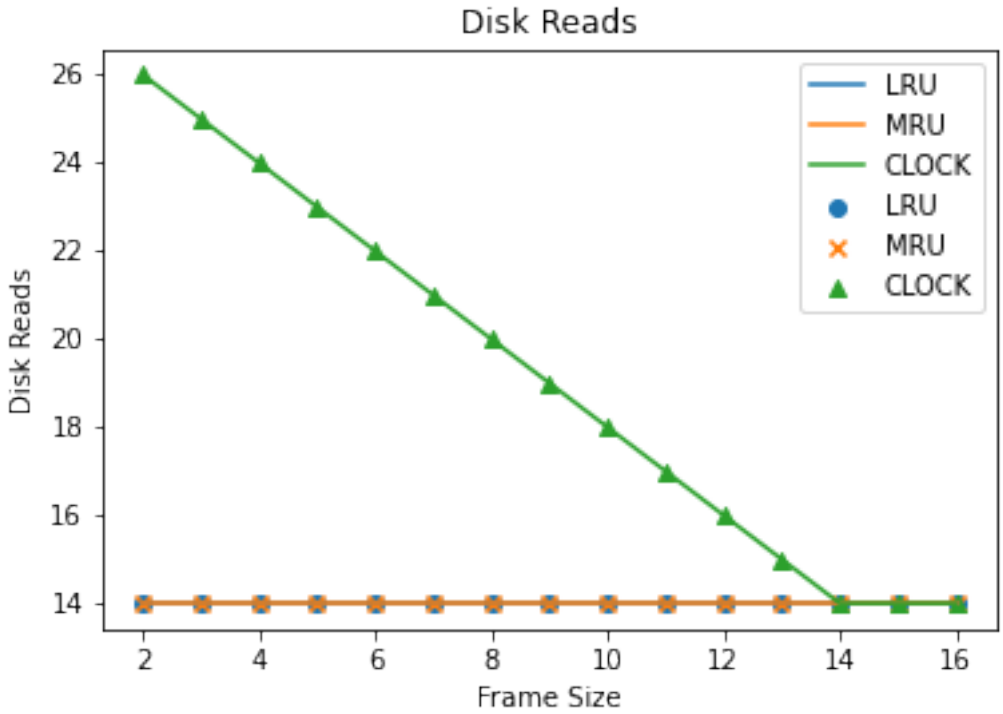

Figure 3: Select - Disk Reads

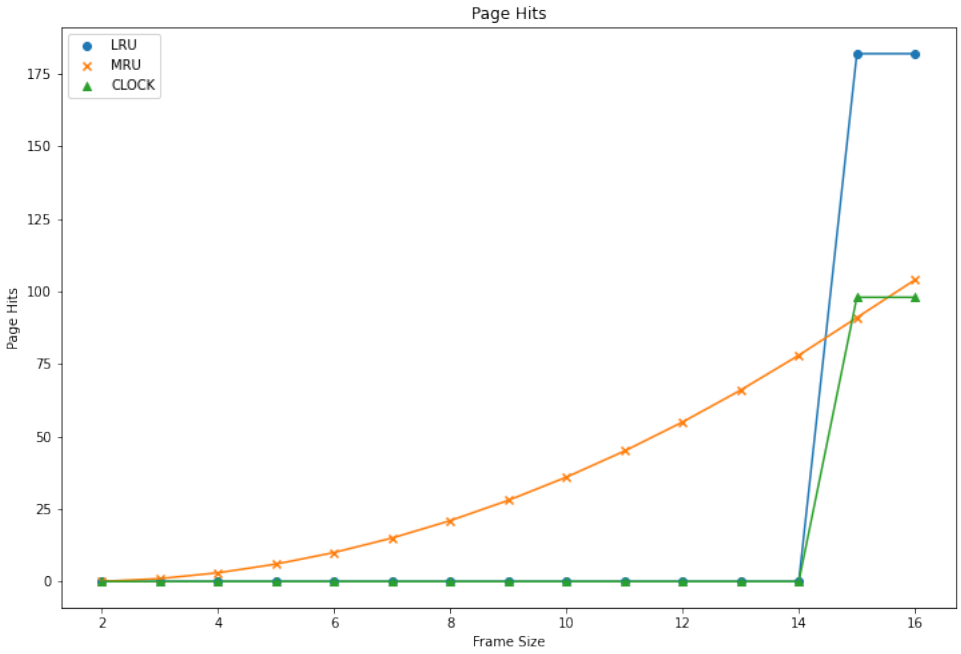

Figure 4: JOIN - Page Hits

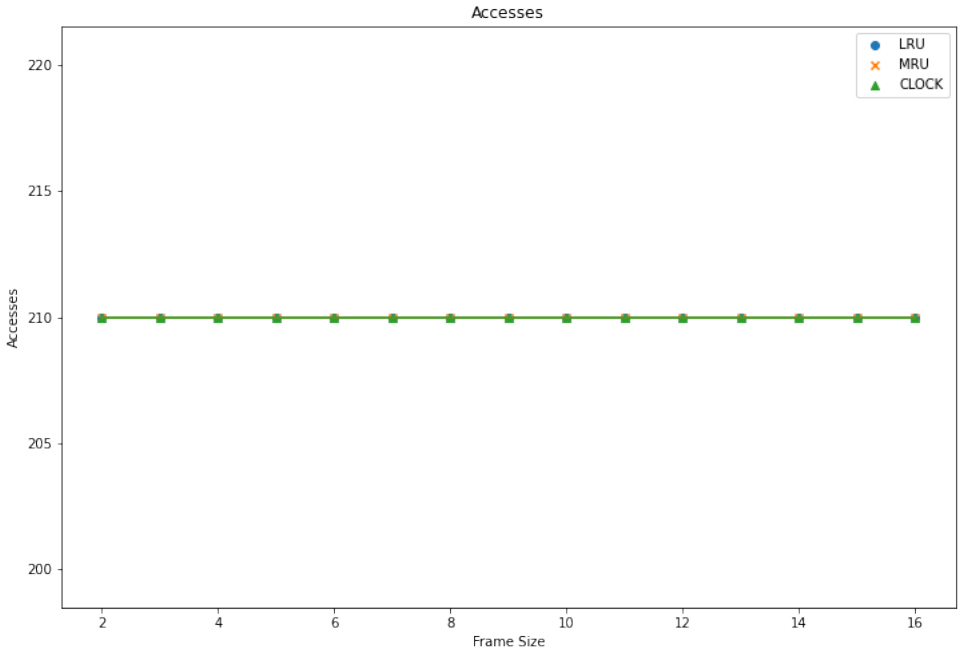

Figure 5: JOIN - Accesses

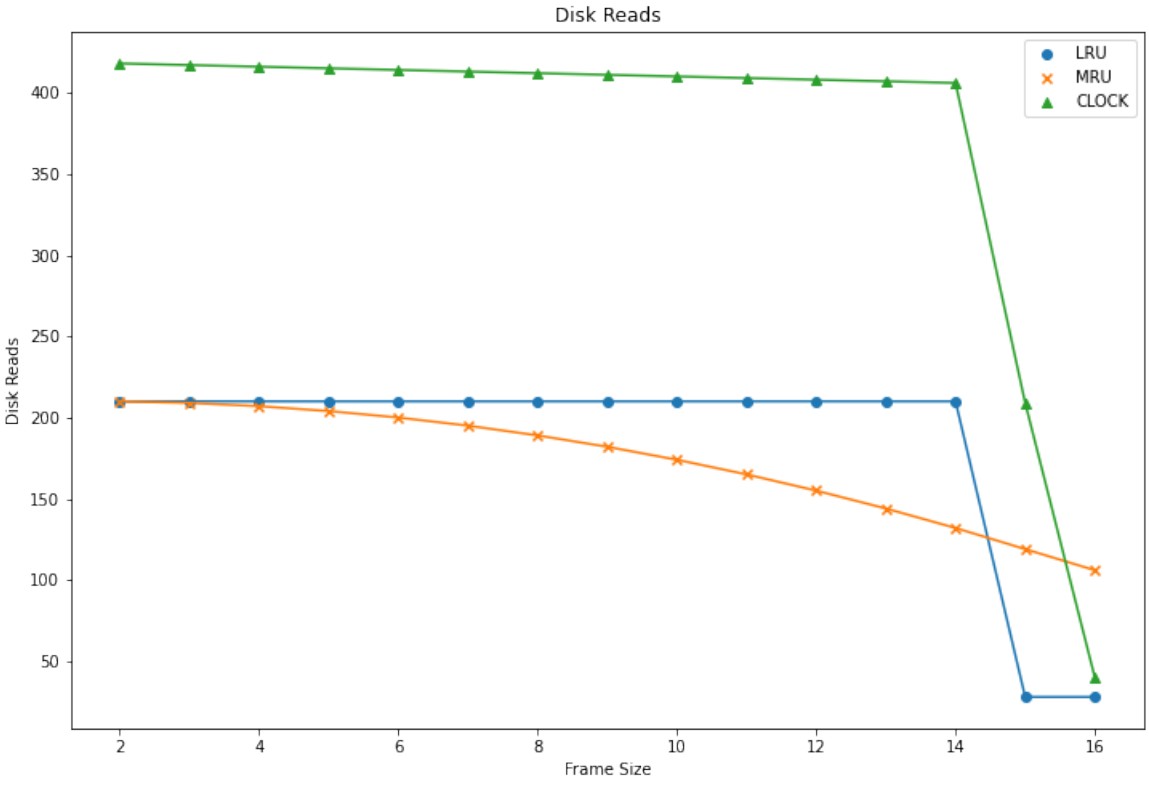

Figure 6: JOIN - Disk Reads