ECS 165B: Database System Implementation
Lecture 4

UC Davis
April 6, 2011

Acknowledgements: some slides based on earlier ones by Raghu Ramakrishnan, Johannes Gehrke, Jennifer Widom, Bertram Ludaescher, and Michael Gertz.

Class Agenda

- Last time:
  - File and buffer management in DBMS
  - File and buffer management in DavisDB
- Today:
  - File and buffer management in DBMS/DavisDB, cont
  - Subversion and project logistics
  - DavisDB API, Project Part 1
- Reading:
  - Chapter 13

Announcements

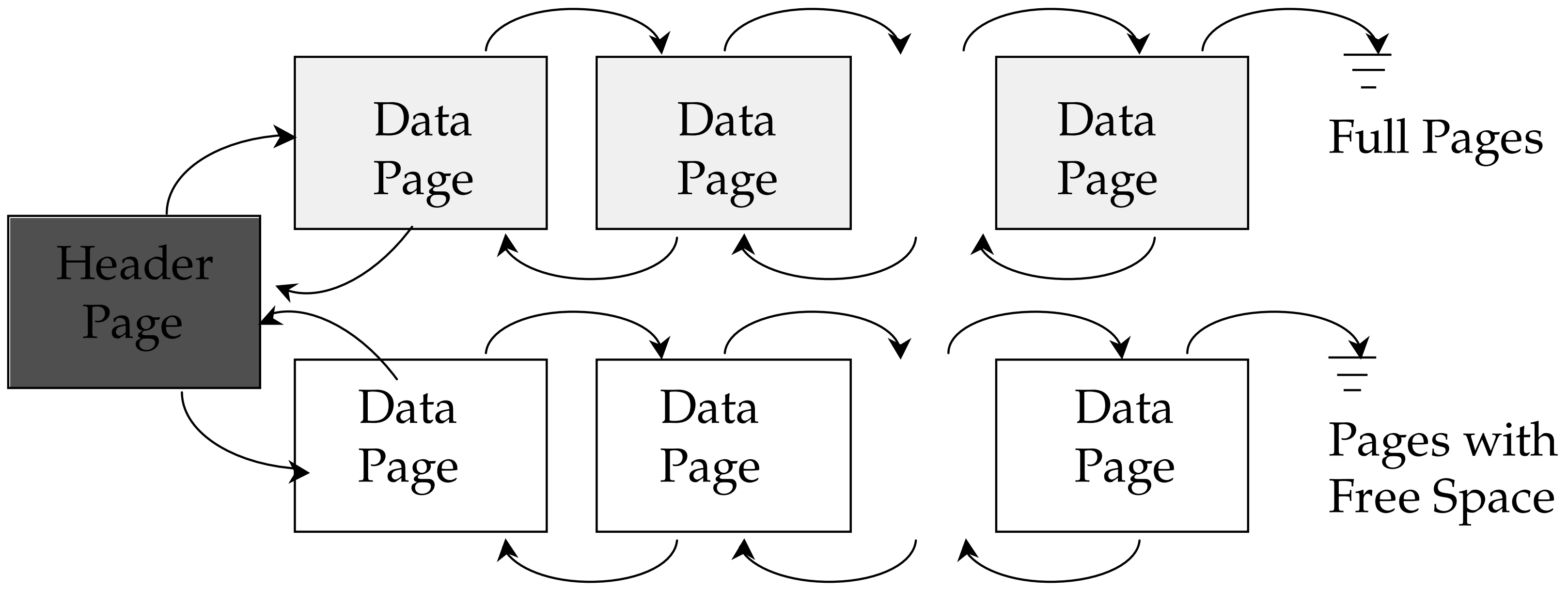
Project Part 1 posted, due Sunday 4/17 @ 11:59pm

> http://www.cs.ucdavis.edu/~green/courses/ecs165b/recordManager.html

Repositories will be set up later this morning

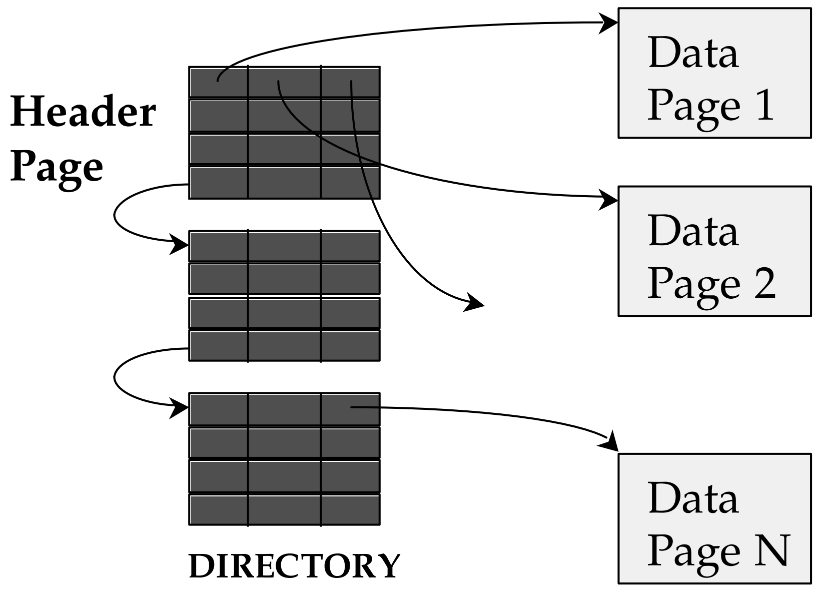
File and Buffer Management, cont’d

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 (will revisit in DavisDB, Part 3)

- 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

attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Summary

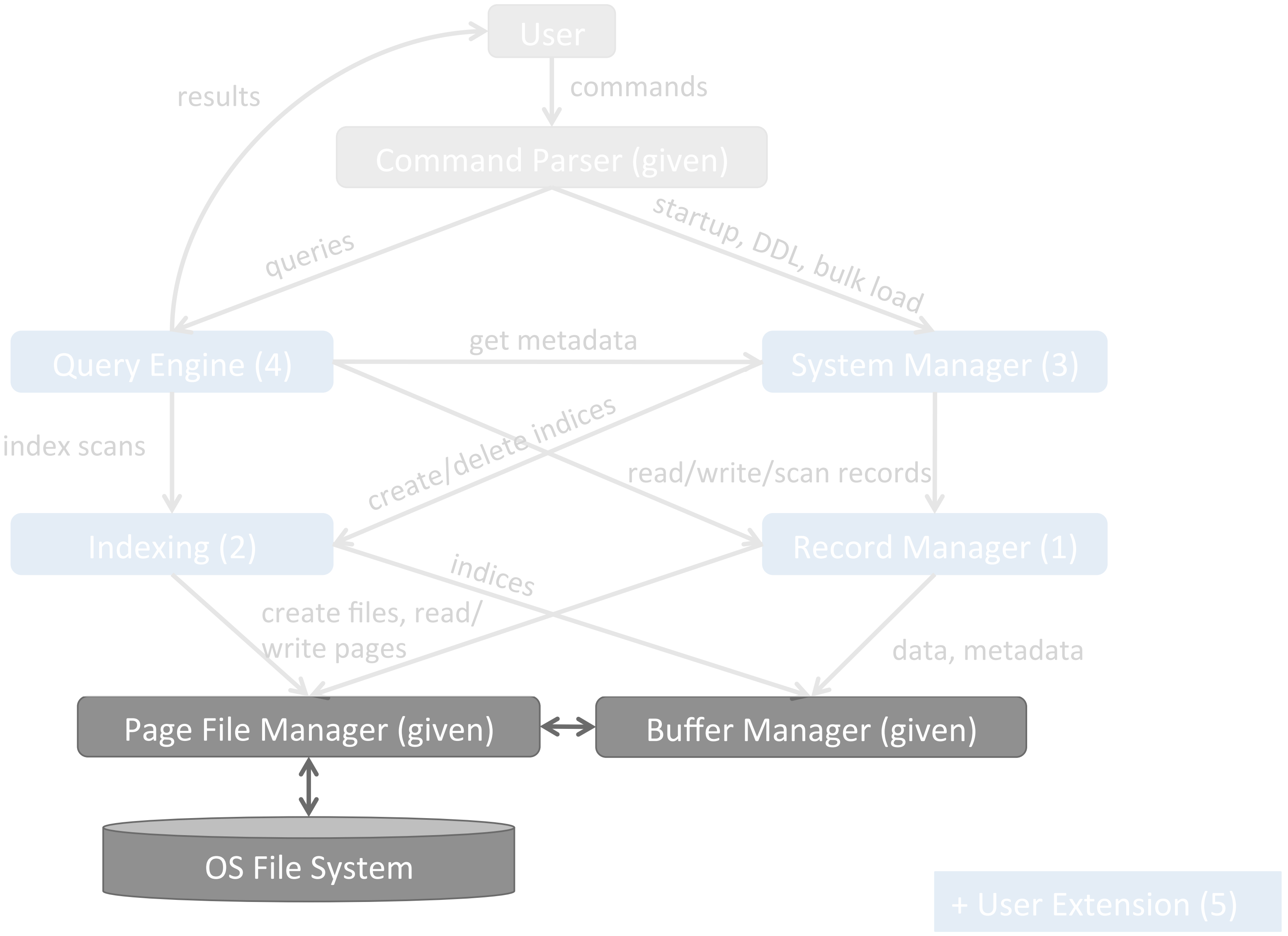
- Disks provide cheap, non-volatile, but slow storage
  - Random access, but cost depends on location of page on disk; important to arrange data sequentially to minimize *seek* delays
    - DavisDB isn't very smart about this
- Buffer manager brings pages into RAM
  - Page stays in RAM until released 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



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 on class web site:

  http://www.cs.ucdavis.edu/~green/courses/ecs165b/pageFile.html

  See also Doxygen docs:

  http://www.cs.ucdavis.edu/~green/courses/ecs165b/docs/index.html
- Where it all begins: **PageFileManager**…

PageFileManager

- Your code will use **one** singleton 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 **PageFileHandle** object to manage pages within a file

PageFileHandle

- 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

Memory Management for Shared Objects

- When several parts of the code are using same object, how do we know when object is “no longer being used”?
  - so that it can be freed (C++ object), or evicted from buffer pool (page), or ...
- Standard solution in C/C++: use **reference counting**
  - When code needs an object, increment the reference count
  - When finished, decrement the reference count
  - Pitfalls?
- For example, Microsoft COM/OLE objects have methods **addRef** and **release** to do this
- In DBMS buffer pool, traditional names for the methods are **pin** and **unpin**

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

```
PageFileHandle* 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**

Coding Tip: Don't Forget to Mark Pages Dirty!

- Be diligent about getting this right from the beginning, else you risk introducing tough-to-track-down bugs

```
FileHandle* file;
PageHandle page;

ReturnCode code = file->getFirstPage(&page);
if (code == RC_OK) {
    // … modify contents of page …
    file->markDirty(page.pageNo);
    file->unpinPage(page.pageNo);
}
```

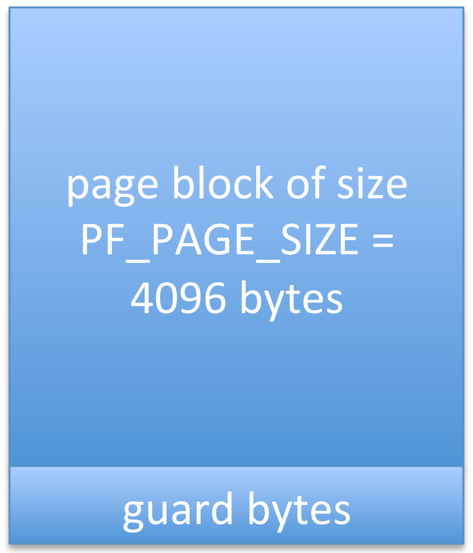
Coding Tip: Assertions in Page File Manager

- Page file manager makes heavy use of runtime assertions; some of these will catch **your** bugs!

```
ReturnCode PageFileManager::allocateBlock
(PageFileHandle* fileHandle, int pageNo, char** data) {
    // first, look for a free block, while also computing
    // the LRU unpinned block to use as backup
    int iLru = -1;
    long epochLru = LONG_MAX;
    for (uint i = 0; i < PF_BUFFER_SIZE; i++) {
        assert(pageBlocks_[i].isConsistent());
    …
    }
    …
}
```

if this assertion fires, it means somebody wrote past the end of a page block!

Coding Tip: Assertions in Page File Manager



…

```
assert(pageBlocks_[i].isConsistent());
```

…

isConsistent() checks for modification of the guard bytes following the page block

Revision Control Systems and Subversion

Why use revision control systems?

- Scenario 1:
  - Your program is working
  - You change "just one thing"
  - Your program breaks
  - You change it back
  - Your program is still broken – *why*?

- Has this ever happened to **you**?

Why use revision control systems (2)?

- Your program worked well enough yesterday
- You made a lot of improvements last night...
  - but you haven't gotten them to work yet
- You need to turn in your program *now*

- Has this ever happened to you?

Revision control for teams

- Scenario:
  - You change one part of a program -- it works
  - Your co-worker changes another part -- it works
  - You put them together -- it doesn't work
  - Some change in one part must have broken something in the other part
  - What were all the changes?

Revision Control for Teams (2)

- Scenario:
  - You make a number of improvements to a class
  - Your co-worker makes a number of *different* improvements to the *same* class
- How can you merge these changes?

Revision control systems

- A *revision control system* (aka *version control system*) does these things:
  - Keeps multiple (older and newer) versions of source code, headers, etc
  - Requests comments regarding every change
  - Displays differences between versions
  - Detect/resolve conflicts
- Many systems out there: sccs, rcs, cvs, Visual SourceSafe, svn, git, ...
  - Most popular in the past: **cvs**
  - Most popular nowadays: **svn** (also **git**)

Subversion commands

- **svn checkout (**aka **svn co)** – check out code from repository
- **svn add** – add a new file/directory to the repository
- **svn delete** – delete a file/directory from the repository
- **svn commit** – commit local changes to repository
- **svn diff** – view differences wrt current or old version
- **svn status** – see pending changes
- **svn info** – get info about repository
- **svn update** – grab new revisions from repository
- **svn help** – list all commands

- See **http://subversion.tigris.org**

- Graphical front-ends: TortoiseSVN (Windows), RapidSVN (cross-platform), Subclipse (eclipse plug-in)
  - Visual diffs, easier browsing of history, …

Logistics: Repository Access

Follow directions on

http://www.cs.ucdavis.edu/~green/courses/ecs165b/project.html

```
[green@pc12 ~]$ svn co file:///home/cs165b/CSIF-Proj/
  cs165b-0/svn/trunk/DavisDB
A     DavisDB/RecordFileHandle.h
A     DavisDB/PageFileHandle.h
A     DavisDB/PageFileManager.cpp
A     DavisDB/RecordFileManager.cpp
…
A     DavisDB/submit.sh
A     DavisDB/CMakeLists.txt
A     DavisDB/writeup.txt
A     DavisDB/Common.h
[green@pc12 ~]$
```

Logistics: Repository Access

- Must tell repository about new files!

```
[green@pc12 ~/DavisDB]$ svn add Foo.cpp Foo.h
A         Foo.cpp
A         Foo.h
[green@pc12 ~/DavisDB]$ svn commit -m ""
Adding         Foo.cpp
Adding         Foo.h
Transmitting file data ..
Committed revision 84.
```

- To get changes from your teammate:

```
[chenmi@pc10 ~/DavisDB]$ svn update
```

Logistics: Submitting Your Homework

```
[green@pc12 DavisDB]$ ./submit.sh
Usage: submit.sh <hw#>
where <hw#> is a number in the range [1,5]
```

Submits your project component by tagging the current version of your subversion repository as the submitted version. **It may be executed multiple times for the same <hw#>.** The most recently submitted version is the one that will be used for grading (and its timestamp will determine any late penalties). This script must be run from your subversion DavisDB directory.

After submitting, **the script will also run a test build of your project**, by checking out the submitted version into a temporary directory and executing "cmake ." then "make".

Logistics: Submitting Your Homework (2)

```
[green@pc12 DavisDB]$ ./submit.sh 1
Submitting HW1...
Submission successful.
Running a test build on the submitted code...
Test build successful.
```

Makefiles in DavisDB

- We won't use makefiles directly, rather, we use **cmake** makefile generator, driven by CMakeLists.txt

```
green@quatchi:~/ecs165b-s11/project/public/DavisDB$ cmake .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
...
-- Build files have been written to: /Users/green/ecs165b-s11/project/public/DavisDB
```

- Can now type **make**
- Cool benefit: can also generate Xcode projects (**cmake –g Xcode**), eclipse projects, ... from **one** spec, CMakeLists.txt