Lecture 10:
Buffer Manager and File Organization

Today's Lecture

1. Recap: Buffer Manager
2. Replacement Policies
3. Files and Records

1. Buffer Manager

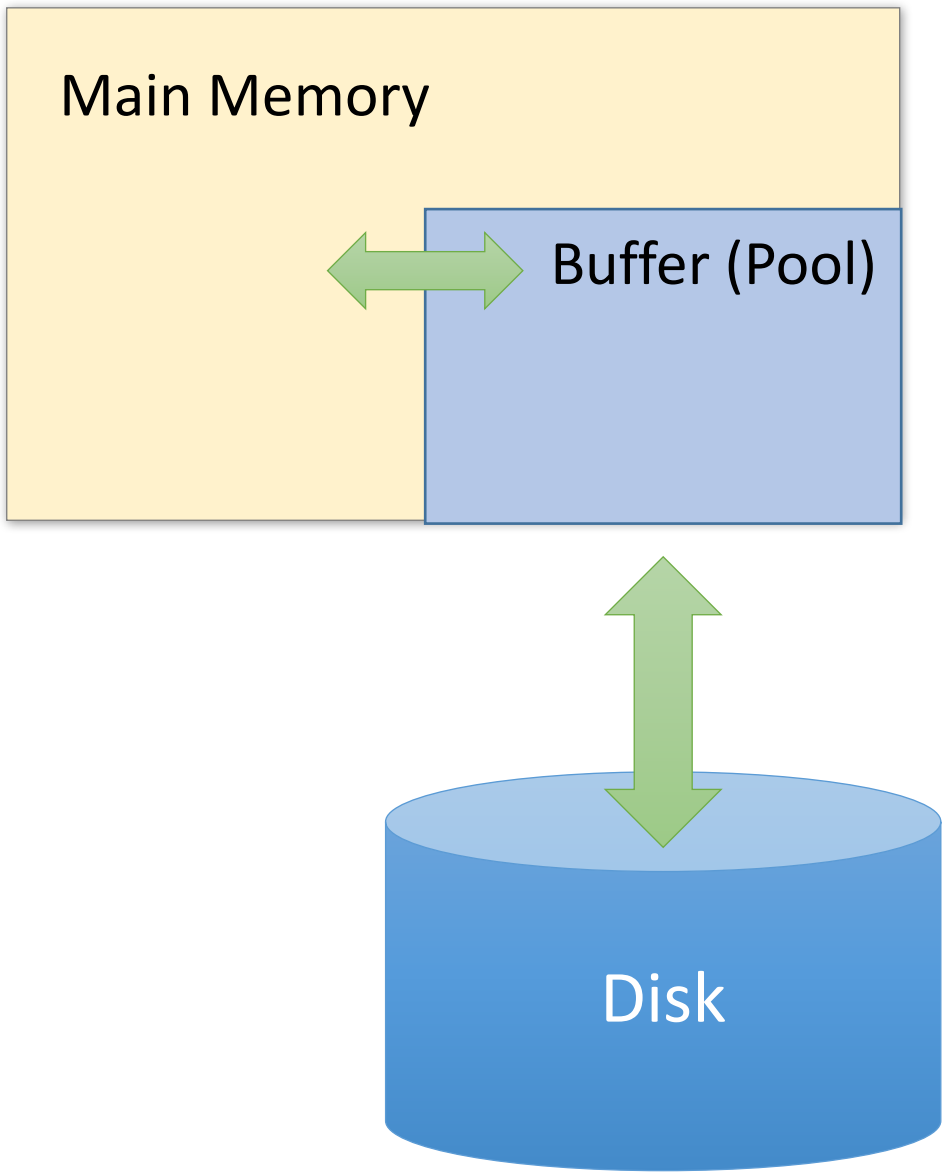
What you will learn about in this section

1. Buffer Pool

2. Buffer Manager

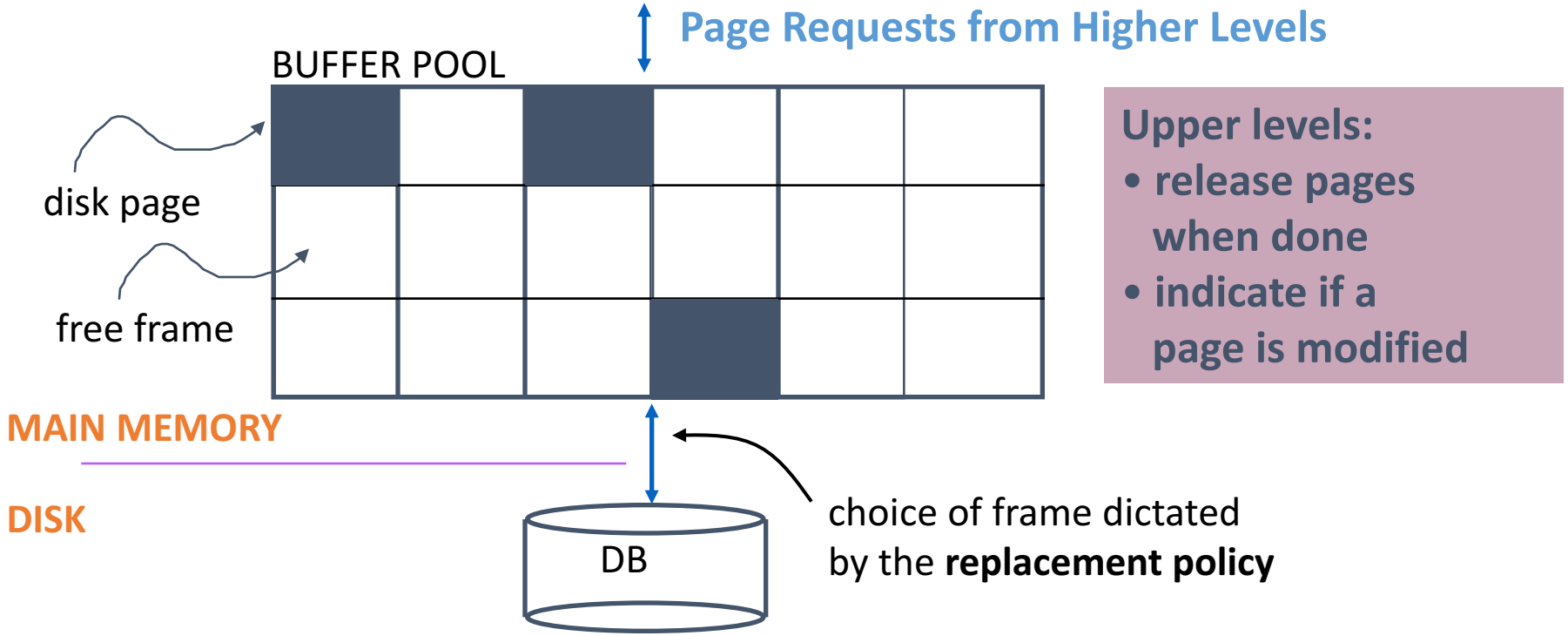
The Buffer (Pool)

- A **buffer** is a region of physical memory used to store *temporary data*
  - *In this lecture:* a region in main memory used to store **intermediate data between disk and processes**
- *Key idea:* Reading / writing to disk is slow- need to cache data!



Buffer Management in a DBMS

- Data must be in RAM for DBMS to operate on it!
  - Can't keep all the DBMS pages in main memory
- Buffer Manager: Efficiently uses main memory
  - Memory divided into buffer frames: slots for holding disk pages



Upper levels:
- **release pages when done**
- **indicate if a page is modified**

Buffer Manager

2 requestors want to modify the same page?

Handled by the concurrency control manager (using locks)

- Bookkeeping per frame:
  - ***Pin count*** : # users of the page in the frame
    - *Pinning* : Indicate that the page is in use
    - *Unpinning* : Release the page, and also indicate if the page is *dirtied*
  - ***Dirty bit*** : Indicates if changes must be propagated to disk

Buffer Manager

- When a Page is requested:
  - In buffer pool -> return a handle to the frame. Done!
    - Increment the pin count
  - Not in the buffer pool:
    - Choose a frame for *replacement*

      (Only replace pages with pin count == 0)
    - If frame is dirty, write it to disk
    - Read requested page into chosen frame
    - Pin the page and return its address

Can you tell the # current users of a page in the BP?

Pin Count!

2. Replacement Policies

What you will learn about in this section

1. Replacement Policy
2. LRU and Clock
3. Sequential Flooding

Buffer replacement policy

- How do we choose a frame for replacement?
  - LRU (**L**east **R**ecently **U**sed)
  - Clock
  - MRU (**M**ost **R**ecently **U**sed)
  - FIFO, random, …
- The replacement policy has big impact on # of I/O's (depends on the access pattern)

LRU

- uses a **queue** of pointers to frames that have pin count = 0
- a page request uses frames only from the *head* of the queue
- when a the pin count of a frame goes to 0, it is added to the *end* of the queue

LRU Example

4 RAM frames

Buffer pool with 4 frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c						
		a	a	a						
			d	d						
				b						
	F	F	F	F						

LRU Example

c a d b

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e					
		a	a	a	a					
			d	d	d					
				b	b					
	F	F	F	F	F					

LRU Example

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e		
		a	a	a	a	a	a	a		
			d	d	d	d	d	d		
				b	b	b	b	b		
	F	F	F	F	F					

LRU Example

d e a b

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e	e	
		a	a	a	a	a	a	a	a	
			d	d	d	d	d	d	c	
				b	b	b	b	b	b	
	F	F	F	F	F				F	

LRU Example

Which page is evicted next?

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e	e	
		a	a	a	a	a	a	a	a	
			d	d	d	d	d	d	c	
				b	b	b	b	b	b	
	F	F	F	F	F				F	

LRU Example

e a b c

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e	e	d
		a	a	a	a	a	a	a	a	a
			d	d	d	d	d	d	c	c
				b	b	b	b	b	b	b
	F	F	F	F	F				F	F

7 page faults for 10 requests

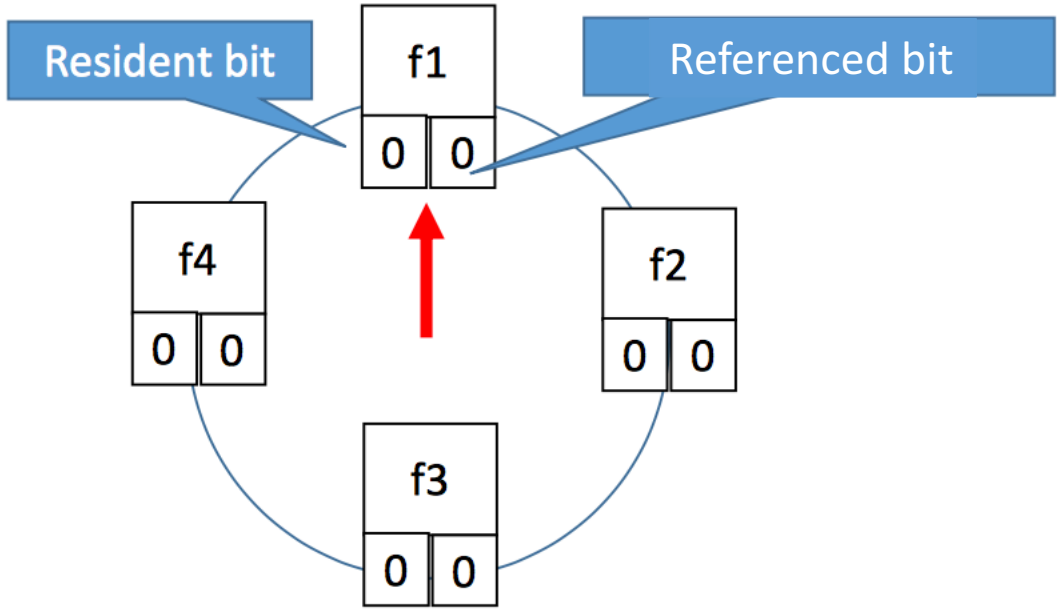
Clock

- Variant of LRU with lower memory overhead
- The *N* frames are organized into a cycle
- Each frame has a referenced bit that is set to 1 when pin count becomes 0
- A current variable points to a frame
- When a frame is considered:
  - If pin count > 0, increment current
  - If referenced = 1, set to 0 and increment
  - If referenced = 0 and pin count = 0, choose the page

Clock Example

Req.	c	a	d	b	e	c	a	b	c	d
f1										
f2										
f3										
f4										

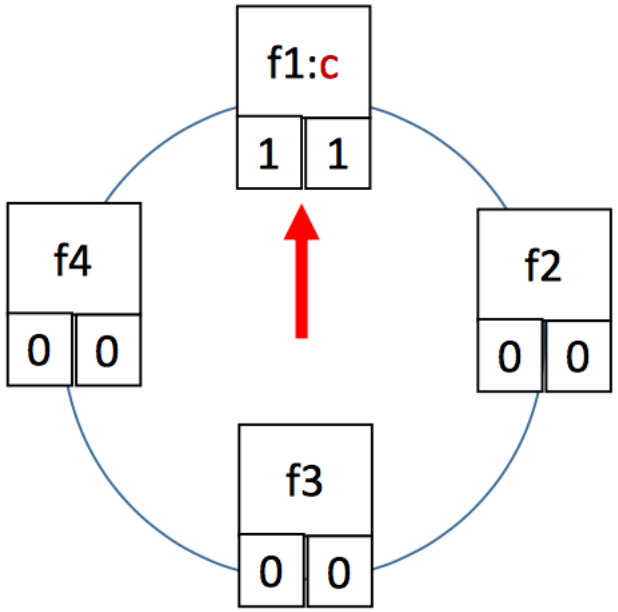
4 RAM frames



Clock Example

4 RAM frames

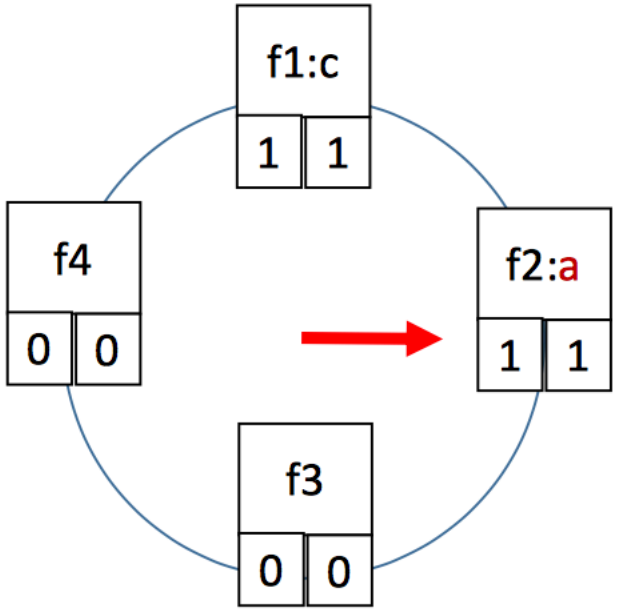
Req.	c	a	d	b	e	c	a	b	c	d
f1	c									
f2										
f3										
f4										
	F									



Clock Example

4 RAM frames

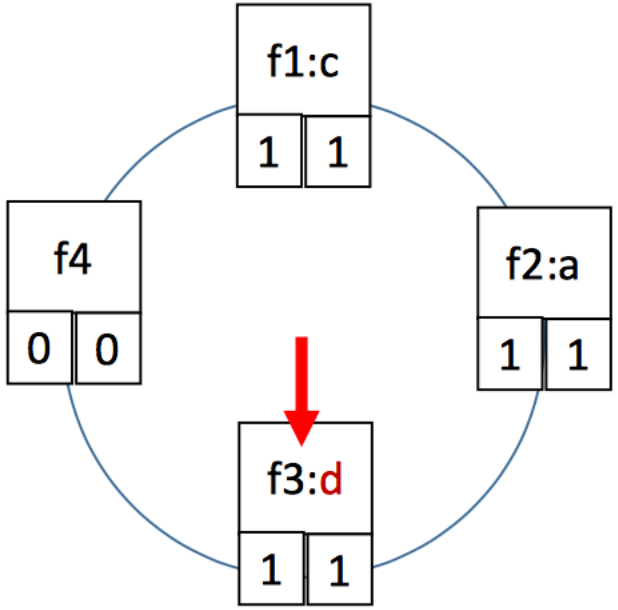
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c								
f2		a								
f3										
f4										
	F	F								



Clock Example

4 RAM frames

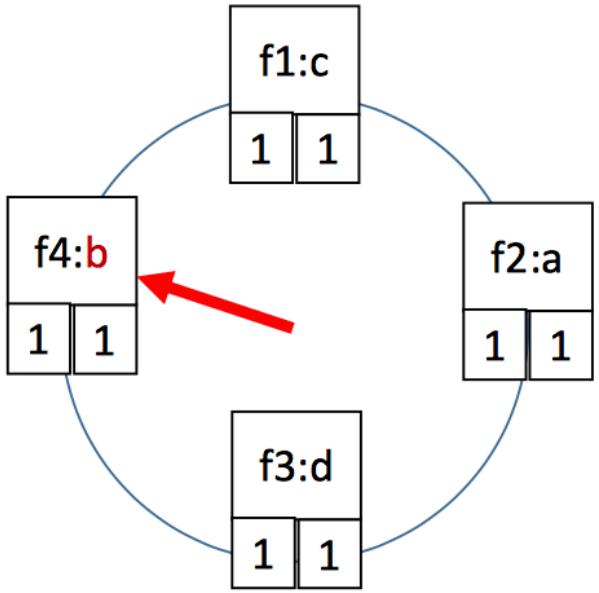
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c							
f2		a	a							
f3			d							
f4										
	F	F	F							



Clock Example

4 RAM frames

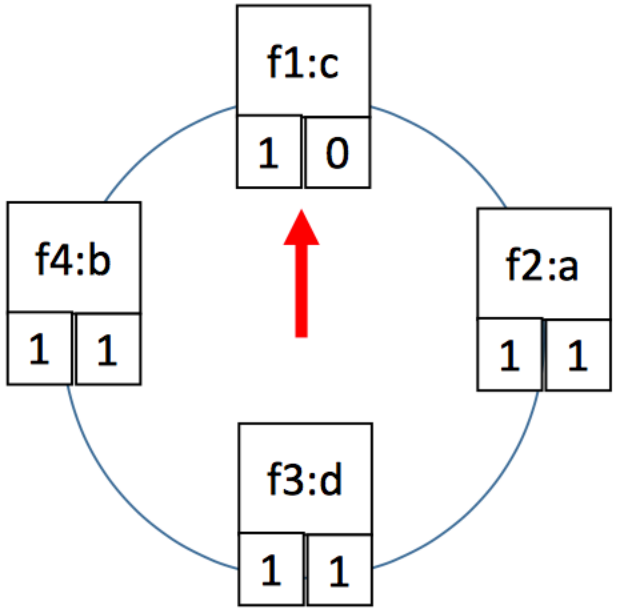
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						



Clock Example

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						

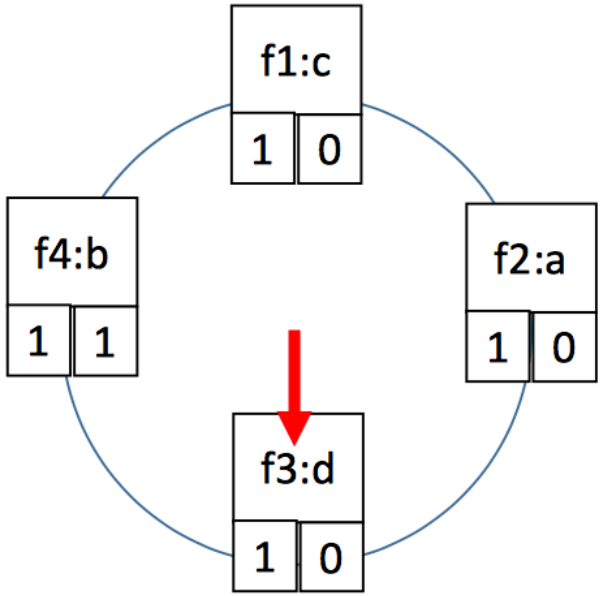


Searching for page to evict

Clock Example

4 RAM frames

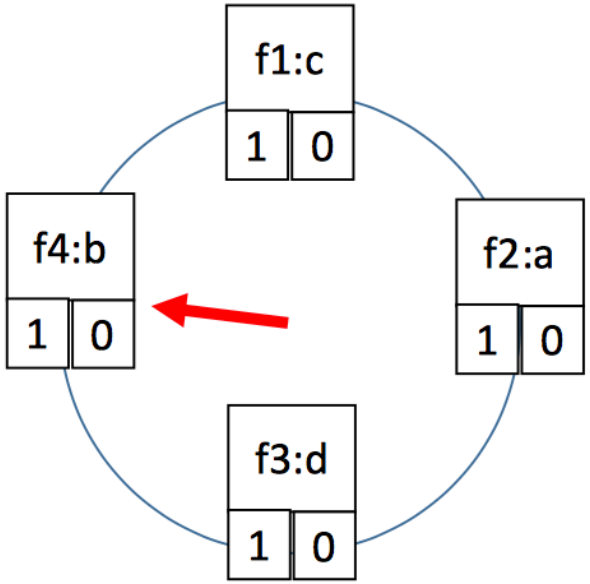
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						



Clock Example

4 RAM frames

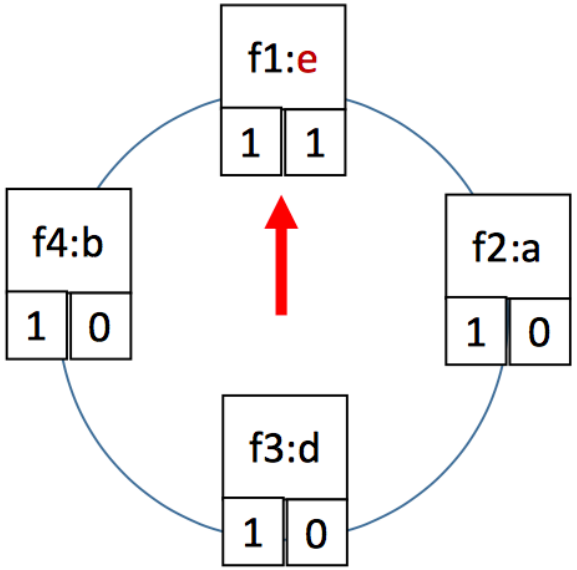
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						



Clock Example

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e					
f2		a	a	a	a					
f3			d	d	d					
f4				b	b					
	F	F	F	F	F					

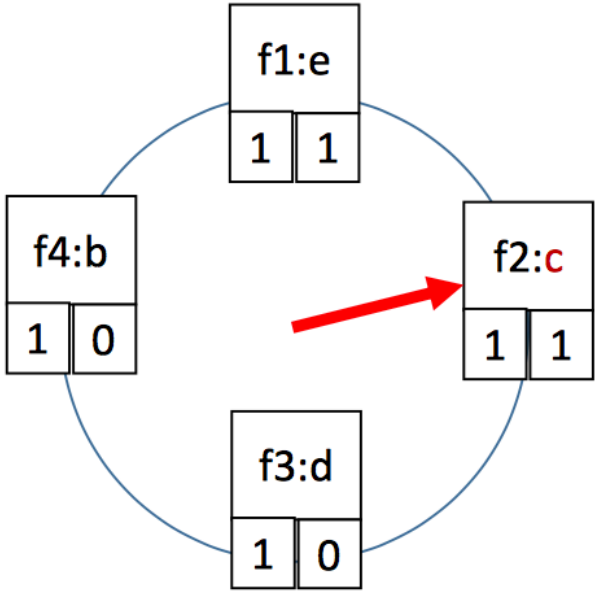


Evicted page c

Clock Example

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e				
f2		a	a	a	a	c				
f3			d	d	d	d				
f4				b	b	b				
	F	F	F	F	F	F				

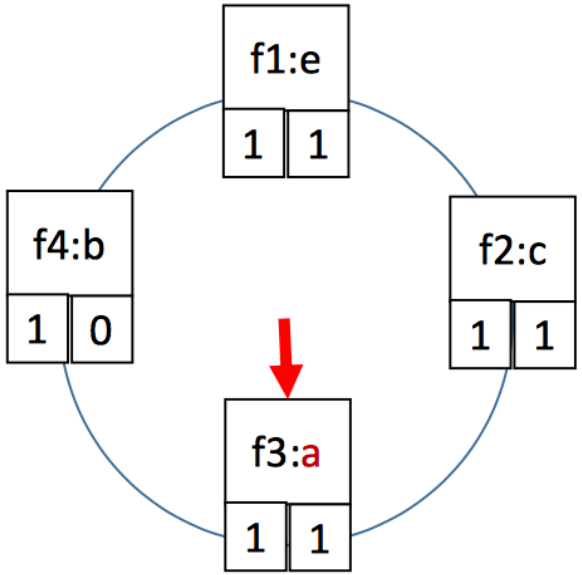


Evicted page a

Clock Example

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e			
f2		a	a	a	a	c	c			
f3			d	d	d	d	a			
f4				b	b	b	b			
	F	F	F	F	F	F	F			

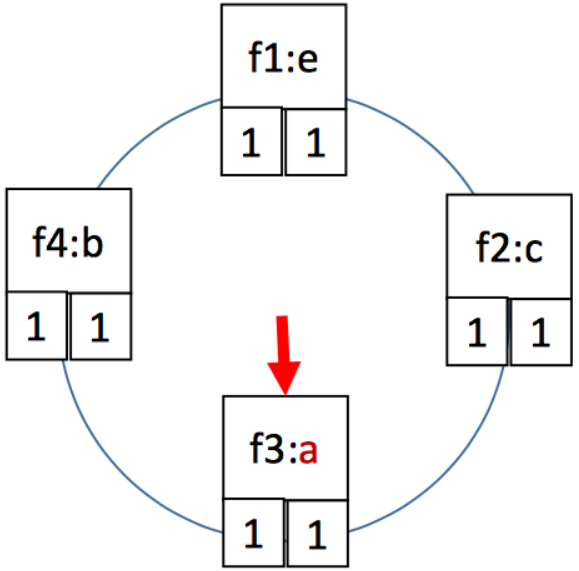


Evicted page d

Clock Example

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e	e		
f2		a	a	a	a	c	c	c		
f3			d	d	d	d	a	a		
f4				b	b	b	b	b		
	F	F	F	F	F	F	F			

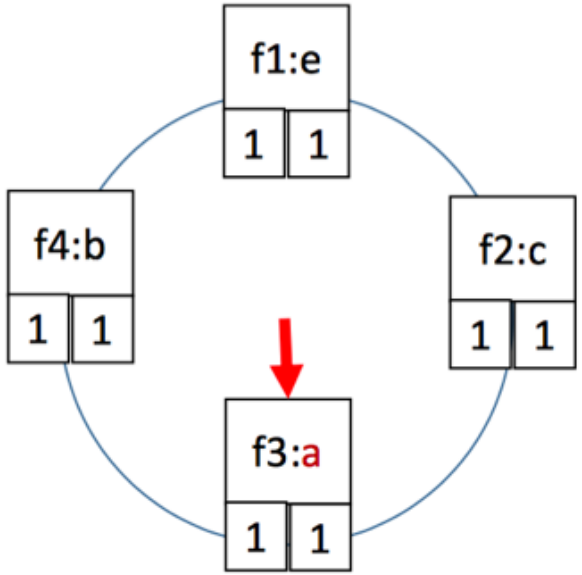


We know that b is in buffer

Clock Example

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e	e	e	
f2		a	a	a	a	c	c	c	c	
f3			d	d	d	d	a	a	a	
f4				b	b	b	b	b	b	
	F	F	F	F	F	F	F			

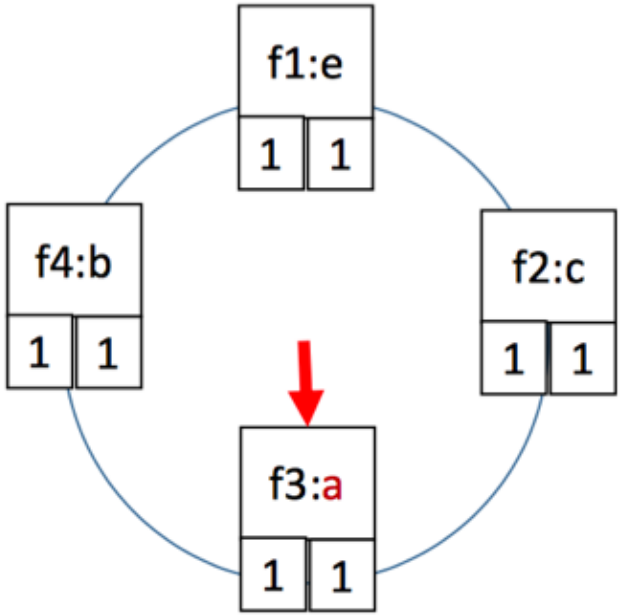


We know that c is in buffer

Clock Example

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e	e	e	
f2		a	a	a	a	c	c	c	c	
f3			d	d	d	d	a	a	a	
f4				b	b	b	b	b	b	
	F	F	F	F	F	F	F			



Where does *d* go:

A: frame 1

B: frame 2

C: frame 3

D: frame 4

Sequential Flooding

- Nasty situation caused by LRU policy + repeated sequential scans
  - # buffer frames < # pages in file
  - each page request causes an I/O !!
  - MRU much better in this situation

Sequential Flooding Example

A: a, b
B: c, d, e, f

3 frames for A - enough

3 frames for B < |B|

req.	a	a	a	a	b	b	b	b
req.	c	d	e	f	c	d	e	f

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

Sequential Flooding Example

A: a, b
B: c, d, e, f

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a					
		F							
	req.	c	d	e	f	c	d	e	f
3 frames for B < |B|		c	c	c					
			d	d					
				e					
		F	F	F					

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

Sequential Flooding Example

A: a, b
B: c, d, e, f

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a				
		F							
	req.	c	d	e	f	c	d	e	f
3 frames for B < |B|		c	c	c	f				
			d	d	d				
				e	e				
		F	F	F	F				

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

f evicts *c*

Sequential Flooding Example

A: a, b
B: c, d, e, f

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

3 frames for A - enough

3 frames for B **< |B|**

req.	a	a	a	a	b	b	b	b
	a	a	a	a	a			
					b			
	F				F			
req.	c	d	e	f	c	d	e	f
	c	c	c	f	f			
		d	d	d	c			
			e	e	e			
	F	F	F	F	F			

c evicts *d*

Sequential Flooding Example

A: a, b
B: c, d, e, f

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a	a	a		
						b	b		
		F				F			
	req.	c	d	e	f	c	d	e	f
3 frames for B < |B|		c	c	c	f	f	f		
			d	d	d	c	c		
				e	e	e	d		
		F	F	F	F	F	F		

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

d evicts *e*

Sequential Flooding Example

A: a, b
B: c, d, e, f

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a	a	a	a	
						b	b	b	
		F				F			
	req.	c	d	e	f	c	d	e	f
3 frames for B < |B|		c	c	c	f	f	f	e	
			d	d	d	c	c	c	
				e	e	e	d	d	
		F	F	F	F	F	F	F	

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

e evicts *f*

Sequential Flooding Example

A: a, b
B: c, d, e, f

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a	a	a	a	a
						b	b	b	b
		F				F			
	req.	c	d	e	f	c	d	e	f
3 frames for B < |B|		c	c	c	f	f	f	e	e
			d	d	d	c	c	c	f
				e	e	e	d	d	d
		F	F	F	F	F	F	F	F

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

f evicts *c*

Sequential Flooding Example

A: a, b
B: c, d, e, f

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a	a	a	a	a
						b	b	b	b
		F				F			
	req.	c	d	e	f	c	d	e	f
3 frames for B < |B|		c	c	c	f	f	f	e	e
			d	d	d	c	c	c	f
				e	e	e	d	d	d
		F	F	F	F	F	F	F	F

Nested Loop

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

Sequential flooding
– each request –
page fault

LRU happens to evict exactly the page which we will need next!!!

Sequential Flooding

- Nasty situation caused by LRU policy + repeated sequential scans
  - # buffer frames < # pages in file
  - each page request causes an I/O !!
  - **MRU much better in this situation**

3. Files and Records

What you will learn about in this section

1. File Organization
2. Page Organization
3. BONUS: Column Stores

Managing Disk Space

- The disk space is organized into files
- Files are made up of pages
- Pages contain records

Page or block is OK for I/O, but
higher levels operate on ***records***, and ***files of records***.

File Operations

- The disk space is organized into files
- Files are made up of pages
- Pages contain records

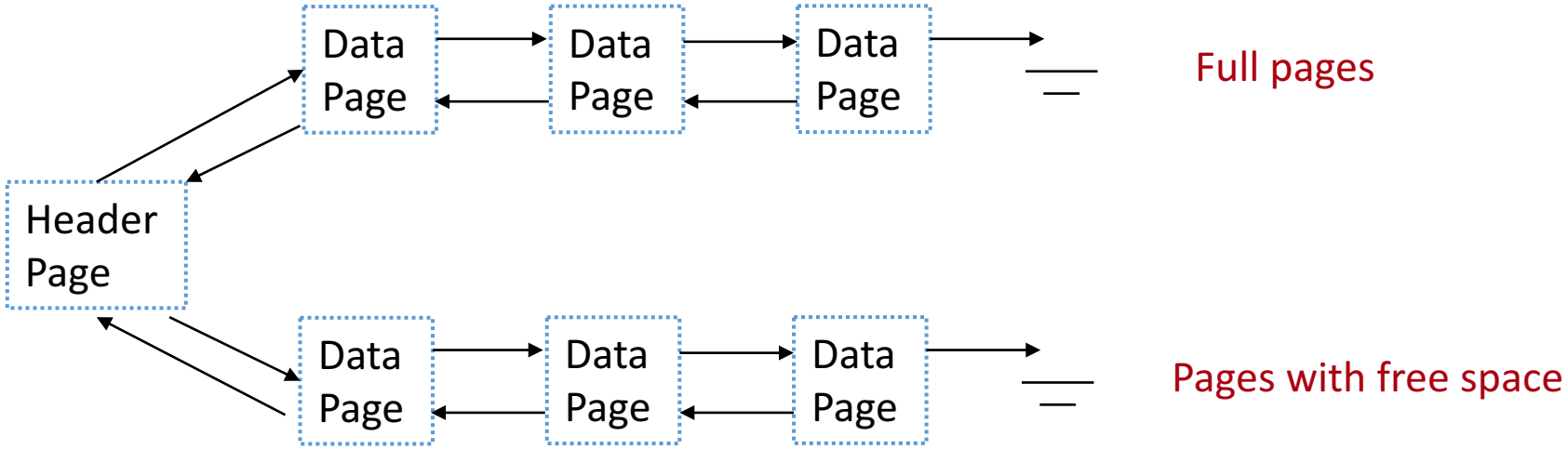
File operations:

- insert/delete/modify record
- read a particular record (specified using the *record id*)
- scan all records (possibly with some conditions on the records to be retrieved)

File Organization: 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: page id (pid)
  - keep track of *free space* on pages
  - keep track of the *records* on a page: record id (rid)
  - Many alternatives for keeping track of this information
- Operations: create/destroy file, insert/delete record, fetch a record with a specified rid, scan all records

Heap File as a List



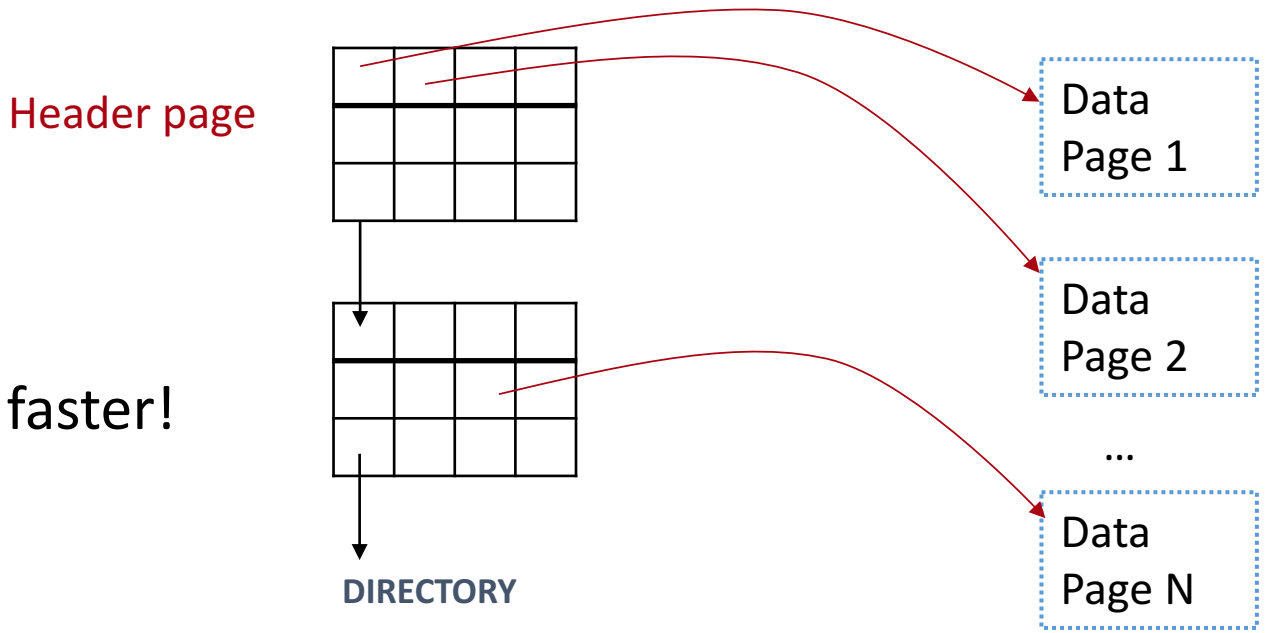
- (heap file name, header page id) recorded in a known location
- Each page contains two pointers plus data: Pointer = Page ID (pid)
- Pages in the free space list have "some" free space

Q: What happens with variable length records?

A: All pages are going to have free space, but maybe we will have to go through a lot of them before we find one with enough space.

Heap File as a Page Directory

- Each entry for a page keeps track of:
  - is the page free or full?
  - how many free bytes are?
- We can now locate pages for new tuples faster!



Managing Disk Space

- Files made up of pages
- and pages contain records
- But file operations are on records:

File operations:

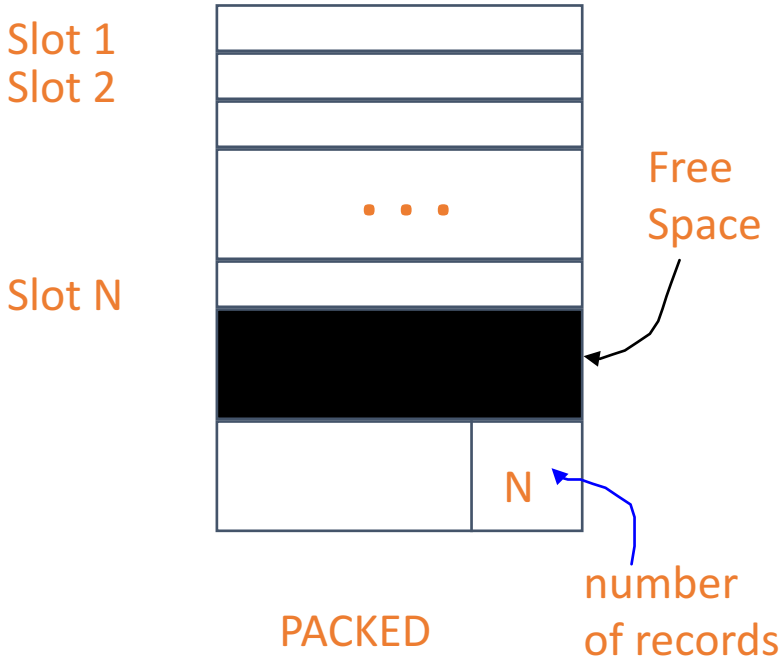
- insert/delete/modify record
- read a particular record (specified using the *record id*)
- scan all records (possibly with some conditions on the records to be retrieved)

Page Organization: Page Formats

- A page is collection of records
- Slotted page format
  - A page is a collection of slots
  - Each slot contains a record
- ***rid = <page id, slot number>***
- There are many slotted page organizations
- We need to have support for:
  - search, insert, delete records on a page

Page Formats: Fixed Length Records

Record id = <page id, slot #>

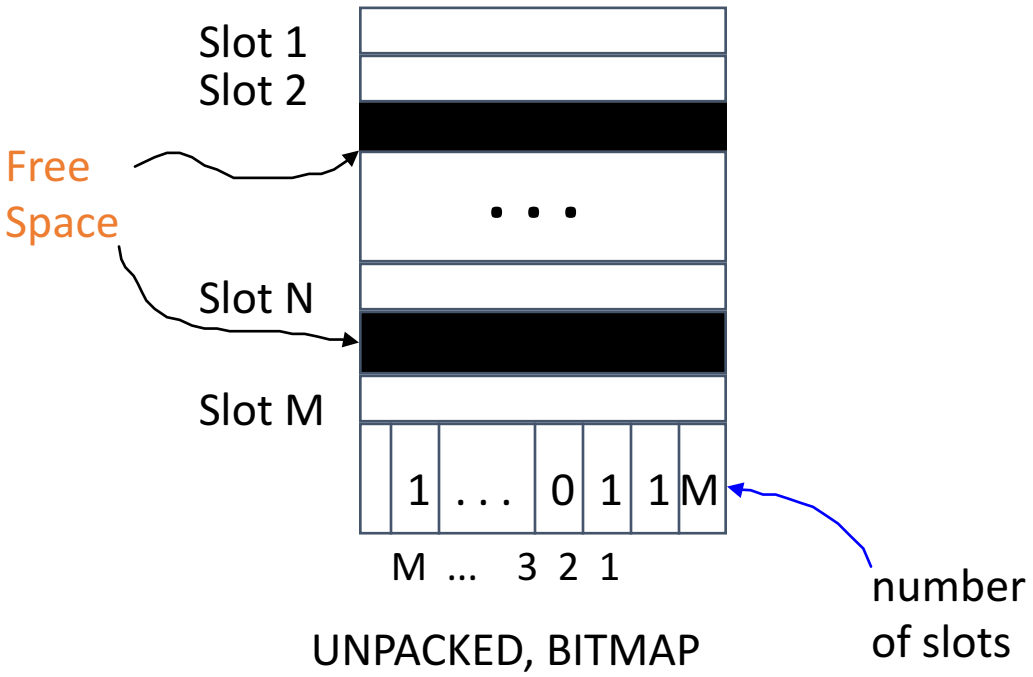


Packed organization: *N* records are always stored in the first *N* slots.

Moving records changes rid!
May not be acceptable.

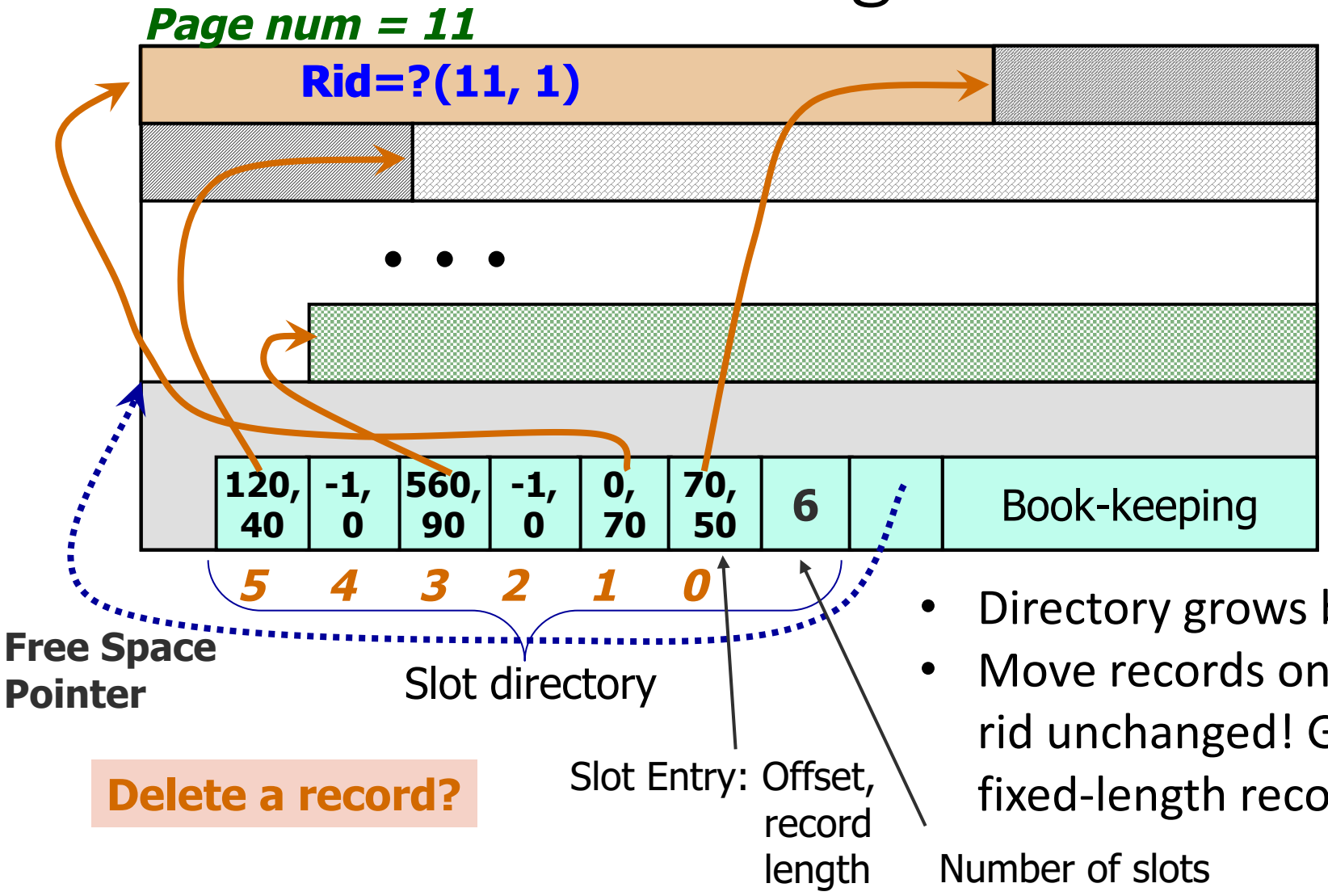
Page Formats: Fixed Length Records

Record id = <page id, slot #>



<u>Unpacked Organization:</u> use a ***bitmap*** to locate records in the page.

Page Formats: Variable Length Records



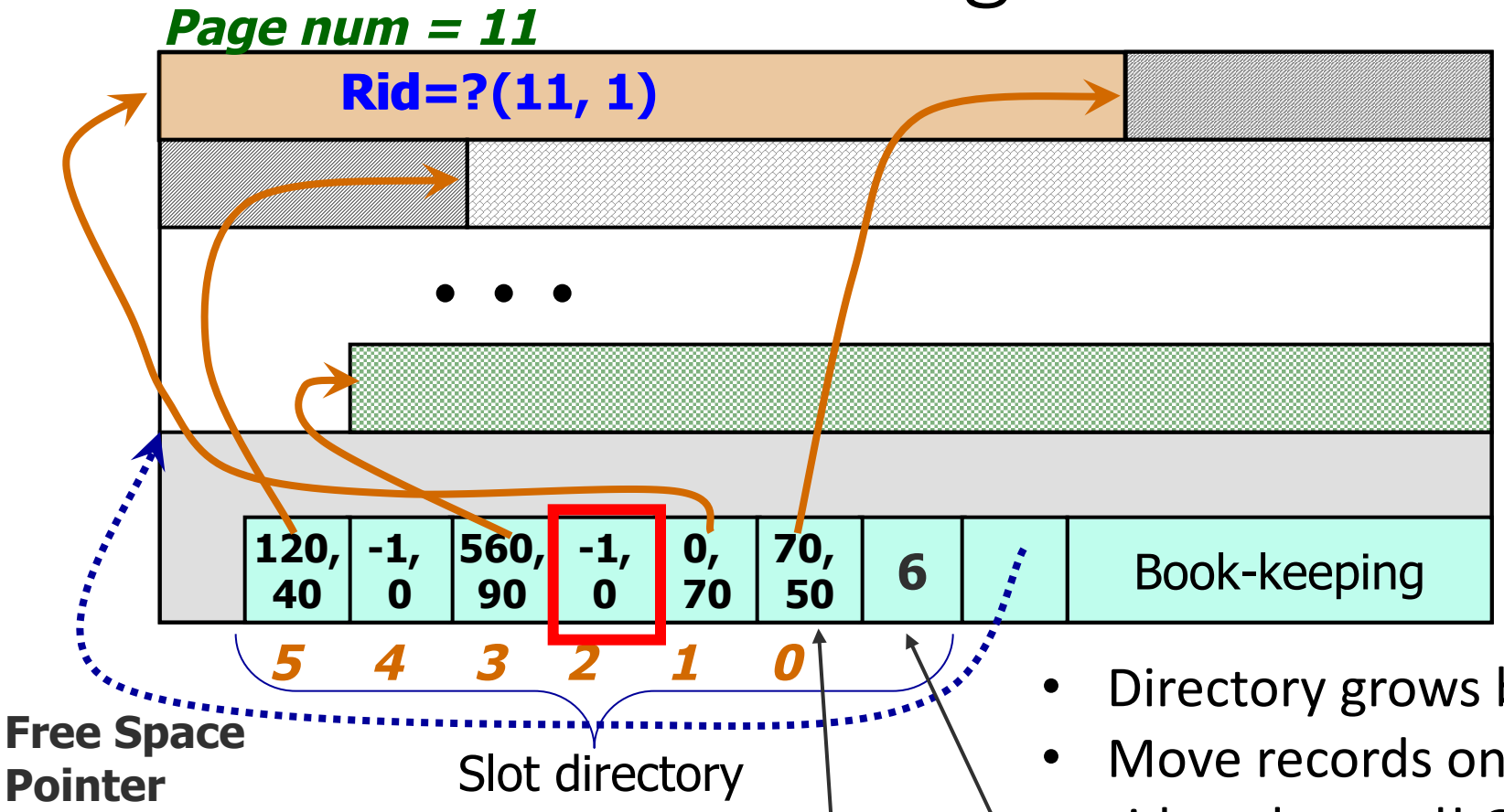
- Directory grows backwards!
- Move records on same page; rid unchanged! Good for fixed-length records too.

Delete a record?

Page Formats: Variable Length Records

- Deletion:
  - offset is set to -1
- Insertion:
  - use any available slot
  - if no space is available, reorganize
- *rid* remains unchanged when we move the record (since it is defined by the slot number)

Page Formats: Variable Length Records



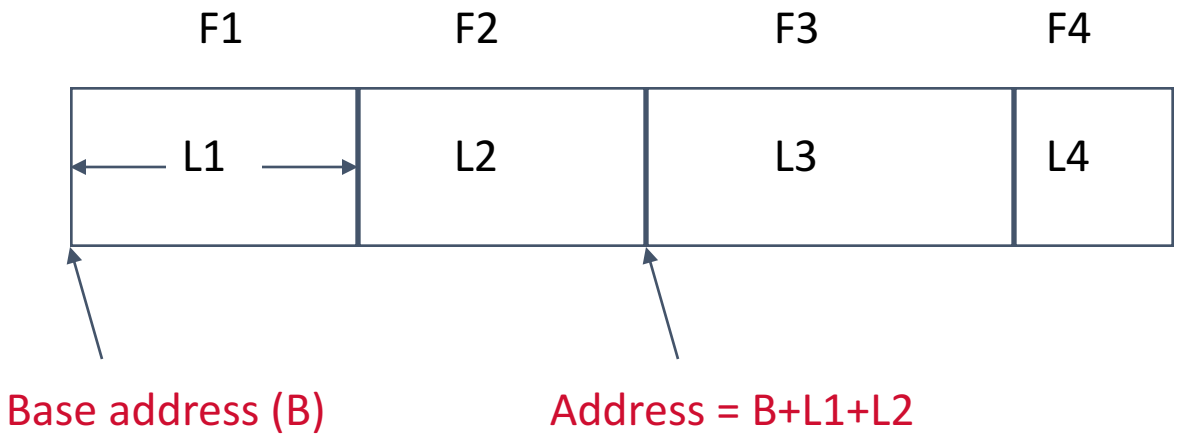
- Directory grows backwards!
- Move records on same page; rid unchanged! Good for fixed-length records too.

Delete a record
Offset is set to -1

Slot Entry: Offset, record length

Number of slots

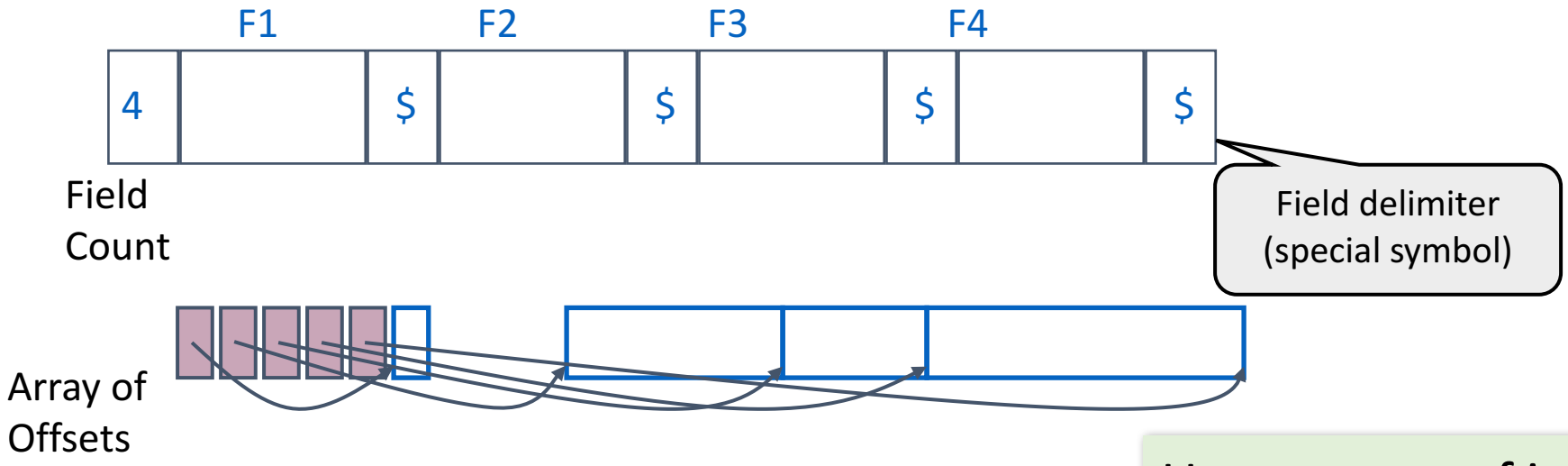
Record Formats: Fixed Length



- All records on the page are the same length
- Information about field types same for all records in a file; stored in *system catalogs.*

Record Formats: Variable Length

Two alternative formats (# fields is fixed):



Use an array of integer offsets in the beginning

- Second alternative offers direct access to i'th field
  - Efficient storage of nulls
  - Small directory overhead.
- Issues with growing records!
  - changes in attribute value, add/drop attributes
- Records larger than pages

Column Stores: Motivation

- Consider a table:
  - Foo (a INTEGER, b INTEGER, c VARCHAR(255), …)
- And the query`:
  - SELECT a FROM Foo WHERE a > 10
- What happens with the previous record format in terms of the bytes that have to be read from the IO subsystem?

Column Stores: Motivation

- Store data "vertically"
- Contrast that with a "row-store" that stores all the attributes of a tuple/record contiguously
  - The previous record formats are "row stores"

111	212	It was a cold morning
222	222	Warm and sunny here
333	232	Artic winter conditions
444	242	Tropical weather

⋮

File 1	File 2	File 3
111	212	It was a cold morning
222	222	Warm and sunny here
333	232	Artic winter conditions
444	242	Tropical weather
⋮	⋮	⋮

Each file is a set of pages.
Columns can be stored in compressed form

Column Stores: Motivation

- Are there any disadvantages associated with column stores?

1. Updates are slower
2. Retrieving back more than one attribute can be slower, e.g. Queries like SELECT * are slower