CS179G : Project in C.S (Databases)

Department of Computer Science & Enginnering
University of California - Riverside
Spring 2003
TA: Demetris Zeinalipour csyiazti@cs.ucr.edu

Buffer Manager Discussion (Class Notes)

Today's issues include
1. General architecture
2. **Buffer Manager** – Replacement Policies / Parameters / Algorithms
3. **Questions**

1. General Architecture

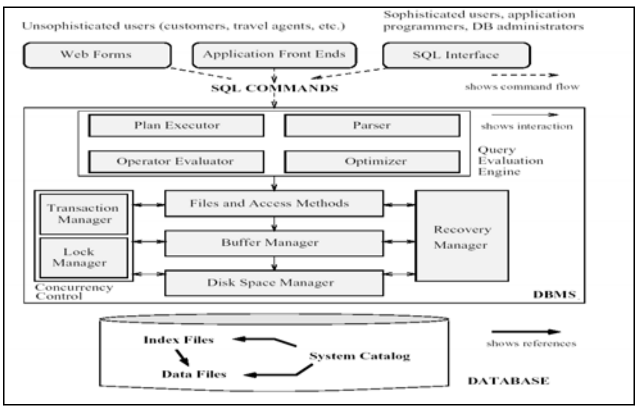
Disk Space Manager (DSM) *db.C*

- It hides the details of the underlying hardware (and OS) and allows higher levels of the software to think of the data as a collection of pages
- It provides commands to read / write / allocate / deallocate units of data (i.e. a PAGE)

```
// read a page
Status MINIBASE_DB->read_page(PageId pageid, Page* page)
// write a page to disk
Status MINIBASE_DB->write_page(PageId pageid, Page* page)
// allocate/deallocate a run of pages
// deallocate does not ensure that the page is actually allocated.
Status MINIBASE_DB->allocate_page(PageId& pageid, intnumber)
Status MINIBASE_DB->deallocate_page(PageId pageid, intnumber)
```

Buffer Manager (7.4) *buf.c*

- The role of the Buffer Manager is to keep track of the pages that are used most (so that access to those pages becomes cheap)



Buffer Manager src/

- Makefile

- main.C

→ **BMTester.C (test1, test2,….) which extends TestDriver.C**

→ **each test calls the BM manager through macros**

e.g. (MINIBASE_BM->pinPage(I,pg,0) !=OK)

- db.C (The storage manager) MINIBASE_DB

- buf.C (The buffer manager) MINIBASE_BM

The Buffer Manager Interface

class BufMgr {

public:

// This is made public just because we need it in your
// driver test.C . It could be private for real use.

Page* bufPool;

// The physical buffer pool of pages.

public:

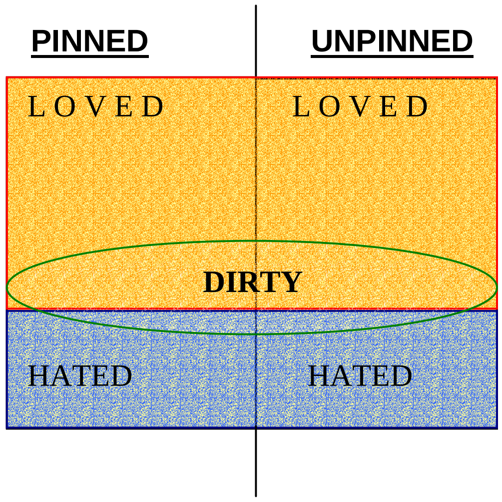
BufMgr(int numbuf, Replacer *replacer = 0);

// Allocate "numbuf" pages (frames) for the pool in main memory.

~BufMgr();

// Should flush all dirty pages in the pool to
// disk before shutting down and deallocate the
// buffer pool in main memory.

Buffer Manager Venn Diagram



This figure is the same with the one presented in class with the difference that the loved and hated areas (from pinned and unpinned area) are now merged.

**** The Query Processing System (QPS) needs to access data on disk***
// READ OPERATION e.g SELECT, UPDATE, DELETE

%%%
Status pinPage(PageId PageId_in_a_DB, Page*& page,int emptyPage=0);
- All **accesses** to pages on disk are done through **BufferManager**
- Given a **PageID** *pinPage* fetches the Page into **page (by reference)**
- Ignore **emptyPage.** In this project should be set to 0**.**
- What steps are involved?
  1) Check (logarithmic) if pageID in bufferpool.
     Use **hashtable** (either your own) or **STL map.**
     The **hashtable** will provide us a **pointer** to the page we are looking for
     Alternative? Exhaustive search (linear cost) 0(n) THEREFORE SHOULD NOT BE USED
  2) If **FOUND pageID.pincount++**
  3) if **WAS pageID.pincount==0** it means that it was a **replacement candidate**
  4) IF **NOT FOUND** fetch from **Storage Manager, Pin It and return it**
  5) Dirty Pages?: If page is removed from BM & Dirty make sure to Write it back to the Storage Manager

%%%
*The **Query Processing System (QPS)** tells us that it **don't need a page** anymore,*
*that its **dirty** (e.g. UPDATE) and that its **not required in near future.***
Status unpinPage(PageId globalPageId_in_a_DB, int dirty, int hate);
1) Pinned: decrease
   Unpinned: error PAGE_NOT_PINNED
   In all cases adjust dirty bit.
2) Lazy policy. It means we always keep the Buffer Pool as full as possible.
3) Love conquers Hate.

%%%
*The **Query Processing System (QPS)** created a new employees record **(i.e. INSERT**).*
We need to allocate a RUN of pages in the BM as well as the Storage Manager
Status newPage(PageId& firstPageId, Page*& firstpage,int howmany=1);
1) Find empty BM Frame
2) Allocate Run of new pages in Storage Manager
3) Pin Page(s)

%%%
//e.g DELETE
Status freePage(PageId globalPageId);
// This method should be called to delete a page that is on disk.
// This routine must call the DB class to deallocate the page.

%%%
e.g. COMMIT
Status flushPage(int pageid);
// Used to flush a particular page of the buffer pool to disk
// Should call the write_page method of the DB class

%%%
e.g. CLOSE DATABASE
Status flushAllPages();
// Flush all **DIRTY** pages of the buffer pool to disk
};

<u>The above are NOT all the required conditions that need to be meet.</u>

Hashtable

Buffer Manager ➔ bufPool[numbuf]

bufDescr[numbuf] *page number, pin_count, dirtybit*

Hashtable?
+ Figure out to which frame does the page we are looking for belongs to.
+ In logarithmic (at least) algorithm
+ Hash into 20 buckets and then figure out where each page/frame belongs to.

Idea: Calculate $h(value) = (a \times value + b) \bmod HTSIZE$, **for a=1, b=0and HTSIZE=20. and 20 buffer pages. Therefore we expect collision to be minimal. Of course if the sequence is biased e.g. numbers 20, 40, 60, 80, …i+20 (all these numbers hash to bucket=0 since that is the remainder) then we will have a lot of collision.**

Buf.h

#define NUMBUF 20 // BM Frames count
#define HTSIZE 20 // Hashtable size

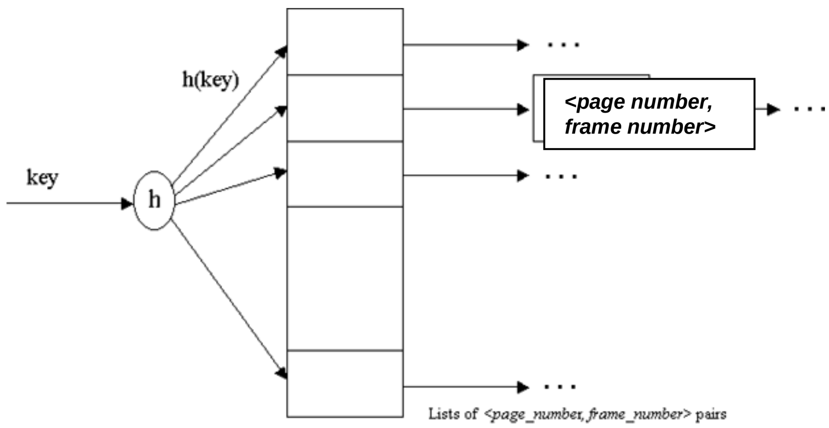


Figure 1: Hash Table

A simple *hash table* should be used to figure out what frame a given disk page occupies.The hash table should be implemented (entirely in main memory) by using an array of pointers to lists of pairs. The array is called the *directory* and each list of pairs is called a *bucket*. Given a *page number*, you should apply a *hash function* to find the directory entry pointing to the bucket that contains the frame number for this page, if the page is in the buffer pool. If you search the bucket and don't find a pair containing this page number, the page is not in the pool. If you find such a pair, it will tell you the frame in which the page resides. This is illustrated in Figure 1: