ECS 165B: Database System Implementation
Lecture 2

UC Davis
March 31, 2010

Acknowledgements: design of course project for this class borrowed from CS 346 @ Stanford's RedBase project, developed by Jennifer Widom, and used with permission. Slides based on earlier ones by Raghu Ramakrishnan, Johannes Gehrke, Jennifer Widom, Bertram Ludaescher, and Michael Gertz.
Class Agenda

• Last time:
  – Logistics and course overview
  – Introduction to the DavisDB project
  – Start file and buffer management review (Chapter 9 of textbook)

• Today:
  – Finish file and buffer management review
  – File and buffer management in DavisDB

• Reading:
  – Chapter 9 of Ramarkrishnan & Gehkre
  – (or Chapter 11 of Silberschatz et al.)
Announcements

**Teaching assistant:**

Mingmin Chen ([michen@ucdavis.edu](mailto:michen@ucdavis.edu))

**Office hours:** Wednesdays, 11:00-11:50am, 055 Kemper Hall

Please send your team requests to Mingmin by email (or edit the online spreadsheet) by **end of day today!**

- We will finalize teams and set up your subversion repositories tomorrow

Project overview posted!

[http://www.cs.ucdavis.edu/~green/courses/ecs165b/project.html](http://www.cs.ucdavis.edu/~green/courses/ecs165b/project.html)

Project Part I will be posted to web page **tomorrow**, due 4/11
File and Buffer Management, Part II
Disk Space Management

- Lowest layer of DBMS software manages space on disk
- Higher levels call upon this layer to:
  - allocate / de-allocate a page
  - read / write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed
  - Simplifying assumption in DavisDB: no requests for *sequences*; pages are accessed one at a time
  - Part of student extension? (Part 5 of project)
Buffer Management in a DBMS

Page Requests from Higher Levels

BUFFER POOL

disk page

free frame

MAIN MEMORY

DISK

DB

choice of frame dictated by replacement policy

• Data must be in RAM for DBMS to operate on it!
• Table of <frameNo, pageNo> pairs is maintained
When a Page is Requested...

• If requested page is not in pool:
  – Choose a frame for *replacement*
  – If frame is dirty, write it to disk
  – Read requested page into chosen frame

• *Pin* the page and return its address

• If requests can be predicted (e.g., sequential scans), pages can be *pre-fetched* several pages at a time
  – Again, opportunity ignored in DavisDB for simplicity
More on Buffer Management

• Requestor of page must *unpin* it, and indicate whether page has been modified
  – *Dirty bit* is used for this

• Page in pool may be requested many times
  – A *pin count* is used. A page is a candidate for replacement iff its *pin count* = 0

• Concurrency control and recovery may entail additional I/O when a frame is chosen for replacement. (*Write-Ahead Log* protocol; more later...)
  – No concurrency control or recovery in DavisDB (good topic for student extension!)
Buffer Replacement Policy

• Frame is chosen for replacement by a *replacement policy*:
  – Least-recently-used (LRU), Clock, MRU, etc
  – DavisDB uses LRU

• Policy can have big impact on # of I/O's; depends on the *access pattern*

• *Sequential flooding*: nasty situation caused by LRU + repeated page scans
  – # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).
DBMS vs. OS File System

- OS does disk space and buffer management; why not let the OS manage these tasks?

- Differences in OS support: portability issues

- Some limitations, e.g., files can't span disks

- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing concurrency control and recovery)
  - adjust *replacement policy*, and pre-fetch pages based on access patterns in typical DB operations
Record Formats: Fixed-Length

- Information about field types same for all records in a file; stored in *system catalogs*
- Finding *i*’th field requires scan of record
- DavisDB uses fixed-length records
• Two alternative formats (# fields is fixed):

```
| 4 | $ | $ | $ | $ |
```

Fields Delimited by Special Symbols

```
F1  F2  F3  F4
```

Array of Field Offsets

• Second offers direct access to $i^{th}$ field, efficient storage of 
  *nulls* (special *don't know* value); small directory overhead
• *Record id = <page id, slot #>*. In first alternative, moving records for free space management changes *record id*; may not be acceptable.
• Can move records on page without changing record id; so, attractive for fixed-length records too!
Files of Records

• Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.

• **FILE**: a collection of pages, each containing a collection of records. Must support:
  
  – insert/delete/modify record
  
  – read a particular record (specified using *record id*)
  
  – scan all records (possibly with some conditions on the records to be retrieved)
Unordered (Heap) Files

• Simplest file structure contains records in no particular order

• As file grows and shrinks, disk pages are allocated and de-allocated

• To support record-level operations, we must:
  – keep track of the *pages* in a file
  – keep track of *free space* on pages
  – keep track of the *records* on a page

• There are many alternatives for keeping track of this
Heap File Implemented as a List

- The header page id and heap file name must be stored someplace
- Each page contains two "pointers" (page ids) plus data
Heap File Using a Page Directory

- The entry for a page can include the number of free bytes on the page

- The directory is a collection of pages; linked list implementation is just one alternative
  - Much smaller than linked list of all heap file pages!
System Catalogs

• For each index:
  – structure (e.g., B+-tree) and search key fields

• For each relation
  – name, file name, file structure (e.g., heap file)
  – attribute name and type, for each attribute
  – index name, for each index
  – integrity constraints

• For each view:
  – view name and definition

• Plus statistics,authorization,buffer pool size, etc
  – Catalogs are themselves stored as relations!
Example: System Catalog Table for Attributes

<table>
<thead>
<tr>
<th>attr_name</th>
<th>rel_name</th>
<th>type</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_name</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>1</td>
</tr>
<tr>
<td>rel_name</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>2</td>
</tr>
<tr>
<td>type</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>3</td>
</tr>
<tr>
<td>position</td>
<td>Attribute_Cat</td>
<td>integer</td>
<td>4</td>
</tr>
<tr>
<td>sid</td>
<td>Students</td>
<td>string</td>
<td>1</td>
</tr>
<tr>
<td>name</td>
<td>Students</td>
<td>string</td>
<td>2</td>
</tr>
<tr>
<td>login</td>
<td>Students</td>
<td>string</td>
<td>3</td>
</tr>
<tr>
<td>age</td>
<td>Students</td>
<td>integer</td>
<td>4</td>
</tr>
<tr>
<td>gpa</td>
<td>Students</td>
<td>real</td>
<td>5</td>
</tr>
<tr>
<td>fid</td>
<td>Faculty</td>
<td>string</td>
<td>1</td>
</tr>
<tr>
<td>fname</td>
<td>Faculty</td>
<td>string</td>
<td>2</td>
</tr>
<tr>
<td>sal</td>
<td>Faculty</td>
<td>real</td>
<td>3</td>
</tr>
</tbody>
</table>
Summary

• Disks provide cheap, non-volatile, but slow storage
  – Random access, but cost depends on location of page on disk; important to arrange data sequentially to minimize seek delays
    • DavisDB isn't very smart about this

• Buffer manager brings pages into RAM
  – Page stays in RAM until released by requestor
  – Written to disk when frame chosen for replacement (which is some time after requestor releases the page)
  – Choice of frame to replace based on replacement policy
  – Tries to pre-fetch several pages at a time
    • DavisDB doesn't worry about this
Summary (Continued)

• File layer keeps track of pages in a file, and supports abstraction of a collection of records.
  
  – Pages with free space identified using linked list or directory structure (similar to how pages in file are kept track of)

• Indexes support efficient retrieval of records based on the values in some fields

• Catalog relations store information about relations, indices, and views. *(Information that is common to all records in a given collection.)*
File and Buffer Management in DavisDB
File and Buffer Management in DavisDB

- **User**: Commands and results
- **Command Parser (given)**
- **Query Engine (4)**: Index scans
- **System Manager (3)**: Startup, DDL, bulk load
- **Indexing (2)**: Create/delete indices
- **Record Manager (1)**: Read/write/scan records
- **Disk Space Manager (given)**: Create files, read/write pages
- **Buffer Manager (given)**
- **OS File System**: Data, metadata

Indices

6.5 User Extension (5)
Paged File Component (Provided)

• Paged File Component has two functions:
  – provides in-memory buffer pool of pages/frames
  – performs low-level file I/O at the granularity of pages

• Overview will be posted tomorrow:
  http://www.cs.ucdavis.edu/~green/courses/ecs165b/pageFile.html

For now, see Doxygen docs:
  http://www.cs.ucdavis.edu/~green/courses/ecs165b/docs/annotated.html

• Where it all begins: PageFileManager…
PageFileManager

• Your code will create **one** instance of this class

• Manages the buffer pool of in-memory pages
  – allocate/de-allocate "scratch" pages
  – coordinates with file handle objects to bring pages to/from disk
  – uses LRU replacement policy

• Used to create/open/close/remove page files
  – Returns **FileHandle** object to manage pages within a file
FileHandle

• Returned by PageFileManager, used to:
  
  – allocate/de-allocate pages in the file
  
  – pages identified by logical **page number** rather than physical offset
  
  – mark page as dirty
  
  – force page to disk
  
  – scan pages in file
Coding Tip: Don't Forget to Free Memory!

• DBMS is a long-running process; memory leaks are unacceptable

• Every new must have a matching delete

• With some coding discipline, can avoid many problems
  – When possible, put new and delete close together in the code, so that a human can easily verify correctness
  – Memory must always be freed, even when handling exceptional conditions

• Use tools like valgrind to track down memory leaks

• We will check for memory leaks when grading your projects
Coding Tip: Pinning/Unpinning Pages

- Whenever you access a page, you must remember to unpin it after you're done (else you leak the page)
- **Best coding practice**: do both tasks nearby, ideally in the same function, so that correctness can easily be verified

```c
FileHandle* file;
PageHandle page;

ReturnCode code = file->getFirstPage(&page);
if (code == RC_OK) {
    // … do stuff with page …
    file->unpinPage(page.pageNo);
}
```

- Same goes for memory allocation/de-allocation
  - make it easy to match every `new` with its corresponding `delete`