Storage and Buffer Manager

Lab Description

For “Advanced Database Systems”

2026

TABLE OF CONTENTS

INTRODUCTION **3**

BUFFER AND FRAMES **3**

BUFFER AND FRAME SIZES 3

BUFFER AND FRAME STORAGE STRUCTURES **3**

PAGE FORMAT **4**

FILE FORMAT **4**

BUFFERING TECHNIQUE **4**

HASHING TECHNIQUE **4**

FILE STORAGE **5**

CLASS DESIGN **5**

DATA STORAGE MANAGER 5
BUFFER MANAGER 6

BUFFER INTERFACE FUNCTIONS **7**

FIXPAGE(INT PAGE_ID, INT PROT) 7
FIXNEWPAGE() 7
UNFIXPAGE(INT PAGE_ID) 7
NUMFREEFRAMES() 7
SELECTVICTIM() 7
HASH(INT PAGE_ID) 7
REMOVEBCB(BCB* PTR, INT PAGE_ID) 8
REMOVELRUELE(INT FRID) 8
SETDIRTY(INT FRAME_ID) 8
UNSETDIRTY(INT FRAME_ID) 8
WRITEDIRTYS() 8
PRINTFRAME(INT FRAME_ID) 8

DATA STORAGE INTERFACE FUNCTIONS **8**

OPENFILE(STRING FILENAME) 8
CLOSEFILE() 9
READPAGE(INT PAGE_ID) 9
WRITEPAGE(INT FRAME_ID, BFRAME FRM) 9
SEEK(INT OFFSET, INT POS) 9
GETFILE() 9
INCNUMPAGES() 9
GETNUMPAGES() 9
SETUSE(INT PAGE_ID, INT USE_BIT) 9
GETUSE(INT PAGE_ID) 10

EXPERIMENT SETUP **10**

IMPLEMENTATION PLAN **11**

Introduction

In this project, we will implement a simple storage and buffer manager. The document addresses the storage and buffer manager. Buffer and frame sizes, buffer and frame storage structures, page formats, page storage structures, file formats, buffering techniques, hashing techniques, file storage structures, and interface functions for the disk space and buffer modules will be discussed. The particular technique is chosen from the material covered in class that are relevant to buffer and storage manager.

Buffer and Frames

Buffer and Frame Sizes

Buffer refers to the space in main memory. CPU can only access the contents present in main memory. The buffer consists of an array of frames. When a page is requested it is loaded up into memory in the buffer. Most commercial database management systems make the frame size the same as the size of a page in order to prevent external fragmentation. The same strategy is employed in the project. The buffer size by default will be set to 1024 for the project.

Buffer and Frame Storage Structures

Buffer is made up of logical partitions called as frames.
The frame will be stored in a globally defined structure, describing what a frame will look like. This structure will be defined as

```
#define FRAMESIZE  4096

        struct bFrame
        {
                Char field [FRAMESIZE ];
        };
```

The buffer array will store a series of frames that will store the pages that are loaded in it. This array will look as follows:

```
        #define DEFBUFSIZE 1024
        bFrame buf[DEFBUFSIZE];  // or the size that the user defined by the input parameter
```

This will be the space allocated for the buffer that will be accessed by the buffer manager and file and access methods to reference a required page.

Page Format

In this project, we need not refer to the detailed structure of a page. The only mattered information is the page_id and page_size. So you need not design page format.

File Format

We recommend using the directory-based structure to organize the database file, as introduced in the class. Every file has a base page that contains a pointer to every page in the file. Each pointer in the base page sits in order for the pages. Data pages in this type of file do not have pointers, only records. The base page, or directory, must be consulted in order to get the next page in the file.

The Directory Based file format was chosen because it seemed to suite itself to finding specific pages for the records that are requested. The directory base for the file will allow for quick access to the correct page without having to search a long list of pages to get to the correct one.

Buffering Technique

We choose LRU as the only replacement policy in our lab. LRU always evicts the least-recently-used page from an LRU queue used to organize the buffer pages which are ordered by time of their last reference. It always selects as a victim the page found at the LRU position. The most important advantage of LRU is its constant runtime complexity. Furthermore, LRU is known for its good performance in case of reference patterns having high temporal locality, i.e., currently referenced pages have a high re-reference probability in the near future.

Hashing Technique

For each frame in the buffer, a buffer control block must be kept. Each buffer control block, or BCB, contains a page_id, a frame_id, page_latch, fix_count, and dirty_bit. The page_ids are used as the key into a hashing function that will map the page_id to a BCB. Two hash tables must be kept: one to map page_ids to frame_ids and BCB's and one that maps frame_ids to page_ids.

We suggest using the simple Static Hashing technique. In Static hashing the number of buckets is fixed. If a bucket is full, an overflow chain is connected for the extra data entries. Using the key value, the hashing function maps it to a bucket. And to search within an individual bucket sequential search is used. The number of buckets does not change as time progresses.

The Static hashing technique for hash tables. The hashing function will look like:

H(k) = (page_id)%buffer_size

Buffer Control Blocks will contain page_id, frame_id, latch, count, dirty_bit.
The hash table for the page_id to BCB will look like: BCB hTable[BufferSize].
The table for the frame_id to page_id will look like: int hTable[BufferSize].

```
struct BCB
 {

  BCB();
  int page_id;
  int frame_id;
  int latch;
  int count;
  int dirty;
  BCB * next;

};
```

File Storage

In our project, we need only one physical file on the disk. All data in the database will be kept in this single file. This file will be kept in the working directory and will be named `data.dbf`. This file should always be found, even if it is empty, which it will be when the system is run for the first time.

Class Design

Data Storage Manager

```
class DSMgr
{
public:
        DSMgr();
        int OpenFile(string filename);
        int CloseFile();
        bFrame ReadPage(int page_id);
        int WritePage(int frame_id, bFrame frm);
        int Seek(int offset, int pos);
        FILE * GetFile();
        void IncNumPages();
        int GetNumPages();
```

```
        void SetUse(int index, int use_bit);
        int GetUse(int index);
private:
        FILE *currFile;
        int numPages;
        int pages[MAXPAGES];
};
```

Buffer Manager

```
class BMgr
{
public:
        BMgr();
        // Interface functions
        int FixPage(int page_id, int prot);
        void NewPage FixNewPage();
        int UnfixPage(int page_id);
        int NumFreeFrames();

        // Internal Functions
        int SelectVictim();
        int Hash(int page_id);
        void RemoveBCB(BCB * ptr, int page_id);
        void RemoveLRUEle(int frid);
         void SetDirty(int frame_id);
         void UnsetDirty(int frame_id);
        void WriteDirtys();
        PrintFrame(int frame_id);
private:
        // Hash Table
        int ftop[DEFBUFSIZE];
        BCB* ptof[DEFBUFSIZE];
};
```

Buffer Interface Functions

These interface functions will provide an interface to the file and access manager above it. The functions required are:

FixPage(int page_id, int prot)

The prototype for this function is FixPage(Page_id, protection) and it returns a frame_id. The file and access manager will call this page with the page_id that is in the record_id of the record. The function looks to see if the page is in the buffer already and returns the corresponding frame_id if it is. If the page is not resident in the buffer yet, it selects a victim page, if needed, and loads in the requested page.

FixNewPage()

The prototype for this function is FixNewPage() and it returns a page_id and a frame_id. This function is used when a new page is needed on an insert, index split, or object creation. The page_id is returned in order to assign to the record_id and metadata. This function will find an empty page that the File and Access Manager can use to store some data.

UnfixPage(int page_id)

The prototype for this function is UnfixPage(page_id) and it returns a frame_id. This function is the compliment to a FixPage or FixNewPage call. This function decrements the fix count on the frame. If the count reduces to zero, then the latch on the page is removed and the frame can be removed if selected. The page_id is translated to a frame_id and it may be unlatched so that it can be chosen as a victim page if the count on the page has been reduced to zero.

NumFreeFrames()

`NumFreeFrames` function looks at the buffer and returns the number of buffer pages that are free and able to be used. This is especially useful for the N-way sort for the query processor. The prototype for the function looks like NumFreeFrames() and returns an integer from 0 to BUFFERSIZE-1(1023).

SelectVictim()

SelectVictim function selects a frame to replace. If the dirty bit of the selected frame is set then the page needs to be written on to the disk.

Hash(int page_id)

Hash function takes the page_id as the parameter and returns the frame id.

RemoveBCB(BCB* ptr, int page_id)

RemoveBCB function removes the Buffer Control Block for the page_id from the array. This is only called if the SelectVictim() function needs to replace a frame.

RemoveLRUEle(int frid)

RemoveLRUEle function removes the LRU element from the list.

SetDirty(int frame_id)

SetDirty function sets the dirty bit for the frame_id. This dirty bit is used to know whether or not to write out the frame. A frame must be written if the contents have been modified in any way. This includes any directory pages and data pages. If the bit is 1, it will be written. If this bit is zero, it will not be written.

UnsetDirty(int frame_id)

`UnsetDirty` function assigns the dirty_bit for the corresponding frame_id to zero. The main reason to call this function is when the setDirty() function has been called but the page is actually part of a temporary relation. In this case, the page will not actually need to be written, because it will not want to be saved.

WriteDirtys()

`WriteDirtys` function must be called when the system is shut down. The purpose of the function is to write out any pages that are still in the buffer that may need to be written. It will only write pages out to the file if the dirty_bit is one.

PrintFrame(int frame_id)

`PrintFrame` function prints out the contents of the frame described by the frame_id.

Data Storage Interface Functions

The current data file will be kept in the DSManager class. This file will be named as data.dbf.

OpenFile(string filename)

`OpenFile` function is called anytime a file needs to be opened for reading or writing. The prototype for this function is OpenFile(String filename) and returns an error code. The function opens the file specified by the filename.

CloseFile()

`CloseFile` function is called when the data file needs to be closed. The protoype is CloseFile() and returns an error code. This function closes the file that is in current use. This function should only be called as the database is changed or a the program closes.

ReadPage(int page_id)

`ReadPage` function is called by the FixPage function in the buffer manager. This prototype is ReadPage(page_id, bytes) and returns what it has read in. This function calls fseek() and fread() to gain data from a file.

WritePage(int frame_id, bFrame frm)

`WritePage` function is called whenever a page is taken out of the buffer. The prototype is WritePage(frame_id, frm) and returns how many bytes were written. This function calls fseek() and fwrite() to save data into a file.

Seek(int offset, int pos)

`Seek` function moves the file pointer.

GetFile()

`GetFile` function returns the current file.

IncNumPages()

`IncNumPages` function increments the page counter.

GetNumPages()

`GetNumPages` function returns the page counter.

SetUse(int page_id, int use_bit)

`SetUse` function looks sets the bit in the pages array. This array keeps track of the pages that are being used. If all records in a page are deleted, then that page is not really used anymore and can be reused again in the database. In order to know if a page is reusable, the array is checked for any use_bits that are set to zero. The fixNewPage function firsts checks this array for a use_bit of zero. If one is found, the page is reused. If not, a new page is allocated.

GetUse(int page_id)

`GetUse` function returns the current use_bit for the corresponding page_id.

Experiment Setup

In our project, you are required to perform a trace-driven experiment to demonstrate your implemental result. The trace has been generated according to the Zipf distribution. There are total 500,000 page references in the trace, which are restricted to a set of pages whose numbers range from 1 to 50,000. Each trace record has the format as "x, ###", where x is 0(read) or 1(write) and ### is the referenced page number. You are required to scan the trace file and print out the total I/Os between memory and disk. The buffer is supposed empty at the beginning of your experiment. All the 50,000 pages involved in the trace need to be first materialized in the disk, which corresponds to the directory-based file `data.dbf`.

Implementation Plan

