ECS 165B: Database System Implementation
Lecture 3

UC Davis
April 4, 2011

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

Class Agenda

- Last time:
  - Serialization and memory management in C++

- Today:
  - File and buffer management in DBMS
  - File and buffer management in DavisDB
  - Subversion (time allowing)

- Reading:
  - Chapter 13

Announcements

Discussion section meets today, 1:10pm-2:00pm, 223 Olson

Armen will cover warmup homework solution and gdb mini-tutorial

Project teams: please sign up by **end of day today** via online Google doc

https://spreadsheets.google.com/ccc?key=0Ag95XzE8poA1dEJiYWhxanRkZHJIUlNQbjdPU09TLUE&hl=en&authkey=CI-svc0N

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

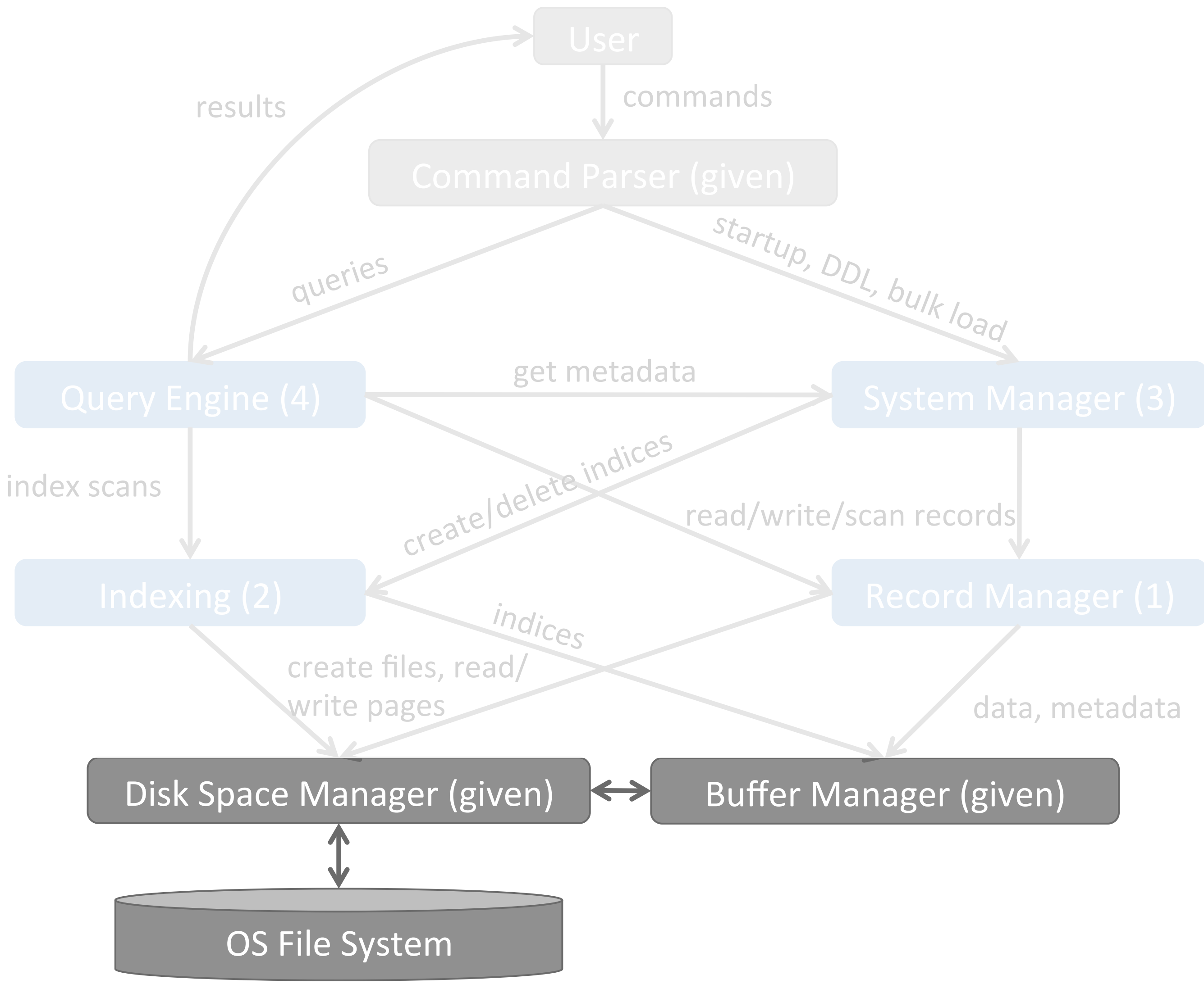
Project overview posted!

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

Project Part I will be finalized and sent out tomorrow, due Sunday 4/17 @ 11:59pm

File and Buffer Management in a DBMS

File and Buffer Management in DavisDB



Disks and Files

- (Traditional) DBMS stores information on hard disks

- This has major implications for DBMS design!

  - READ: transfer data from disk to memory (RAM)
  - WRITE: transfer data from RAM to disk
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
  - DavisDB I/O efficiency contest: minimize total READS and WRITES

Why Not Store Everything in Main Memory?

- Traditional arguments:
  - *It costs too much*. In 1995, $1000 would buy you either 128MB of RAM or 7.5GB of disk.
  - *Main memory is volatile*. We want data to be saved between runs. (Obviously!)
- Traditional storage hierarchy:
  - Main memory (RAM) for currently-used data
  - Disk for the main database (secondary storage)
  - Tapes for archiving older versions of the data (tertiary storage)
- DavisDB follows traditional model (minus the tapes ☺)
- Discussion: do the traditional arguments still hold water?

Disks and Paged Files

- Secondary storage device of choice
- Main advantage over tapes: *random access* versus *sequential*
- Data on hard disks is stored and retrieved in units called *disk blocks* or (as we'll term them in DavisDB) *pages*
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk…
- …therefore, relative placement of pages on disk has major impact on DBMS performance!
  - For simplicity, we'll overlook this in DavisDB
- File is organized as a sequence of pages

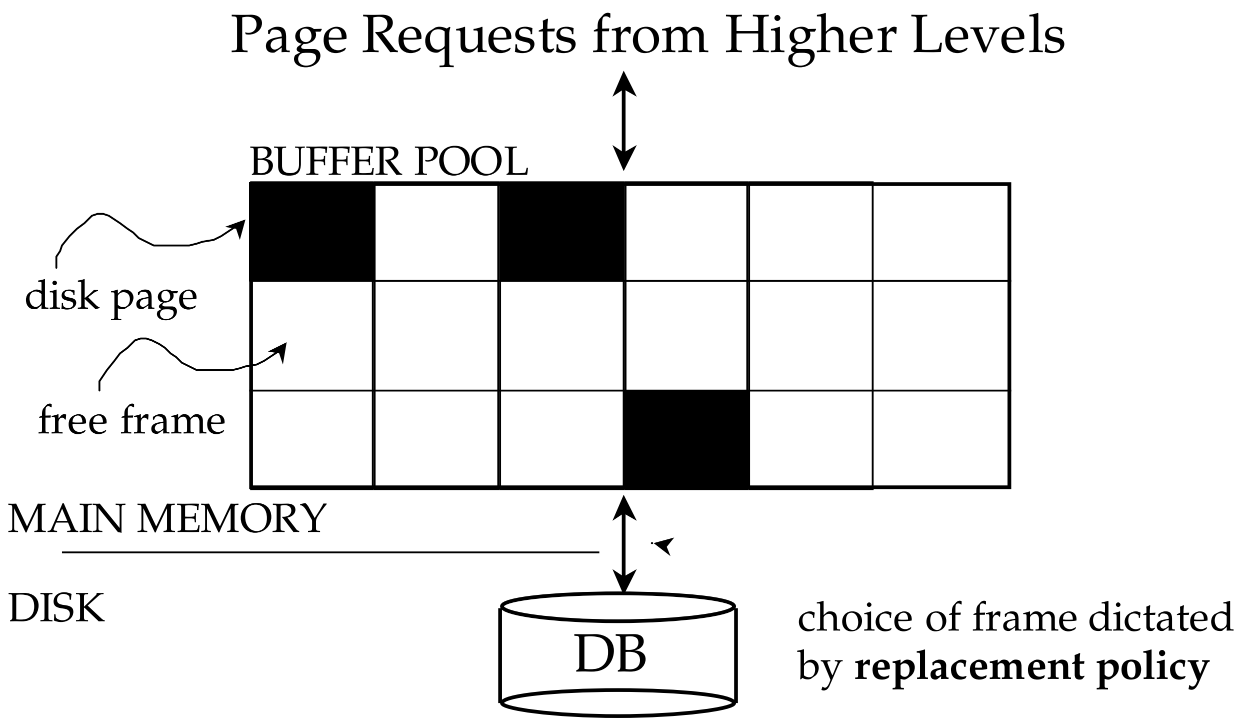
Buffer Management

- Main memory is limited
- Pages of disk files move in/out of in-memory *buffer pool*
- DavisDB # pages in buffer pool = 40
- Total buffer size (40 pages @4K pages) = 160K (tiny!)

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

Buffer Management in a DBMS



- 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 ("*write on replacement*")
  - 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* (aka *reference 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

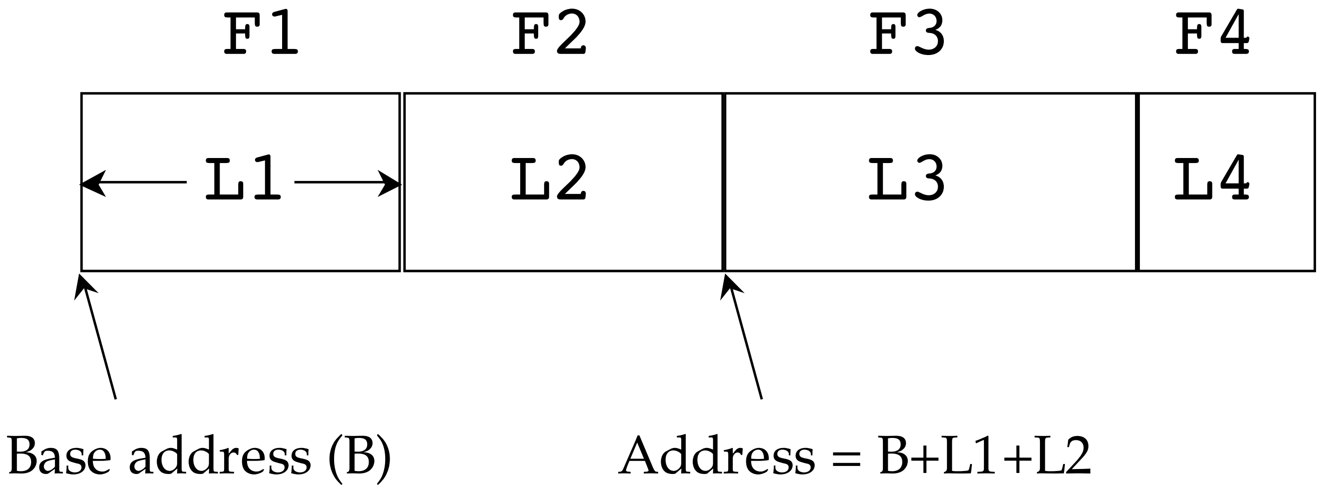
Buffer Replacement Policy

- 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 technical 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

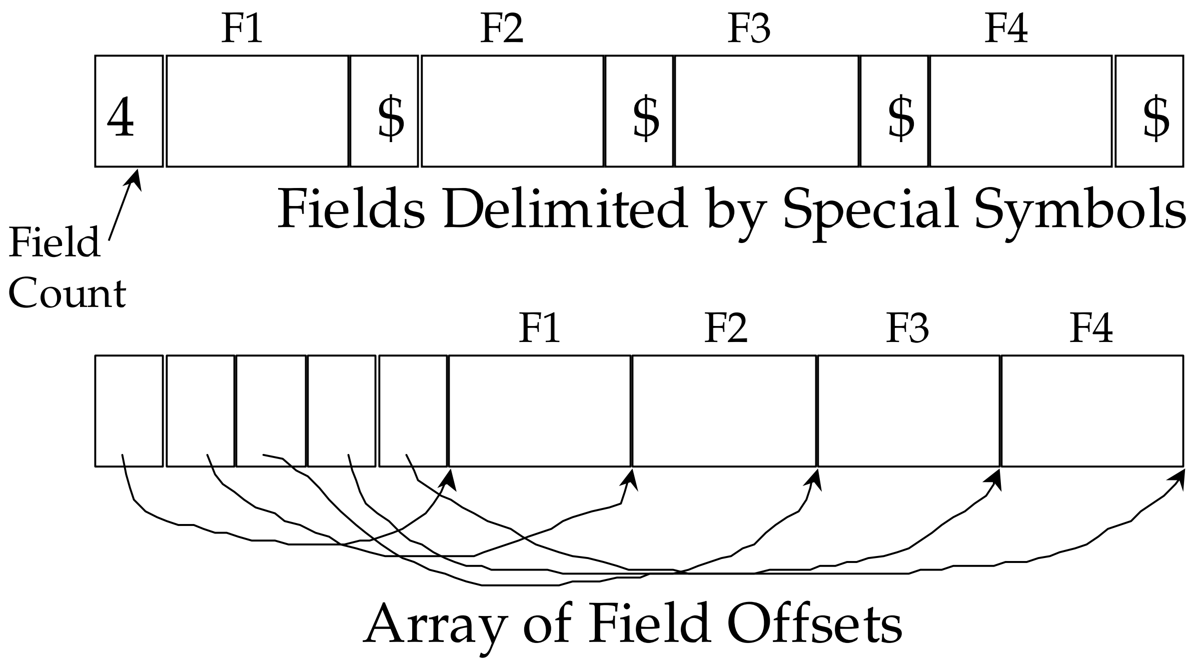
Record Formats: Fixed-Length



- 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

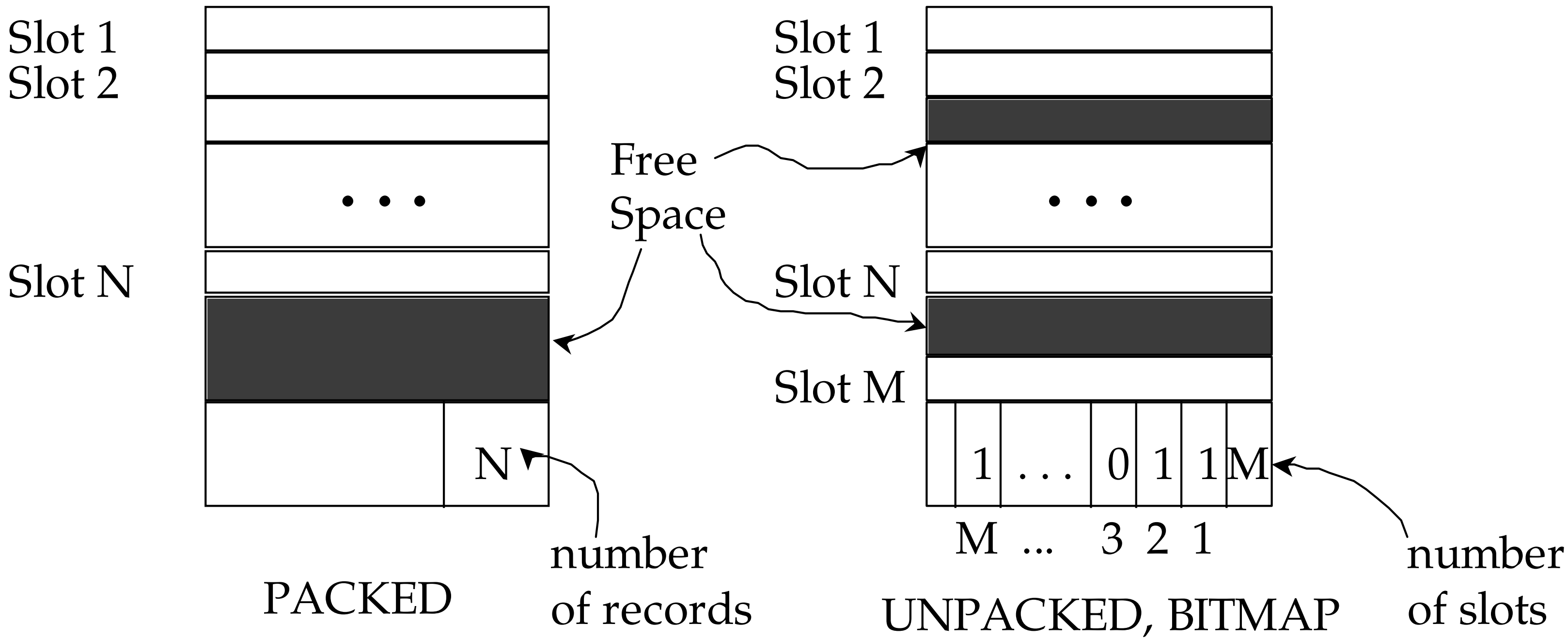
Record Formats: Variable-Length

- Two alternative formats (# fields is fixed):



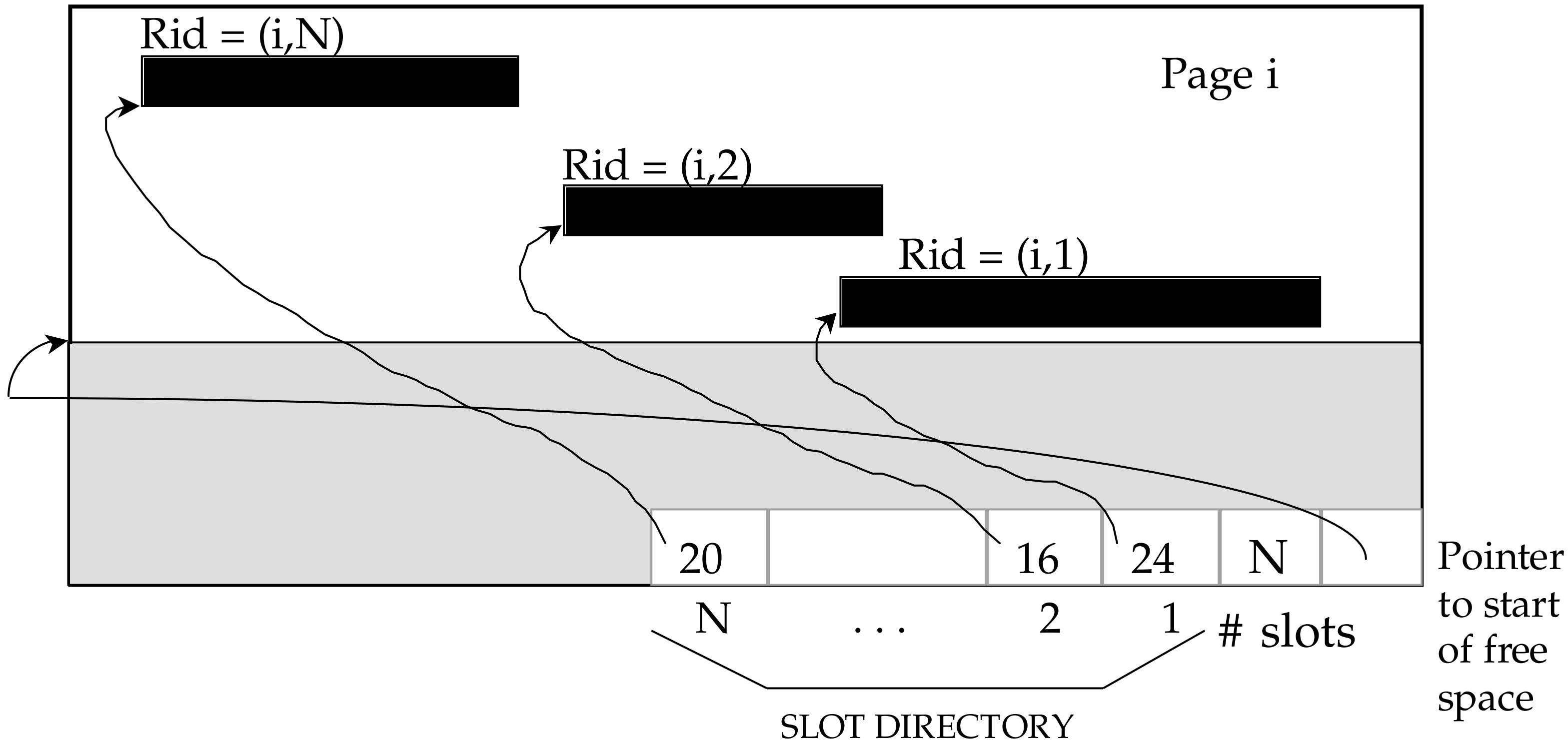
- Second offers direct access to *i'th* field, efficient storage of *nulls* (special *don't know* value); small directory overhead

Page Formats: Fixed-Length Records



- *Record id* = <*page id*, *slot #*>. In first alternative, moving records for free space management changes *record id*; may not be acceptable.

Page Formats: Variable-Length Records



- Can move records on page without changing *record id*; so, attractive for fixed-length records too!

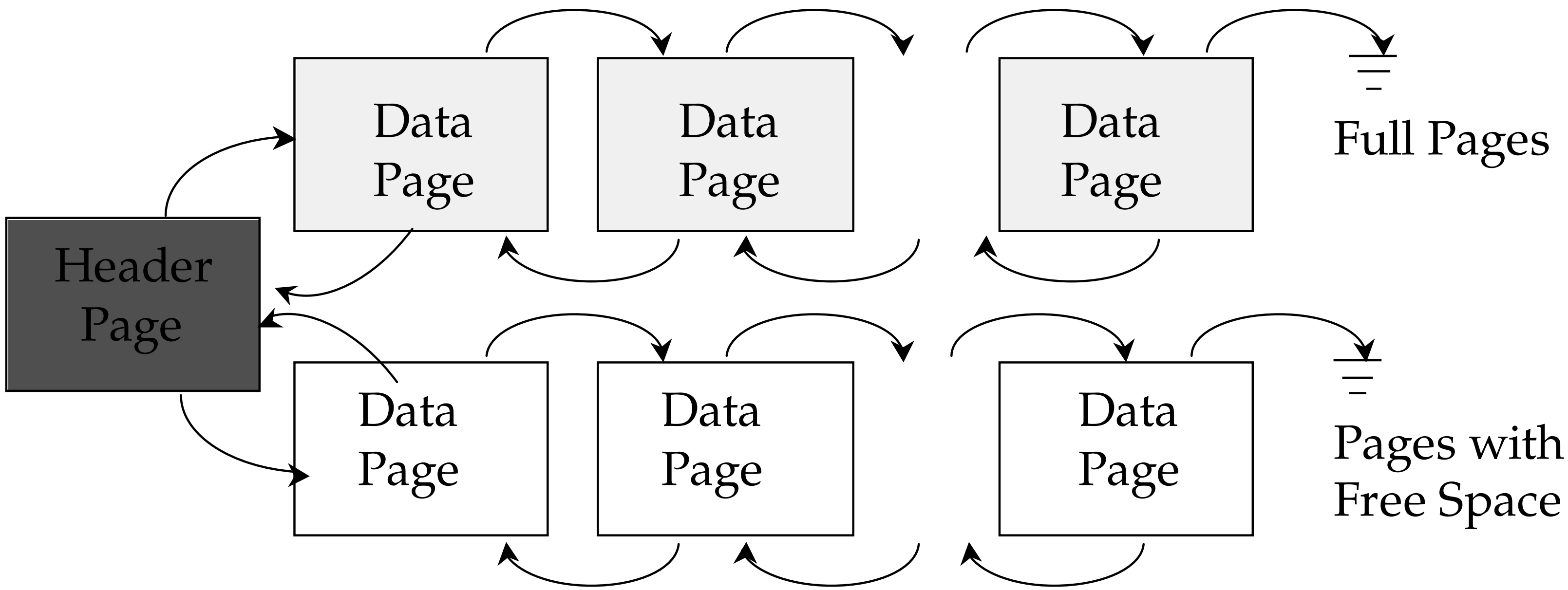
Files of Records

- 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