Final Project Report
Bufferpool Implementation System Project

[Github Repository Link]

Jiayu Lu Joel Franklin Stalin Vijayakumar

Abstract

The performance of database systems heavily relies on the effective management of the buffer pool, which serves as a caching layer between the database and the underlying storage. In this project, we present the design and implementation of an efficient buffer pool management system that aims to optimize the caching of data pages, minimize disk I/O, and improve overall query performance. Our buffer pool manager incorporates several eviction algorithms, including Least Recently Used (LRU), Clean-First LRU (CFLRU), and LRU Write Sequence Reordering (LRUWSR), to adaptively select the best eviction strategy based on the workload characteristics.

We also implement a flexible and modular buffer pool architecture that enables the exploration of various buffer pool sizes, workload configurations, and storage options. To ensure realistic performance evaluation, our project includes a workload generator that produces diverse workloads, capturing both uniform and skewed data access patterns. The workloads are then executed against our buffer pool implementation, and key performance metrics such as latency, buffer hit ratio, read IOs, and write IOs are analyzed.

Our experimental results demonstrate the effectiveness of our buffer pool manager in improving database performance across different workload scenarios. By selecting the appropriate eviction algorithm and buffer pool size, our system can adapt to the data access patterns and storage constraints, leading to reduced latency and minimized disk I/O. The insights gained from this project can be leveraged to design more efficient and scalable database systems, capable of catering to the ever-growing data management requirements of modern applications.

1. Introduction

The bufferpool of a Database Management System (DBMS) is a memory area used to temporarily store a subset of the database data and is used to store frequently or recently accessed data, allowing the DBMS to retrieve data from memory instead of reading it from disk. The bufferpool reduces the number of disk accesses required to retrieve the

data and improves the performance of the database system, as disk access is much slower than memory access.

When data is requested from the database, the DBMS checks to see if the requested data is already present in the buffer pool. If the data is in the buffer pool, it can be retrieved quickly without needing to access the disk. If the data is not in the buffer pool, the DBMS must read it from disk into the buffer pool before it can be accessed.

2. Problem Statement

The bufferpool uses a part of the main memory to store data. Each time an access request comes, the bufferpool returns the page in the memory or brings that page from disk and stores it in the bufferpool. However, if the bufferpool has no space for a new page to write, we have to evict a page and replace that with the page the access requests. With proper page replacement policies, we can optimize the performance of bufferpool by reducing disk access.

The page replacement policy determines the order in which pages are evicted from the buffer pool and written back to disk.

Since pages can be evicted from the buffer pool in any order, the page replacement policy can affect the order in which pages are written back to disk. For example, if the policy is designed to prioritize the eviction of dirty pages, then dirty pages will be written back to disk before clean pages. This can result in more frequent writes to disk, but it ensures that the most up-to-date data is always stored on disk. By controlling the order in which pages are written back to disk, the page replacement policy can impact the performance of the DBMS. A well-designed page replacement policy can help to reduce disk I/O and improve overall system performance.

Our goal is to implement different page replacement policies and understand how they work in a bufferpool and perform a comparative analysis on the three page-replacement policies (LRU, CFLRU, LRU-WSR) based on different workloads with/without simulation on disk. The comparison is based on metrics such as page misses, page hits, total writes and execution latency.

3. Background

We began the project by reading resources on Buffer pool and related concepts such as Page Hits, Page Misses, Clean/Dirty pages. We understood what improves the performance of a bufferpool. We also read about various other additional page replacement algorithms such as LFU and FIFO. We understood the working principles behind LRU, CFLRU and LRU-WSR.

4. Solution Design

4.1. We implemented the following helper functions.

- **search** (arguments - buffer_instance and pageId) - It searches for a particular pageId in the Bufferpool and returns the index of the page in the Bufferpool. If the page is not found, it returns -1.
- **getContentByPageId** (arguments - pageId) - It returns the content stored in the pageId of the Bufferpool.
- **write_back_to_disk** (arguments - pageId and content) - It writes back the updated content to the page in disk having the respective pageId.

4.2. We implemented the read and write functions.

4.2.1. **read** (arguments - buffer_instance, pageId, offset, algorithm, simulation_on_disk, is_write_operation)

- Initially it calls the search helper function to check if the pageId is in the Bufferpool.
- If the pageId is found in the Bufferpool, it returns the index of the pageId in the Bufferpool.
- If the pageId is not found in the Bufferpool and if the Bufferpool has not reached it's maximum size, it pushes the page into the Bufferpool.
- If the pageId is not found in the Bufferpool and if the Bufferpool has reached it's maximum size, it calls the page replacement algorithm, evicts 1 page from Bufferpool and pushes the page of respective pageId into the Bufferpool.
- If the evicted page from Bufferpool is dirty, the helper function **write_back_to_disk** is called to write back the updated content to the page in disk having the respective pageId.
- In all the above 3 cases, it reads the contents at the offset in the page having the respective pageId.

4.2.2. write (arguments - buffer_instance, pageId, offset, new_entry, algorithm, simulation_on_disk)

- Initially it calls the read function which returns the index of the page having the respective pageId in the Bufferpool.
- If simulation_on_disk is true, it updates the content at the offset in the page having the respective pageId in the Bufferpool.

4.3. We implemented 3 page replacement policies namely LRU (Least Recently Used), CFLRU (Clean-First LRU) (2 approaches) and LRU-WSR (LRU Write Sequence

Reordering) page replacement algorithms.

4.3.1. LRU (Least Recently Used)

The principle of LRU (Least Recently Used) page replacement policy is to evict the page from the buffer pool that has not been used for the longest time. In other words, the page that has been accessed the least recently is chosen for eviction. This is based on the assumption that pages that have not been accessed for a long time are less likely to be accessed in the future.

In our implementation, we maintained a *timestamp* attribute for each page element in the bufferpool. Each time we access the page, we update the *timestamp* with a new value which is larger than the last value. In this way, we can keep the page elements in an order of accessed time. When a page needs to be evicted, we simply evict the page with the minimum value of *timestamp*.

The following is the code for LRU:

```
int Buffer::LRU()
{
  int index = 0;
  int min_timestamp = buffer_pool[0].timestamp;

// We return the index of the minimum time stamp
  for (int i = 1; i < buffer_pool.size(); ++i) {
    if (buffer_pool[i].timestamp < min_timestamp) {
      min_timestamp = buffer_pool[i].timestamp;
      index = i;
    }
  }
  return index;
}
```

LRU Algorithm Time Complexity Analysis

The algorithm is called only when the Bufferpool reaches it's maximum size. In this algorithm, we traverse through the entire Bufferpool and search for the page with minimum timestamp. Hence time complexity = O(n) where n is the maximum Bufferpool size.

4.3.2. CFLRU (Clean-First LRU)

CFLRU divides the LRU list into two regions: the working region and the clean-first region. The working region contains the pages that have been accessed most recently and

are considered to be "hot" or likely to be accessed again soon. The clean-first region contains the pages that have not been accessed recently and are considered to be "cold".

CFLRU first evicts clean pages from the clean-first region. When there are no more clean pages in the clean-first region, it acts like a classical LRU and evicts the page that has not been used for the longest time.

In our implementation, we maintained the same *timestamp* attribute like we did in the LRU implementation. We used a *window_size* parameter to logically divide the LRU list into 2 parts. The first part is the clean-first region and the second part is the working region.

We implemented CFLRU using 2 approaches. In the first approach we used an **Array** and in the second approach we used an **Min Heap**.

4.3.2.1. **CFLRU Approach 1**

- We maintain a vector '**lru_list**' of size equal to the Bufferpool. The 'lru_list' contains the indices of the pages in the Bufferpool.
- We sort the 'lru_list' based on the timestamp of the pages in the increasing order.
- We start at i = 0 and search for the first clean page till i reaches the value of max_buffer_size / window_size. We assign a value of 3 to window_size.
- If no clean pages are found, then CFLRU behaves like a classical LRU and returns the page at lru_list[0] as the page to be evicted.

The following is the code for CFLRU Approach 1

```
int Buffer::CFLRU_using_array() {
  int index = -1;
  int window_size = 3;

  // Creating lru_list_array
  vector<int> lru_list(buffer_pool.size());
  for (int i = 0; i < buffer_pool.size(); i++){
    lru_list[i] = i;
  }

  // Sorting the lru_list_array according to the timestamps of the
bufferpool pages
  std::sort(lru_list.begin(), lru_list.end(),[&](int A, int B) -> bool
{return buffer_pool[A].timestamp < buffer_pool[B].timestamp;});

  // Find the first clean page in the clean-first region
  for (int i = 0; i < max_buffer_size / window_size; ++i) {
```

```
    if (!buffer_pool[lru_list[i]].dirty) {
      return i;
    }
  }
  // Return the classical LRU index for bufferpool
  index = lru_list[0];
  return index;
}
```

CFLRU Approach 1 Time Complexity Analysis

The algorithm involves the following steps. Here n represents the maximum size of Bufferpool.

- Forming the initial ‘lru_list’ vector - Time complexity of O(n).
- Sorting the ‘lru_list’ vector - Time complexity of O(n * log(n)).
- Iterating through the ‘lru_list’ vector to find the first clean page in the clean-first region - Time complexity of O(n).
- Overall Time complexity = O(n) + O(n * log(n)) + O(n) = O(n * log(n))

4.3.2.2. CFLRU Approach 2

We maintain 2 vectors of indices - '**clean_first_indices**' and '**working_region_indices**'.

- We also maintain a **min heap** (timestamp, index) where the priority is based on timestamp.
- We keep popping the minimum elements index of min heap into the 'clean_first_indices' vector till size of 'clean_first_indices' vector = max_buffer_size/3.
- We pop the remaining minimum elements index of min heap into the 'working_region_indices' vector.
- We linearly search for the first clean page in 'clean_first_indices' vector starting from i = 0 and return the index of the first clean page that is least recently used.
- If there are no clean pages in the 'clean_first_indices' vector, we return the page at clean_first_indices[0] as this is the least recently used page in the overall bufferpool.

The following is the code for CFLRU Approach 2

```
int Buffer::CFLRU_using_min_heap() {
  int index = -1;
  int clean_first_region_size = max_buffer_size / 3; //
Calculating clean_first_region_size
```

```
  vector<int> clean_first_indices;
  vector<int> working_region_indices;

  // Form a min heap of the time stamps and the indexes of the
pages in buffer pool
  priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> min_heap;
  for (int i = 0; i < buffer_pool.size(); ++i) {
    min_heap.push({buffer_pool[i].timestamp, i});
  }

  // Populate the clean_first_indices vector
  for (int i = 0; i < clean_first_region_size; ++i) {
    clean_first_indices.push_back(min_heap.top().second);
    min_heap.pop();
  }

  // Populate the working_region_indices_vector
  while (!min_heap.empty()) {
    working_region_indices.push_back(min_heap.top().second);
    min_heap.pop();
  }

  // Find the first clean page in the clean-first region that
is least recently accessed
  for (int i : clean_first_indices) {
    if (buffer_pool[i].dirty == false) {
      index = i;
      break;
    }
  }

  // If there are no clean pages in the clean-first region,
find the least recently used page in the overall buffer pool
  if (index == -1) {
    index = clean_first_indices[0];
  }

  return index;
```

```
}
```

CFLRU Approach 2 Time Complexity Analysis

The algorithm involves the following steps. Here n represents the maximum size of Bufferpool.

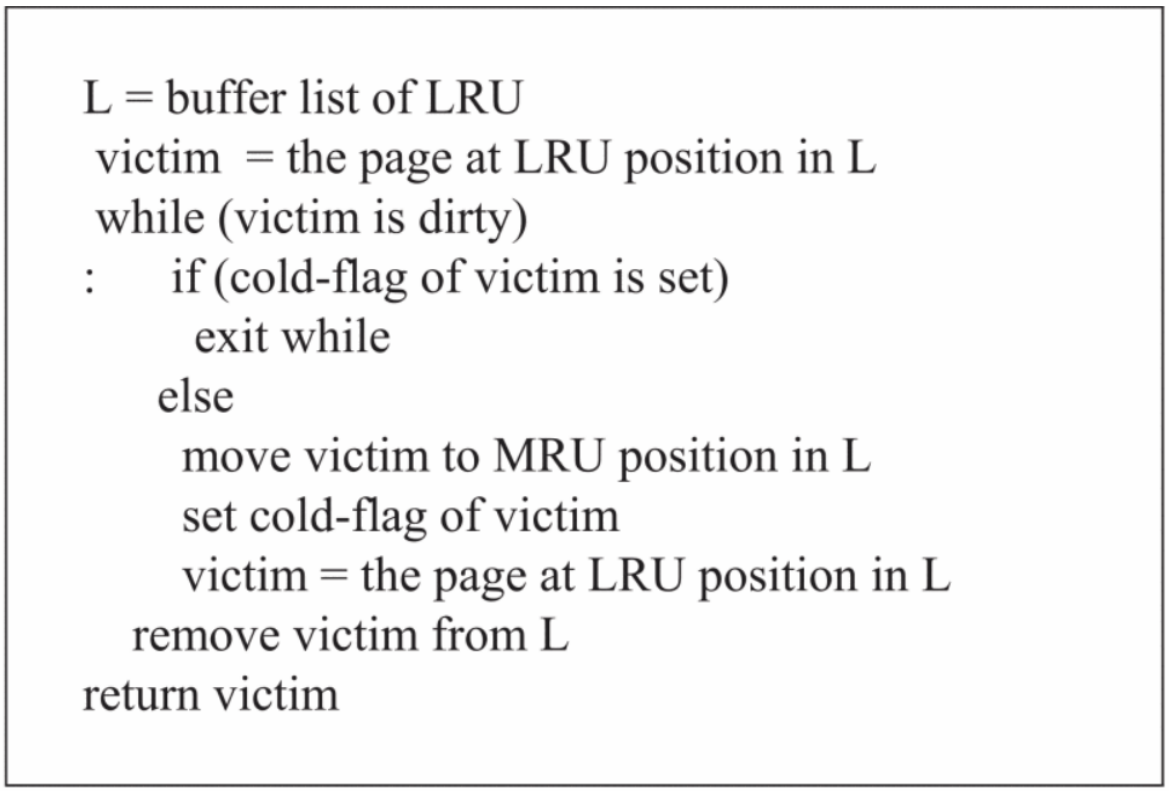
- Pushing elements into the Min Heap - Time complexity of O(n * log(n)).
- Removing minimum elements from Min Heap into ‘clean_first_indices’ and ‘working_region_indices’ - Time complexity of O(n * log(n)).
- Iterating through the ‘clean_first_indices’ - Time complexity of O(n).
- Overall Time complexity = O(n * log(n)) + O(n * log(n)) + O(n) = O(n * log(n)).

4.3.3. LRU-WSR (LRU Write Sequence Reordering)

The implementation of LRU-WSR is as follows

- Initially we search for the least recently used page.
- If the least recently used page is clean, we evict it.
- If it is dirty, we check if it is cold or hot.
- If it is cold, we evict it.
- If it is hot, we set the cold flag and update the timestamp as the 1 + maximum value of timestamp of the pages in the Bufferpool and repeat the process again.

The below snapshot is taken from the Research Paper (link attached).



The following is the code for LRU-WSR

```
int Buffer::LRUWSR()
{
  int index;
  int min_timestamp;
  bool found = false;

  while (!found)
  {
    index = 0;
    min_timestamp = buffer_pool[0].timestamp;
    // First the least recently used page is found
    for (int i = 1; i < buffer_pool.size(); ++i)
    {
      if (buffer_pool[i].timestamp < min_timestamp)
      {
        min_timestamp = buffer_pool[i].timestamp;
        index = i;
      }
    }
```

```
    // We check if the LRU page is dirty. If it is clean, we evict it
    if (!buffer_pool[index].dirty)
    {
      found = true;
    }
    else
    {
      // We then check if the dirty page is cold. If it is cold, we
evict it
      if (buffer_pool[index].cold)
      {
        found = true;
      }
      // If the dirty page is hot, we set the cold flag and update the
timestamp. We give a second chance to the dirty hot page
      else
      {
        buffer_pool[index].cold = true;
        buffer_pool[index].timestamp = Buffer::global_clock;
        ++Buffer::global_clock;
      }
    }
  }
  return index;
}
```

LRU-WSR Algorithm Time Complexity Analysis

The algorithm involves the following steps.

- Iterating through the Bufferpool to find the page with minimum timestamp - Time complexity of O(n).
- The worst case scenario is when all the pages in the Bufferpool are dirty and hot which means we would have to iterate through the Bufferpool n times,
- Overall Time complexity = O(n^2).

5. Results & Conclusion

In this part, we evaluated the performance of the buffer pool both only in memory and on disk. When evaluating in the memory, we only count the hit rate and the miss rate. With simulation_on_disk option enabled, we also evaluated the latency of IOs. Both

experiments are evaluated on uniform data and skewed data (90% operations on 10% data).

Parameters Description

S.No.	Parameter	Description
1.	b	Buffer size in pages
2.	n	Disk size in pages
3.	e	Entry size in bytes
4.	x	Total no. of operations to be performed
5.	r	Percentage of reads in workload
6.	r’	Percentage of writes in workload
7.	a	Algorithm of page eviction
8.	s	Probability of selecting any of the skewed pages
9.	d	Percentage of skewed pages in disk
10.	pin	Pin mode is enabled or not
11.	simulation_on_disk	Simulation on disk is enabled or not

5.1 Skewed vs Non Skewed Buffer Hit Ratio

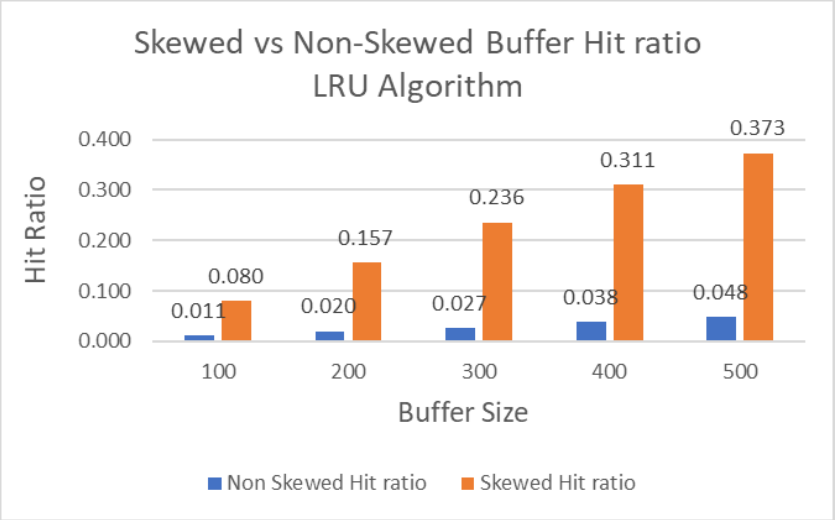
Skewed data refers to an uneven distribution of data access patterns where some data items (pages) are accessed more frequently than others. This means that the workload is not uniformly distributed across all the data items, and a small subset of the data items is responsible for a large number of requests.

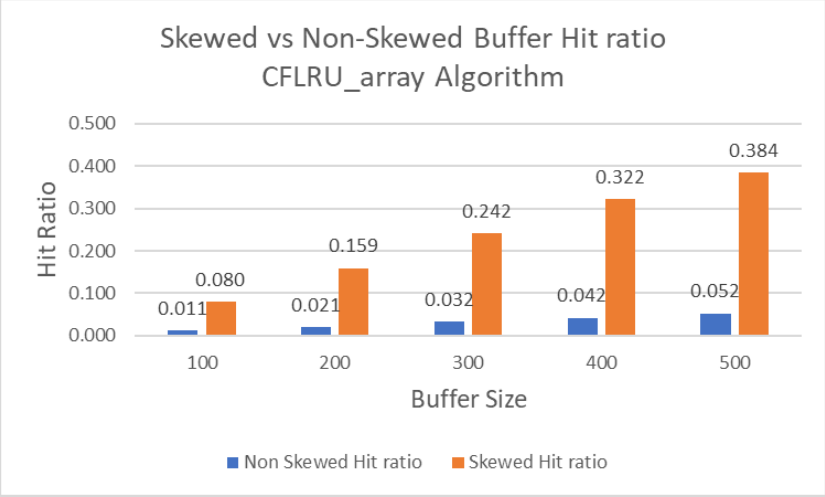
This uneven access pattern can be characterized by the presence of hot items, which are data items that are frequently accessed or updated, and cold items, which are accessed less frequently or not at all.

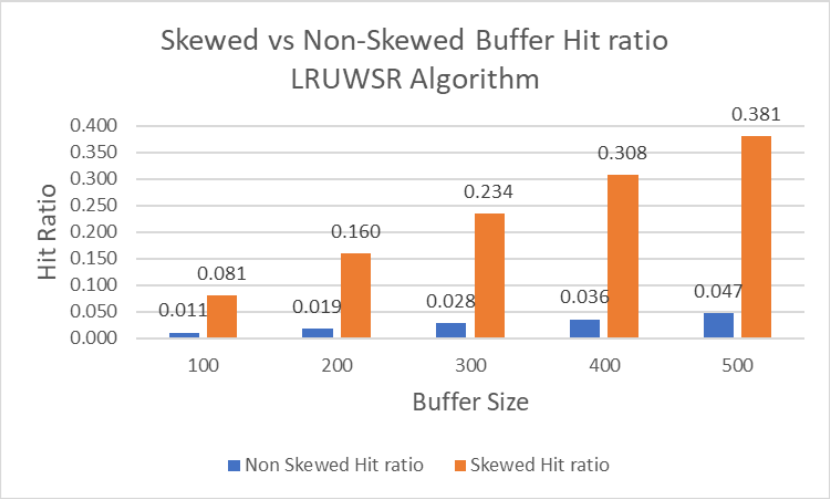
We evaluated the buffer hit ratios for different algorithms on skewed and non skewed data. Skewed data is when s = 90 and d = 10 (i.e performing 90% operations on 10% data - This is the measure of skewness used in all skewed experiments). The **parameters values** are as mentioned below.

S.No.	Parameter	Value
1.	n	10000
2.	e	128
3.	x	10000
4.	r	70

The **visualizations** are as follows.







The most frequently accessed data items (also known as hot items) are likely to be found in the buffer pool, as they are used more often and have a higher probability of being kept in memory. When the workload is skewed, the buffer pool hit ratio tends to increase because:

- Hot items are accessed more frequently, increasing the chances of them being in the buffer pool.
- Buffer replacement algorithms, like Least Recently Used (LRU) or variations of it, prioritize keeping frequently accessed items in the buffer pool.
- As the hot items remain in the buffer pool, they help satisfy a larger number of requests, leading to fewer disk accesses and a higher hit ratio.

So, in the case of skewed data, the buffer pool can better utilize its available space to keep the most relevant and frequently accessed data in memory, resulting in a higher hit ratio and improved performance.

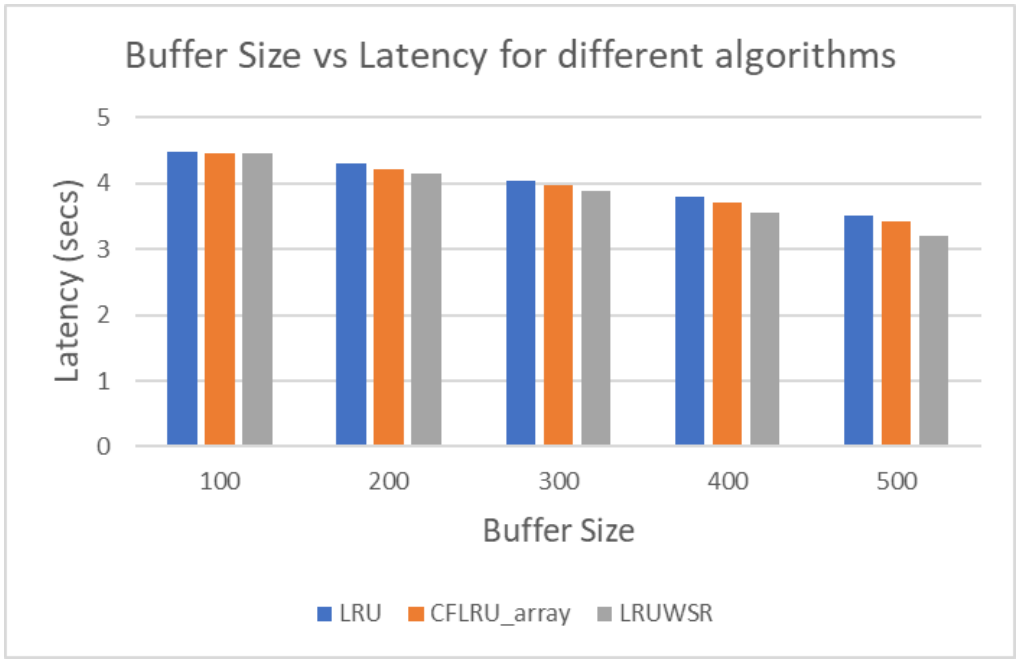
5.2 Buffer Pool Size Evaluation

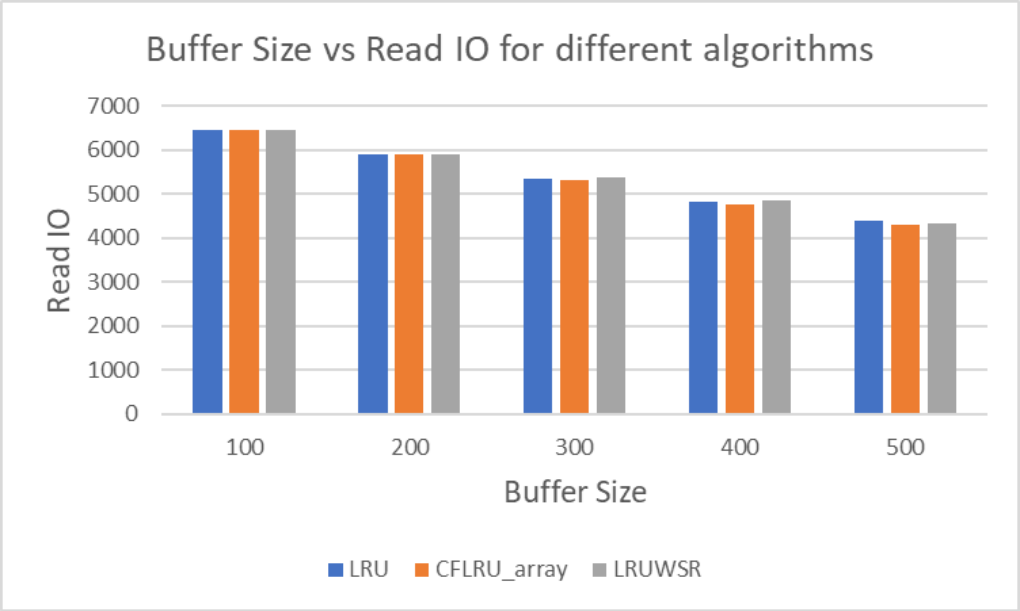
We measured the **Latency values**, **Read IOs**, and **Write IOs** of different buffer sizes and algorithms by turning on the ‘simulation_on_disk’. The parameter values are as mentioned below.

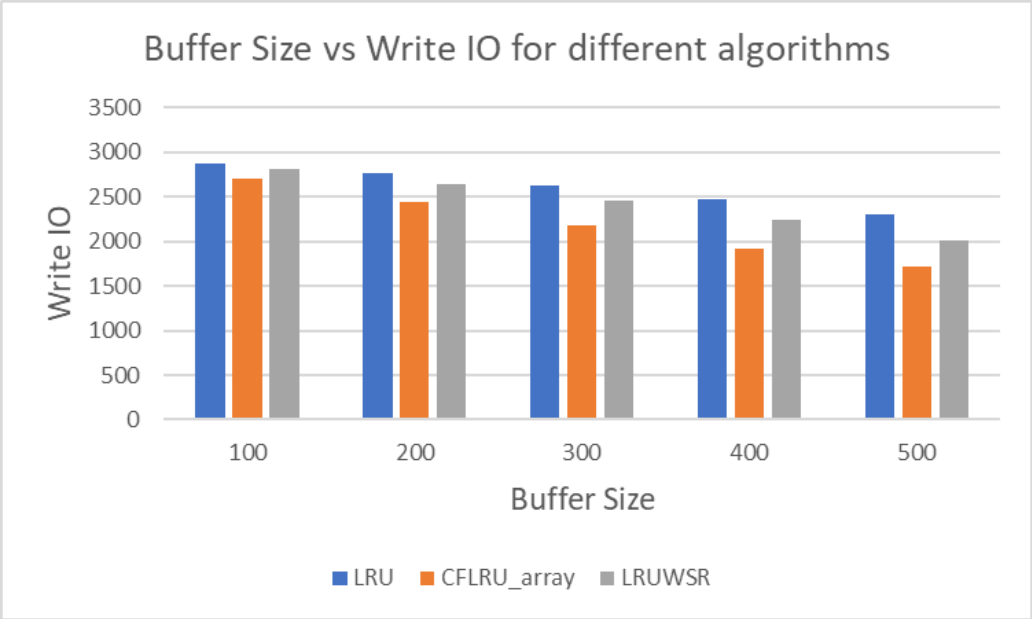
S.No.	**Parameter**	**Value**
1.	n	10000
2.	e	128
3.	x	10000

4.	r	70
5.	s	90
6.	d	10
7.	simulation_on_disk	true

The visualizations are as follows.







We observe that as the buffer size increases, there is a decrease in Latency, Read IOs and Write IOs.

As the buffer size increases, more data pages can be stored in the buffer pool at a given time. This has several effects on the overall performance of the system:

- **Decrease in latency**: A larger buffer size means that more data pages can be cached in memory, which reduces the need to fetch them from disk. Disk accesses are significantly slower than memory accesses, so minimizing disk I/O will generally result in lower latency for the read and write operations. With more data

available in the buffer pool, the chances of finding the required page in memory (buffer hit) increase, leading to faster access times.

- **Decrease in read IOs**: With a larger buffer size, more data pages can be stored in memory. This reduces the likelihood of a buffer miss, where the required data page is not found in the buffer pool and has to be fetched from disk. As a result, the number of disk read operations decreases, which in turn reduces the overall read IOs.

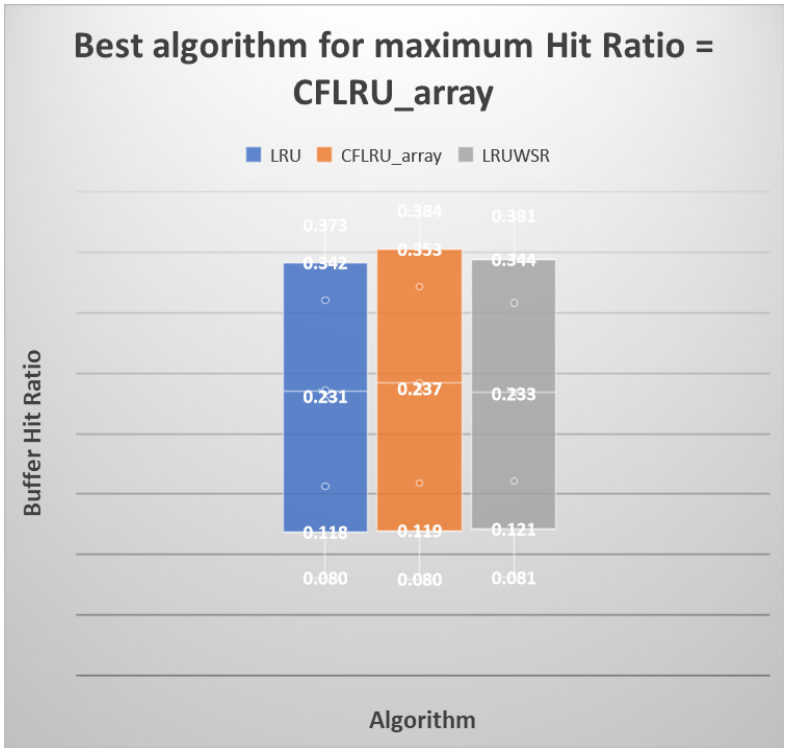
- **Decrease in write IO**: With a larger buffer size, there's a higher chance that the data page is already in memory, which means fewer disk write operations are needed to update the data. Moreover, some buffer management algorithms may also perform optimizations, such as delaying write operations until the page is evicted from the buffer pool or utilizing group commit, which further reduces the number of write IOs.

5.3 **Algorithm Evaluation**

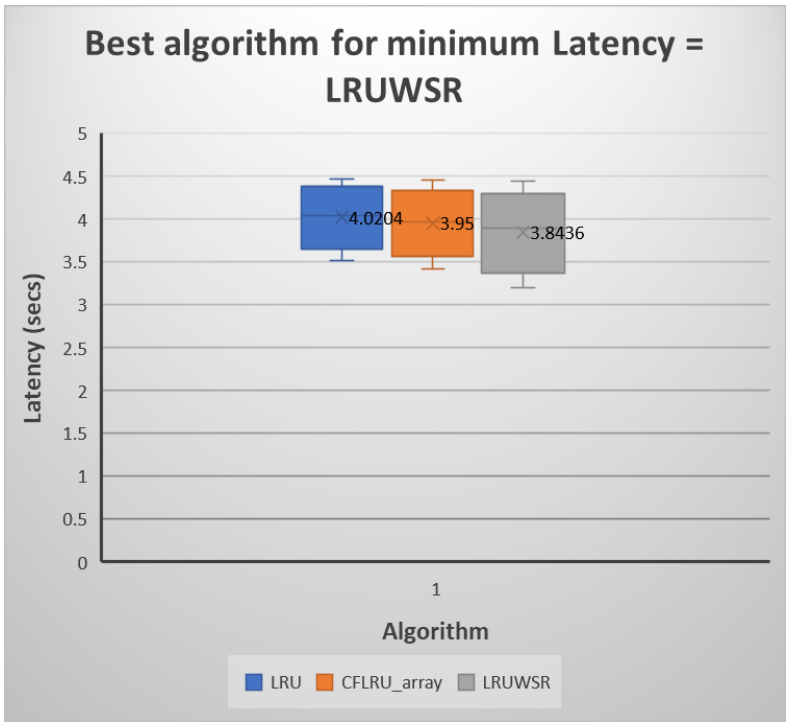
We evaluated the algorithms on skewed data for average values of maximum Buffer Hit ratio, minimum Latency, minimum no. of Read IOs, and minimum number of Write IOs. The average is calculated by running the algorithms over 5 different buffer sizes (100, 200, 300, 400, 500). The parameter values are as mentioned below.

S.No.	**Parameter**	**Value**
1.	n	10000
2.	e	128
3.	x	10000
4.	r	70
5.	s	90
6.	d	10
7.	simulation_on_disk	true

The visualizations are as follows.



CLFRU_array recorded the maximum average Buffer Hit ratio.



LRUWSR recorded the minimum average Latency value.

CFLRU_array recorded the minimum average number of Read IOs.

6. References

1. Park, S. Y., Jung, D., Kang, J. U., Kim, J. S., & Lee, J. (2006, October). CFLRU: a replacement algorithm for flash memory. In Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (pp. 234-241).

2. Jung, H., Shim, H., Park, S., Kang, S., & Cha, J. (2008). LRU-WSR: integration of LRU and writes sequence reordering for flash memory. IEEE Transactions on Consumer Electronics, 54(3), 1215-1223.

3. O'neil, E. J., O'neil, P. E., & Weikum, G. (1993). The LRU-K page replacement algorithm for database disk buffering. Acm Sigmod Record, 22(2), 297-306