

Introduction to DBMS Internals

DBMS Architecture

Data storage

Alvin Cheung

Reading: R & G Chapter 9



Course Overview

- Unit 1: Relational model and SQL
- Unit 2: Storage and indexing
- Unit 3: Query execution
- Unit 4: Query optimization
- Unit 5: Conceptual design
- Unit 6: Transactions
- Unit 7: Recovery
- Unit 8: NoSQL



DBMS Architecture

Architecture of a DBMS: SQL Client



- How is a SQL query executed?



DBMS: Parsing & Optimization



Purpose:

Parse, check, and verify the SQL

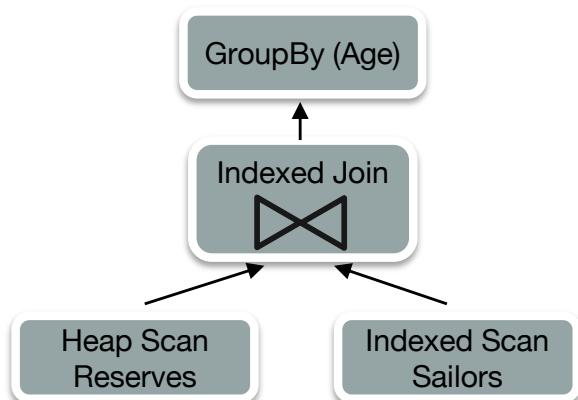
```
SELECT S.sid, S.sname, R.bid  
FROM Sailors R, Reserves R  
WHERE S.sid = R.sid and S.age > 30  
GROUP BY age
```

And translate into an efficient relational query plan



DBMS: Relational Operators

Purpose: Execute query plan by operating on **records** and **files**

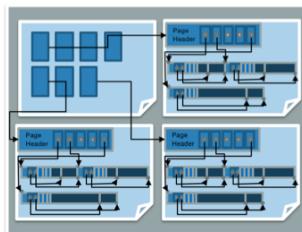


DBMS: Files and Index Management



Purpose: Organize tables and Records as groups of pages in a logical file

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	\$400
443	Grouch	Oscar	32	\$300
244	Oz	Bert	55	\$140
134	Sanders	Ernie	55	\$400

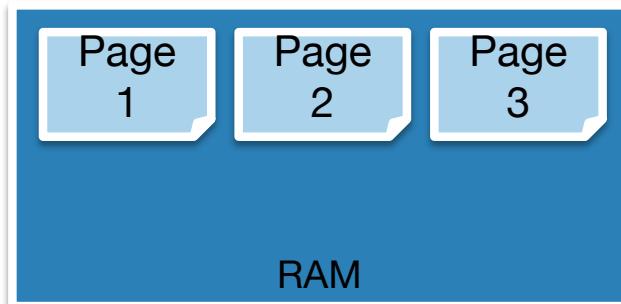


DBMS: Buffer Management



Purpose:

Provide the illusion of operating
in memory



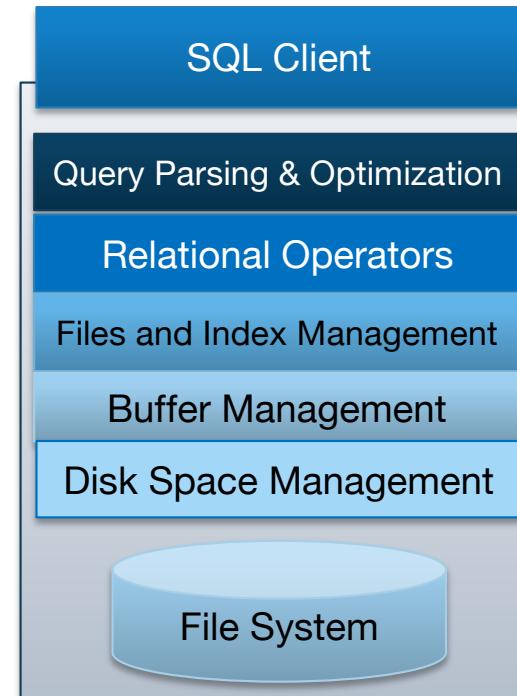
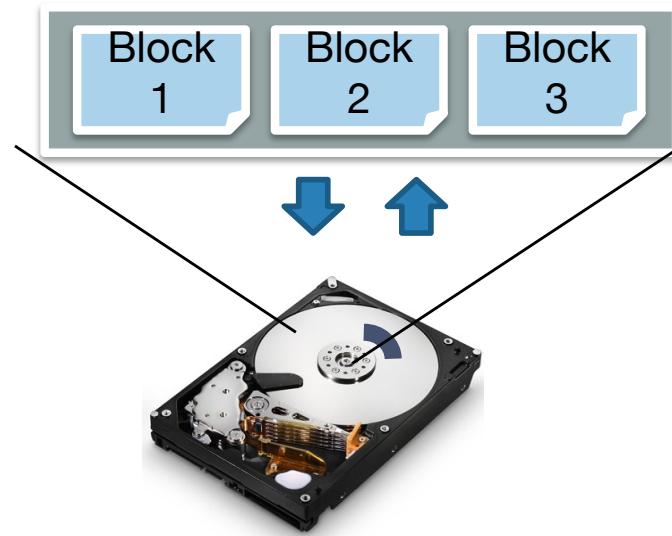
Disk Space Management



DBMS: Disk Space Management



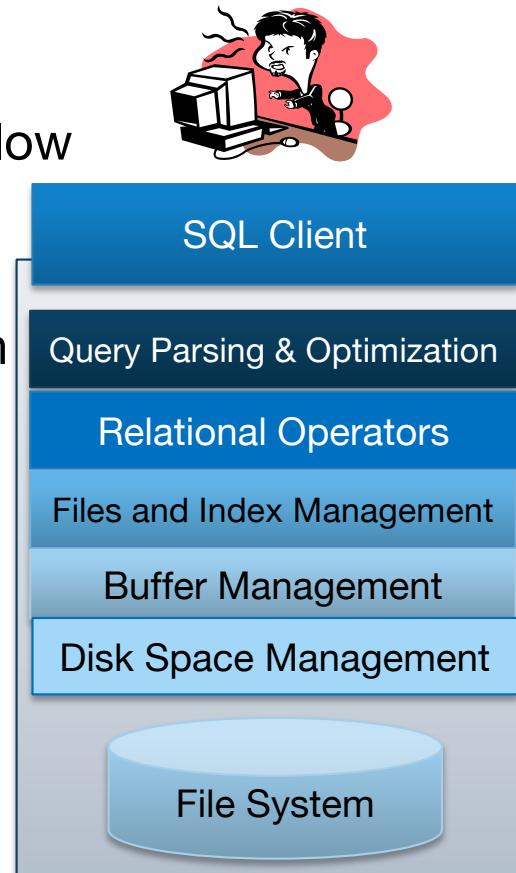
Purpose: Translate page requests into physical bytes on one or more device(s)



Architecture of a DBMS



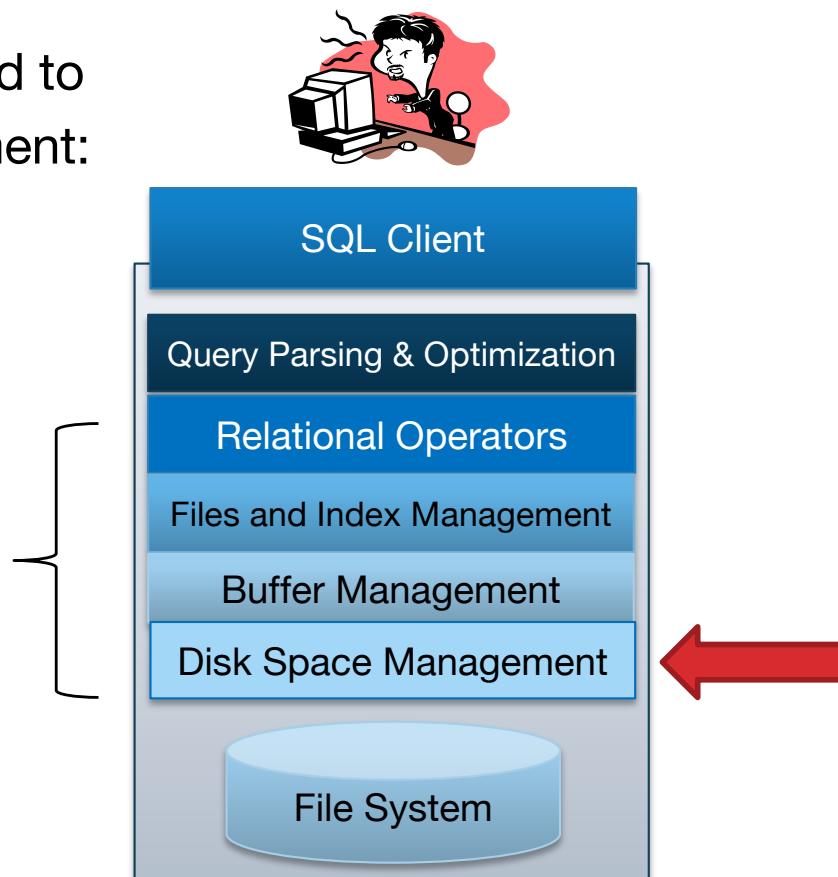
- Organized in layers
- Each layer abstracts the layer below
 - Manage complexity
 - Performance assumptions
- Example of good systems design
- Many non-relational DBMSs are structured similarly



DBMS: Concurrency & Recovery

Two cross-cutting issues related to storage and memory management:

Concurrency Control
Recovery



STORAGE MEDIA

Disks

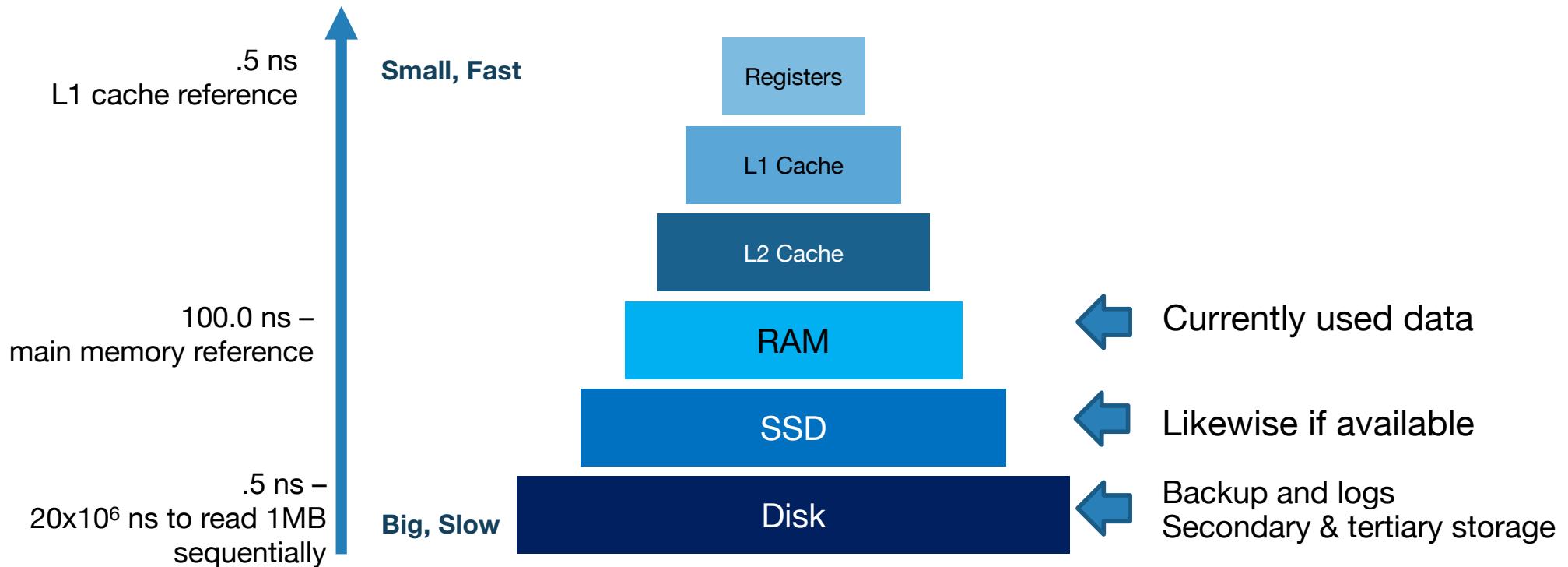


- Most database systems were originally designed for magnetic “spinning” disks
 - Disk are a mechanical anachronism!
 - Instilled design ideas that apply to using solid state disks as well
- Major implications:
 - Disk API:
 - **READ**: transfer “page” of data from disk to RAM.
 - **WRITE**: transfer “page” of data from RAM to disk.
 - No random reads / writes!!
 - Both API calls are very, very slow!
 - Plan carefully!



CS 162: Operating Systems
and System Programming

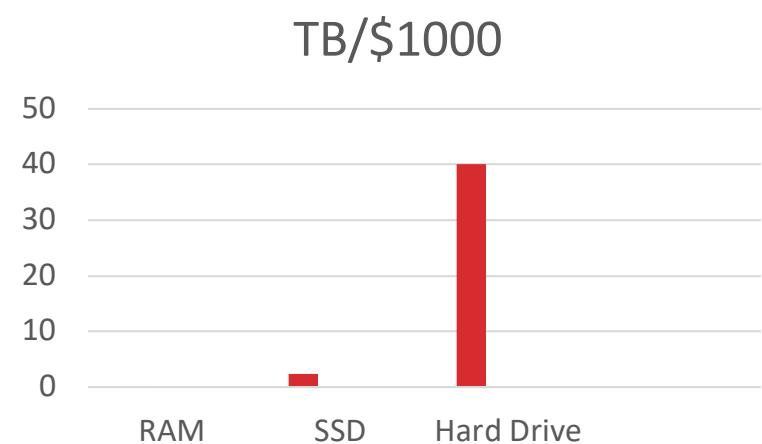
Storage Hierarchy



Economics



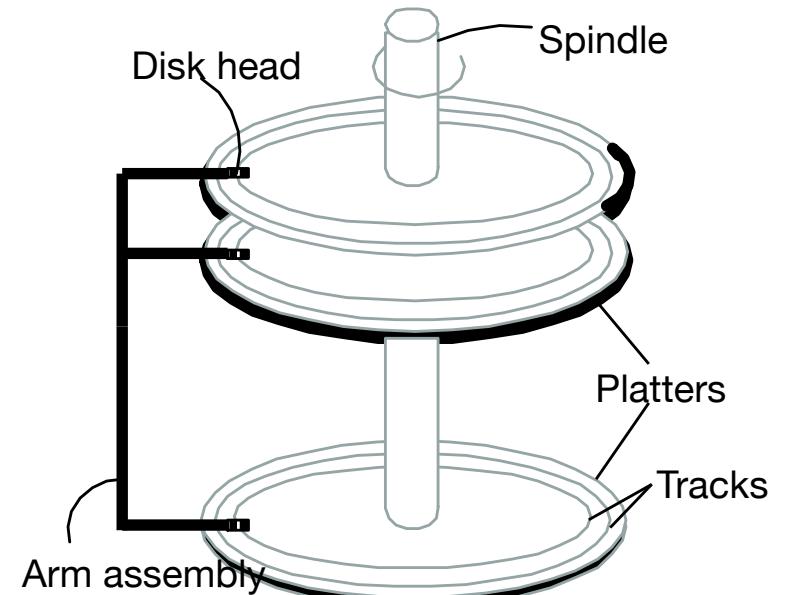
- \$1000 at NewEgg 2022:
 - Mag Disk: ~40TB for \$1000
 - SSD: ~2.3TB for \$1000
 - RAM: 128GB for \$1000



Components of a Disk



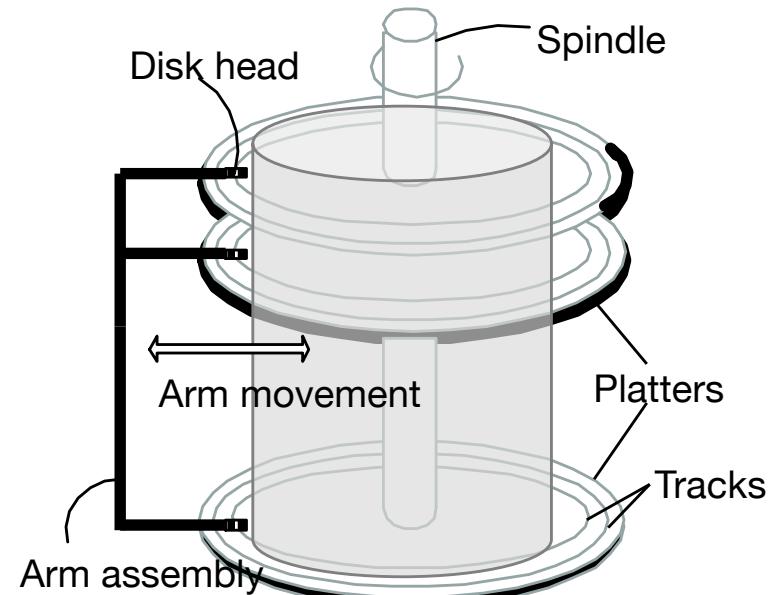
- **Platters** spin (say 15000 rpm)
- **Arm assembly** moved in or out to position a **head** on a desired **track**
 - Tracks under heads make a “cylinder”



Components of a Disk



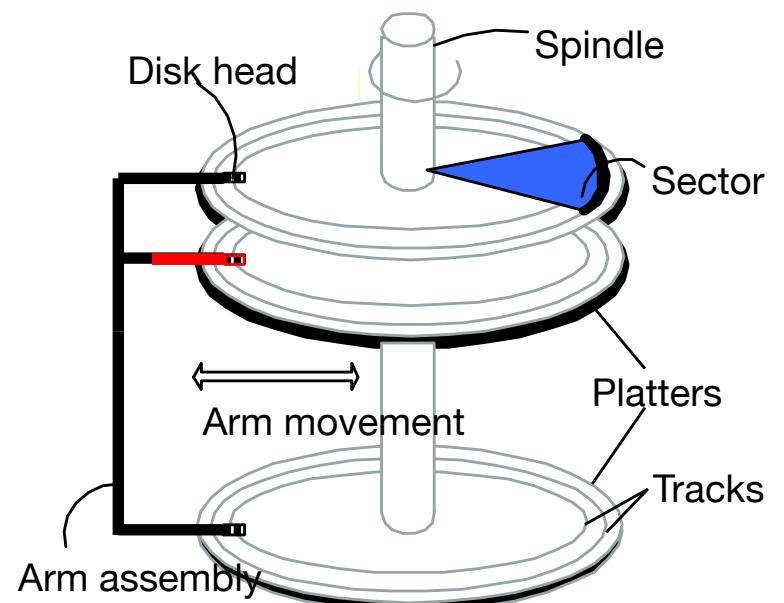
- **Platters** spin (say 15000 rpm)
- **Arm assembly** moved in or out to position a **head** on a desired **track**
 - Tracks under heads make a “cylinder”



Components of a Disk



- **Platters** spin (say 15000 rpm)
- **Arm assembly** moved in or out to position a **head** on a desired **track**
 - Tracks under heads make a “cylinder”
- Only **one head reads/writes at any one time**
- Block/page size is a multiple of (fixed) **sector** size



An Analogy



Accessing a Disk page



- Time to access (read/write) a disk block:
 - **seek time** (moving arms to position disk head on track)
 - ~2-3 ms on average
 - **rotational delay** (waiting for block to rotate under head)
 - ~0-4 ms (15000 RPM)
 - **transfer time** (actually moving data to/from disk surface)
 - ~0.25 ms per 64KB page
- Key to lower I/O cost: reduce seek/rotational delays

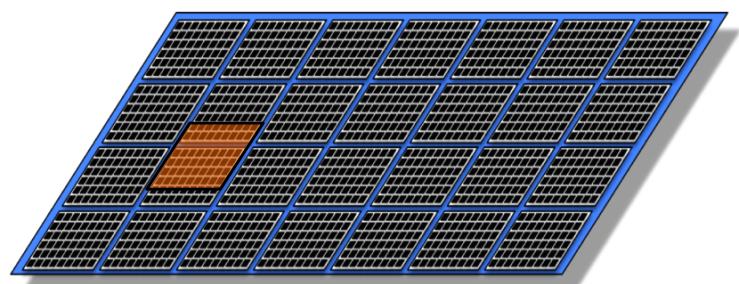
Flash (SSD)



- Organized into “cells”
 - Current generation (NAND)
 - Random reads and writes, but:
 - Fine-grain reads (4-8K reads), coarse-grain writes (1-2MB writes)
- {

Small/fast read

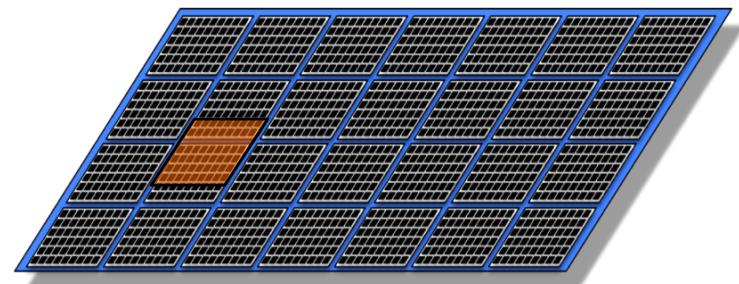
Big/slow write



Flash (SSD), Pt. 2



- So... **read is fast and predictable**
 - 4KB random reads: ~500MB/sec
- But **write is not!**
 - 4KB random writes: ~120 MB/sec
 - Why? Only 2k-3k erasures before failure
 - so keep moving write units around (“wear leveling”)



(writes)
When find faulty cell, need + find another
cell w/ same info

DISK SPACE MANAGEMENT

Block Level Storage



- Read and Write **large chunks of sequential bytes**
- Sequentially: “Next” disk block is fastest (*& opposite of random*)
- Maximize usage of data per Read/Write
 - “Amortize” seek delays (HDDs) and writes (SSDs):
if you’re going all the way to Pluto, pack the spaceship full!
- Predict future behavior
 - Cache popular blocks
 - Pre-fetch likely-to-be-accessed blocks
 - Buffer writes to sequential blocks
 - More on these as we go

A Note on Terminology

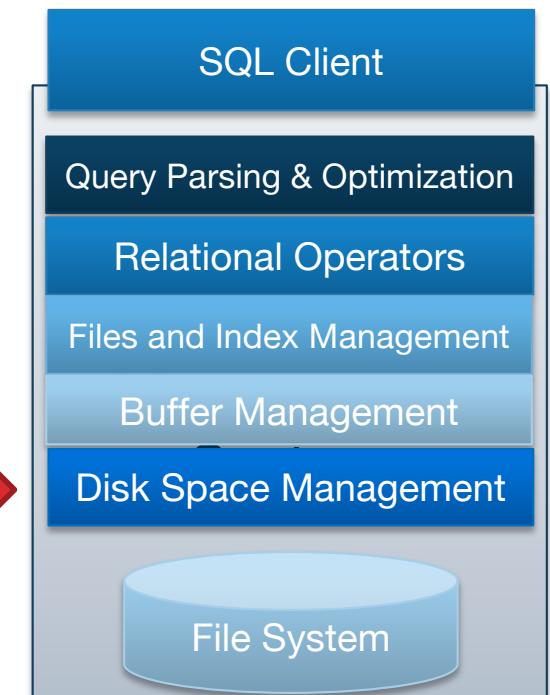


- **Block** = Unit of transfer for disk read/write
 - 64KB – 128KB is a good number today
 - Book says 4KB
 - We'll use this unit for all storage devices
- **Page**: a common synonym for “block”
 - In some texts, “page” = a block-sized chunk of RAM
- We'll treat “block” and “page” as synonyms

Disk Space Management



- Lowest layer of DBMS, manages space on disk
- Purpose:
 - Map pages to locations on disk
 - Load pages from disk to memory
 - Save pages back to disk & ensuring writes
- Higher levels call upon this layer to:
 - Read/write a page
 - Allocate/de-allocate logical pages



Disk Space Management: Requesting Pages



- page = getFirstPage("Sailors");
while (!done) {
 process(page);
 page = page.nextPage();
}
- Physical details hidden from higher levels of system
- Higher levels may “safely” assume nextPage is fast
 - Hence sequential runs of pages are quick to scan

↑

b/c read is small and
-art and reading sequentially
rather than random

Disk Space Management: Implementation



- **Proposal 1:** Talk to the storage device directly
 - Could be very fast if you knew the device well
 - Hard to program when each device has its own API
 - What happens when devices change?

Disk Space Management: Implementation



- **Proposal 2:** Run our own over filesystem (FS)
 - Bypass the OS, allocate single large “contiguous” file on an empty disk
 - assume sequential/nearby byte access are fast
 - Most FS optimize disk layout for sequential access
 - Gives us what we want if we start with an empty disk
 - DBMS “file” may span multiple FS files on multiple disks/machines

Disks and Files: Summary



- Magnetic (hard) disks and SSDs
 - Basic HDD and SSD mechanics
 - Concept of “near” pages and how it relates to cost of access
- Relative cost of
 - Random vs. sequential disk access (10x)
 - Disk (Pluto) vs RAM (Sacramento) vs. registers (your head)
 - Big, big differences!

Files: Summary



- DB File storage
 - Typically over FS file(s)
- Disk space manager loads and stores pages
 - Block level reasoning
 - Abstracts device and file system; provides fast “next page”

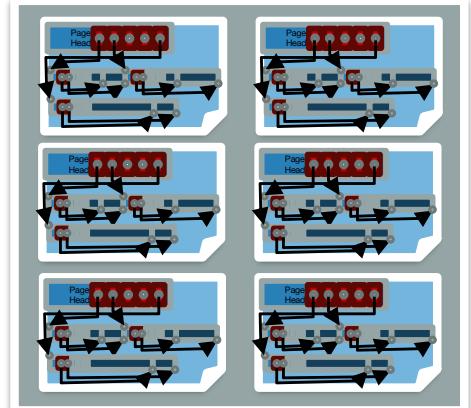
Disk Representations: Files, Pages, Records



Overview: Files of Pages of Records



- It's all about abstractions!
 - Each table is stored in one or more OS files
 - Each file contains many pages
 - Each page contains many records
- Pages are the common currency understood by multiple layers:
 - Managed on disk by the disk space manager:
pages read/written to physical disk/files
 - Managed in memory by the buffer manager:
higher levels of DBMS only operate in memory



Files of Pages of Records

- Let's talk about a single table for now
- **DB FILE:** A collection of pages, each containing a collection of records.
- API for higher layers of the DBMS:
 - **Reads:**
 - Fetch a particular record by **record id** ...
 - Record id is a pointer encoding pair of **(pageID, location** on page)
 - Scan all records
 - Possibly with some conditions on the records to be retrieved
 - **Updates:** Insert/delete/modify record
- This abstraction could span multiple OS files and even machines



Many DB File Structures



Information is stored in files in multiple different ways

- **Unordered Heap Files**
 - Records placed arbitrarily across pages
- **Clustered Heap Files**
 - Records and pages are grouped in some meaningful way
- **Sorted Files**
 - Pages and records are in strict sorted order
- **Index Files**
 - B+ Trees, Linear Hashing, ...
 - May contain records or point to records in other files
- Focus on Unordered Heap Files for now...

Unordered Heap Files

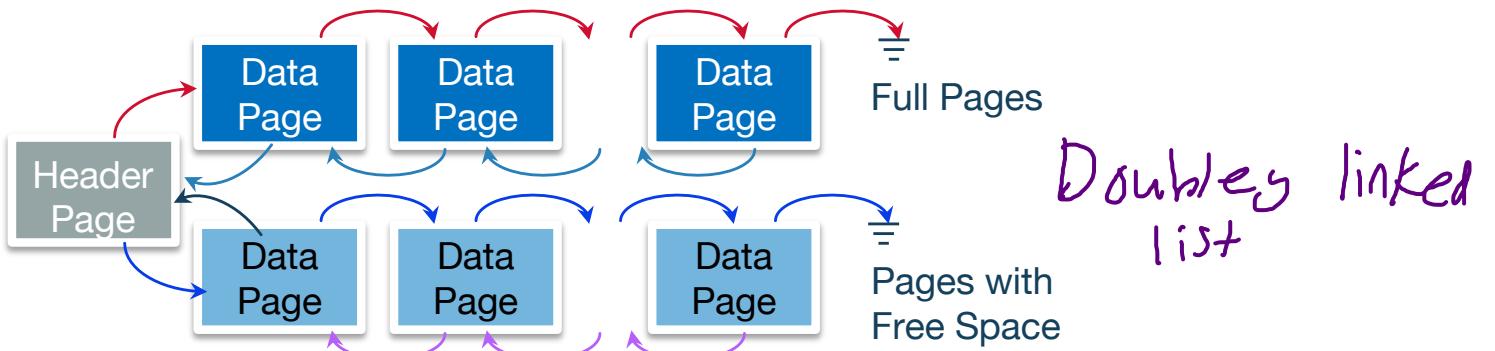


- Collection of records in no particular order
 - Not to be confused with “heap” data-structure: efficient max/min
- As file shrinks/grows, pages (de)allocated
- To support record level operations, we must
 - Keep track of the pages in a file
 - Keep track of free space on pages
 - Keep track of the records on a page

Take 1: Heap File as List

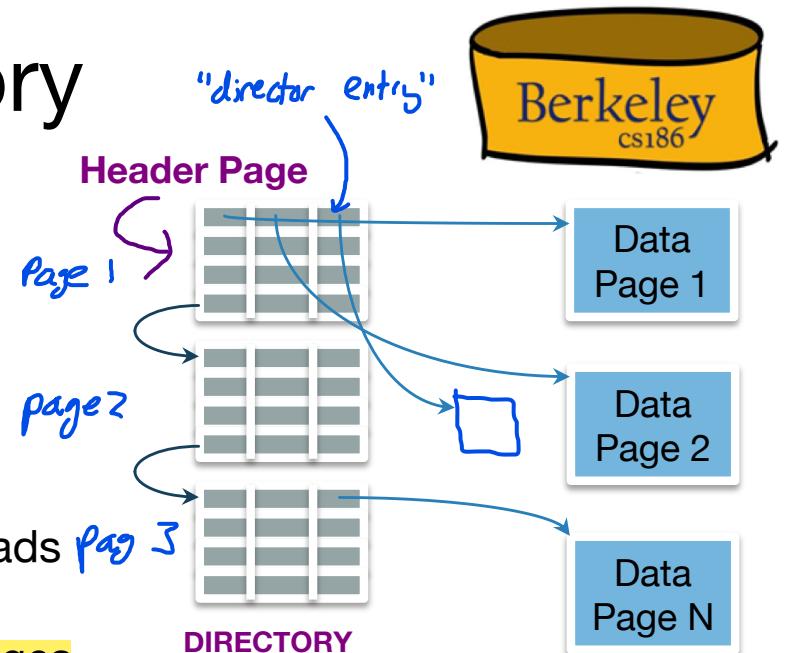


- Heap file has one special “Header page”
 - Location of the heap file and the header page saved e.g., in catalog
- Each page contains 2 “pointers” plus **free space** and **data**
- What is wrong with this?
 - Find a page with enough space for a 20 byte record?
 - A: Might need to access many pages (w/ free space) to check



Take 2: Use a Page Directory

- Directory, with multiple **Header Pages**, each encoding:
 - A pointer to page
 - #free bytes on the page
- There can be multiple such header pages
- Header pages accessed often → likely in cache
- Finding a page to fit a record required far fewer page loads than linked list. Why?
 - One header page load reveals free space of many pages
- We can optimize the page directory further:
 - E.g., compressing header page, keeping header page in sorted order based on free space, etc.
 - But diminishing returns?

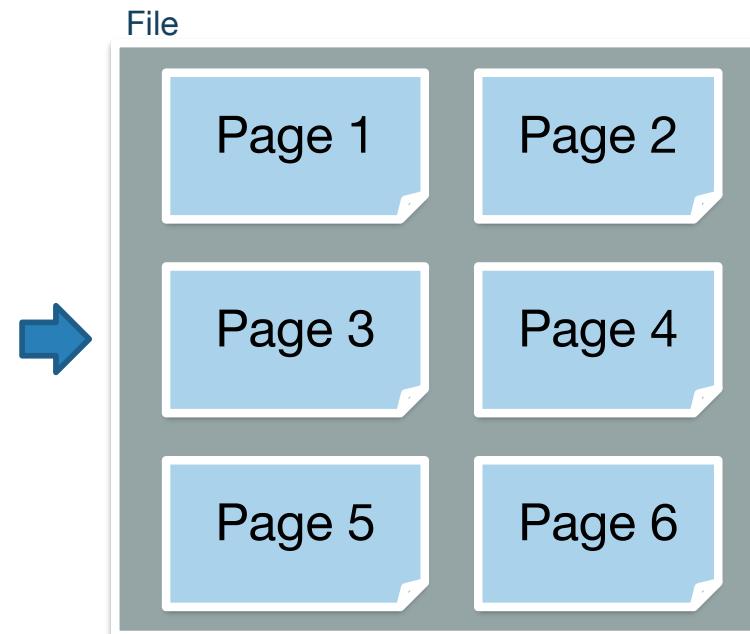


Summary



- Table encoded as files which are collections of pages
- Page directory provides locations of pages and free space

SSNz	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	\$400
443	Grouch	Oscar	32	\$300
244	Oz	Bert	55	\$140
134	Sanders	Ernie	55	\$400





PAGE LAYOUT

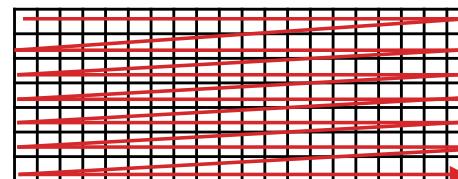
A Note On Imagery



- Data (in memory or disk) is stored in linear order



- This doesn't fit nicely on screen
 - So we will “wrap around” the linear order into a rectangle



Page Basics: The Header



- Header may contain “**metadata**” about the page, e.g.
 - Number of records
 - Free space
 - Maybe a next/last pointer
 - Bitmaps, Slot Table
 - (We’ll talk about why all of these later)

Page Header

Things to Address



Some options:

- Record length? **Fixed** or Variable
- Page layout? **Packed** or Unpacked

Some questions:

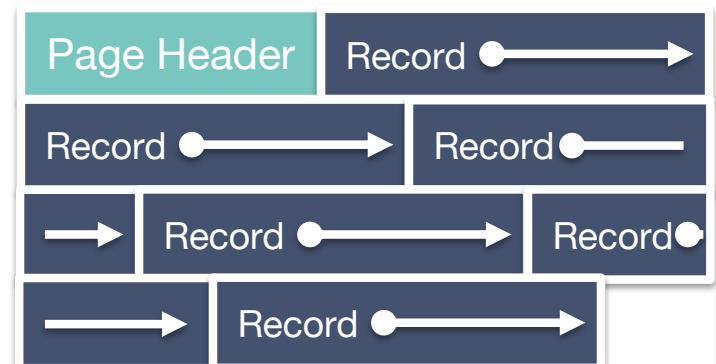
- Find records by record id?
 - Record id = (Page, Location in Page)
- How do we add and delete records?

Page Header

Fixed Length Records, Packed



- ① • Pack records densely
- ② • Record id = (pageId, “location in page”)?
 - (pageId, record number in page)!
 - We know the offset from start of page!
 - Offset = header + (record size) x (n-1)
- Easy to add: just append
- Delete?

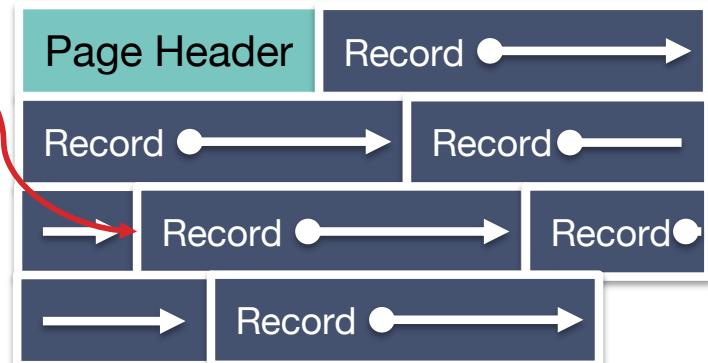


Fixed Length Records, Packed, Pt 2.



- Pack records densely
- Record id = (pageId, “location in page”)?
 - (pageId, record number in page)!
 - We know the offset from start of page!
- Easy to add: just append
- Delete?
 - Say we delete (Page 2, Record 3)

Record id:
(Page 2, Record 4)

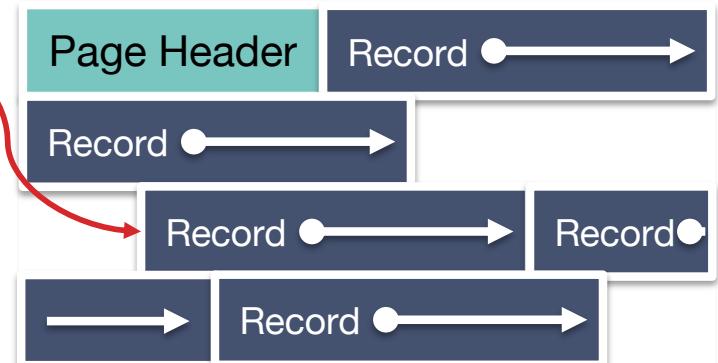


Fixed Length Records: Packed, Pt 3.



- Pack records densely
- Record id = (pageld, “location in page”)?
 - (pageld, record number in page)!
 - We know the offset from start of page!
- Easy to add: just append
- Delete?
 - Say we delete (Page 2, Record 3)
 - Now free space... need to reorg

Record id:
(Page 2, Record 4)



Fixed Length Records: Packed, Pt. 5



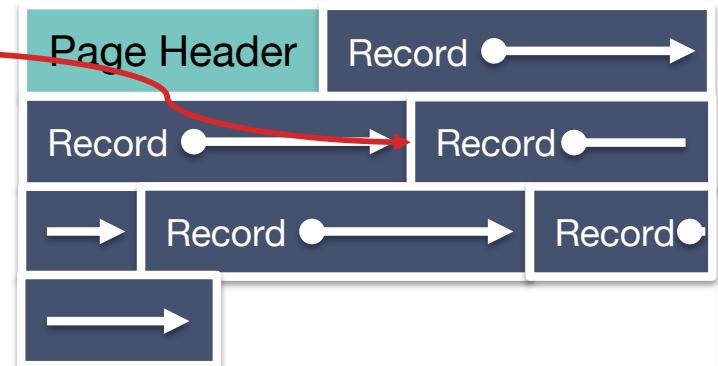
Pack records densely

Record id = (pageld, “location in page”)?

- (pageld, record number in page)!
- We know the offset from start of page!
- Easy to add: just append
- Delete?
 - Packed implies re-arrange!
 - “record id” - (Page 2, Record 4) now need to be updated to (Page 2, Record 3)
 - Record Ids need to be updated!
 - Could be expensive if they’re in other files.

Record id:

(Page 2, Record 3)

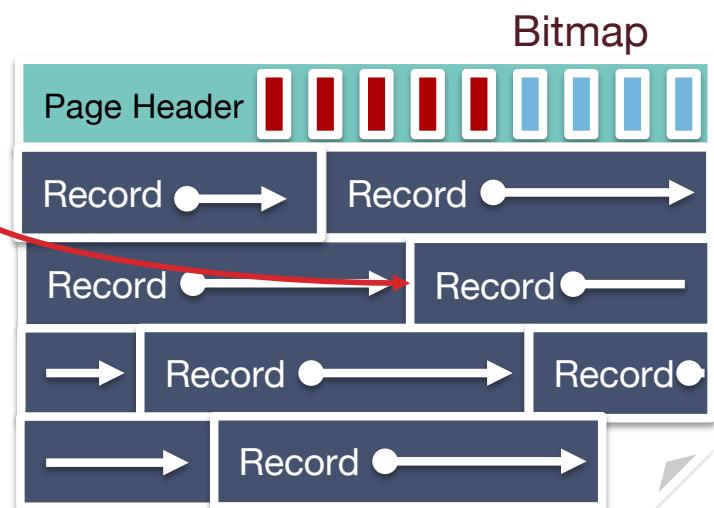


Fixed Length Records: Unpacked

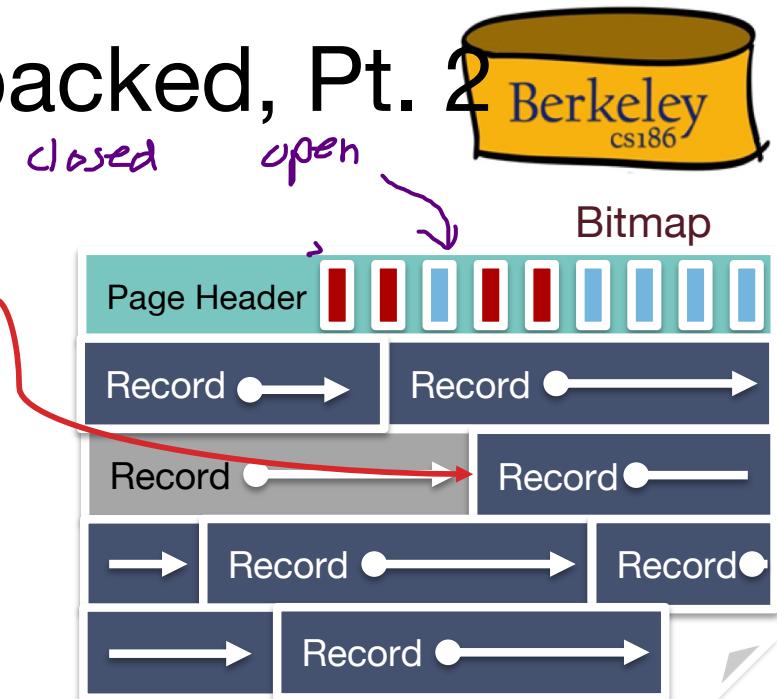


Record id:
(Page 2, Record 4)

- Bitmap denotes “slots” with records
 - Record id = (pageId, “location in page”)?
 - (pageId, slotId)
 - **Insert:** find first empty slot in bitmap
 - **Delete:** ?



Fixed Length Records: Unpacked, Pt. 2

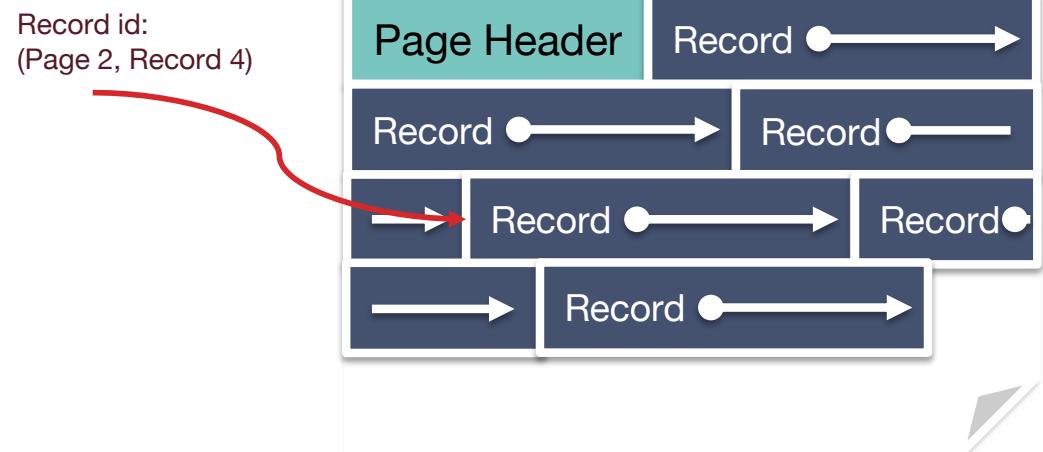


- Bitmap denotes “slots” with records
- Record id = (pageld, “location in page”)?
 - (pageld, slotId)
- **Insert:** find first empty slot in bitmap
- **Delete:** clear bit
 - No reorganization needed!
 - Small cost of a bitmap, which can be very compact

Variable Length Records



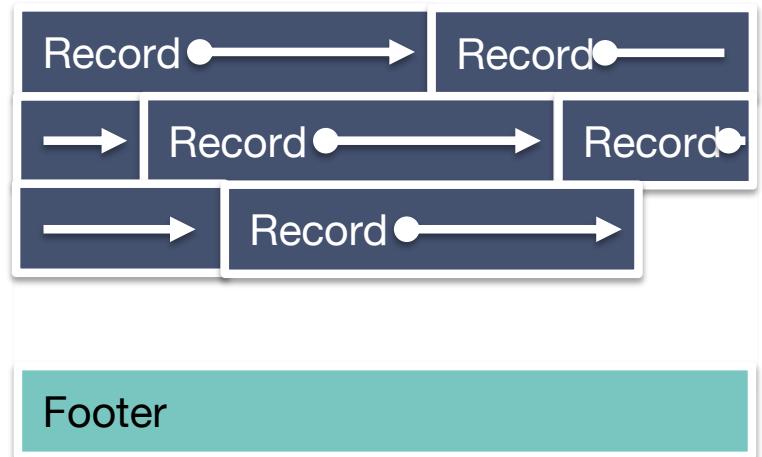
- We've already seen that packed isn't the best idea, so let's consider the unpacked case
- How do we know where each record begins (mapping recordid to location)?
- What happens when we add and delete records?



First: Relocate metadata to footer

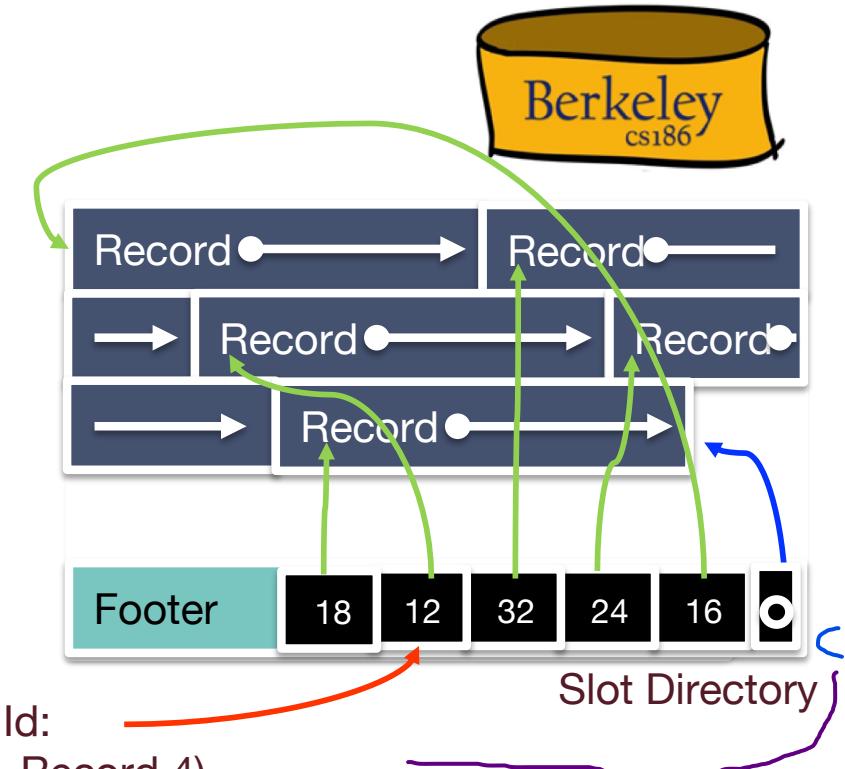


- We'll see why this is handy shortly...



Slotted Page

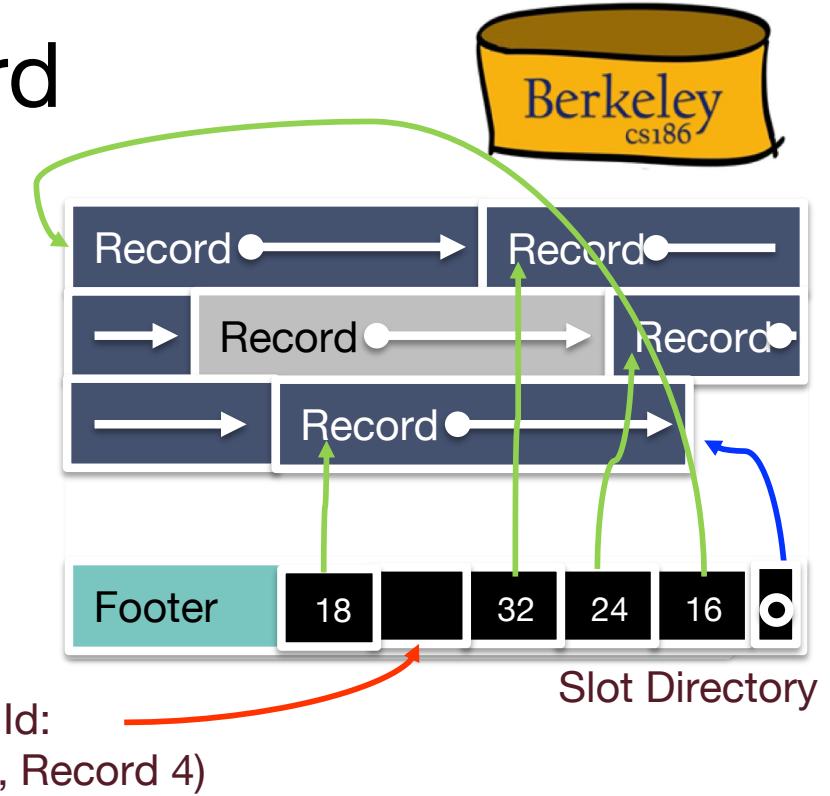
- Introduce slot directory in footer
 - Pointer to free space
 - Length + Pointer to beginning of record
 - reverse order
- Record ID = location in slot table
 - from right
- Delete?
 - e.g., 4th record on the page



free space pointer
(to next free space)

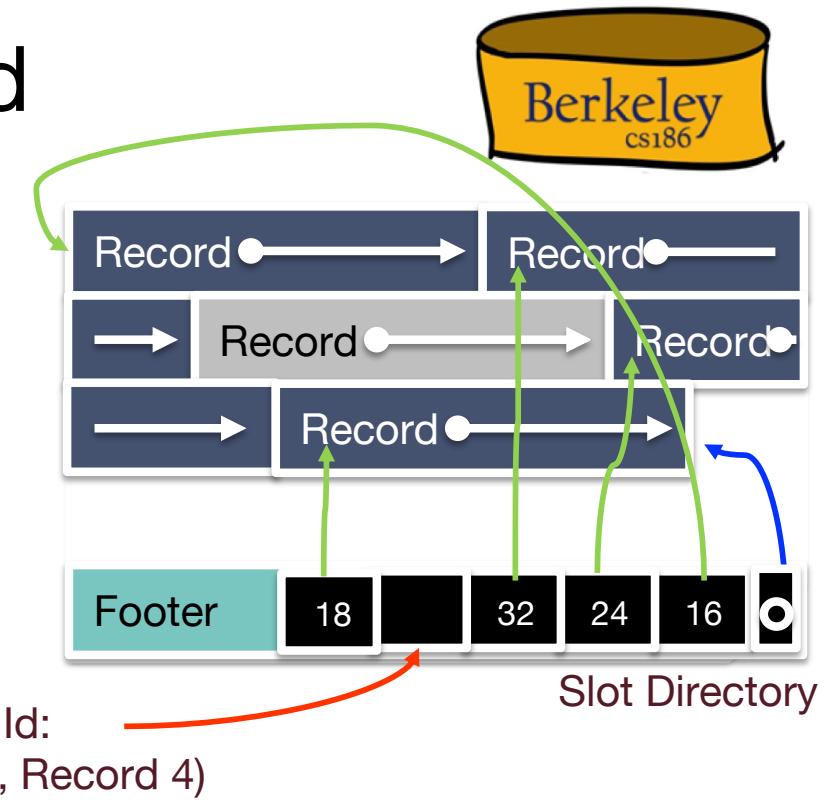
Slotted Page: Delete Record

- Delete record (Page 2, Record 4):
 - Set 4th slot directory pointer to null
 - Doesn't affect pointers to other records (no internal reorg, and no updating of external pointers)



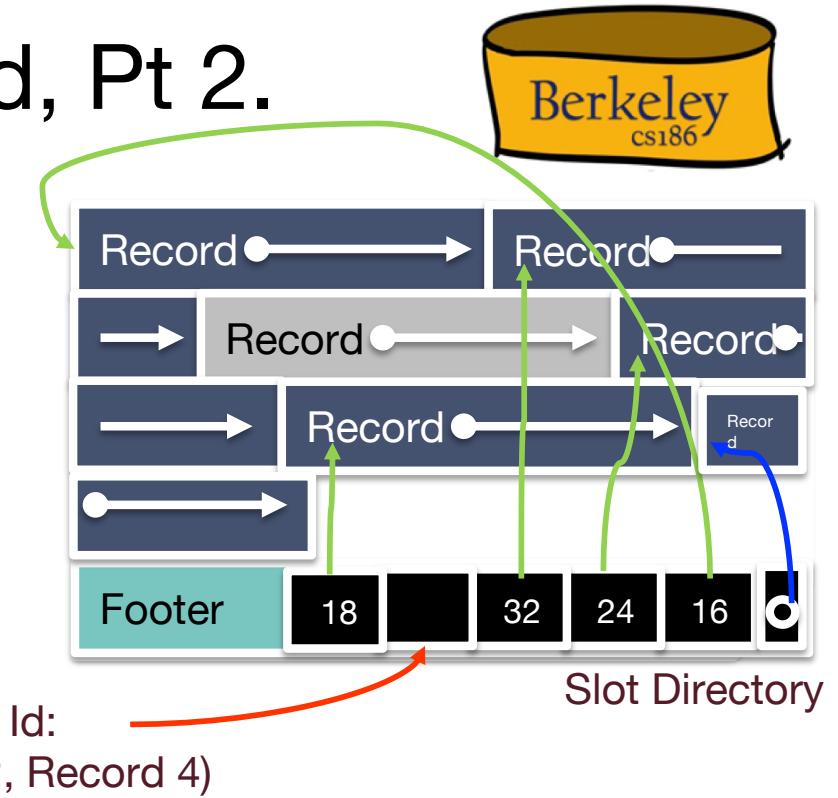
Slotted Page: Insert Record

- Insert:

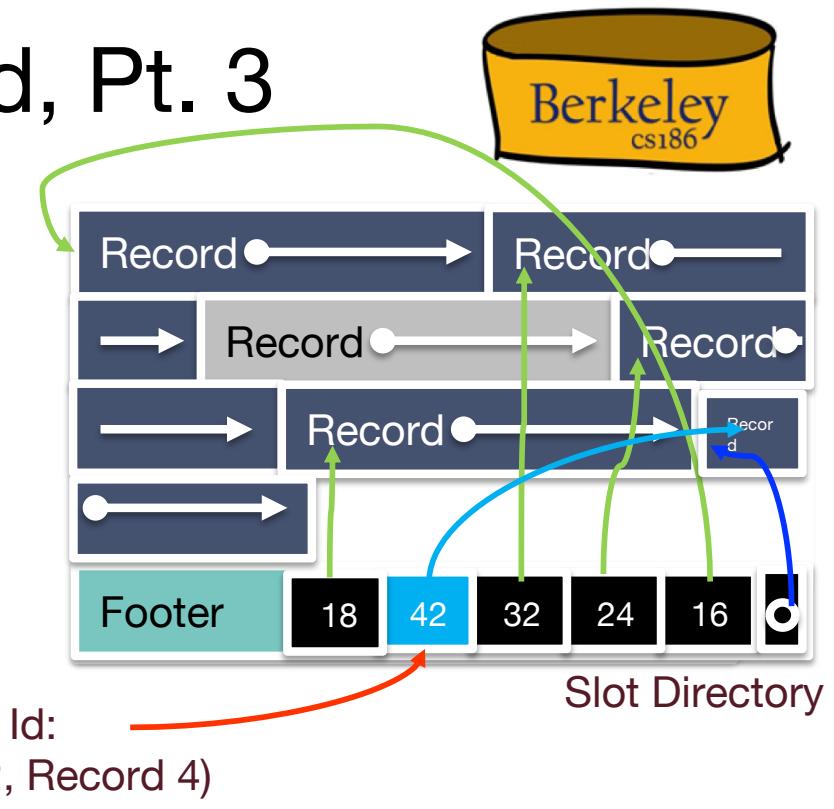


Slotted Page: Insert Record, Pt 2.

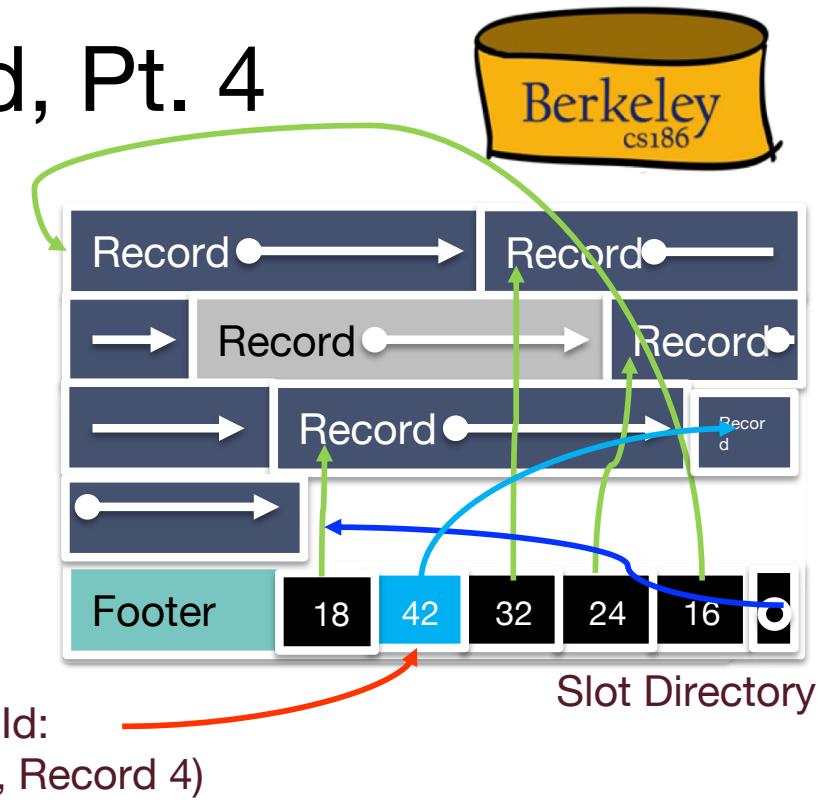
- Insert:
 - Place record in free space on page



Slotted Page: Insert Record, Pt. 3

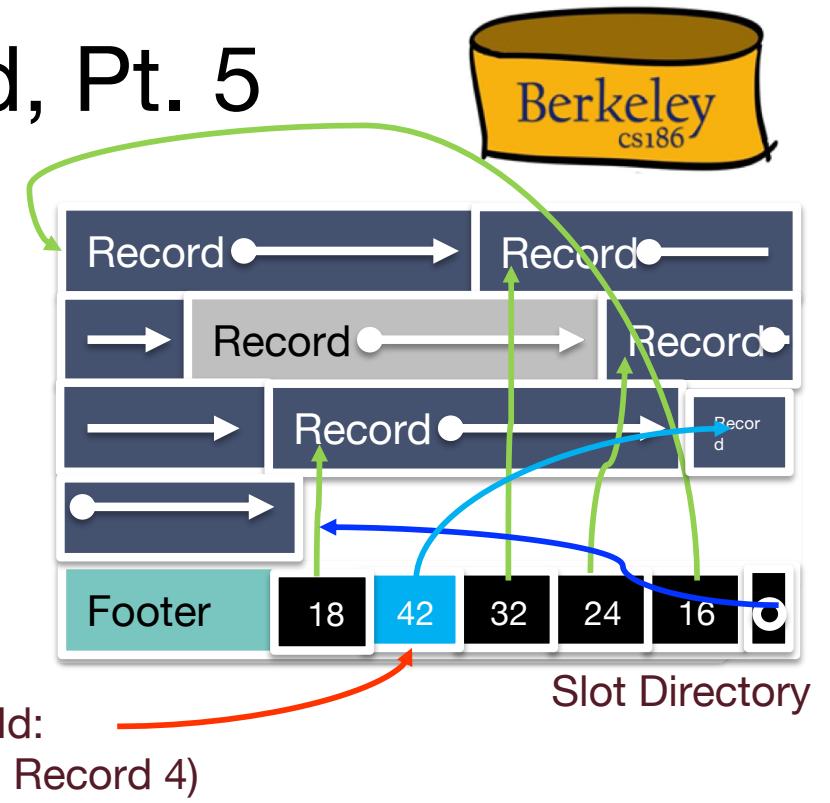


Slotted Page: Insert Record, Pt. 4



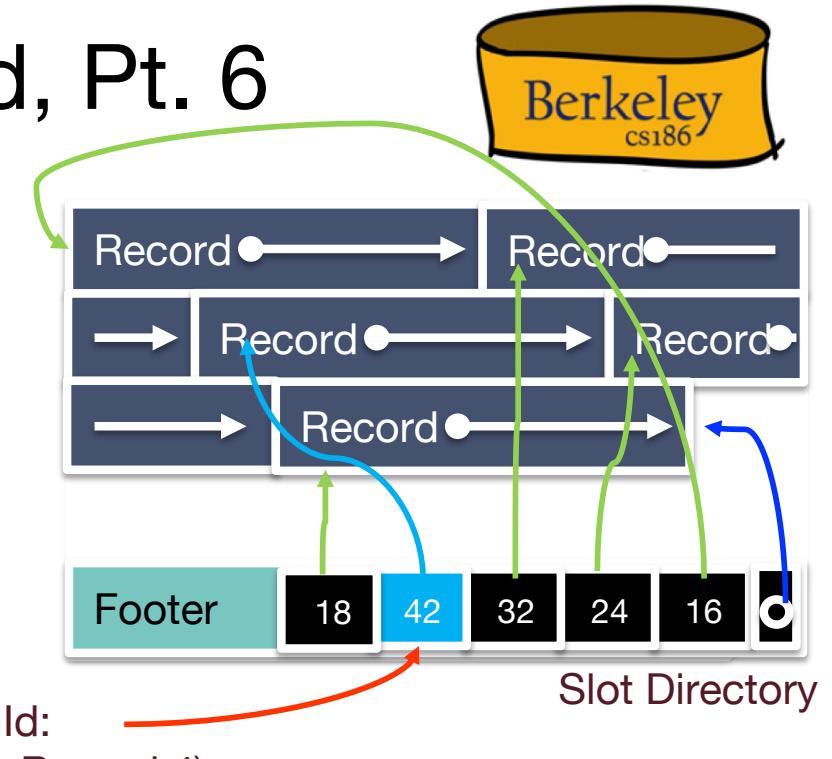
- Insert:
 - Place record in free space on page
 - Create pointer/length pair in next open slot in slot directory
 - Update the free space pointer

Slotted Page: Insert Record, Pt. 5



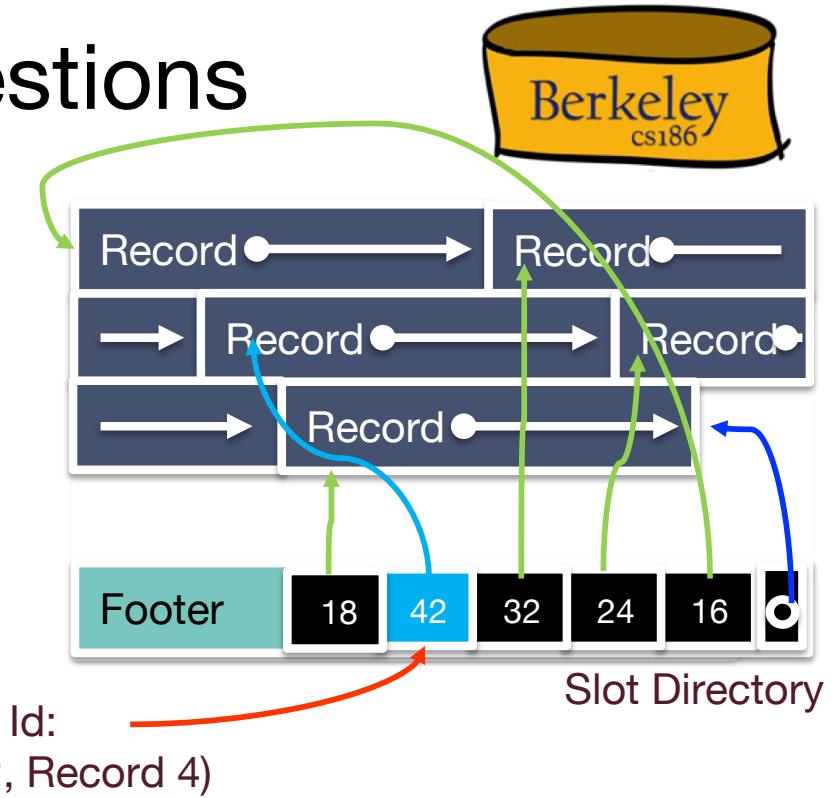
Slotted Page: Insert Record, Pt. 6

- Insert:
 - Place record in free space on page
 - Create pointer/length pair in next open slot in slot directory
 - Update the free space pointer
 - Fragmentation?
 - Reorganize data on page!
*if it can fit in
fragmented chunk*



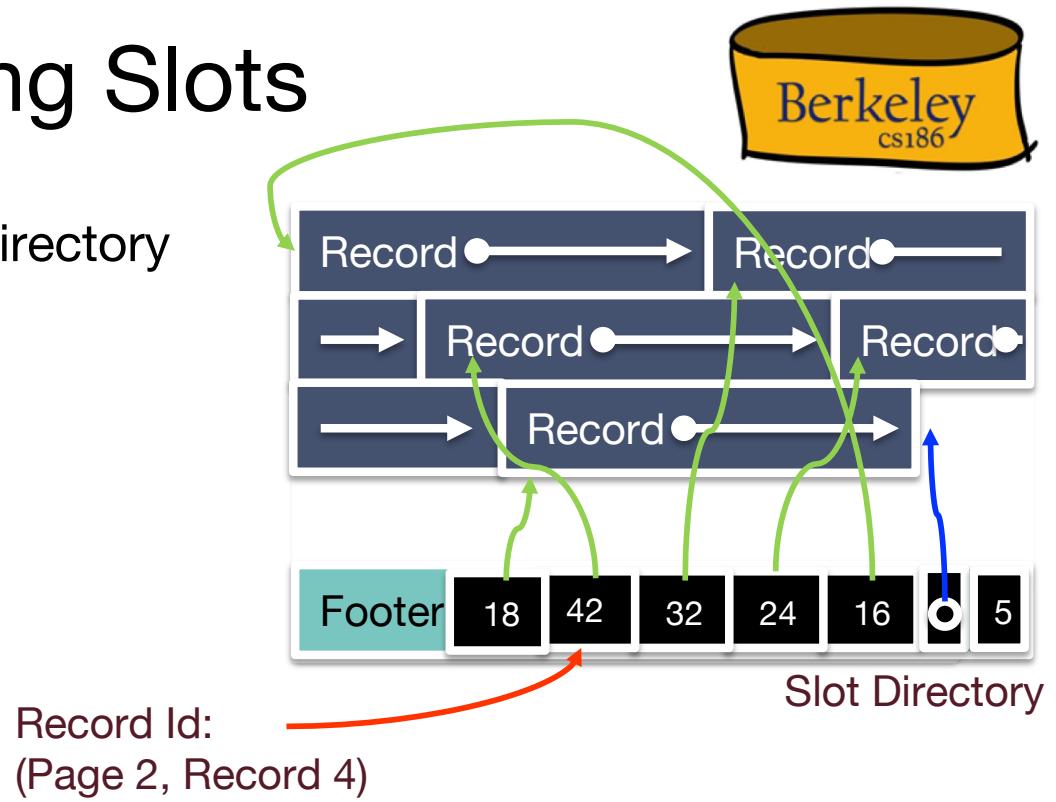
Slotted Page: Leading Questions

- Reorganize data on page
 - Is this safe?
 - Yes this is safe because records ids don't change. Record ids refer to slots
- When should I reorganize?
 - We could re-organize on delete
 - Or wait until fragmentation blocks record addition and then reorganize.
 - Often pays to be a little sloppy if page never gets more records.
- What if we need more slots?
 - Let's see...



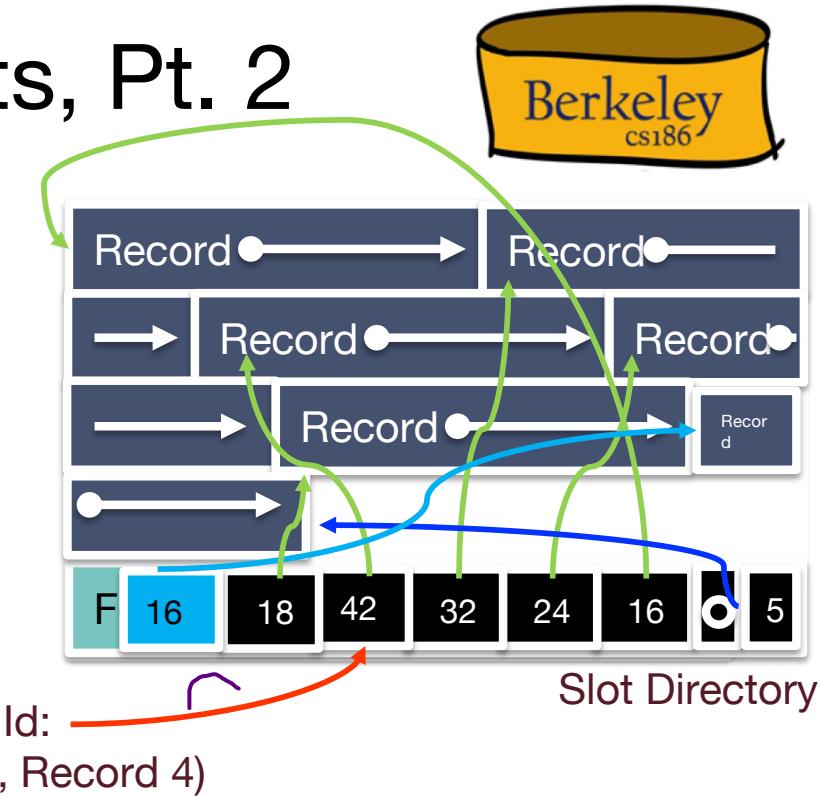
Slotted Page: Growing Slots

- Tracking number of slots in slot directory
 - Empty or full



Slotted Page: Growing Slots, Pt. 2

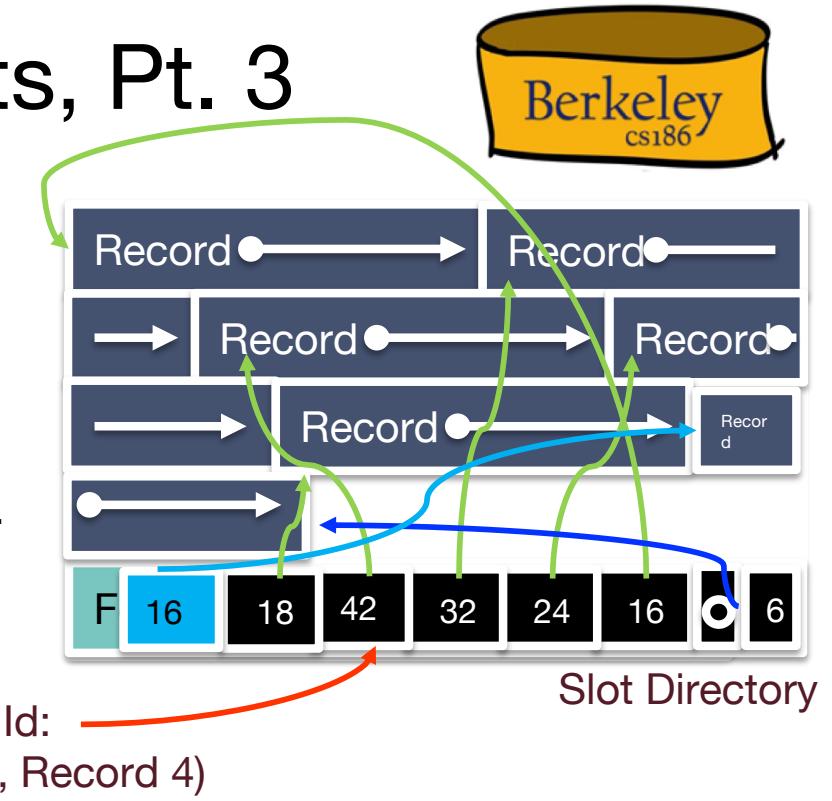
- Tracking number of slots in slot directory
 - Empty or full
- If full slots = number of slots, then extend slot directory
- To extend slot directory
 - Slots grow from end of page inward
 - Records grow from beginning of page inward.
 - Easy!



Do this so we can have a set size defined for the page and stops growing when we hit footer

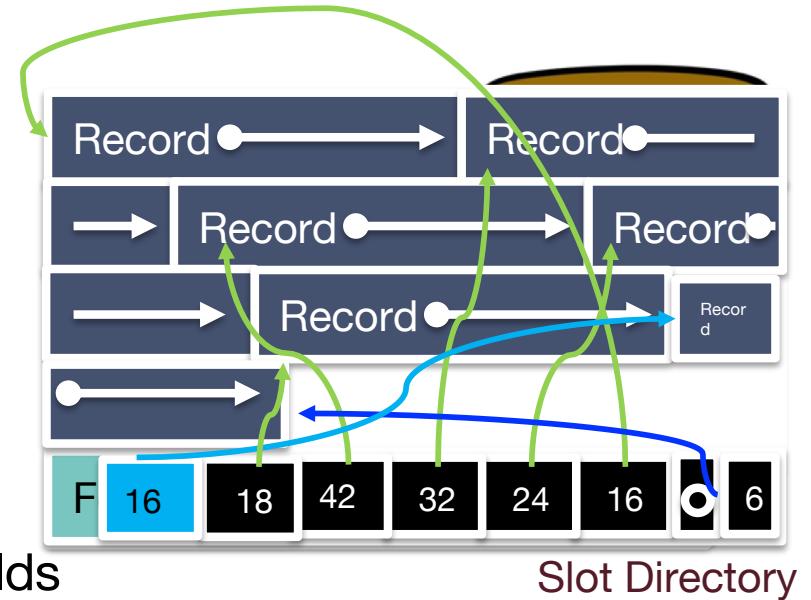
Slotted Page: Growing Slots, Pt. 3

- Tracking number of slots in slot directory
 - Empty or full
- Extend slot directory
 - Slots grow from end of page inward
 - Records grow from beginning of page inward.
 - Easy!
- And update count



Slotted Page: Summary

- Typically use Slotted Page
 - Good for variable and fixed length records
- Not bad for fixed length records too.
 - Why?
 - Fixed length records also have NULL fields
 - NULL values can be “squashed” and indicated using a flag, avoiding full attribute length storage
 - But, if we have only non-NULL fields, can be worth the optimization of fixed-length format





RECORD LAYOUT

Record Formats



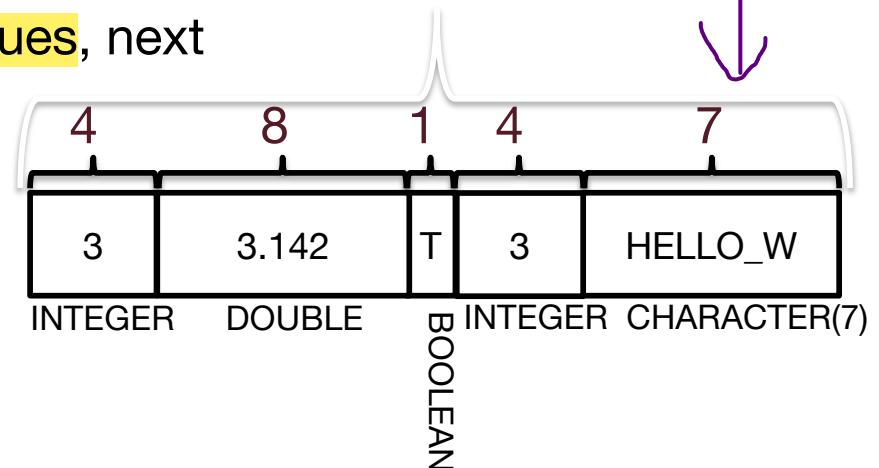
- Each record in a table/relation has a fixed combo of types
- Relational databases use same page format for data on disk or in memory
 - Save cost of conversion (known as serialization/deserialization)
- Assume System Catalog stores the Schema
 - No need to store type information with records (save space!)
 - Catalog is just another table
- Goals:
 - Fast access to fields (why?)
 - Records should be compact
- Easy Case: Fixed Length Fields
- Interesting Case: Variable Length Fields

Record Formats: Fixed Length



- Finding i'th field?
 - done via arithmetic (fast) $= 1 + 8 + 1 + 4$
- Making it more compact?
 - If all fields are not-null, no good way of compacting
 - Else apply variable length techniques, next

back-to-back empty slots?



Record Formats: Variable Length

What happens if fields are variable length?



Record				
Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Could store with padding? (Essentially fixed length)
Wasted Space

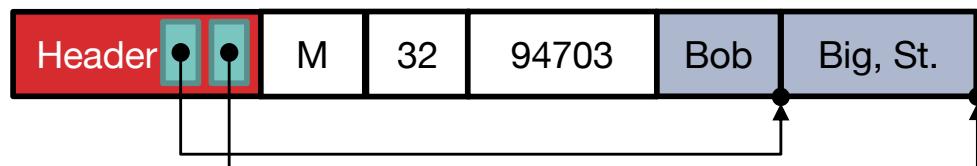
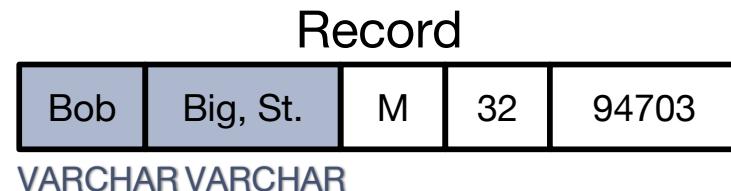
Bob	Big, St.	M	32	94703
CHAR(20)	CHAR(18)	CHAR	INT	INT

- Must account for largest possible string (wasteful) or rearrange as soon as a larger string comes (inefficient).
- Could store with delimiters (e.g., commas)?
 - Hard to find fields and strings with commas

Record Formats: Variable Length, Pt. 7



- What happens if fields are variable length?



- Solution: introduce a record header
- Easy access to fields, and almost as compact as can be (modulo header)
 - Same approach can be used to squash fixed length null fields w. many nulls

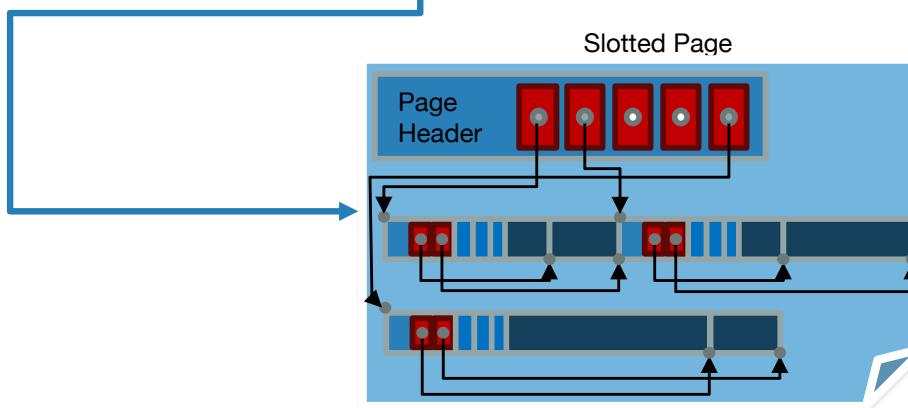
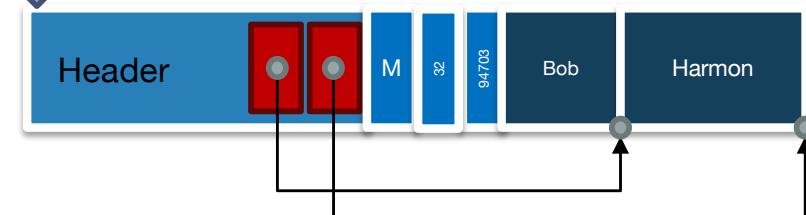
Overview: Representations



Record

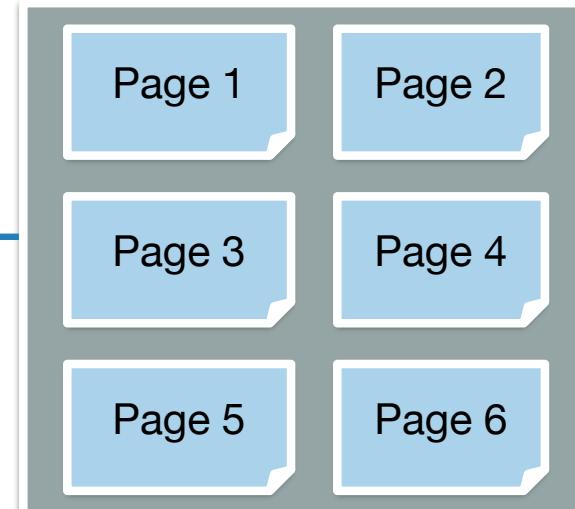
Bob	Harmon	M	32	400
Varchar	Varchar	Char	Int	Int

Byte Representation of Record



SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	\$400
443	Grouch	Oscar	32	\$300
244	Oz	Bert	55	\$140
134	Sanders	Ernie	55	\$400

File



Files: Summary



- DBMS “File” contains pages, and records within pages
 - Heap files: unordered records organized with directories
- Page layouts
 - Fixed-length packed and unpacked
 - Variable length records in slotted pages, with intra-page reorg
- Variable length record format
 - Direct access to i'th field and null values