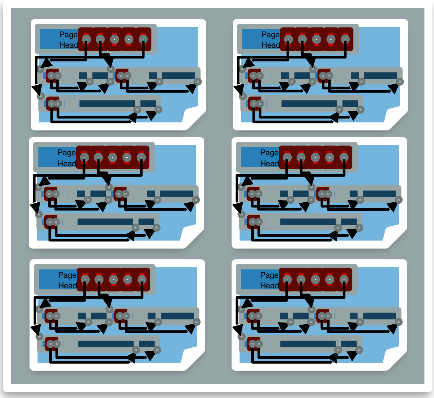
Disk Representations:
Files, Pages, Records

Overview: Files of Pages of Records

- Overall:
  - Each table is stored in one or more OS files
  - Each file contains many pages
  - Each page contains 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
- **<u>DB FILE</u>:** 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

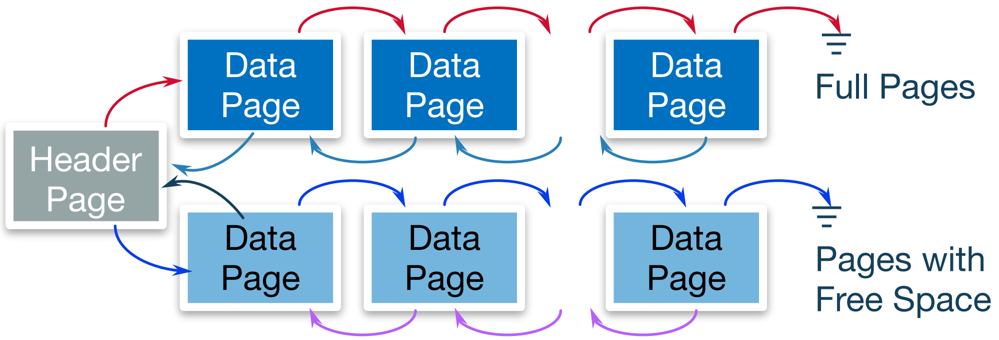
- Unordered Heap Files
  - Records placed arbitrarily across pages
- Clustered Heap Files
  - Records and pages are grouped in some meaningful way
- Sorted Files
  - Pages and records are in strict sorted order
- Index Files
  - B+ Trees, Linear Hashing, …
  - May contain records or point to records 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

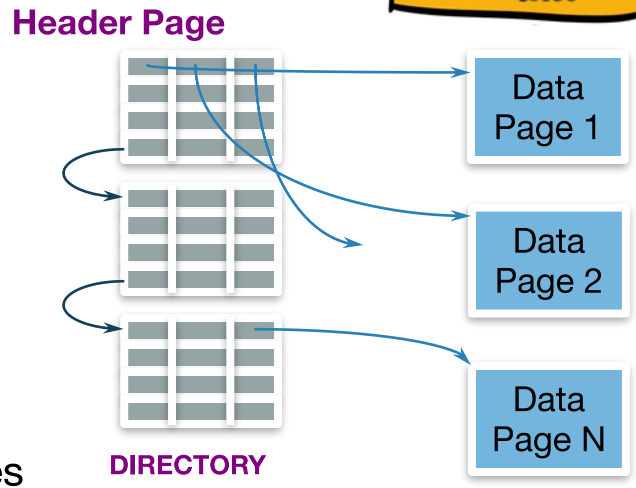
Take 1: Heap File as List

- 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?
  - How do I find a page with enough space for a 20 byte record
  - A: Need to access many pages (w/ free space) to check



Take 2: Use a Page Directory

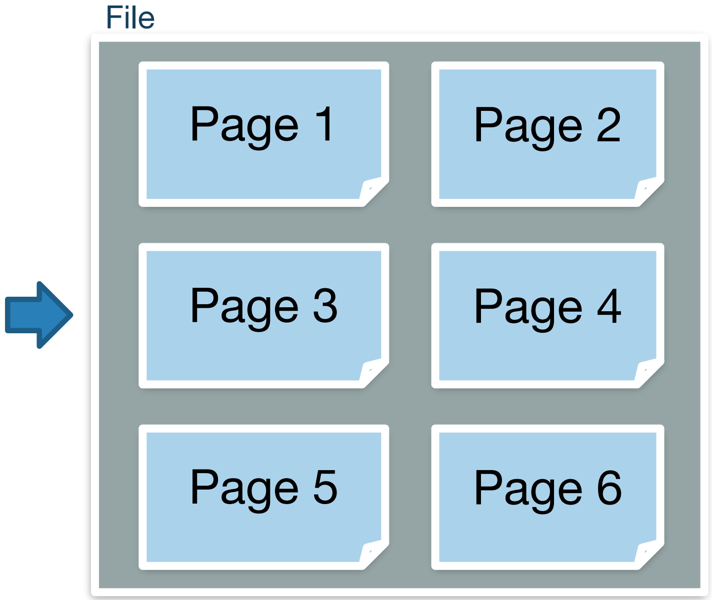
- Directory, with multiple Header Pages, each encoding:
  - A pointer to page
  - #free bytes on the page
- There can be multiple such header pages
- Header pages accessed often → likely in cache
- Finding a page to fit a record required far fewer page loads than linked list. Why?
  - One header page load reveals free space of many pages
- You can imagine optimizing 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

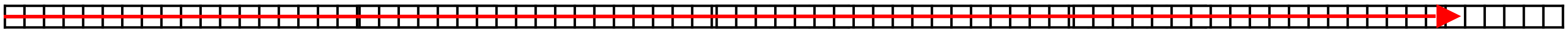
SSNz	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$400
443	Grouch	Oscar	32	$300
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400



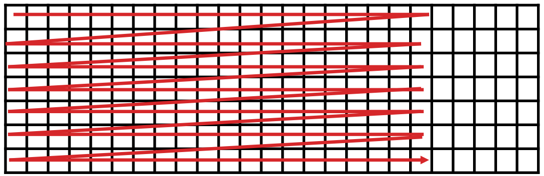
PAGE LAYOUT

A Note On Imagery

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

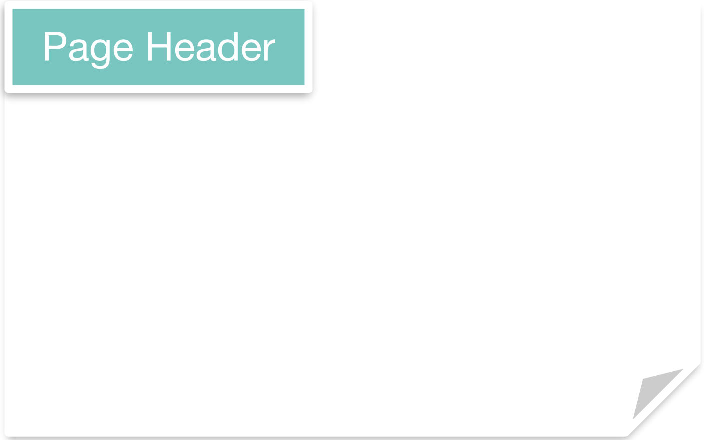


- This doesn’t fit nicely on screen
  - So we will “wrap around” the linear order into a rectangle



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)



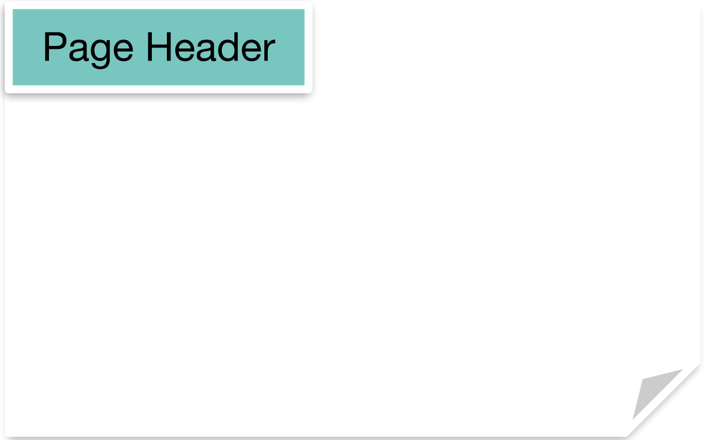
Things to Address

Some options:

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

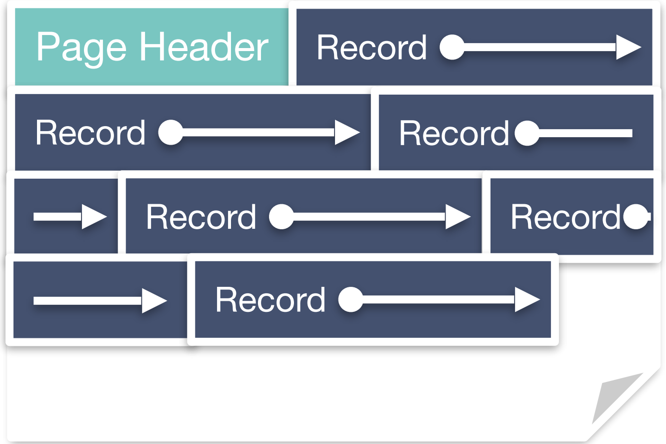
Some questions:

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



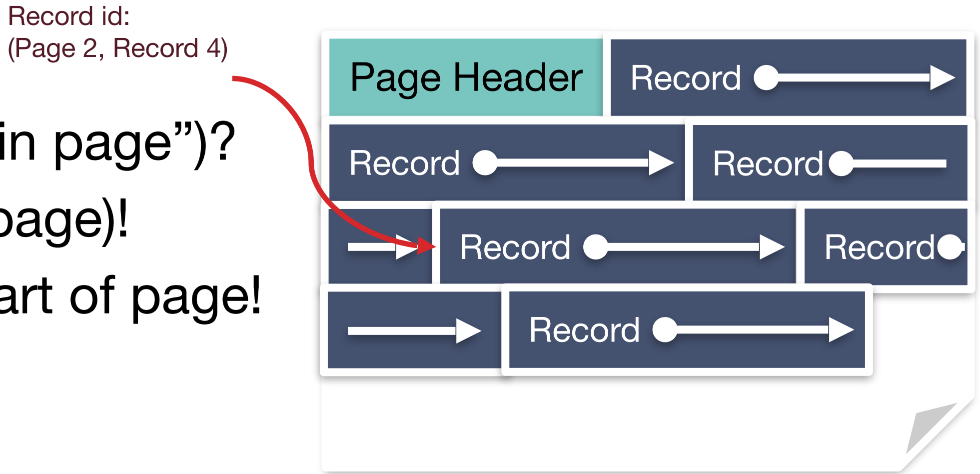
Fixed Length Records, Packed

- Pack records densely
- 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?



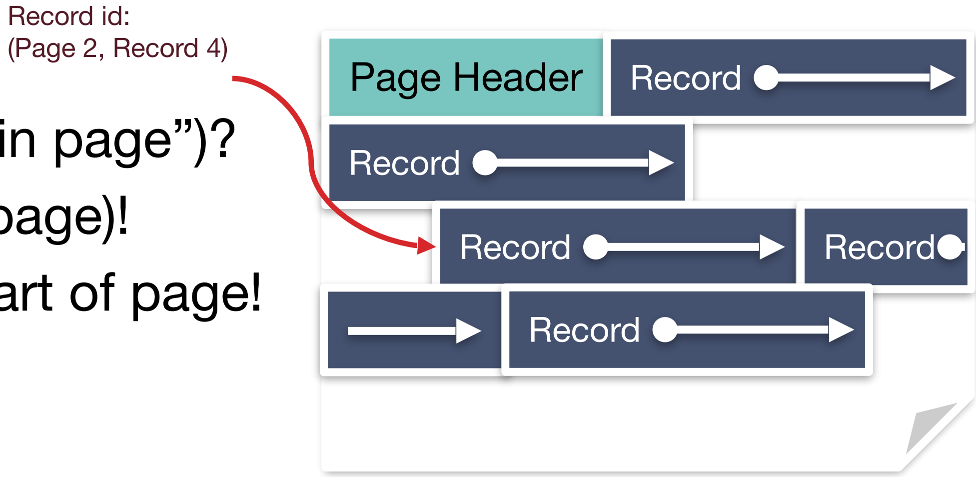
Fixed Length Records, Packed, Pt 2.

- 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)



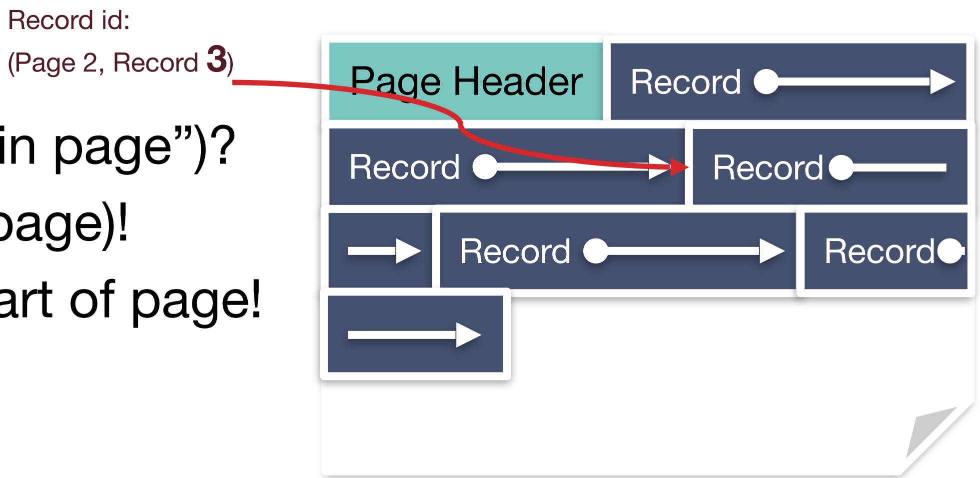
Fixed Length Records: Packed, Pt 3.

- 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)
  - Now free space… need to reorg

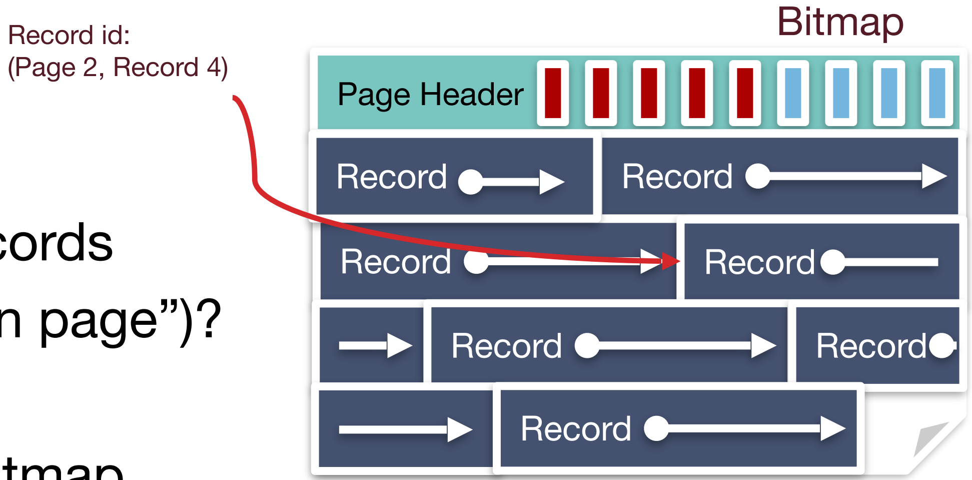


Fixed Length Records: Packed, Pt. 5

- 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?
  - 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.

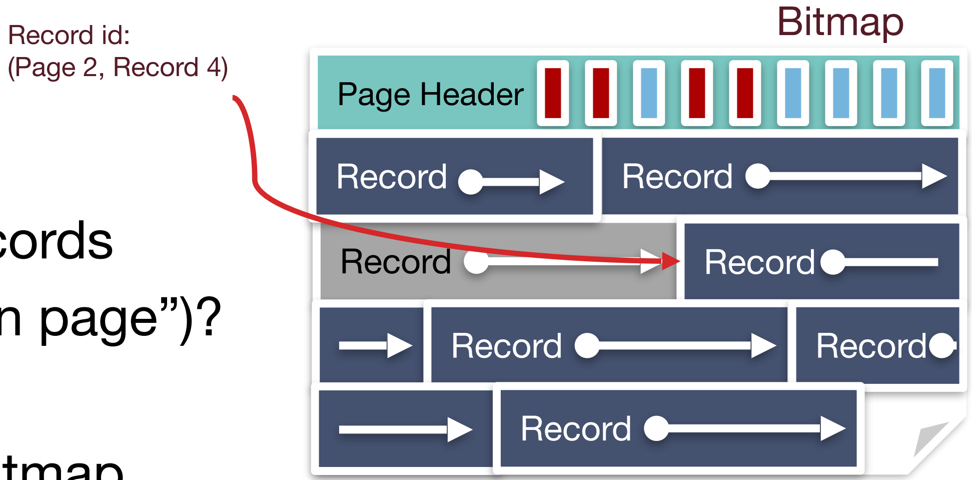


Fixed Length Records: Unpacked



- 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

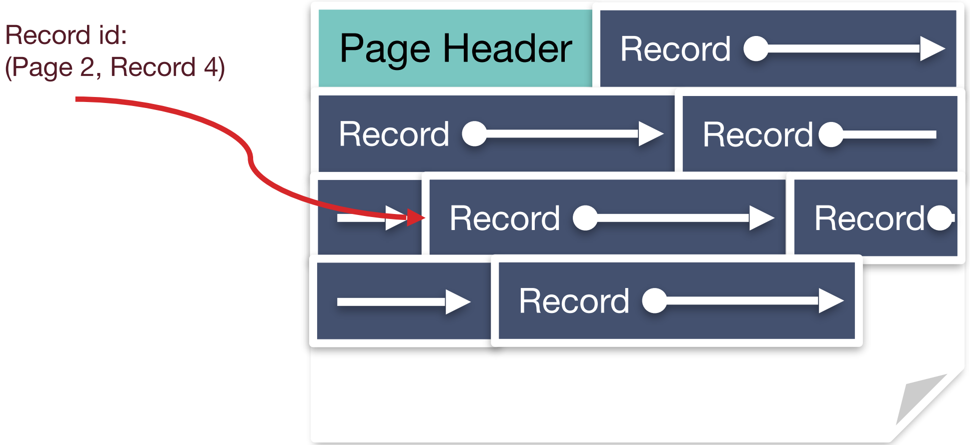


- Bitmap denotes "slots" with records
- Record id = (pageId, "location in page")?
  - (pageId, slotId)
- **Insert**: find first empty slot in bitmap
- **Delete**: clear bit
  - No reorganization needed!
  - Small cost of a bitmap, which can be very compact

Variable Length Records

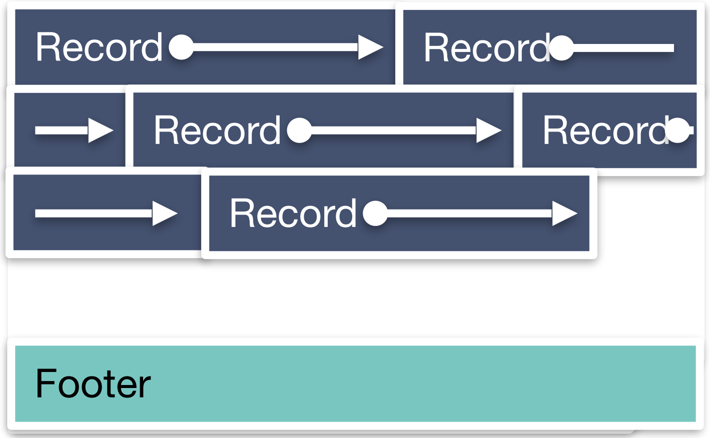
- We've already seen that packed isn't the best idea, so let's consider the unpacked case

- How do we know where each record begins (mapping recordid to location)?
- What happens when we add and delete records?



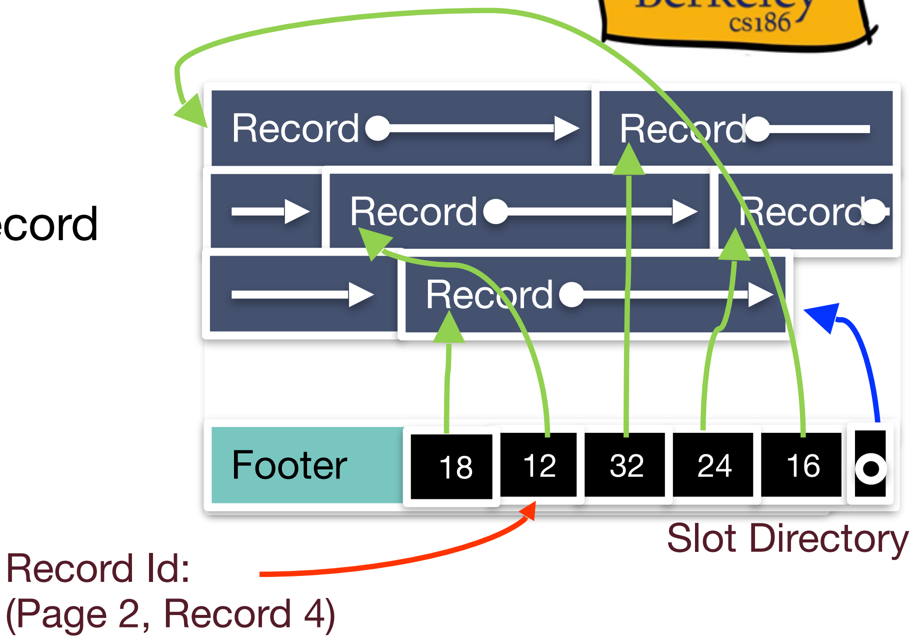
First: Relocate metadata to footer

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



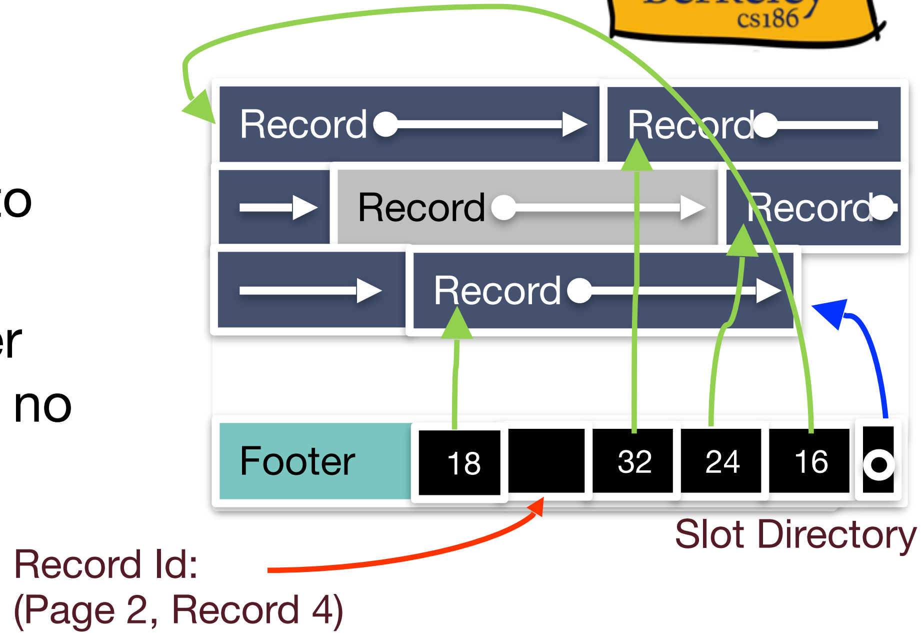
Slotted Page

- 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



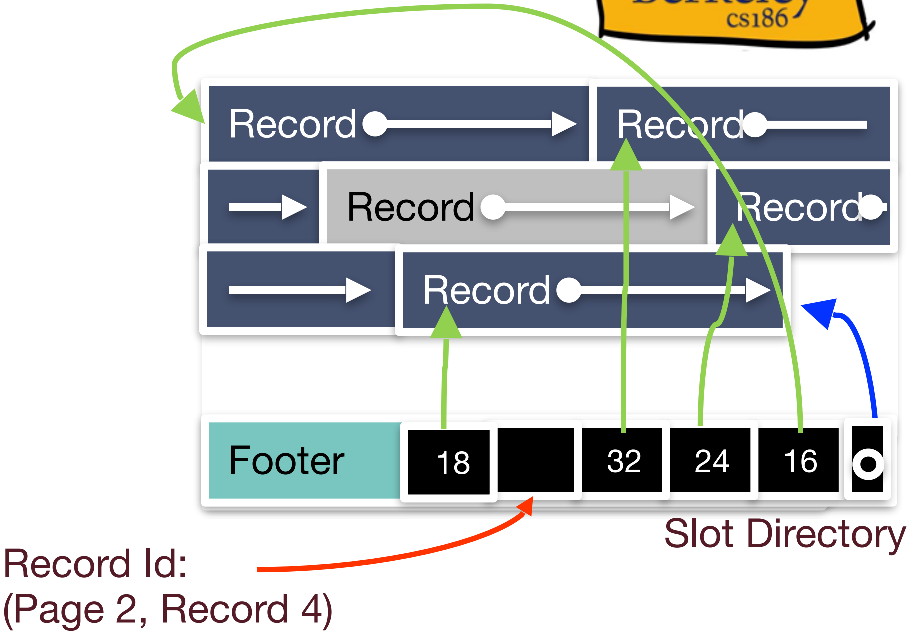
Slotted Page: Delete Record

- 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)



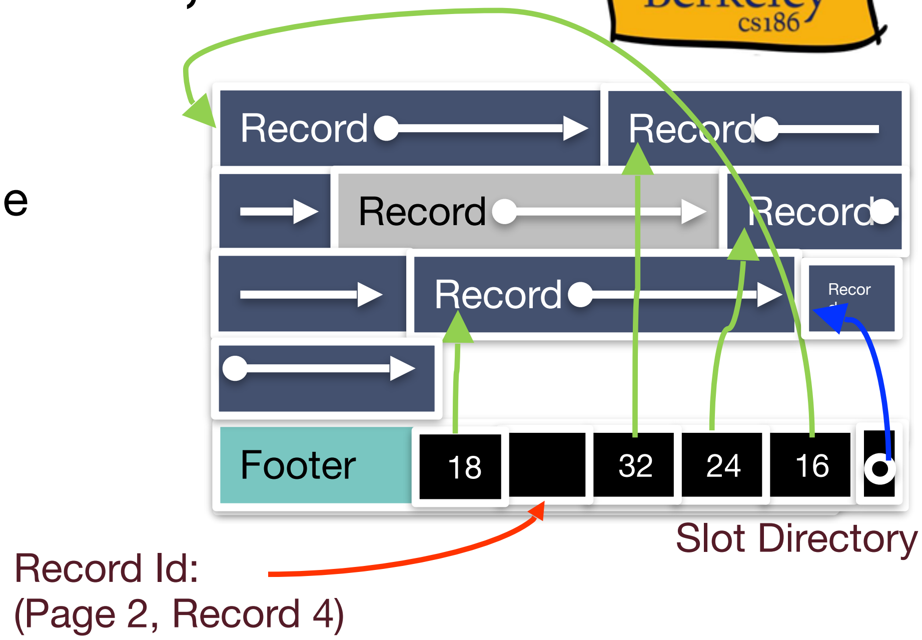
Slotted Page: Insert Record

- Insert:



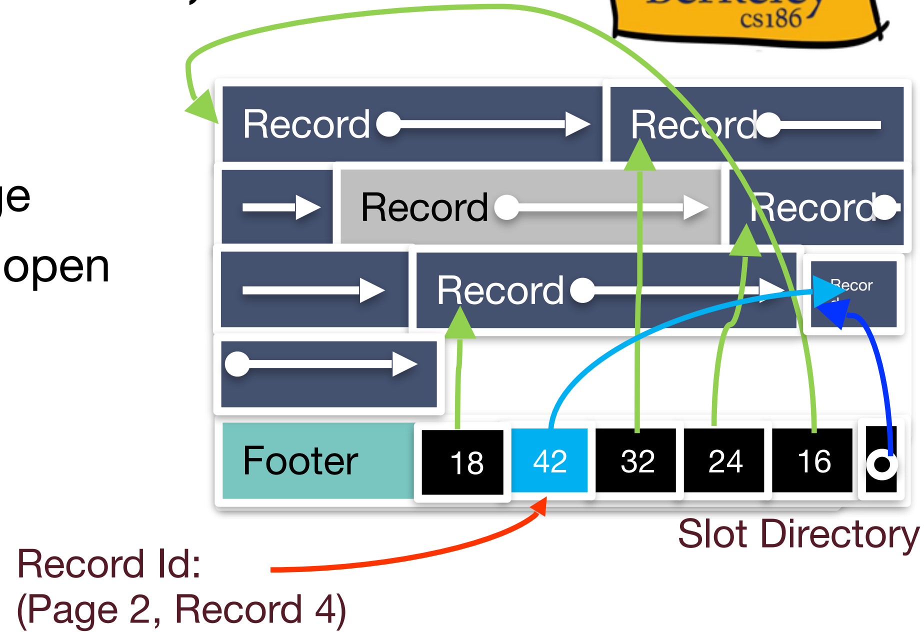
Slotted Page: Insert Record, Pt 2.

- Insert:
  - Place record in free space on page



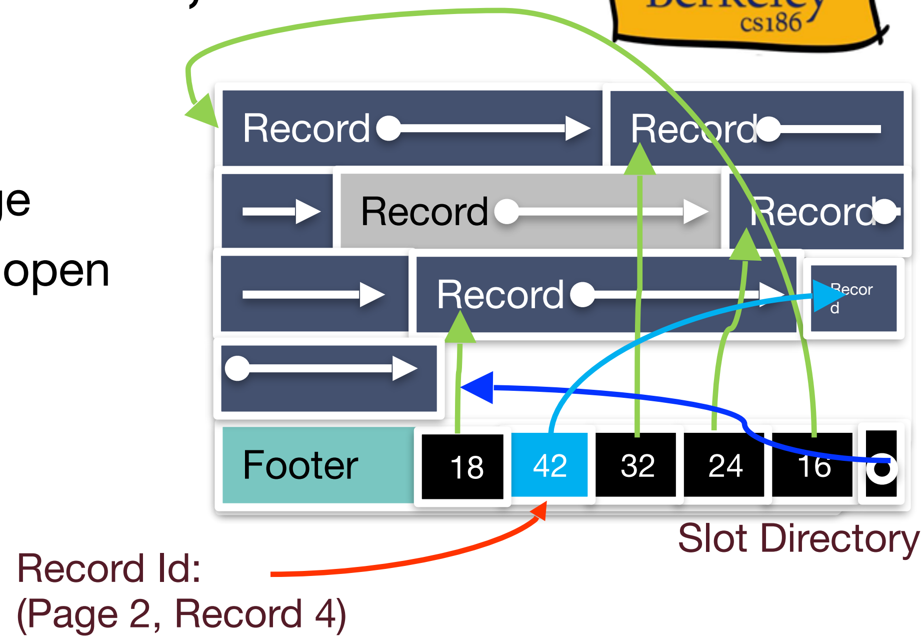
Slotted Page: Insert Record, Pt. 3

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



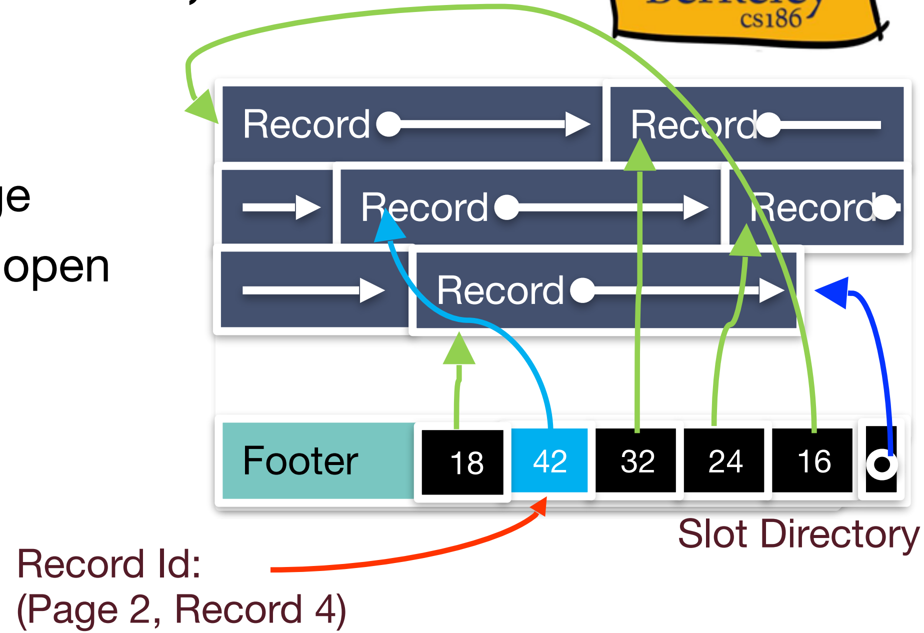
Slotted Page: Insert Record, Pt. 4

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



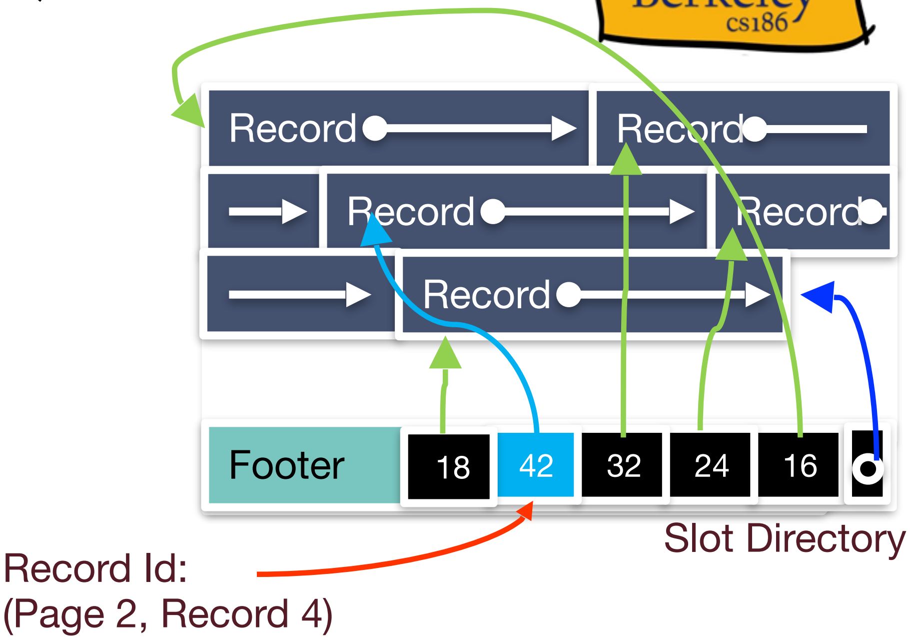
Slotted Page: Insert Record, Pt. 6

- 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!



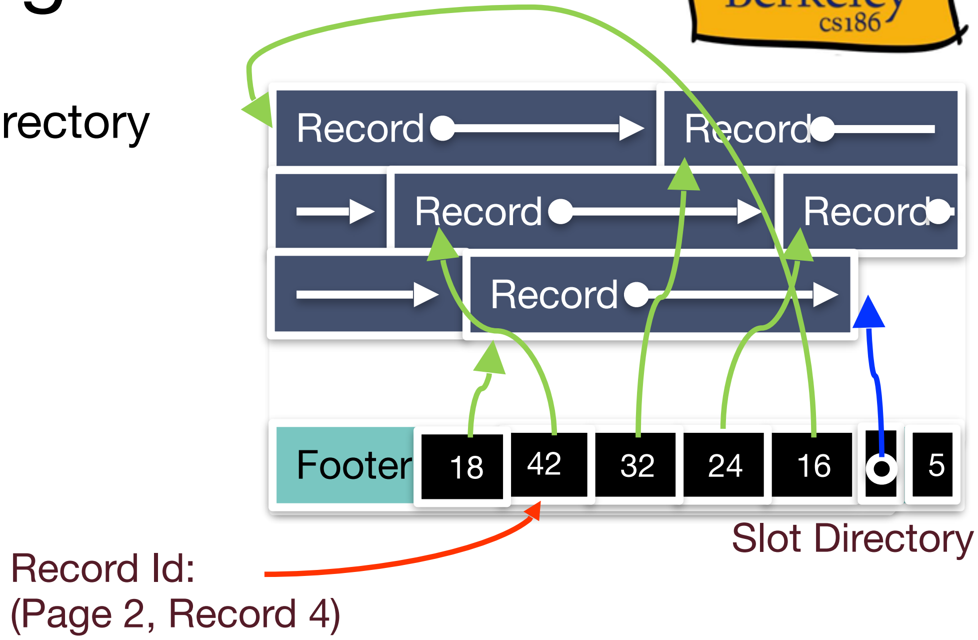
Slotted Page: Leading Questions

- 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

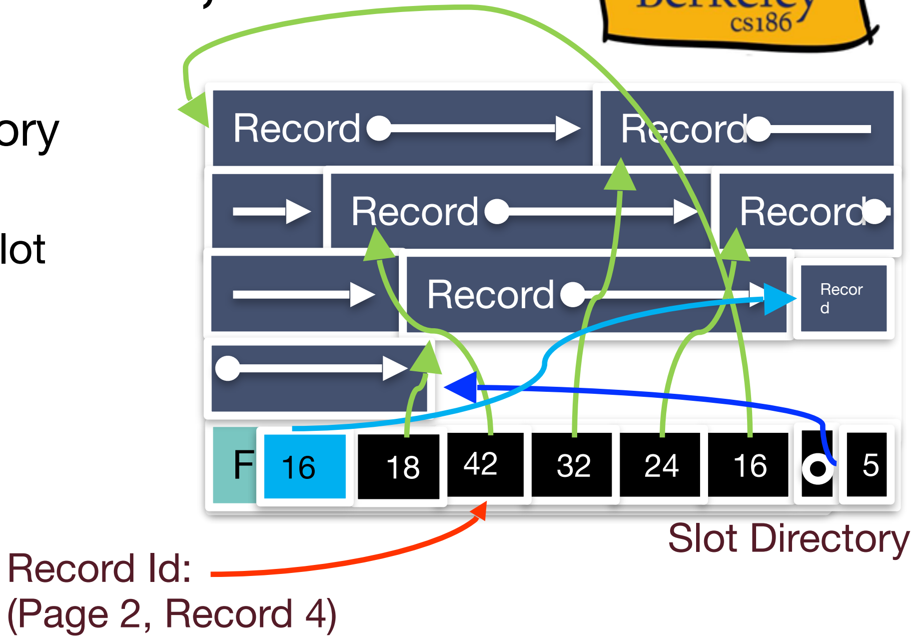
- Tracking number of slots in slot directory
  - Empty or full



Slotted Page: Growing Slots, Pt. 2

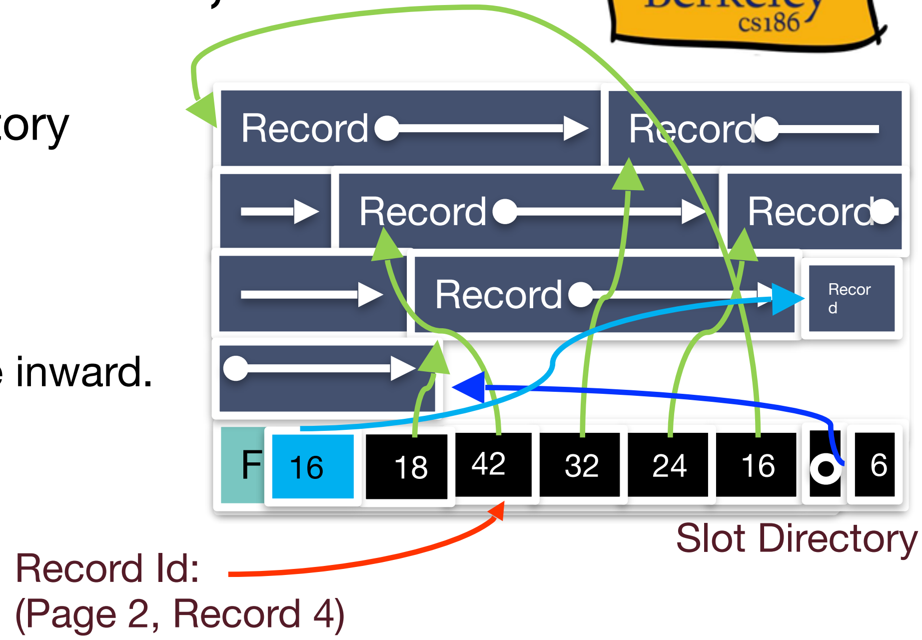
- Tracking number of slots in slot directory
  - Empty or full
- If full slots = number of slots, then extend slot directory

- To extend slot directory
  - Slots grow from end of page inward
  - Records grow from beginning of page inward.
  - Easy!



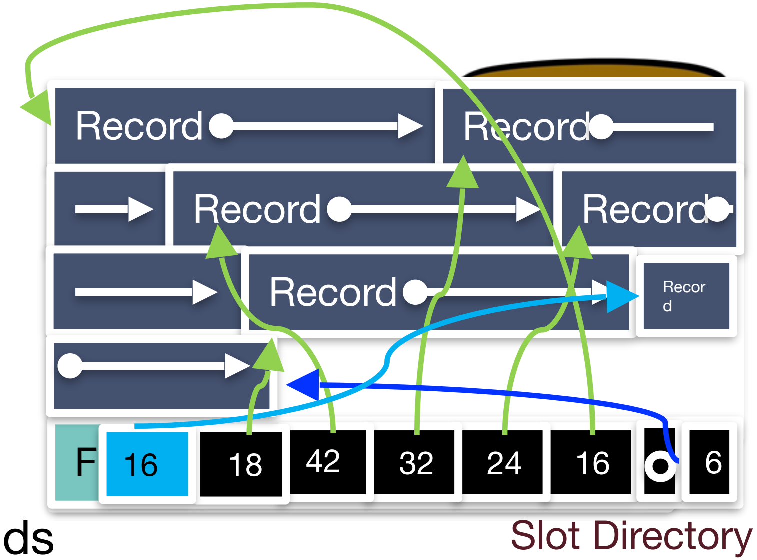
Slotted Page: Growing Slots, Pt. 3

- 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

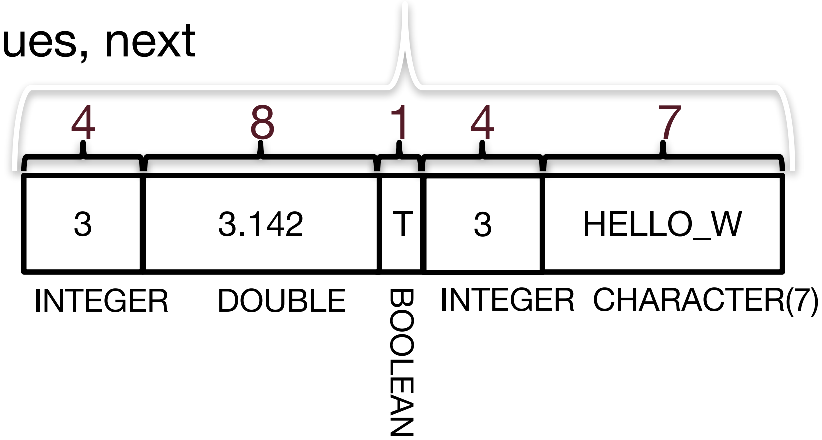
Record Formats

- Each record in a table/relation has a fixed combo of types
- Relational databases also 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

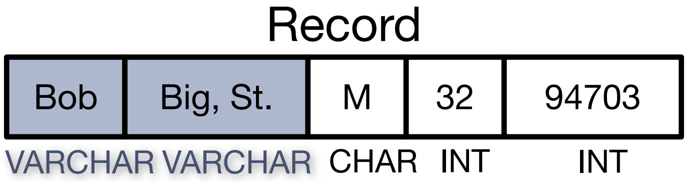
Record Formats: Fixed Length

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

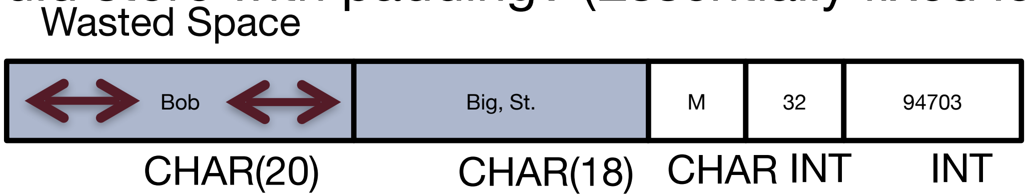


Record Formats: Variable Length

What happens if fields are variable length?



Could store with padding? (Essentially fixed length)



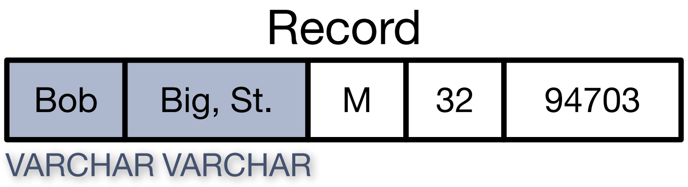
But have to account for largest possible string (wasteful) or rearrange as soon as a larger string comes (inefficient).

Could store with delimiters (e.g., commas)?

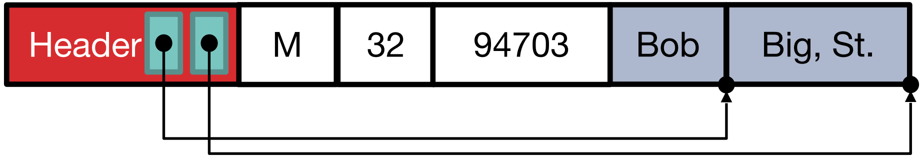
But makes it hard to find fields and also ensure that commas are not part of the string

Record Formats: Variable Length, Pt. 7

- What happens if fields are variable length?

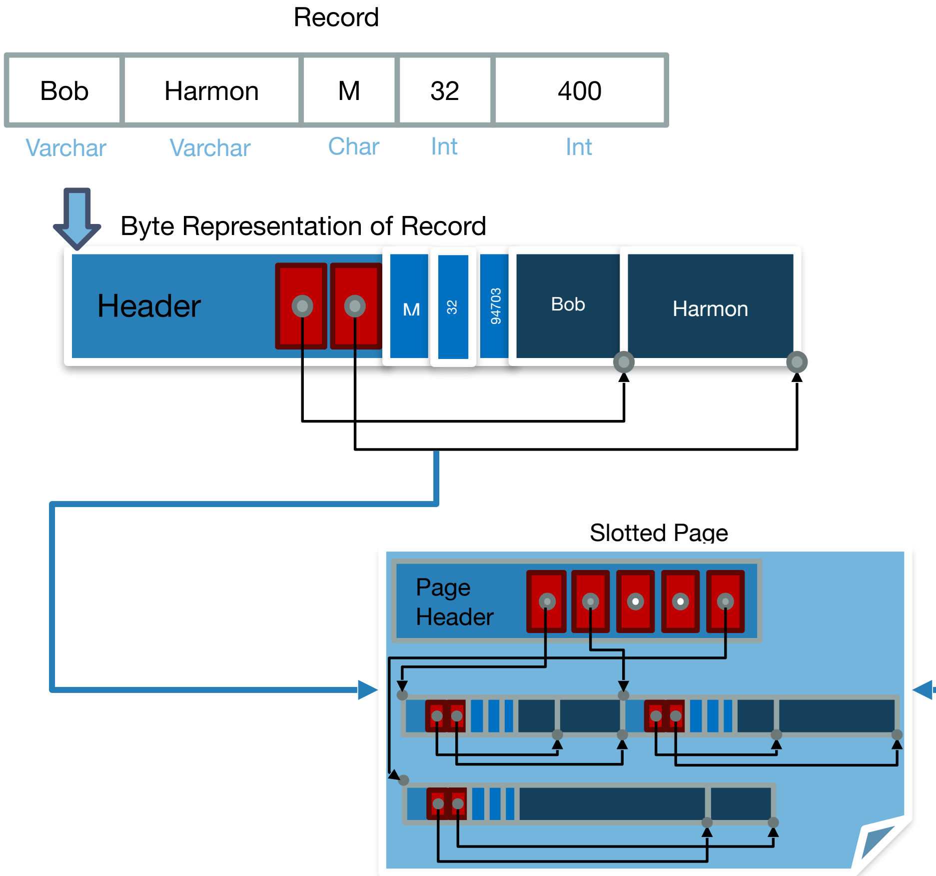


- Solution: introduce a record header

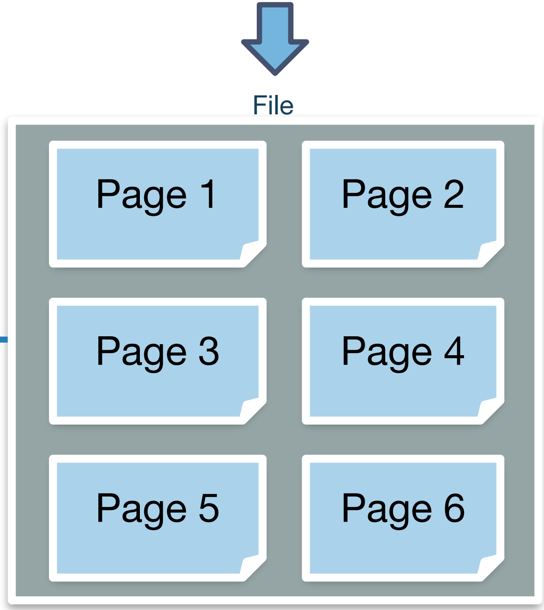


- Easy access to fields, and almost as compact as can be (modulo header)
  - Same approach can be used to squash fixed length null fields w. many nulls

Overview: Representations



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



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