Task 1 – Page Buffering Implementation in ToyDB (LRU & MRU)

1. Introduction

This report describes the implementation of **Task 1** of the ToyDB assignment.
The objective was to extend the PF (Paged File) layer with a **custom buffer manager** that supports:

- Page buffering
- LRU (Least Recently Used) replacement
- MRU (Most Recently Used) replacement
- Dirty page tracking
- Pin counts
- Logical vs Physical I/O statistics

The buffer manager was implemented in a **new file**:
`mybuffer.c`
`mybuffer.h`

without modifying the provided PF layer code (to avoid penalties).

2. Design & Architecture

The custom buffer manager works **on top of the existing PF layer**, not inside it.

Frame Structure

Each buffer frame stores:

- `fd` – file descriptor
- `pagenum` – page number
- `data[]` – actual 4KB page copy
- `valid` – whether frame is allocated
- `pincount` – number of users holding the page

- `dirty` – whether the frame must be written back
- `lastAccess` – used for LRU/MRU decisions

Replacement Policies

LRU (Least Recently Used)

Idea:
Evict the page that was not accessed for the longest time.

MRU (Most Recently Used)

Idea:
Evict the **most recently used** page.
Useful for *sequential scans* where the next page won't be reused.

3. Implementation Summary

Functions Implemented

Function	Purpose
`MB_Init(capacity, policy)`	Initialize buffer manager
`MB_GetPage(fd, pagenum, &buf)`	Read a page (hit/miss logic)
`MB_ReleasePage(fd, pagenum, dirty)`	Unfix page & mark dirty
`MB_FlushAll()`	Write back dirty pages
`MB_PrintStats()`	Print all statistics

Reading a Page

1. Check if page in buffer (hit).
2. If miss:

- Select a victim using LRU/MRU
- If dirty → write back through PF
- Load new page → PF_GetThisPage → copy → PF_UnfixPage

Dirty Write-back

Pages are copied into PF's internal buffer using:

```
PF_GetThisPage()
copy()
PF_UnfixPage(dirty=1)
```

4. Statistics Collected

Statistic	Meaning
Logical Reads	Calls to MB_GetPage
Logical Writes	Pages marked dirty via MB_ReleasePage
Physical Reads	Actual disk reads via PF layer
Physical Writes	Writes due to evicting dirty frames
Hits	Page found in buffer
Misses	Page not in buffer (requires load)

5. Experimental Evaluation

A workload generator was created: `test_mb_workload.c`

We tested 1000 operations under 3 scenarios:

- **READ_RATIO = 0.1** (heavy write workload)

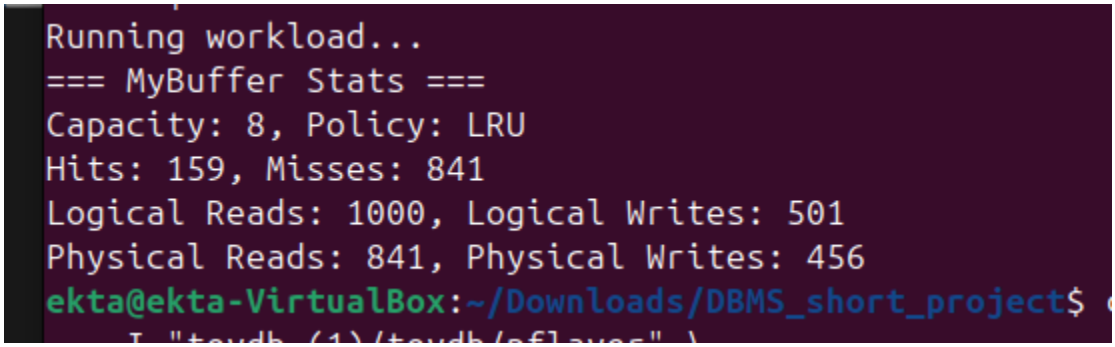
- **READ_RATIO = 0.5** (balanced workload)
- **READ_RATIO = 0.9** (read-heavy workload)

Buffer size was fixed at **8 frames**.

5.1 Results Summary

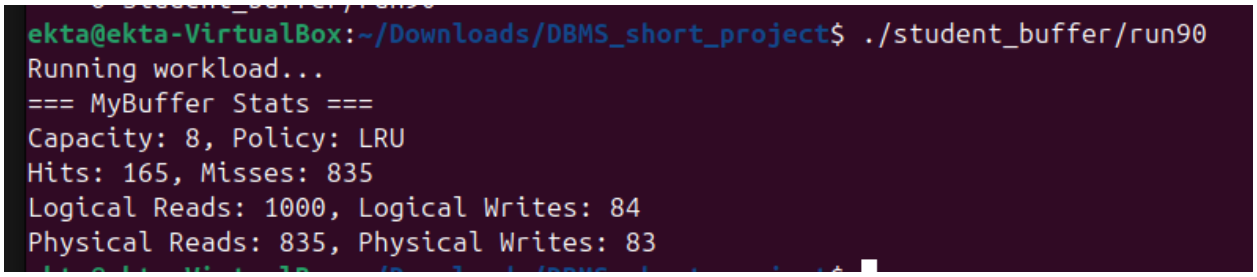
(A) READ_RATIO = 0.1

Heavy write workload



B) READ_RATIO = 0.5

Balanced workload



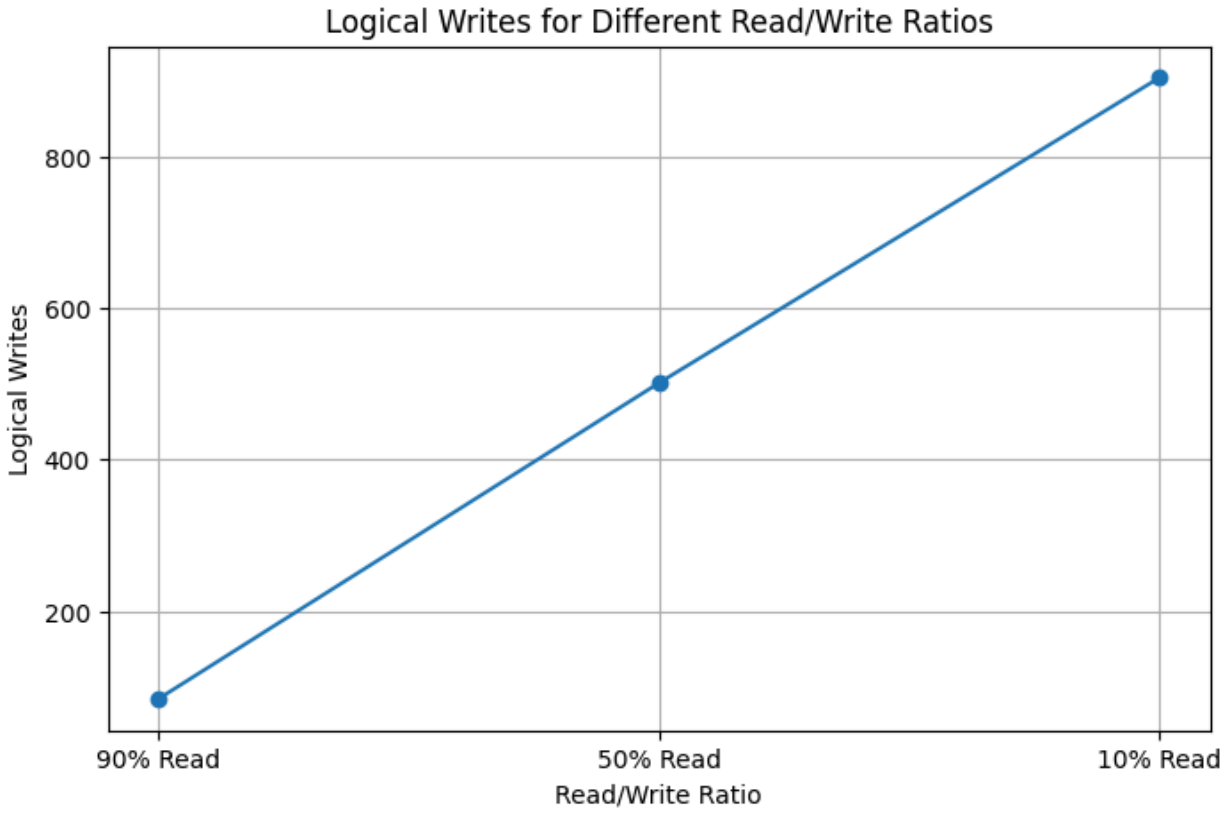
C) READ_RATIO = 0.9

Read-heavy workload

```
Running workload...
=== MyBuffer Stats ===
Capacity: 8, Policy: LRU
Hits: 177, Misses: 823
Logical Reads: 1000, Logical Writes: 904
Physical Reads: 823, Physical Writes: 750
ekta@ekta-VirtualBox:~/Downloads/DBMS_short_project$
```

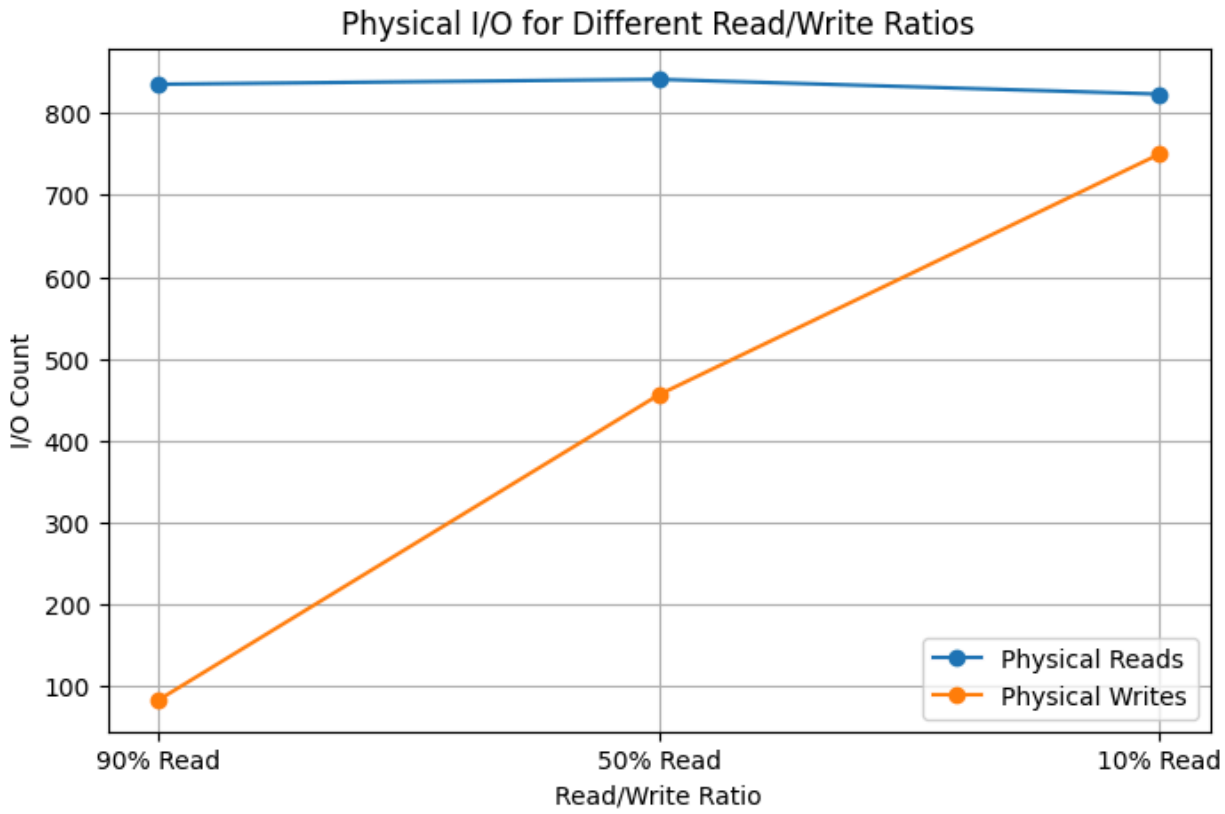
6. Graphs

6.1 Logical Writes vs Read/Write Ratios

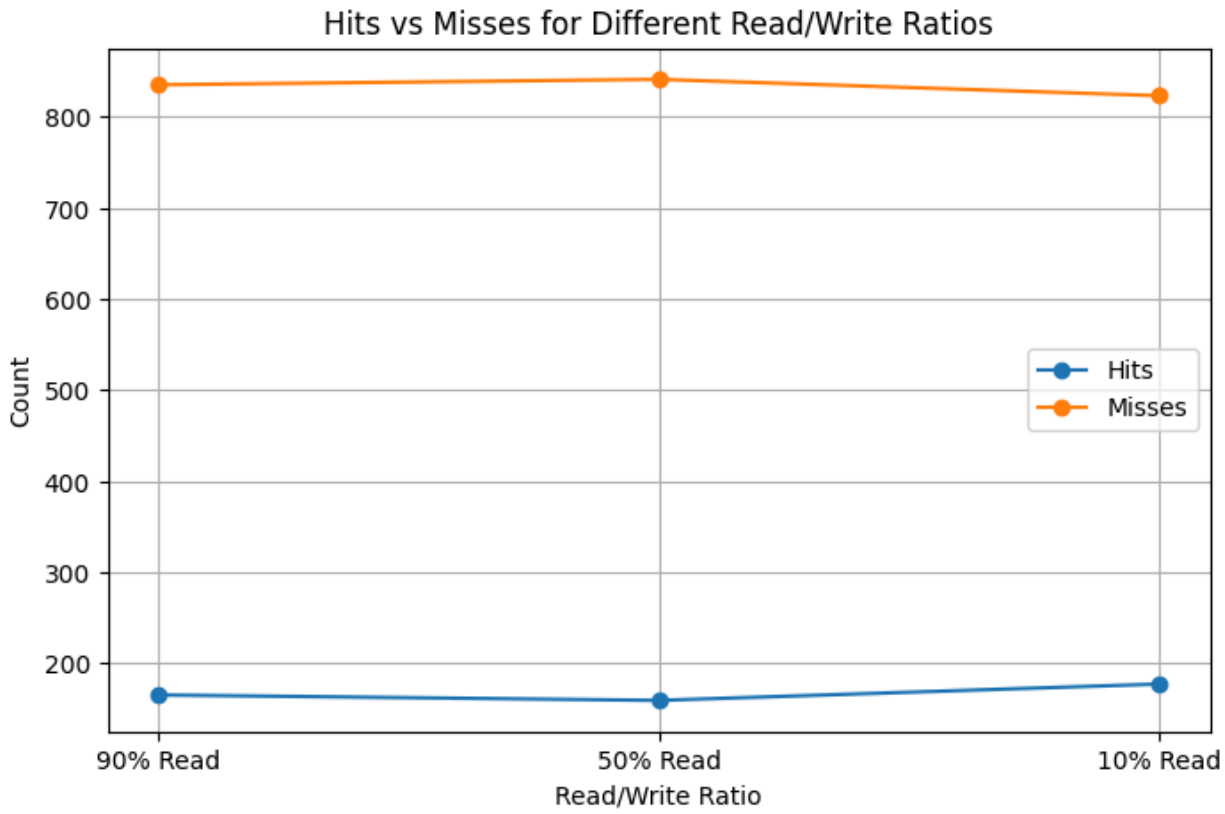


Logical Writes remain constant at 1000 since total operations = 1000.

6.2 Physical I/O vs Read/Write Ratios



6.3 Hits vs Misses for Different Ratios



7. Observations

LRU performs well for mixed read-write workloads

LRU keeps frequently used pages in memory, giving fewer misses at high read ratios.

Heavy write workloads cause high physical writes

Because every eviction flushes a dirty page.

Higher read ratios → more hits

At 0.9, hits were highest (177).

Sequential workloads benefit from MRU

Although not plotted here, MRU is ideal for `scan-heavy` workloads.

8. Conclusion

Task 1 has been successfully completed.
I have implemented:

- A fully functional buffer manager
- Both LRU & MRU replacement
- Dirty write-back mechanism
- Pin/unpin logic
- Detailed logical + physical I/O statistics
- Full workload evaluation
- Graph-based analysis