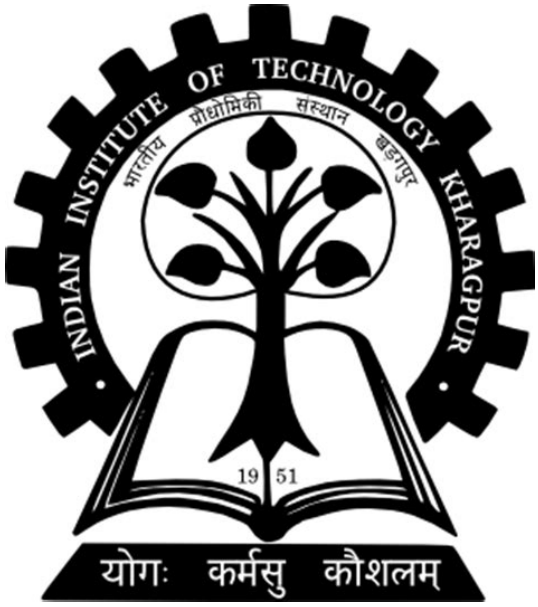
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Term Project

(Database Management Systems Laboratory)

Team **React-ers**

Abhay Kumar Keshari 20CS10001

Likhith Reddy Moreddigari 20CS10037

Shivansh Shukla 20CS10057

Venkata Sai Suvvari 20CS10067

Shashank Goud Boorgu 20CS30013

Chapter 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 Background

A buffer pool is a reserved memory space for storing data temporarily to improve database system performance. It works by caching data pages that are frequently accessed by the queries. When a query needs a specific data page, the buffer manager searches for the page in the buffer pool. If the page is found, it is returned to the query without performing any disk I/O. If the page is not in the buffer pool, the buffer manager reads it from the disk and adds it to the buffer pool. The buffer manager needs to replace an existing page in the buffer pool if the buffer pool is full.

There are several buffer replacement policies, such as LRU (Least Recently Used), MRU (Most Recently Used), CLOCK. Each of these policies has its strengths and weaknesses, and it's important to choose the right policy depending on the database workload.

1.3 Implementation

The buffer pool simulation is implemented in C++ language. A simple database schema is created, consisting of a table Person with attributes name,age and marks. There are 1000 entries in the table. The Page size is 4096 bytes .The buffer pool size is a variable and given at the time of input by the user. Each buffer management policy is implemented and tested against a simple join/selection query workload.

Frame

The Frame structure:

```
1 class Frame
2 {
```

```
    private:
    int pageNum;        // page number of page
    char* pageData;     // data in page
    FILE *fp;           // file to which this page belongs to
    bool pinned;        // either pinned or unpinned
    bool second_chance;         // for clock replacement algorithm

    void setFrame(FILE*fp, int pageNum, char* pageData, bool pinned);
    void unpinFrame();


    public:
    Frame();
    Frame(const Frame& f);
    ~Frame();
    friend class LRUBufferManager;
    friend class ClockBufferManager;
    friend class MRUBufferManager;
};
```

We are using Frames in the Memory to store the pages and some headers.

Data Members

- **pageNum:** an integer that stores the page number of the page.
- **pageData:** a pointer to a character array that stores the data in the page.
- **fp:** a pointer to a FILE object that represents the file to which this page belongs to.
- **pinned:** a boolean variable that indicates whether the page is pinned in memory or not.
- **second_chance:** a boolean variable that is used in clock replacement algorithm.

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 pageData pointer.
- **friend class LRUBufferManager:** declares the LRUBufferManager class as a friend of the Frame class.
- **friend class ClockBufferManager:** declares the ClockBufferManager class as a friend of the Frame class.
- **friend class MRUBufferManager:** declares the MRUBufferManager class as a friend of the Frame class.

BufStats

```
class BufStats
{
    public:
    int accesses;
    int diskreads;
    int pageHits;

    BufStats();
    void clear();
};
```

It contains how many access and disk reads are made.

Data Members

- **accesses:** an integer that keeps track of the total number of page accesses made
- **diskreads:** an integer that keeps track of the total number of page reads from disk made
- **pageHits:** an integer that keeps track of the total number of page hits, i.e. number of times a page was found in the buffer cache and did not require a disk read.

Functions

- **BufStats():** 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.

Replacement Policy

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

It has the virtual methods to replace pages, unpin pages, get a new page, get and clear the Buf stats.

1.3.1 LRU Policy

```
class LRUBufferManager: public ReplacementPolicy
{
    private:
    int numFrames;    // number of frames that can be fit in pool
```

```
    list<Frame> lru;  // list to implement LRU
    unordered_map<pair<FILE*, int>, list<Frame>::iterator, PairHash> mp;   //
   map to identify whether a page is present in buffer or not
    BufStats stats;

    public:
    LRUBufferManager(int numFrames);
    char* getPage(FILE*fp, int pageNum);
    ~LRUBufferManager();
    BufStats getStats();
    void clearStats();
    void unpinPage(FILE*fp, int pageNum);
};
```

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 numFrames is the number of frames that can be fit in the pool. lru is a list which contains various frames sorted in the order of their usage. mp is used for checking whether there is a frame with the given page number of a file. If it is present then present in some frame, we will remove it from the list and insert it at the front. If it is not present then we would remove the last frame(unpinned). The least recently used one will be present at the last. Hence when a new page has to be loaded into the frame, this last unpinned frame will be replaced and it will be inserted at the front of the list. And the mp is updated appropriately.

Data Members

- **numFrames:** 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 pageNum to an iterator in the lru list.
- **stats:** an object of type BufStats that contains statistics about buffer usage.

Functions

- **LRUBufferManager(int numFrames):** a constructor that takes an integer argument numFrames and initializes the numFrames member variable.
- **getPage(FILE* fp, int pageNum):** a function that takes a FILE* pointer fp and an integer pageNum as arguments and returns a pointer to a character buffer. This function is used to retrieve a page from the buffer. If the page is already in the buffer, it is moved to the front of the lru list, and the corresponding iterator in the mp map is updated. If the page is not in the buffer, and there is space available in the buffer, the page is read from disk, added to the front of the lru list, and a new entry is added to the mp map. If there is no space available in the buffer, the least recently used page is evicted and replaced with the new page. This function also updates the statistics in the stats object.

- ~ **LRUBufferManager():** 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* fp, int pageNum):** a function that unpins a page in the buffer. It sets the pinned flag of the corresponding Frame object to false.

1.3.2 MRU Policy

```
class MRUBufferManager: public ReplacementPolicy
{
    private:
    int numFrames;    // number of frames that can be fit in pool
    list<Frame> mru;  // list to implement MRU
    unordered_map<pair<FILE*, int>, list<Frame>::iterator, PairHash> mp;   //
   map to identify whether a page is present in buffer or not
    BufStats stats;

    public:
    MRUBufferManager(int numFrames);
    char* getPage(FILE*fp, int pageNum);
    ~MRUBufferManager();
    BufStats getStats();
    void clearStats();
    void unpinPage(FILE*fp, int pageNum);
};
```

The MRU policy works by replacing the most recently used page in the buffer pool. In other words, the page that was accessed the most recently and hasn't been accessed since is replaced. This policy is based on the assumption that pages that were recently accessed are less likely to be accessed again in the near future.

The numFrames is the number of frames that can be fit in the pool. mru is a list which contains various frames sorted in the order of their usage. mp is used for checking whether there is a frame with the given page number of a file. When we want to access a page it is checked in the mp, if it is present in some frame, we will remove it from the list and insert it at the front. If it is not present then we would remove the first frame(unpinned). The most recently used one will be present at the front. Hence when a new page has to be loaded into the frame, this recently used (unpinned) frame will be replaced and it will be inserted at the front of the list. And the mp is updated appropriately.

Data Members

- **numFrames:** 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:** This is an unordered map that is used to identify whether a page is present in the buffer pool or not. It maps a pair of a file pointer and a page number to an iterator 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.
- **stats:** This is an object of the BufStats class that stores statistics about the buffer pool.

Functions

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

1.3.3 CLOCK Policy

```
class ClockBufferManager: public ReplacementPolicy
{
    private:
    int numFrames;    // number of frames that can be fit in pool
    Frame* bufferPool;  // list to implement clock
    int clock_hand;   // clock hand
    BufStats stats;
    int numPages;

    public:
    ClockBufferManager(int numFrames);
    char* getPage(FILE*fp, int pageNum);
    ~ClockBufferManager();
    void unpinPage(FILE*fp, int pageNum);
    BufStats getStats();
    void clearStats();
};
```

The CLOCK policy works by using a circular buffer to keep track of the pages in the buffer pool. Each page has 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 set to 0, and the clock hand moves to the next page. If the reference bit is 0, the page is replaced, and the new page is added to the buffer pool.

When we want to access a page, we iterate over the bufferPool to check if the page is present or not. In case if the number of pages present in the memory is less than number of Frames. New page is added at the end. If the frame size is full we iterate over the bufferPool. As long as we do not find the a page which is unpinned and second chance marked as False, we change the second chance status to False and continue the iteration. we set the second chance of the given frame as False. If we find a frame which is unpinned and second status set to False we replace it with the new page.

Data Members

- **numFrames:** This is an integer variable that represents the number of frames that can be fit in the buffer pool.
- **bufferPool:** 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 BufStats class that stores statistics about the buffer pool.
- **numPages:** This is an integer variable that represents the number of pages in the buffer pool.

Functions

- **ClockBufferManager(int numFrames):** This is the constructor of the class, which takes an integer argument representing the number of frames in the buffer pool.
- **getPage(FILE* fp, int pageNum):** This function is used to get a page from the buffer pool. It takes two arguments: FILE* fp represents the file from which the page needs to be fetched, and int pageNum represents the page number of the page that needs to be fetched. It returns a pointer to the page data.
- **∼ClockBufferManager():** This is the destructor of the class.
- **unpinPage(FILE* fp, int pageNum):** This function is used to unpin a page from the buffer pool. It takes two arguments: FILE* fp represents the file from which the page was fetched, and int pageNum 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.

1.4 Results

Each of the buffer management strategies viz., LRU, MRU and CLOCK were tested on a table. For each strategy, JOIN/SELECTION queries were executed 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 is averaged over multiple iterations, as a measure of strategy's performance.

From the results, the three buffer strategies viz., LRU, MRU and Clock performed equally good for selection queries, with the least number of disk I/O operations required. From the fact that everytime a page is used, a new page is to be read again, irrespective of replacement algorithm as select needs new pages always. So, the page hits stay zero and disk reads stay 7, since the size of database is 7 pages, irrespective of replacement policy and buffer pool size.

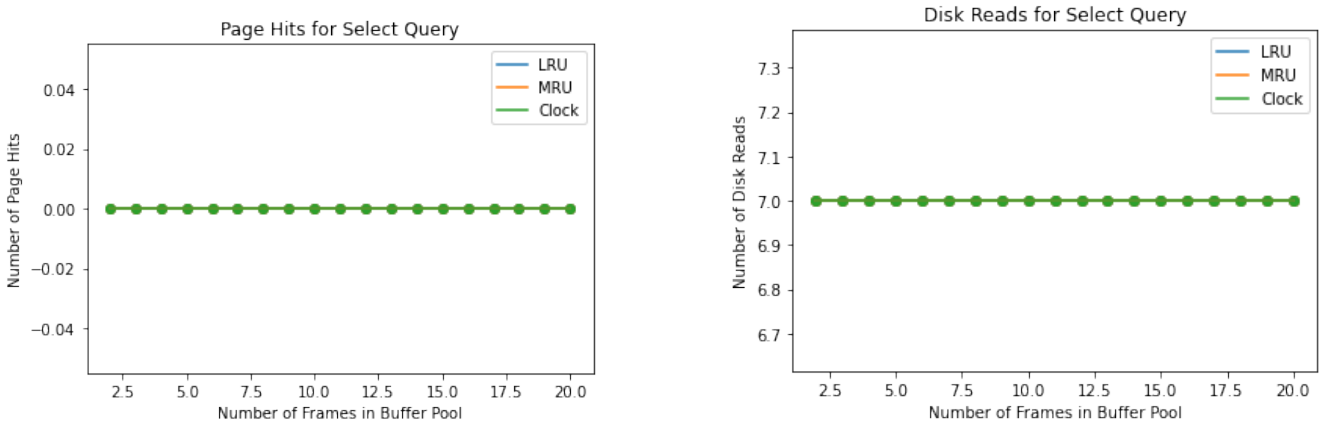


Figure 1.1: Select Query

For join queries, on the other hand, MRU was the best performing strategy, with the least number of disk I/O operations required. 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.

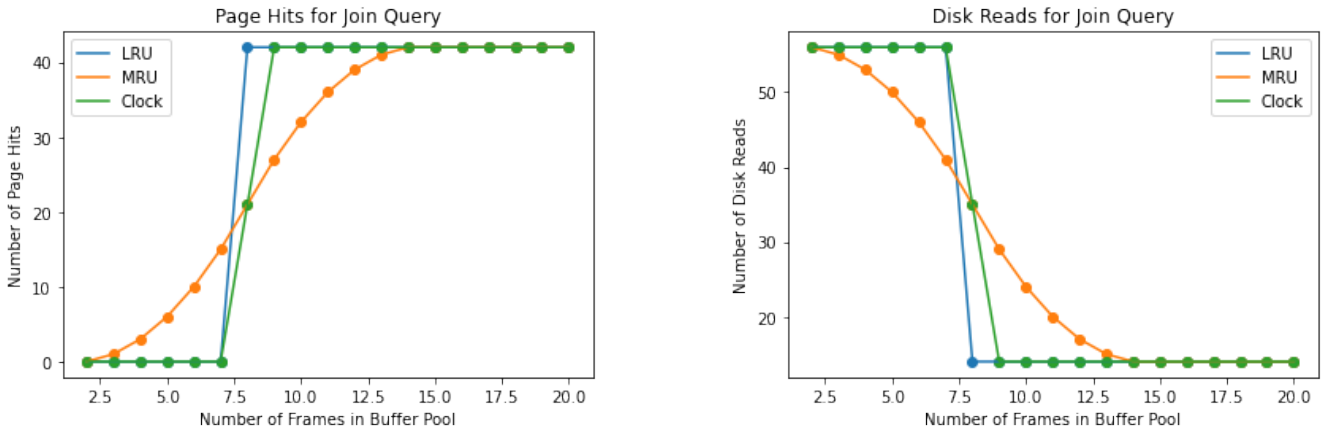


Figure 1.2: Join Query

CLOCK performed as good as LRU with a few exceptions. 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 others.
Overall, our experiments suggest that LRU and MRU are good choices for optimizing the performance of selection and join queries, respectively and Clock is a fair choice.

1.5 References

- https://en.wikipedia.org/wiki/Cache_replacement_policies
- https://web.stanford.edu/class/cs346/2015/notes/Lecture_One.pdf
- https://www.cs.utexas.edu/users/witchel/372/lectures/16.PageReplacementAlgos.pdf
- https://www.cs.cornell.edu/courses/cs4410/2015su/lectures/lec15-replacement.html

Link to the project code: https://github.com/likhnic/DBMS_Project