CSE190D: Topics in Database System Implementation

Project 1: BadgerDB Buffer Manager Implementation

Discussion: Friday, April 12, 2019

Presenter: Costas Zarifis
Slide Contributors: Apul Jain, Arun Kumar
Goal

Allows students to learn about the internals of data processing. Hands-on experience building the internals of a simple RDBMS

Two parts:

1. Buffer manager
2. B+-tree
BadgerDB: IO Layer

- BadgerDB: an RDBMS “skeleton” in C++
- It provides programmatic API that supports:
  - Create/destroy database files
  - Allocate/deallocate pages
  - Read/write pages
  - More!
BadgerDB: IO Layer

- File Class(File.h)
  - Provides API calls used to access and manipulate database files (create, open, remove etc)

- Page Class(Page.h)
  - API calls to allocate, read, write and delete pages
Buffer Management
Buffer Management

Page Requests from other modules of DBMS

Buffer Pool

RAM

Disk

DB

Buffer Replacement Policy decides which frame to evict

Page in an occupied frame

Free frames
The Structure of the Buffer Manager

The BadgerDB buffer manager uses three C++ classes:

- **BufDesc**: Keep track of the state of a frame
- **BufHashTbl**: Keep track of all pages in buffer pool
- **BufMgr**: Actual buffer manager, this is where you add your code
BufDesc Class - Clock Algorithm

Used to keep track of the state of each frame

Buffer is described by 4 attributes:

- **Dirty bit**: if true indicates that the page is dirty (i.e. has been updated) and thus must be written to disk before the frame is used to hold another page.
- **RefBit**: used by the clock algorithm to indicate how recently the page was used (set to 1 when frame is first fetched)
- **PinCount**: indicates how many times the page has been pinned.
- **Valid bit**: is used to indicate whether the frame contains a valid page.
Buffer Replacement: Clock Algorithm

Frame b/k:
- pinCnt
- dirty
- refbit
- ...

Requested page in Buffer Pool:
- pinCnt++/ Return handle to frame
- Else read page in from disk
  - find space in the buffer pool!

Page in use and dirty
Page in use but not dirty
Page unpinned and now not referenced
Page dirty but not ref/pinned
Use this frame!
BufDesc Class

- Use the void Set(File* filePtr, PageId pageNum) to initialize the buffer description.
- Use the void Clear() method to reset the buffer description.

```cpp
class BufDesc {
private:
    File* file;     // pointer to file object
    PageId pageNo;  // page within file
    FrameId frameNo; // buffer pool frame number
    int pinCnt;     // number of times this page has been pinned
    bool dirty;     // true if dirty; false otherwise
    bool valid;     // true if page is valid
    bool refbit;    // true if this buffer frame been referenced recently

    void Clear();     // initialize buffer frame
    void Set(File* filePtr, PageId pageNum);  //set BufDesc member variable values
    void Print();     //Print values of member variables
    BufDesc();        //Constructor
};
```
Clock Algorithm - Flowchart

1. Advance Clock Pointer
   - Valid set?
     - No
     - refbit set?
       - Yes
         - Clear refbit
       - No
         - Page pinned?
           - Yes
             - Dirty bit set?
               - Yes
                 - Flush page to disk
               - No
                 - Call "Set()" on the Frame
           - No
             - Use Frame
     - Yes
   - Use Frame
BufHashTbl Class

- Used to keep track of the pages in the buffer pool.
- Maps file and page numbers to buffer pool frames.
- Specifically, provides insert, remove and lookup functionality.
- Implemented using chained bucket hashing.

```cpp
// insert entry into hash table mapping (file, pageNo) to frameNo
void insert(const File* file, const int pageNo, const int frameNo);

// Check if (file, pageNo) is currently in the buffer pool (ie. in
// the hash table. If so, return the corresponding frame number in
frameNo.
void lookup(const File* file, const int pageNo, int& frameNo);

// remove entry obtained by hashing (file, pageNo) from hash table.
void remove(const File* file, const int pageNo);
```
BufMgr Class

The BufMgr class is the heart of the buffer manager. This (buffer.cpp) is where you write your code for this assignment.

You need to implement:

- `~BufMgr();`
- `void advanceClock();`
- `void allocBuf(FrameId& frame);`
- `void readPage(File *file, const PageId PageNo, Page*& page);`
- `void unPinPage(File *file, const PageID PageNo, const bool dirty)`
- `void allocPage(File *file, const PageID PageNo, Page *& Page)`
- `void disposePage(File * file, const PageId PageNo)`
- `void flushFile(File *file)`
Overview of Implementation Methods

`BufMgr(const int bufs)` This is the class constructor. It allocates an array for the buffer pool with bufs page frames and a corresponding BufDesc table. The way things are set up all frames will be in the clear state when the buffer pool is allocated. The hash table will also start out in an empty state.

`~BufMgr()` Flushes out all dirty pages and deallocates the buffer pool and the BufDesc table.

`void advanceClock()` Advance clock to next frame in the buffer pool.

`void allocBuf(FrameId& frame)` Allocates a free frame using the clock algorithm; if necessary, writing a dirty page back to disk. Throws BufferExceeded Exception if all buffer frames are pinned. This private method will get called by the readPage() and allocPage() methods described below. Make sure that if the buffer frame allocated has a valid page in it, you remove the appropriate entry from the hash table.
void readPage(File* file, const PageId PageNo, Page*& page) First check whether the page is already in the buffer pool by invoking the lookup() method, which may throw HashNotFoundException when page is not in the buffer pool, on the hashtable to get a frame number. There are two cases to be handled depending on the outcome of the lookup() call:

Case 1: Page is not in the buffer pool. Call allocBuf() to allocate a buffer frame and then call the method file->readPage() to read the page from disk into the buffer pool frame. Next, insert the page into the hashtable. Finally, invoke Set() the method of BufDesc on the frame to set it up properly. Set() will leave the pinCnt for the page set to 1. Return a pointer to the frame containing the page via the page parameter.

Case 2: Page is in the buffer pool. In this case set the appropriate refbit, increment the pinCnt for the page, and then return a pointer to the frame containing the page via the page parameter.
Overview of Implementation Methods

void allocPage(File* file, PageId& PageNo, Page*& page) The first step in this method is to allocate an empty page in the specified file by invoking the file->allocatePage() method. This method will return the page number of the newly allocated page. Then allocBuf() is called to obtain a buffer pool frame. Next, an entry is inserted into the hash table and Set() is invoked on the frame to set it up properly. The method returns both the page number of the newly allocated page to the caller via the pageNo parameter and a pointer to the buffer frame allocated for the page via the page parameter.

void disposePage(File* file, const PageId pageNo) This method deletes a particular page from file. Before deleting the page from file, it makes sure that if the page to be deleted is allocated a frame in the buffer pool, that frame is freed and correspondingly entry from hash table is also removed.
Overview of Implementation Methods

void flushFile(File* file) Should scan bufTable for pages belonging to the file. For each page encountered it should:

1. if the page is dirty, call file->writePage() to flush the page to disk and then set the dirty bit for the page to false
2. remove the page from the hashtable (whether the page is clean or dirty)
3. invoke the Clear() method of BufDesc for the page frame.

Throws PagePinnedException if some page of the file is pinned. Throws BadBufferException if an invalid page belonging to the file is encountered
Thanks!