CS 525: Advanced Database Organization

Programming Assignment 2: Buffer Manager

Group no 17: Individual Contribution			
CWID	**Name**	**Contribution (description)**	**Percent Contribution**
A20594489	Kori Kamoljonov	Integrated the buffer manager with the storage manager, handled error management and force flushing of dirty pages, and contributed to final testing and documentation	33.3%
A20568393	Chaitanya Datta Maddukuri	Implemented LRU_k and CLOCK strategies, developed test cases, and helped debug and optimize memory management to ensure system reliability	33.3%
A20582646	Vamshi Krishna Cheeti	Implemented core buffer manager functions, including initialization and page handling. He developed FIFO and LRU replacement strategies and worked on tracking buffer pool statistics and optimizing performance.	33.4%

1 Overview:

The **Buffer Manager** is responsible for managing a fixed number of page frames in memory, storing pages from a page file controlled by the Storage Manager. It supports multiple page replacement strategies to optimize memory utilization and performance, including:

- **FIFO (First-In-First-Out)** - where the oldest page in memory is replaced first.
- **LRU (Least Recently Used) -** which replaces the least recently accessed page.
- **CLOCK (Clock Page Replacement Algorithm)** - an approximation of LRU using a circular queue.
- **LFU (Least Frequently Used) -** which evicts the page with the lowest access frequency.
- **LRU-K (Least Recently Used - K distance tracking) -** considering the past K accesses for replacement decisions.

The buffer manager works closely with the **Storage Manager** for handling disk read/write operations and maintains various performance metrics, such as reference counts and I/O operations. The efficient functioning of the buffer manager is essential for improving database system performance and reducing disk I/O overhead.

2 Implemented Features:

The buffer manager implements the following key features:

1. Buffer Pool Initialization: Initializes a buffer pool with a specified page file, number of pages, and replacement strategy (FIFO, LRU, LRU k, CLOCK). Allocates memory for page frames, metadata (dirty flags, fix counts, page numbers), and strategy-specific structures like FIFO queues or LRU timestamp arrays.

2. Buffer Pool Disposal: Free up all resources associated with a buffer pool.

3. Page Management:

- **Pinning Pages:** loads a page into the buffer pool.
- **Unpinning Pages:** decrements a page's fix count, allowing it to be replaced.
- **Marking Pages as Dirty:** flags a page as modified.
- **Forcing Pages to Disk:** immediately writes a dirty page to disk, resetting its dirty flag.

4. Replacement Strategies: Implement various page replacement strategies such as FIFO, LRU, LRU k and CLOCK.

5. Statistics Tracking: Provides detailed statistics, including frame contents, dirty flags, fix counts, and disk I/O counts. Functions like getFrameContents, getDirtyFlags, getFixCounts, getNumReadIO, and getNumWriteIO allow users to monitor buffer pool performance and behavior.

6. Error Handling: Detects and handles errors such as invalid page numbers, memory allocation failures, and disk I/O errors.

3 Buffer Manager:

Part 1: Buffer Pool Initialization

This section explains the functions used to initialize a new buffer pool with a specific page file, page size, and replacement strategy.

void initBufferPool(BM_BufferPool *const bm, const char *const pageFileName, const int numPages, ReplacementStrategy strategy, void *stratData)

Initializes a new buffer pool. It allocates memory for the buffer pool, assigns the specified replacement strategy, and sets up required data structures to manage the pool efficiently. The function also ensures that the buffer pool is ready to interact with disk pages.

void shutdownBufferPool(BM_BufferPool *const bm)

Shuts down the buffer pool. It ensures that all dirty pages are written back to disk before deallocating memory and releasing resources. Any associated data structures are cleaned up to prevent memory leaks.

forceFlushPool(BM_BufferPool *const bm)

Forces all dirty pages in the buffer pool to be written to disk. It is useful for ensuring data consistency by persisting any unsaved modifications before shutting down the buffer pool.

Part 2: Page Management

This section describes the functions used to manage the reading, writing, and pinning of pages in the buffer pool.

pinPage(BM_BufferPool *const bm, BM_PageHandle *const page, const PageNumber pageNum)

Loads a requested page into the buffer pool if it is not already present. It increases the fix count to indicate that the page is actively in use, preventing it from being replaced.

unpinPage(BM_BufferPool *const bm, BM_PageHandle *const page)

Decreases the fix count of the specified page. Once the fix count reaches zero, the page becomes eligible for replacement, allowing the buffer manager to free up space when necessary.

*forcePage(BM_BufferPool *const bm, BM_PageHandle *const page)*

Forces the immediate writing of a dirty page back to disk. It ensures that any changes made to the page are persisted, preventing data loss in case of a system failure.

*markDirty(BM_BufferPool *const bm, BM_PageHandle *const page)*

Marks a page as dirty, indicating that it has been modified and should be written back to disk before being replaced. This ensures that the most recent updates are not lost.

Part 3: Replacement Strategies

*FIFO(BM_BufferPool *const bm, BM_PageHandle *const page)*

Implements the First-In-First-Out (FIFO) page replacement strategy. It replaces the oldest unpinned page in the buffer pool when a new page needs to be loaded.

*LRU(BM_BufferPool *const bm, BM_PageHandle *const page)*

Implements the Least Recently Used (LRU) strategy. It keeps track of page usage and selects the least recently accessed unpinned page for replacement.

*LRU_k(BM_BufferPool *const bm, BM_PageHandle *const page)*

Extends LRU by considering the last k accesses before making a replacement decision. It provides better adaptability to varying workload patterns.

*CLOCK(BM_BufferPool *const bm, BM_PageHandle *const page)*

Implements the CLOCK page replacement algorithm. It maintains a circular list of pages and uses a clock hand to track page references, replacing the next unpinned page with a low reference bit.

Part 4: Statistics Interface

*PageNumber *getFrameContents(BM_BufferPool *const bm)*

Returns an array of page numbers representing the current contents of the buffer pool. It provides insight into which pages are currently loaded.

bool *getDirtyFlags(BM_BufferPool *const bm)

Returns an array indicating which pages in the buffer pool have been modified. It helps track changes that need to be written back to disk.

int *getFixCounts(BM_BufferPool *const bm)

Provides an array showing the fix count for each page in the buffer pool. The fix count indicates how many processes are using a page.

int getNumReadIO(BM_BufferPool *const bm)

Rturns the total number of pages read from disk into the buffer pool. It helps in analyzing the efficiency of disk I/O operations.

int getNumWriteIO(BM_BufferPool *const bm)

Returns the total number of pages written back to disk from the buffer pool. It is useful for tracking write operations and optimizing performance.

4. How to Run the Code

Step 1: Navigate to the assign2 branch in the GitHub repository and download the .zip file.

Step 2: Run "***make clean***" to remove any previously compiled files.

Step 3: Use the "***make***" command to compile the program by running the **Makefile**.

Step 4: Execute the test case 1 by running "***./test_case1***". (Basic Functionality & FIFO Strategy)

Step 5: Execute the test case 2 by running "***./test_case2***". (LRU_k & CLOCK Strategy):

Step 6: Clean up Executables by running "**make clean**"

5. Memory Leak Check

There are no memory leaks in the code. Below are commands used to check. For macOS users, run the following to check for leaks:

- `***leaks -atExit -- ./test_case1 | grep LEAK:***`

- `***leaks -atExit -- ./test_case2 | grep LEAK:***`