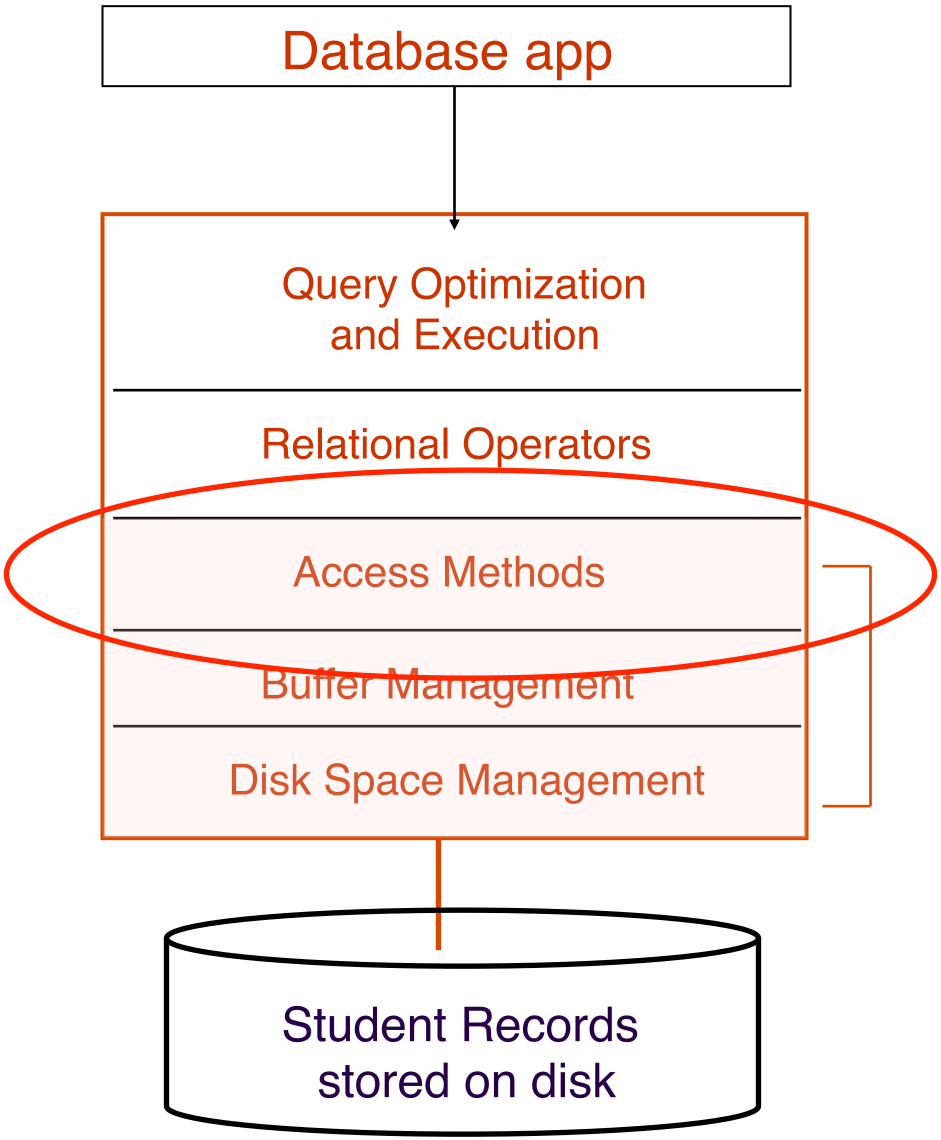
# CAS CS 460/660
# Introduction to Database Systems

# File Organization

Slides from UC Berkeley

# Context




These layers must consider concurrency control and recovery

# Files of Records

- Disk blocks are the interface for I/O, but…
- Higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a number of records. The File API must support:
  - **insert**/**delete**/**modify** record
  - **fetch** a particular record (specified by *record id*)
  - **scan** all records (possibly with some conditions on the records to be retrieved)
- Typically: file page size = disk block size = buffer frame size

# "MetaData" - System Catalogs

- How to impose structure on all those bytes??
- MetaData: "Data about Data"
- For each relation:
  - name, file location, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each view: view name and definition
- Plus statistics, authorization, buffer pool size, etc.

# Catalogs are Stored as Relations!

| attr_name | rel_name | type | position | length |
|---|---|---|---|---|
| attr_name | Attr_Cat | string | 1 | 50 |
| rel_name | Attr_Cat | string | 2 | 40 |
| type | Attr_Cat | string | 3 | 40 |
| position | Attr_Cat | integer | 4 | 4 |
| sid | Students | string | 1 | 10 |
| name | Students | string | 2 | 50 |
| login | Students | string | 3 | 40 |
| age | Students | integer | 4 | 4 |
| gpa | Students | real | 5 | 8 |
| fid | Faculty | string | 1 | 10 |
| fname | Faculty | string | 2 | 50 |
| sal | Faculty | real | 3 | 8 |

Attr_Cat(attr_name, rel_name, type, position, length)

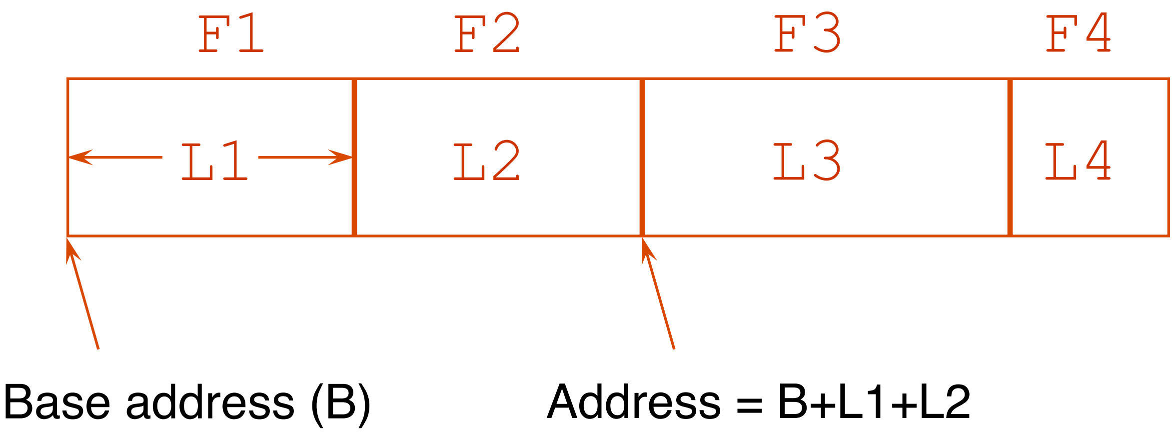
# It’s a bit more complicated…

Terminal — psql — 99×28

```
joeh=# \dt pg_attribute
No matching relations found.
joeh=# \d pg_attribute
   Table "pg_catalog.pg_attribute"
    Column     |   Type   | Modifiers
---------------+----------+-----------
 attrelid      | oid      | not null
 attname       | name     | not null
 atttypid      | oid      | not null
 attstattarget | integer  | not null
 attlen        | smallint | not null
 attnum        | smallint | not null
 attndims      | integer  | not null
 attcacheoff   | integer  | not null
 atttypmod     | integer  | not null
 attbyval      | boolean  | not null
 attstorage    | "char"   | not null
 attalign      | "char"   | not null
 attnotnull    | boolean  | not null
 atthasdef     | boolean  | not null
 attisdropped  | boolean  | not null
 attislocal    | boolean  | not null
 attinhcount   | integer  | not null
Indexes:
    "pg_attribute_relid_attnam_index" UNIQUE, btree (attrelid, attname)
    "pg_attribute_relid_attnum_index" UNIQUE, btree (attrelid, attnum)

joeh=# 
```

# Record Formats: Fixed Length

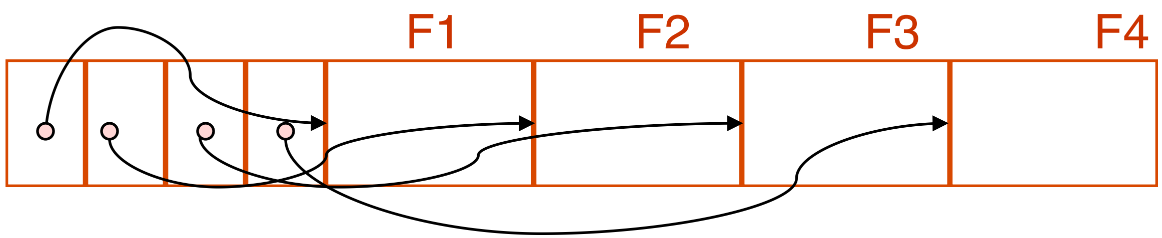



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field done via arithmetic.

# Record Formats: Variable Length

- Two alternative formats (# fields is fixed):


Fields Delimited by Special Symbols




Array of Field Offsets

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

# How to Identify a Record?

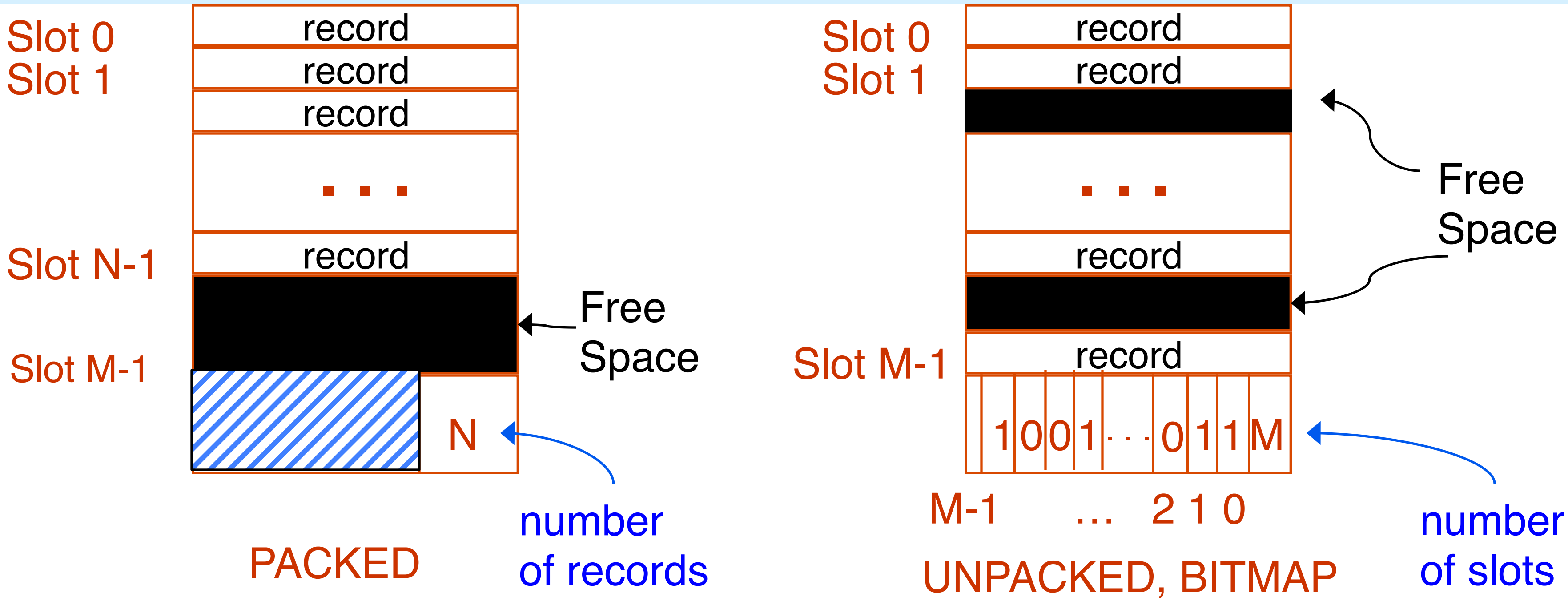
- The Relational Model doesn't expose "pointers", but that doesn't mean that the DBMS doesn't use them internally.

- Q: Can we use memory addresses to permanently "point" to records?

- Systems instead use a "Record ID" or "RecID"

Typically: *Record ID* = *<page id, slot #>*

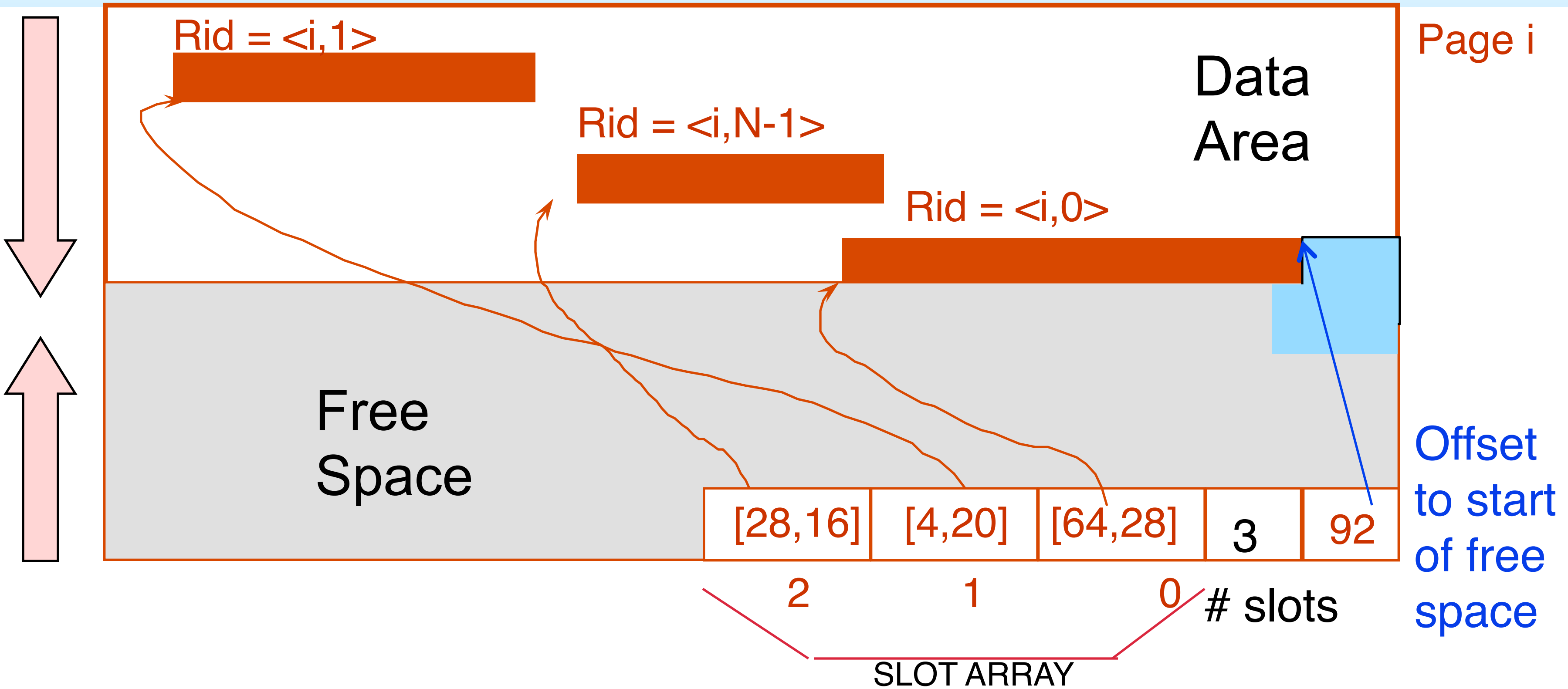
# Page Formats: Fixed Length Records




*In first alternative, free space management* requires record movement.

*Changes RIds - may not be acceptable.*

# "Slotted Page" for Variable Length Records

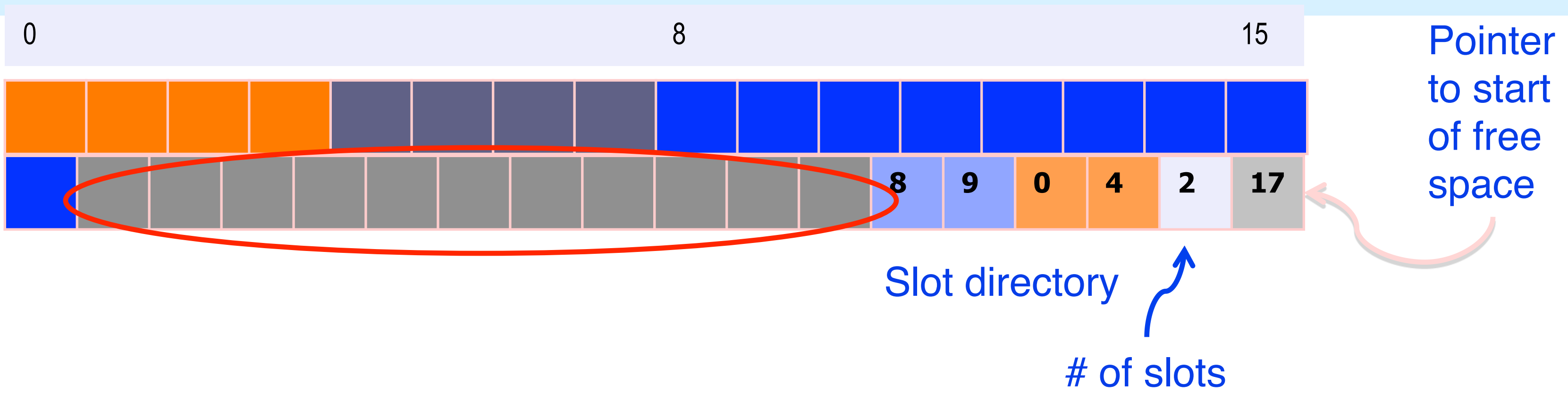



- Slot contains: [offset (from start of page), length]
  - both in bytes
- *Record id = <page id, slot #>*
- Page is full when data space and slot array meet.

# Slotted Page (continued)

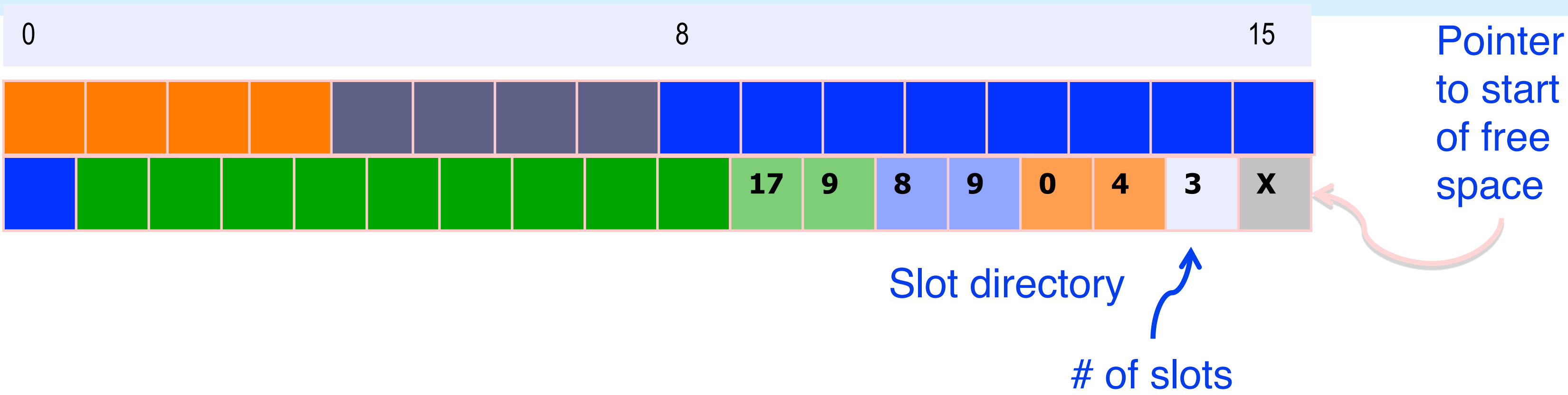
- When need to allocate:
  - If enough room in free space, use it and update free space pointer.
  - Else, try to compact data area, if successful, use the freed space.
  - Else, tell caller that page is full.
- Advantages:
  - Can move records around in page without changing their record ID
  - Allows lazy space management within the page, with opportunity for clean up later

# Slotted page (continued)



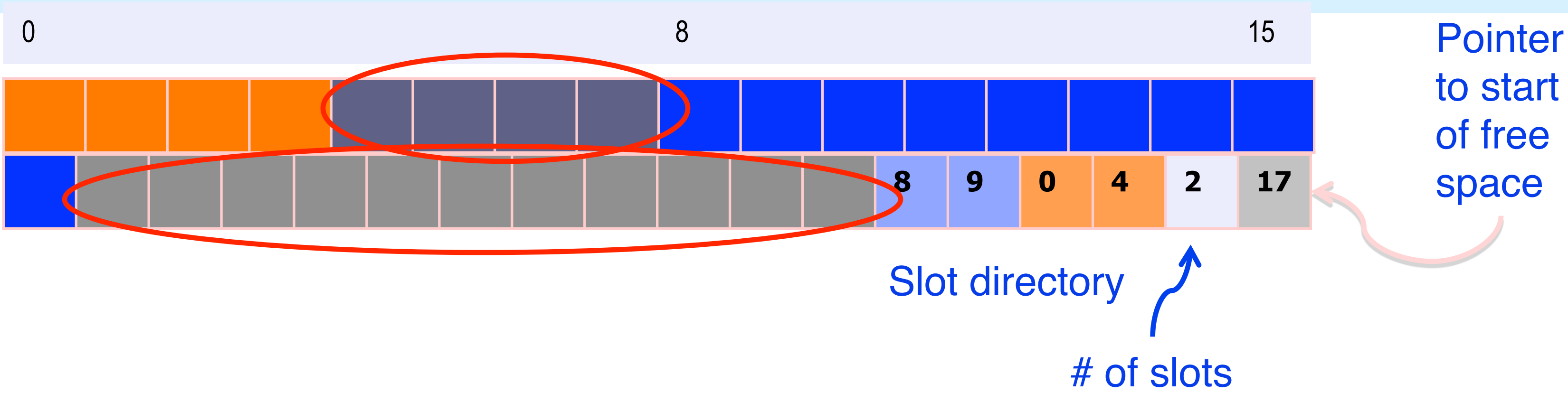

- What's the biggest record you can add to the above page without compacting?
  - Need 2 bytes for slot: [offset, length] plus record.

# Slotted page (continued)



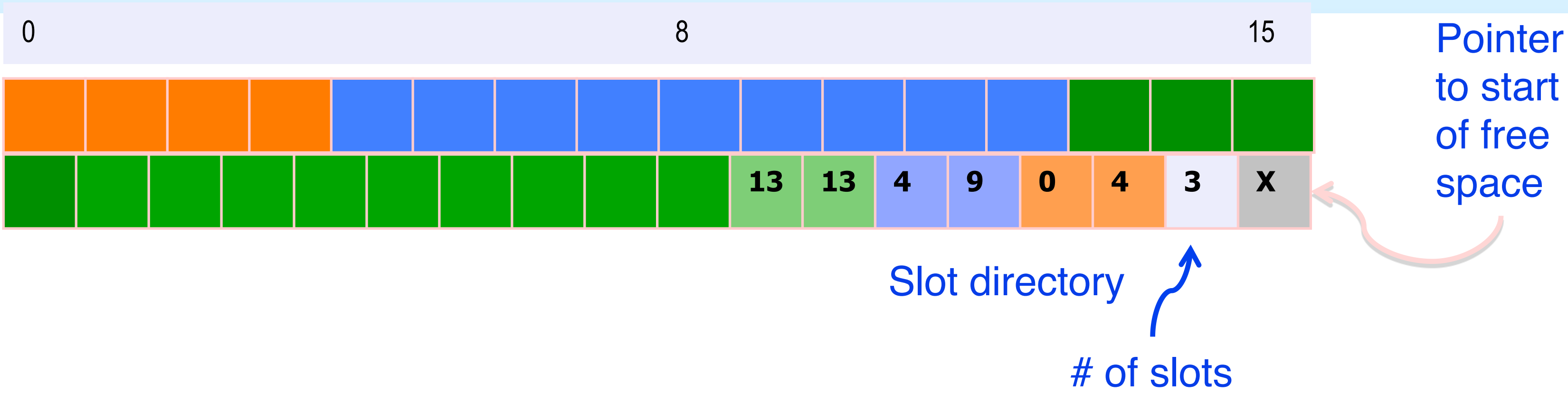

- What’s the biggest record you can add to the above page without compacting?
  - Need 2 bytes for slot: [offset, length] plus record.

# Slotted page (continued)




- What's the biggest record you can add to the above page with compacting?
  - Need 2 bytes for slot: [offset, length] plus record.

# Slotted page (continued)




- What do you do if a record needs to move to a different page?
  - Leave a special “tombstone” object in place of record, pointing to new page & slot.
    - Record id remains unchanged
- What if it needs to move again?
  - Update the original tombstone – so one hop max.

# So far we've organized:

- Fields into Records (fixed and variable length)
- Records into Pages (fixed and variable length)

Now we need to organize Pages into Files

# Alternative File Organizations

Many alternatives exist, *each good for some situations, and not so good in others:*

**Heap files:** Unordered. Fine for file scan retrieving all records. Easy to maintain.

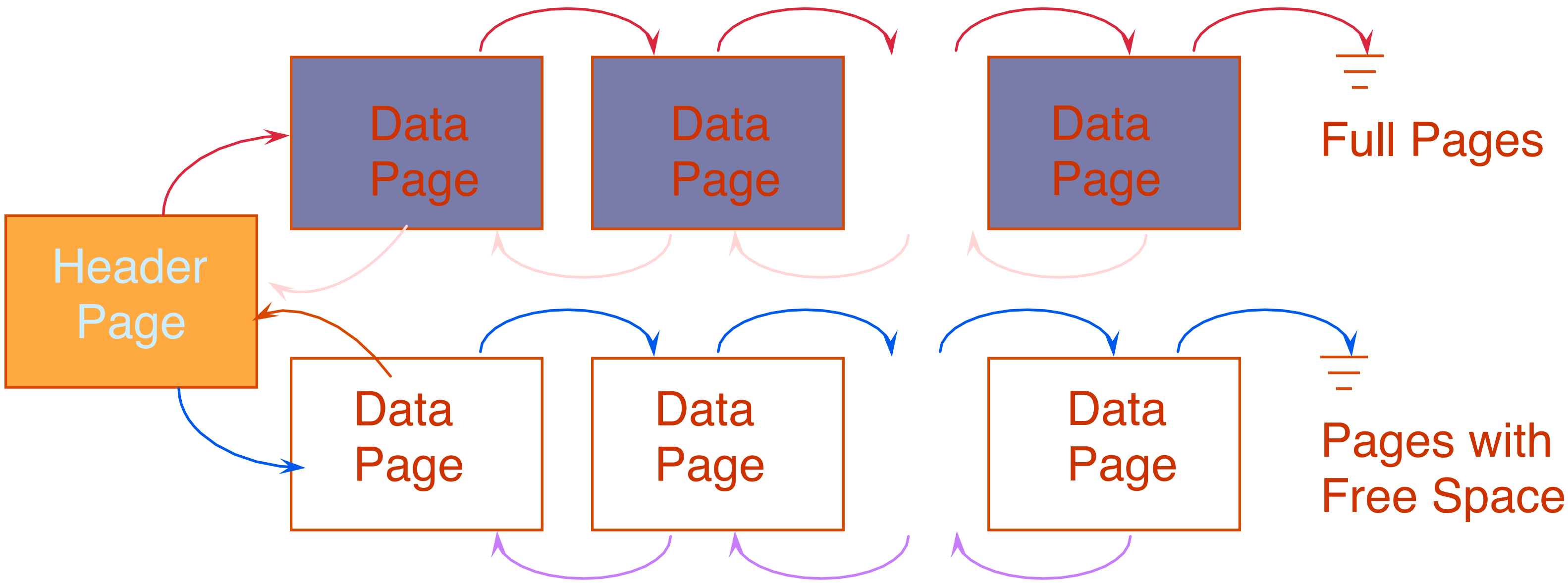
**Sorted Files:** Best for retrieval in *search key* order, or if only a `range' of records is needed. Expensive to maintain.

**Clustered Files** (with Indexes): A compromise between the above two extremes.

# Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, 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
- Can organize as a list, as a directory, a tree, …

# Heap File Implemented as a List

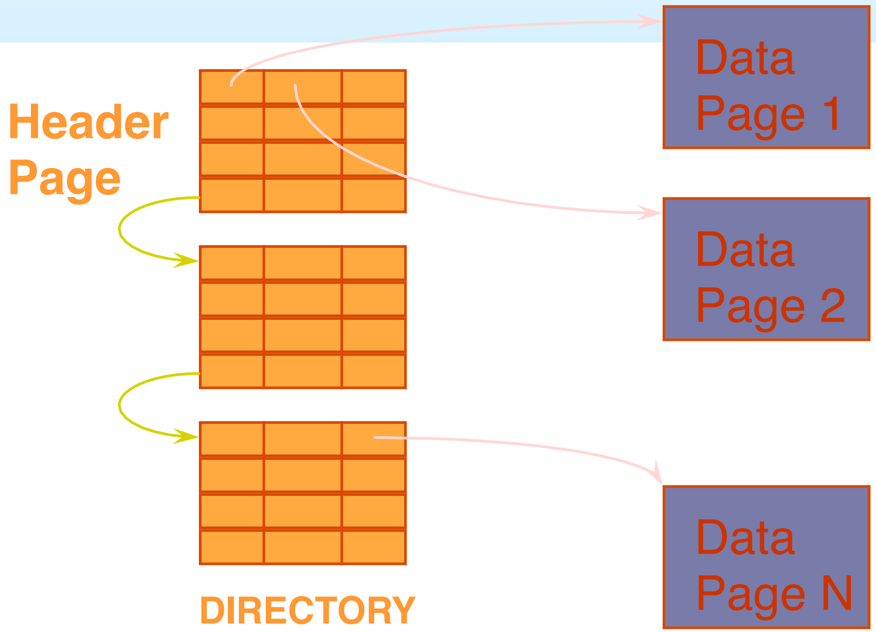



- The Heap file name and header page id must be stored persistently.

  The catalog is a good place for this.

- Each page contains 2 `pointers' 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.

# Cost Model for Analysis

- Average-case analysis; based on several simplistic assumptions.
  - Often called a "back of the envelope" calculation.

- we ignore CPU costs, for simplicity:
  - **B:** The number of data blocks
  - **R:** Number of records per block

- We simply count number of disk block I/O's
  - ignores gains of pre-fetching and sequential access; thus, even I/O cost is only loosely approximated.

# Some Assumptions in the Analysis

- Single record insert and delete.
- Equality selection - exactly one match (what if more or less???).
- For Heap Files we'll assume:
  - Insert always appends to end of file.
  - Delete just leaves free space in the page.
  - Empty pages are not de-allocated.
  - If using directory implementation assume directory is in-memory.

# Average Case I/O Counts for Operations (B = # disk blocks in file)

| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| Scan all records | B | | |
| Equality Search (1 match) | 0.5 B | | |
| Range Search | B | | |
| Insert | 2 | | |
| Delete | 0.5 B+1 | | |

# Sorted Files

- Heap files are lazy on update - you end up paying on searches.
- Sorted files eagerly maintain the file on update.
  - The opposite choice in the trade-off
- Let's consider an extreme version
  - No gaps allowed, pages fully packed always
  - Q: How might you relax these assumptions?
- Assumptions for our BotE Analysis:
  - Files compacted after deletions.
  - Searches are on sort key field(s).

# Average Case I/O Counts for Operations (B = # disk blocks in file)

| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | B | B | |
| **Equality Search (1 match)** | 0.5 B | $\log_2$ B (if on sort key)<br>0.5 B (otherwise) | |
| **Range Search** | B | ($\log_2$ B) + selectivity * B | |
| **Insert** | 2 | ($\log_2$B)+ B | |
| **Delete** | 0.5B+1 | Same cost as Insert | |