Database Systems

Data Storage and Buffer Management II

何明昕 HE Mingxin, Max

Send your email to **c.max@yeah.net** with

a subject like: **DBS***345-Andy: On What …*

Download from **c.program@yeah.net**

/文件中心/网盘/DatabaseSystems2021

Data Storage and Buffer Management II

Where it hits the metal

Review Exercise

- Suppose a hard disk drive has a sector size of 512 bytes, 5000 tracks per surface, 50 sectors per track, and average seek time of 10 msec.

If the disk's RPM is 5400 and one track of data can be transferred per revolution, what is the transfer rate?

Suppose the block size is chosen to be 1024 bytes. Then to store a file containing 100,000 records of 100 bytes each, how many blocks are required?

Review Exercise Answer

- Suppose a hard disk drive has a sector size of 512 bytes, 5000 tracks per surface, 50 sectors per track, and average seek time of 10 msec.

If the disk's RPM is 5400 and one track of data can be transferred per revolution, what is the transfer rate?

$$\frac{\#bytes}{min} = \frac{\#bytes}{sector} \times \frac{\#sectors}{track} \times \frac{\#tracks}{rev} \times \frac{\#revs}{min}$$

$$= 512 \times 50 \times 1 \times 5400$$

$$= 138240000 \frac{\text{bytes}}{\text{min}} = 2250 \frac{KB}{sec} \sim 2.19 \frac{MB}{sec}$$

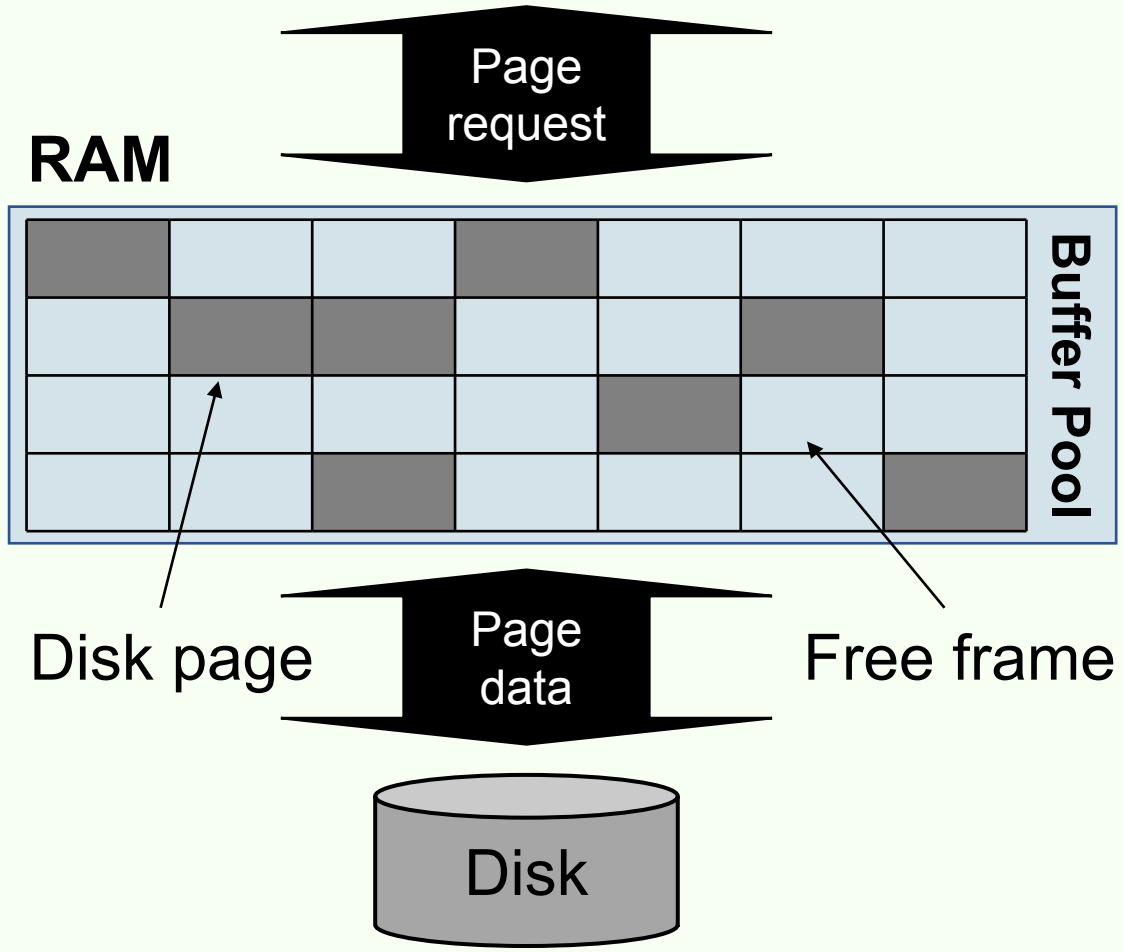
Suppose the block size is chosen to be 1024 bytes. Then to store a file containing 100,000 records of 100 bytes each, how many blocks are required?

$$\frac{\#bytes}{block} = \frac{\#bytes}{record} \times \frac{\#records}{block} \Rightarrow \frac{\#records}{block} = \left\lfloor \frac{1024}{100} \right\rfloor = 10$$

$$\frac{\#records}{file} = \frac{\#records}{block} \times \frac{\#blocks}{file} \Rightarrow \frac{\#blocks}{file} = \frac{100000}{10} = 10000$$

Buffer Manager

- Data must be in RAM for DBMS to operate on it
- However, can't keep all pages DBMS needs in RAM
- *Buffer manager* is in charge of bringing pages from disk to memory as needed by DBMS
- Keeps a pool of slots (i.e. *frames*) to load pages into, i.e. *buffer pool*



Buffer Manager (Cont.)

- Requests to buffer manager:
  - Request a page
  - Release a page when it is no longer needed
  - Notify the buffer manager when a page is modified
- Information saved for each frame:
  - Pin count: number of users of the page
    - Pin a page: indicate that the page is in use (⇒ pin count > 0)
    - Unpin a page: release the page, and also indicate whether the page is *dirtied* (⇒ pin count = 0)
  - Dirty bit: indicates whether the page has been modified and the changes need to be propagated to the disk

Serving a Page

- When a page a requested, if:
  1. the page is in the buffer pool, then
     - return a handle to the frame in the buffer pool
     - increment the pin count
  2. the page is not in the buffer pool, then
     - choose a frame for replacement (i.e. with pin count = 0)
     - if the frame is dirty, write it to disk
     - read requested page into the chosen frame
     - pin the page and return a handle to it

Can you tell the number of users currently using some page in the buffer pool?

Buffer Replacement Policy

- How to choose a frame for replacement
  1. Least recently used (LRU)
  2. Clock
  3. Most recently used (MRU)
  4. FIFO
  5. Random, …
- Replacement policy has a big impact on number of I/Os required to answer various queries
  - Depends on the *access pattern(s)* of queries accessing the data

Least Recently Used (LRU) Replacement Policy

- Keep a *queue* of pointers to frames available to be replaced (i.e. pin count = 0)
  - Add available frame pointers to the tail of the queue
  - Pick the next available frame to be replaced from the head of the queue
- Example
  - Buffer pool size = 3
  - #database pages on disk = 5 (A, B, C, D, E)
  - Determine the states of page replacement queue and the buffer pool when the following sequence of requests are submitted to the buffer manager:
    - Request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, request A, request E

Clock Replacement Policy

- Variant of LRU with lower memory overhead
- *N* frames are ordered as a cycle
- A *current* variable takes on values 1 to *N*, pointing to the next frame to be considered for replacement
- A *referenced bit* for each page which turns on (is set to 1) when the page pin count becomes 0
- Procedure: when need to load a new frame, consider the page pointed to by *current*
  1. If pin count > 0, increment current (mod *N*)
  2. Else if referenced = 1, set it to 0 and increment current (mod *N*)
     - Less likely to replace a recently referenced page
  3. Else (referenced = 0 and pin count = 0), choose current page for replacement

Some sources call it the "second chance" algorithm

Repeat the previous example with clock

Sequential Flooding

- Nasty situation caused by LRU policy when repeated sequential accesses happen
  - #buffer frames < #pages in file
  - Each page request causes an I/O!
  - MRU works much better in these situations

Why?

DBMS vs. OS File System

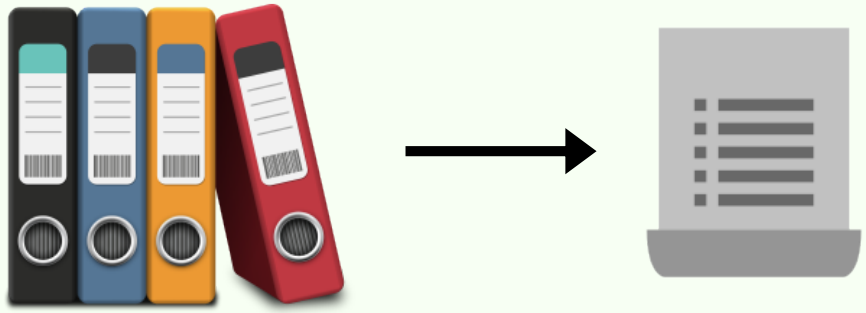
- Q: why not let the OS handle disk space and buffer management?
- A: DBMS is better at predicting reference patterns
- Buffer management in DBMS requires ability to
  - “pin” a page in buffer pool
  - force a page to disk (for recovery and concurrency)
  - adjust the replacement policy
  - pre-fetch pages based on predictable access patterns
- DBMS can better control the overlap of I/O with computation
  - Also can leverage multiple disks more effectively

Recap: Tables on Disk

- Data is stored in tables
- Each table is stored in a file on disk
- Each file consists of multiple pages
- Each page consists of multiple records (i.e. tuples)
- Each records consists of multiple (attribute, value) pairs
- Data is allocated/deallocated in increments of pages
- Logically-close pages should be nearby in the disk

Files of (Pages of) Records

- I/O is performed in page units
- However, higher level data management operations require accessing and manipulating records and files of records
- File = collection of pages
  Page = collection of records
- File operations
  - Insert/delete/modify a record
  - Read a particular record (specified using a *record ID*)
  - Scan all records (possibly with some condition(s) on the records to be retrieved)
- Records are organized in files using various formats, i.e. *file organizations*
  - Each file organizations makes certain operations efficient but other operations expensive

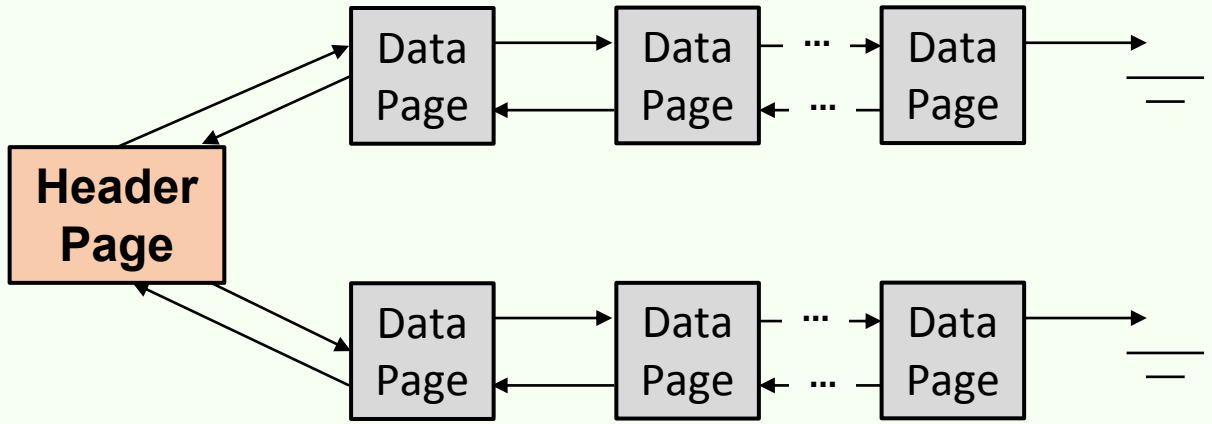


Unordered (Heap) File

- Simplest file organization
- Each page contains records in no particular order
- As the file grows and shrinks, pages are allocated and de-allocated
- To support record level operations, we must keep track of:
  - Pages in a file (using page ID (pid))
  - Free space on pages
  - Records on a page (using record ID (rid))
- Operations: create/destroy file, insert/delete record, fetch a record with a specified rid, scan all records

Heap File Implemented as a List

- (heap file name, header page id) are stored somewhere
- Each (data) page has two pointers + data
  - Each pointer stores a pid
- Pages in the free space list have “some” free space



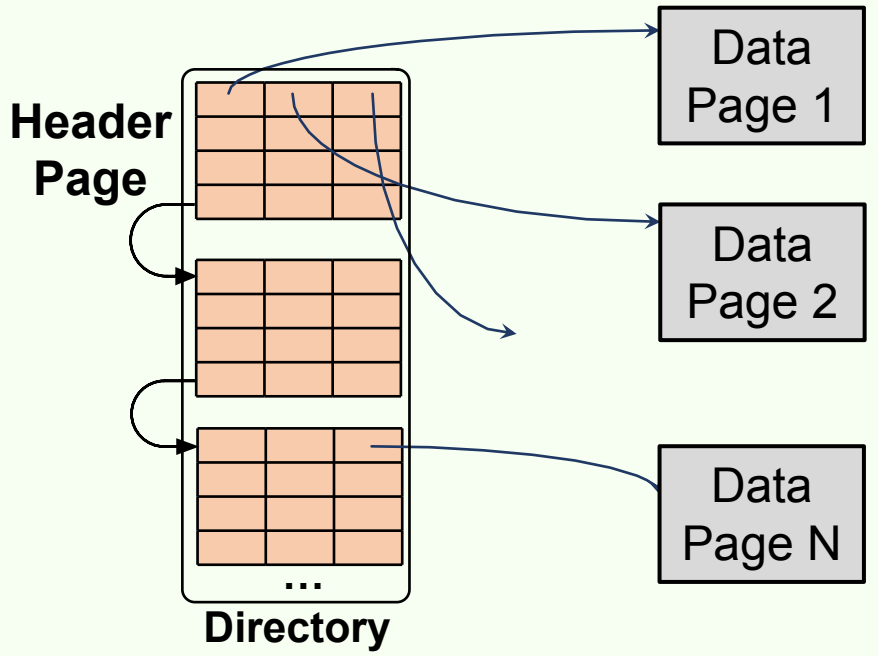
Q: What happens with fixed- vs. variable-length records in terms of full and non-full pages?

Q: How would you find a record with a specific rid?

Q: Why do you need the backward pointers?

Heap File Implemented Using a Page Directory

- Each entry for a page also keeps track of
  - whether the page full or not, or
  - number of free bytes on the page
- Can locate pages for new tuples faster
- Directory is a collection of pages; linked list implementation is just one alternative



Q: What happens with fixed- vs. variable-length records in terms of full and non-full pages?

Q: How would you find a record with a specific rid?

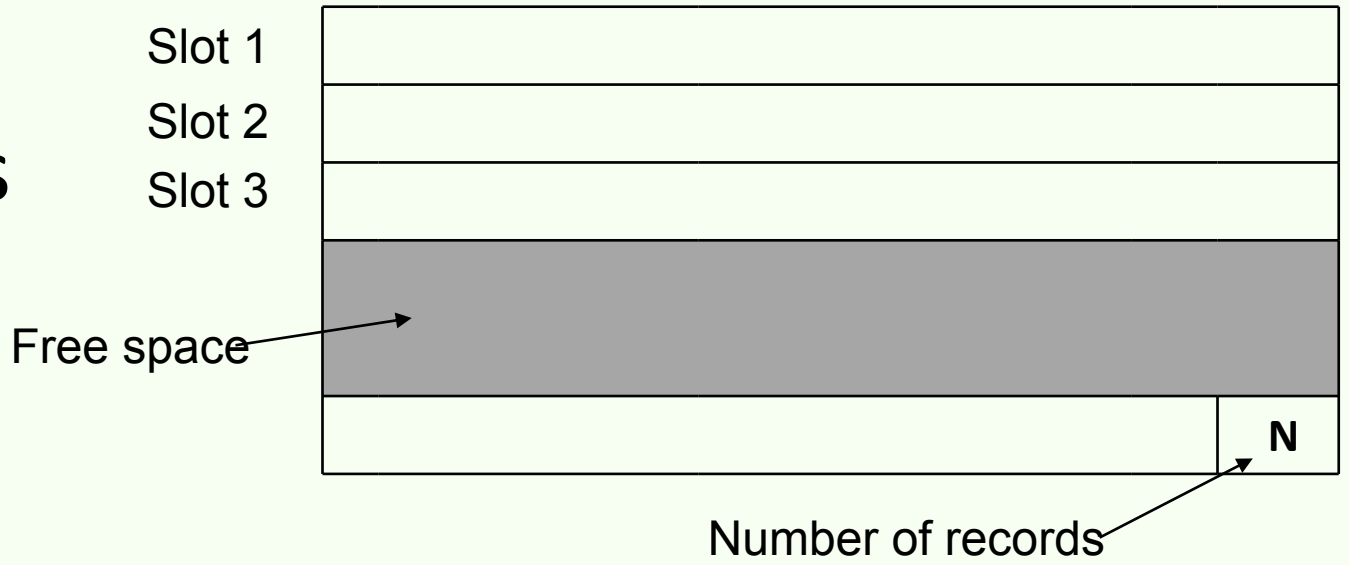
Page Organizations

- A page as a collection of records/tuples
- Queries deal with tuples
- Hence, the *slotted page format*
  - A page is a collection of *slots*
  - Each slot may contain a record
- rid = <pid, slot number>
- Many slotted page organizations
  - Need to support search, insert and delete records on a page
- Page organizations for various *record organizations*
  - Fixed-length records
  - Variable-length records

Other ways of generating rids?

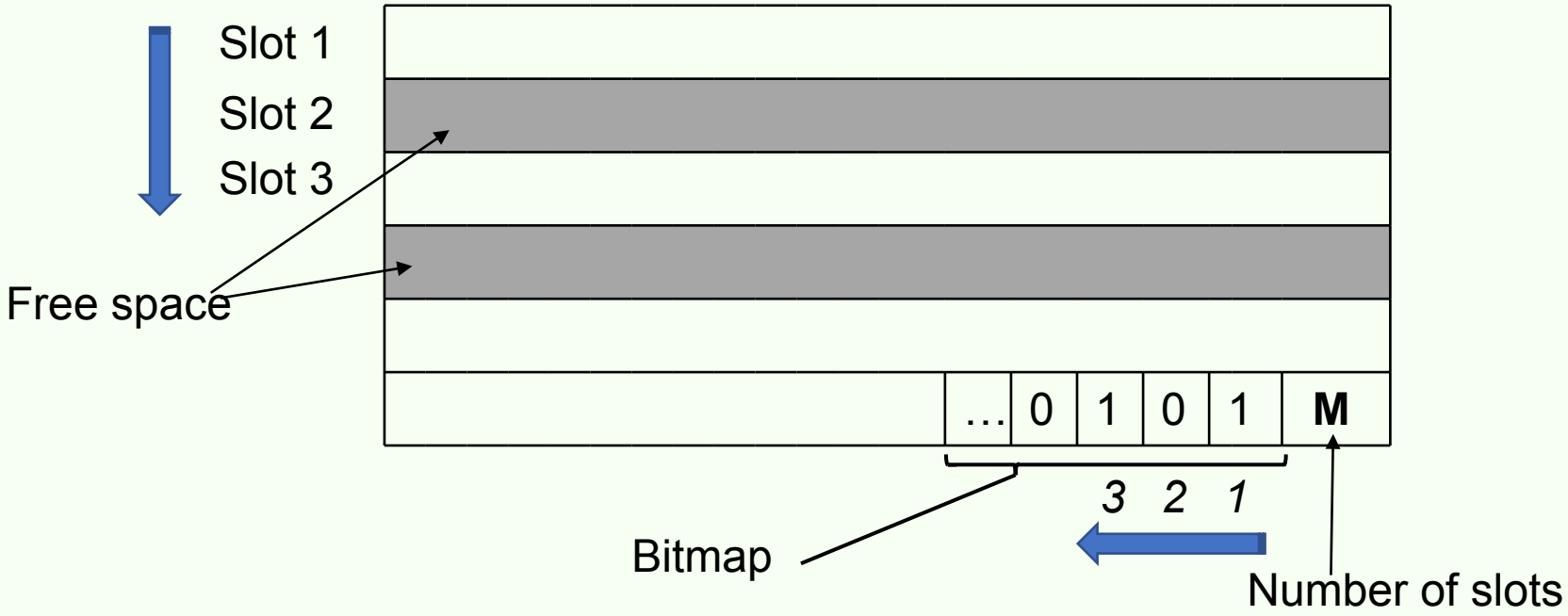
Organization of Pages of Fixed-length Record

1. Packed organization
   - N records are always stored in the first N slots
- Problem: moving records (for free space management) changes the record ID
   - Might not be acceptable



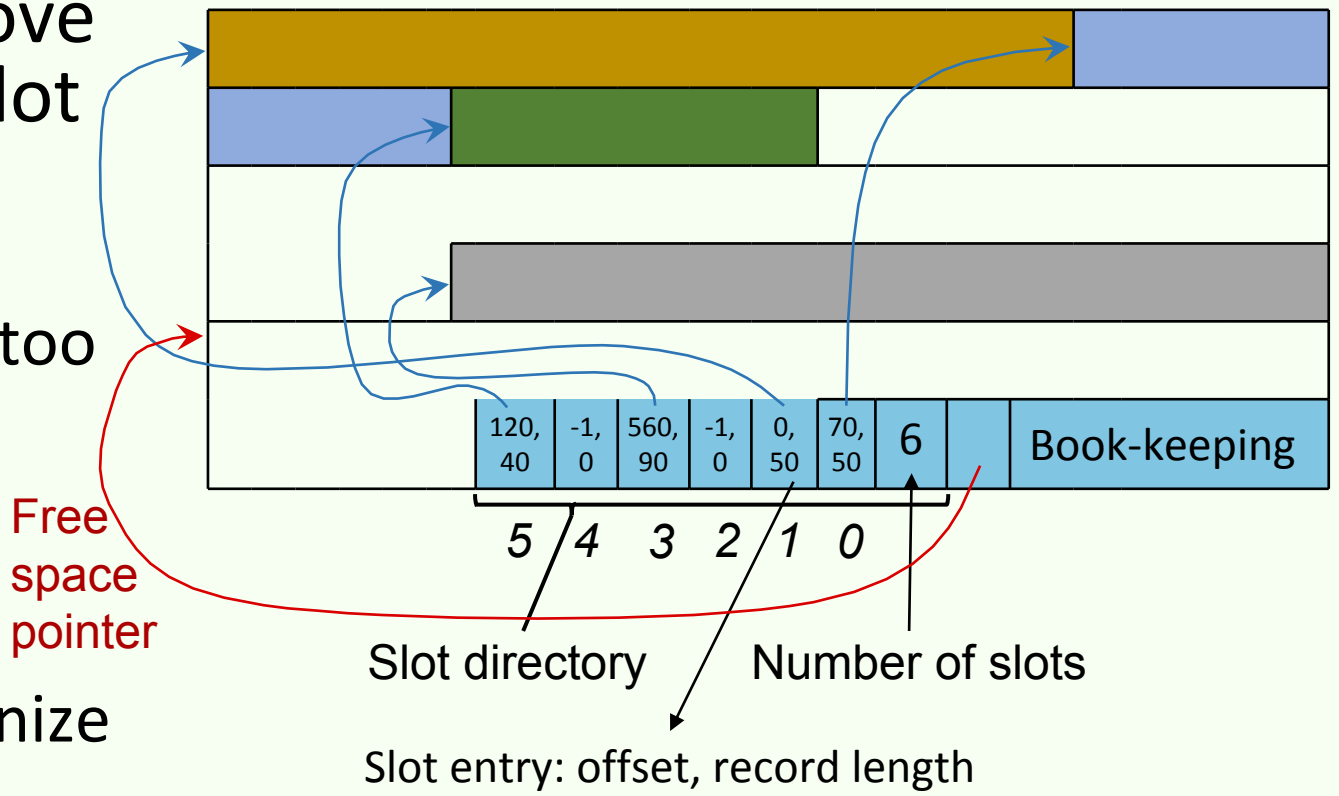
Organization of Pages of Fixed-length Record (Cont.)

2. Unpacked organization
   - Use a *bitmap* to locate records in the page



Organization of Pages of Variable-length Record

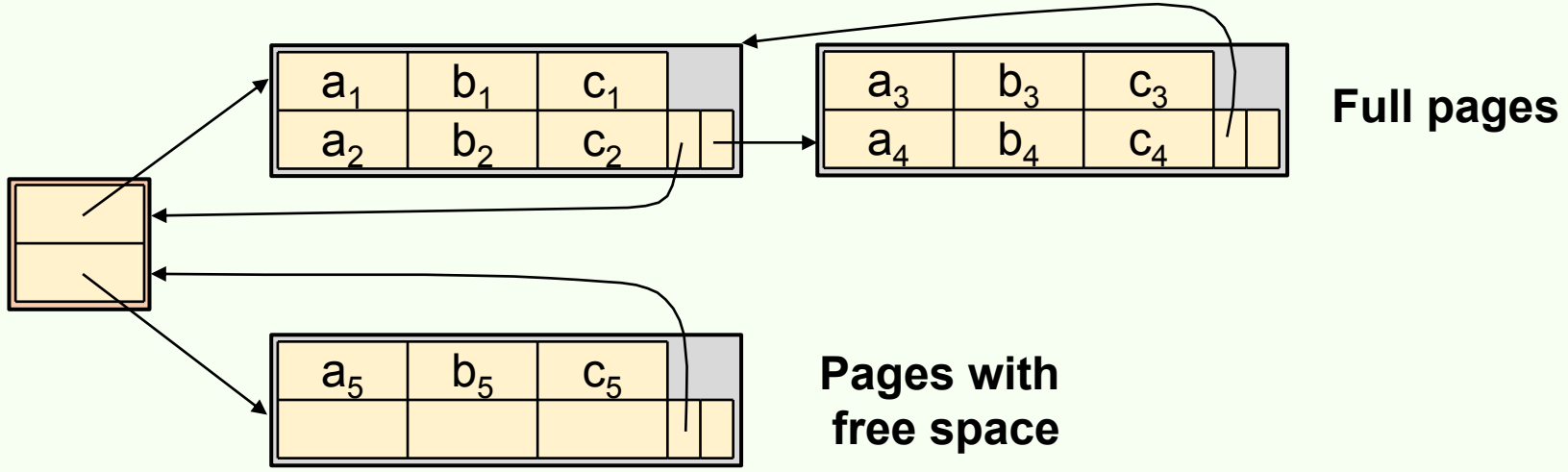
- Directory grows backwards
- rid does not change if you move a record on the same page (slot number is determined by position in slot directory)
  - Good for fixed-length records too
- Deletion: offset is set to -1
- Insertion
  - Use any available slots
  - if no space is available, reorganize (move records around)



(Grossly Simplified) Example

R

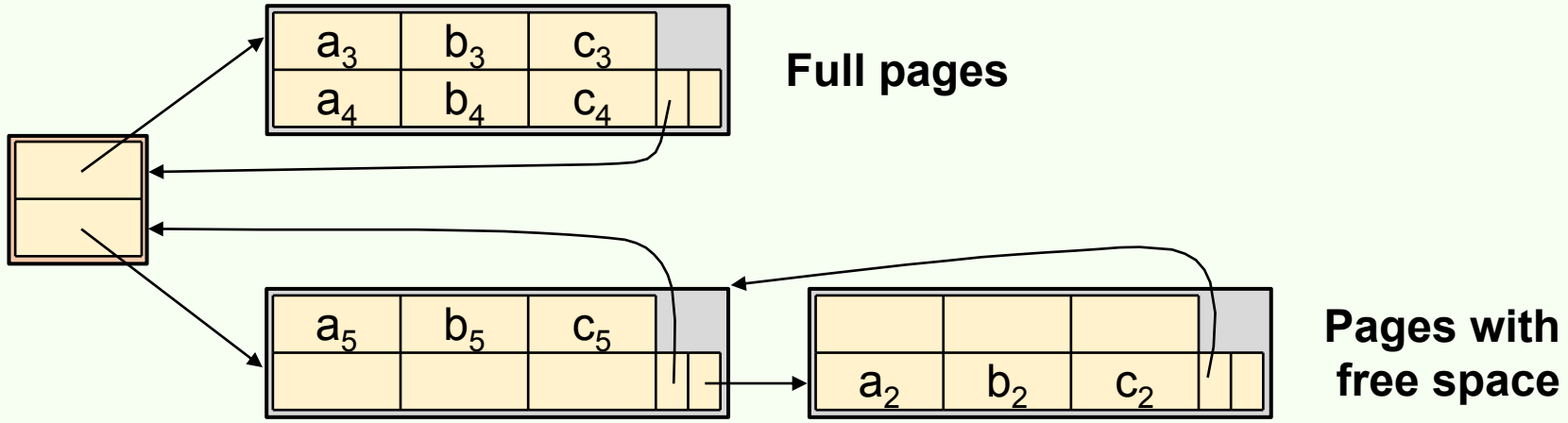
a	b	c
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3
a_4	b_4	c_4
a_5	b_5	c_5




```
DELETE FROM R WHERE a = a1;
```

R

a	b	c
a_2	b_2	c_2
a_3	b_3	c_3
a_4	b_4	c_4
a_5	b_5	c_5



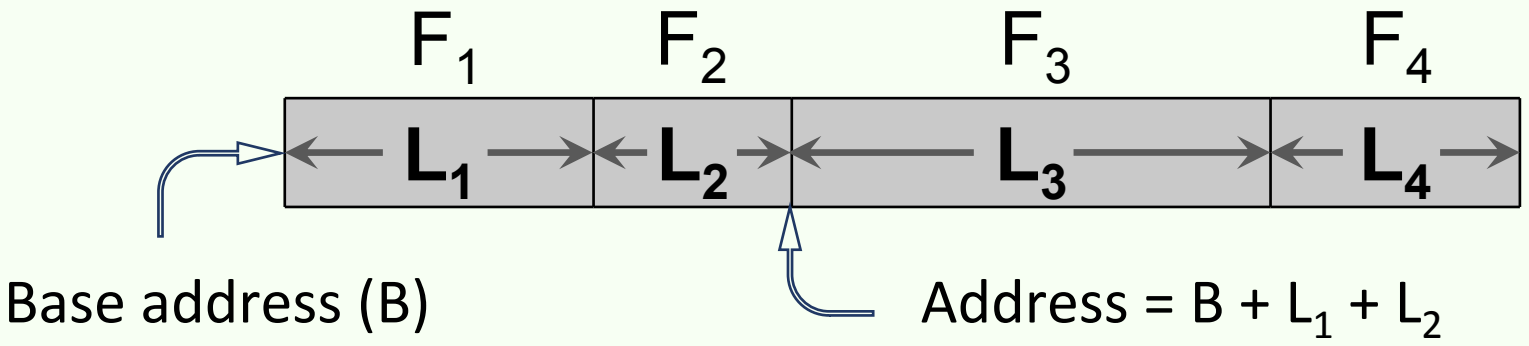
Record Organization (Format)

- Recap
  - File organization
    - Heap file
      - As doubly-linked list
      - Using page directory
  - Page organization
    - For fixed-length records
      - Packed
      - Unpacked
    - For variable-length records

Let's see fixed- and variable-length record formats now.

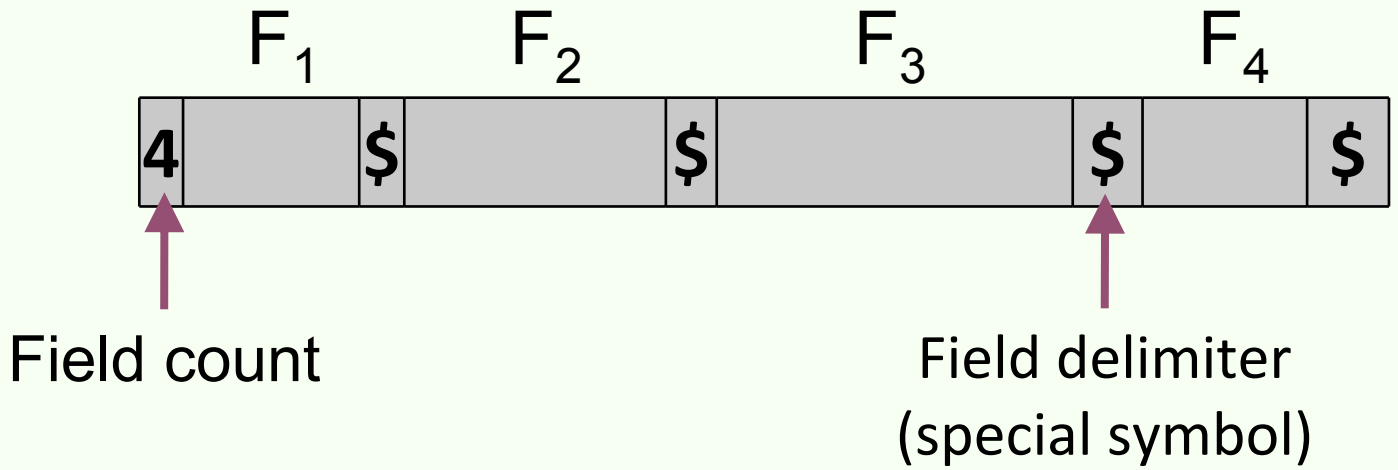
Fixed-length Records

- All records are of the same length, number of fields and field types
  - These information are stored in *system catalog*
- The address of each field can be computed from the information in the catalog



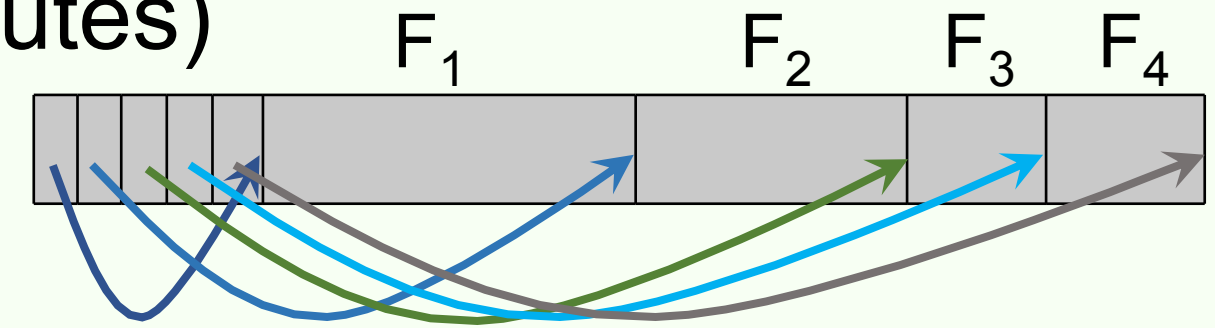
Variable-length Records, Option 1

- Store fields consecutively
- Use delimiters to denote the end of each field
- Need to scan the whole record to locate a field



Variable-length Records, Option 2

- Store fields consecutively
- Use an array of integer offsets in the beginning
  - Efficient storage of NULLs, small directory overhead, direct access to the i^{th} field
- Issues with growing records (change in attribute value, add/drop attributes)



Motivating Example: Looking Up One Attribute

```
CREATE TABLE T (a INTEGER, b INTEGER, c VARCHAR(255), …);
```

```
SELECT a
FROM T
WHERE a > 10;
```

- What can potentially go wrong if we use any of the previous record formats?

Column Stores

- Store columns *vertically*
- Each column of a relation is stored in a different file
  - Can be compressed as well
- Contrast with a *row store* that stores all the attributes of a tuple/record contiguously

File 1

1234	45	Here goes a very long sentence 1
4657	2	Here goes a very long sentence 2
3578	45	Here goes a very long sentence 3

Row store

File 1 *File 2* *File 3*

1234	45	Here goes a very long sentence 1
4657	2	Here goes a very long sentence 2
3578	45	Here goes a very long sentence 3

Column store

Recap

- Architecture of a typical DBMS
- Memory hierarchy
  - CPU cache, main memory, SSD, disk, tape
- Secondary storage (disk and SSD)
- Buffer manager
  - Buffer replacement policies: LRU, clock, MRU, FIFO, random, …
- File organization
  - Heap file (as doubly-linked list or directory)
- Page organization
  - For fixed-length (packed and unpacked) and variable-length records
- Record organization
  - Fixed-length and variable-length (two variations) records
- Row-store vs. column-store

Next Up

Indexing

Questions?