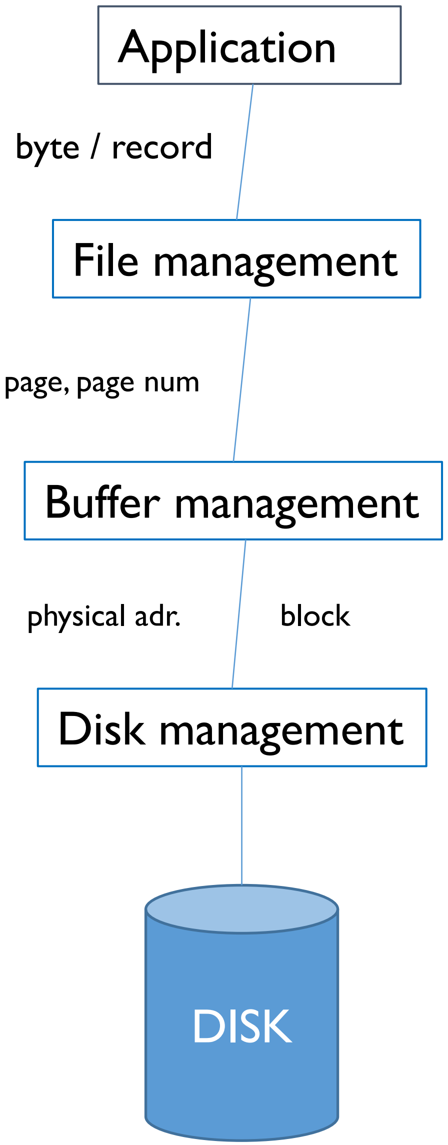
BBM 371 – Data Management

Lecture 3: File Concepts

25.10.2018

Journey of Byte



Request a record/byte (i.e. fscanf(fp, «%d», &a);)

Convert requested byte/record address to block/page address

Decode requested byte/record from coming block/page

Convert logical address to physical address

(#head, #track, #sector) (#cylinder, #head, and #sector)

Manage active pages in the memory

Read/write requested page/block by using physical address.

Disk Space Management

- Lowest layer of DBMS software manages space on disk.

- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page

- Request for a sequence of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done.

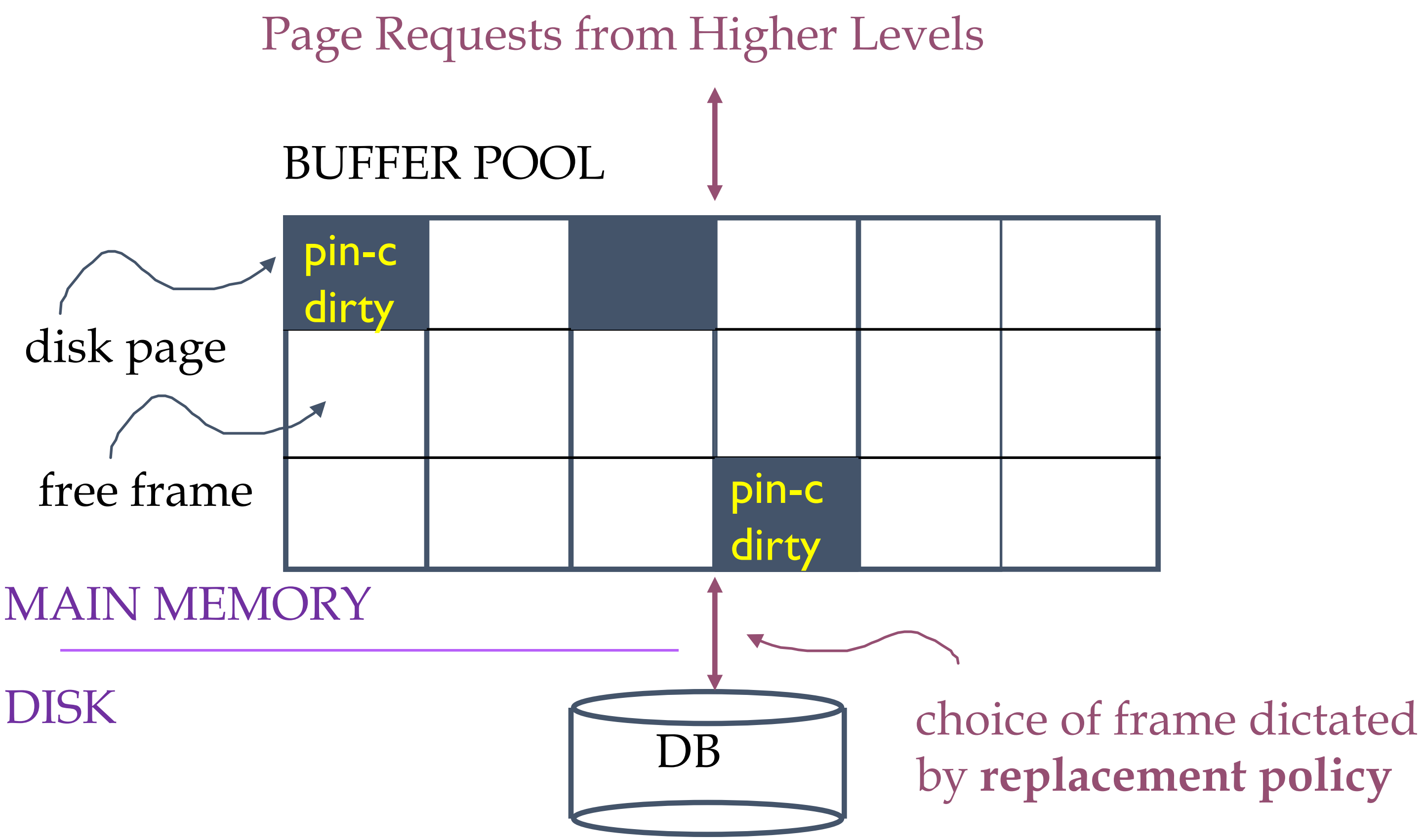
Buffer Management

- All Data Pages must be in memory in order to be accessed

- Buffer Manager
  - Deals with asking Disk Space Manager for pages from disk and store them into memory
  - Sends Disk Space Manager pages to be written to disk

- Memory is faster than Disk
  - Keep as much data as possible in memory
  - If enough space is not available, need a policy to decide what pages to remove from memory. Replacement policy

Buffer Management in a DBMS



- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained.

Buffer Pool

- Frame
  - Data structure that can hold a data page and control flags

- Buffer pool
  - Array of frames of size N

- In C

```
#define POOL_SIZE  100
#define PAGE_SIZE 4096
typedef struct frame {
   int pin_count;
   bool dirty;
   char page[PAGE_SIZE];
} frame;
frame buffer_pool[POOL_SIZE];
```

Operational mode

- All requested data pages must first be placed into the buffer pool.
- `pin_count` is used to keep track of number of transactions that are using the page
  - zero means nobody is using it
- `dirty` is used as a flag (dirty bit) to indicate that a page has been modified since read from disk
  - Need to flush it to disk if the page is to be evicted from pool
- Page is an array of bytes where the actual data is stored in
  - Need to interpret these bytes as int, char, Date data types supported by SQL
    - This is very complex and tricky!

Buffer replacement

- If we need to bring a page from disk, we need to find a frame in the buffer to hold it
- Buffer pool keeps track on the number of frames in use
  - List of frames that are free (Linked list of free frame nums)
- If there is a free frame, we use it
  - Remove from the list of free frames
  - Increment the pin_count
  - Store the data page into the byte array (page field)
- If the buffer is full, we need a policy to decide which page will be evicted

Buffer access & replacement algorithm

- Upon request of page X do
  - Look for page X in buffer pool
  - If found, ++pin_count, then return it
  - else, determine if there is a free frame Y in the pool
  - If frame Y is found
    - Increment its pin_count (++pin_count)
    - Read page from disk into the frame's byte array
    - Return it
  - else, use a replacement policy to find a frame Z to replace
    - Z must have pin_count == 0
  - If dirty bit is set, write data currently in Z to disk
  - Read the new page into the byte array in the frame Z
  - Increment the pin_count in Z (++pin_count)
  - Return it
  - else wait or abort transaction (insufficient resources)

Some remarks

- Need to make sure pin_count is 0
    - Nobody is using the frame
- Need to write the data to disk if dirty bit is true
- This latter approach is called Lazy update
    - Write to disk only when you have to!!!
    - Careful, if power fails, you are in trouble.
    - DBMS need to periodically flush pages to disk
        - Force write
- If no page is found with pin_count equal to 0, then either:
    - Wait until one is freed
    - Abort the transaction (insufficient resources)

Buffer Replacement policies

- LRU – Least Recently Used
  - Evicts the page that is the least recently used page in the pool.
  - Can be implemented by having a queue with the frame numbers.
  - Head of the queue is the LRU
  - Each time a page is used it must be removed from current queue position and put back at the end
    - This queue need a method erase() that can erase stuff from the middle of the queue
- LRU is the most widely used policy for buffer replacement
  - Most cache managers also use it

Other policies

- Most Recently Used
  - Evicts the page that was most recently accessed
  - Can be implemented with a priority queue
- FIFO
  - Pages are replaced in a strict First-In-First Out
  - Can be implemented with a FIFO List (queue in the strict sense)
- Random
  - Pick any page at random for replacement

Sample Buffer Pool

Page_no = 1 Pin_count = 3 Dirty = 1 Last Used: 12:34:05	Page_no = 2 Pin_count = 0 Dirty = 1 Last Used: 12:35:05	Page_no = 3 Pin_count = 1 Dirty = 0 Last Used: 12:36:05	Page_no = 4 Pin_count = 2 Dirty = 0 Last Used: 12:37:05	Page_no = 5 Pin_count = 0 Dirty = 0 Last Used: 12:38:05
Page_no = 6 Pin_count = 0 Dirty = 0 Last Used: 12:29:05	Page_no = 7 Pin_count = 1 Dirty = 1 Last Used: 12:20:05	Page_no = 8 Pin_count = 0 Dirty = 1 Last Used: 12:40:05	Page_no = 9 Pin_count = 2 Dirty = 0 Last Used: 12:27:05	Page_no = 10 Pin_count = 0 Dirty = 1 Last Used: 12:39:05

Which page should be removed if LRU is used as the policy:…………………………...........

Which page should be removed if MRU is used as the policy :…………………………………

Which pages do not need to be written to disc, if it is removed:…………………...………..

Which pages could not be removed in this situation:…………………………………………...

DBMS vs. OS File System

- OS does disk space & buffer management: why not let OS manage these tasks?

- Some limitations, e.g., files can't span disks.

- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk,
  - adjust replacement policy, and pre-fetch pages based on access patterns in typical DB operations.

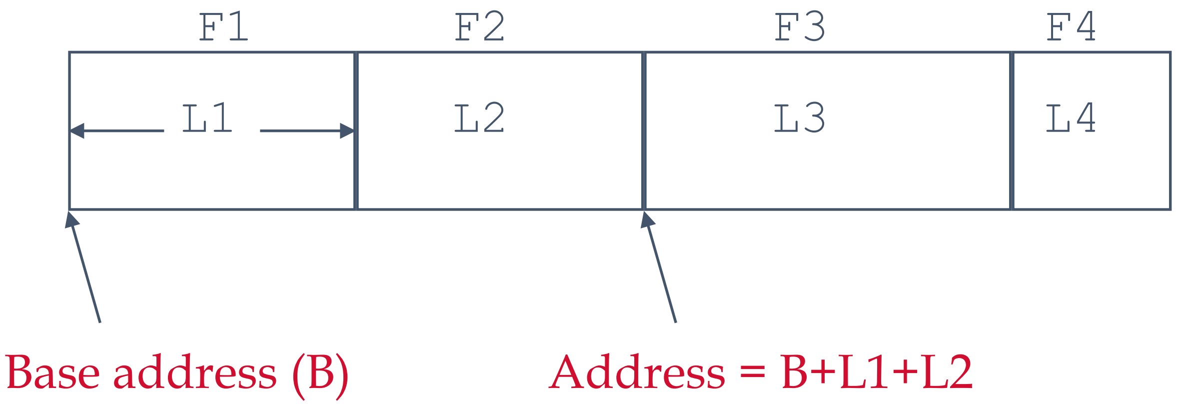
Record Formats

- Organization of records whether field length of record
  - Fixed
  - Variable

Note:Type and number of fields are identical for all tuples

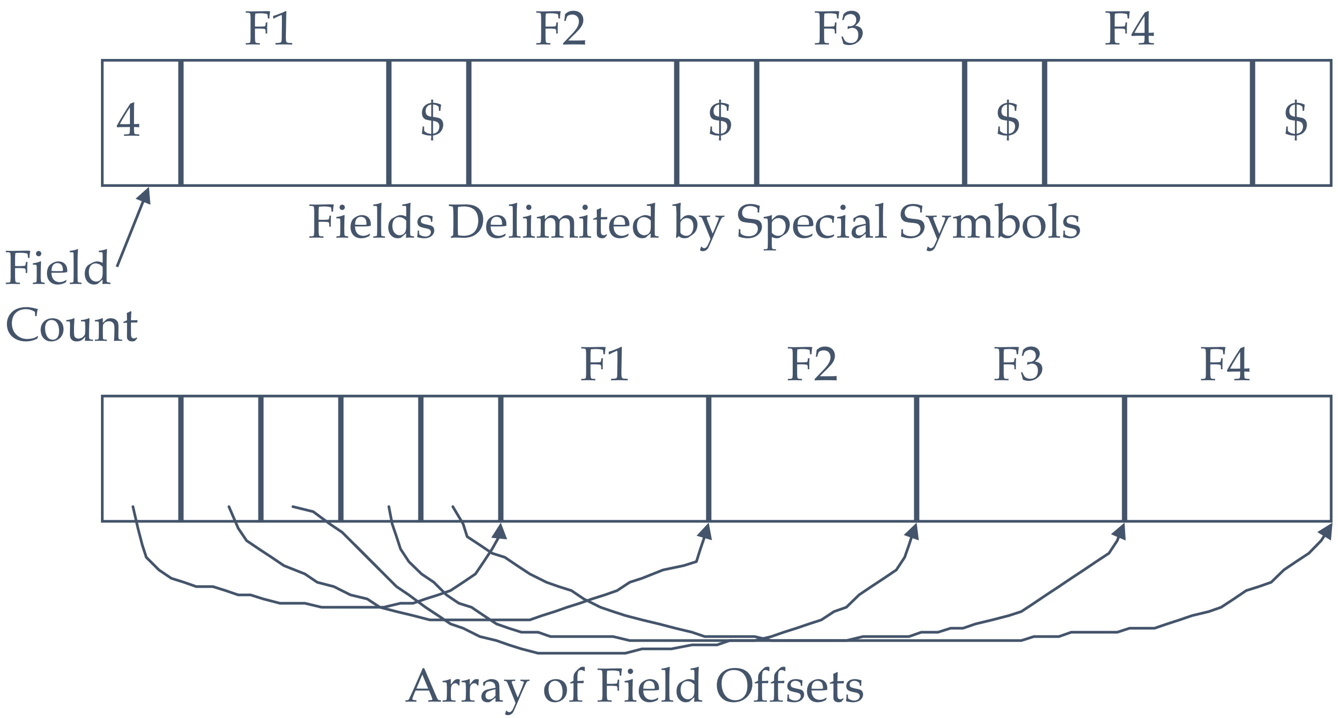
Fixed Length Records

- All fields can be placed continuous
- Finding i^{th} field address requires adding length of previous fields to base address.



Variable Length Records

- Two alternative formats (# fields is fixed):



☛

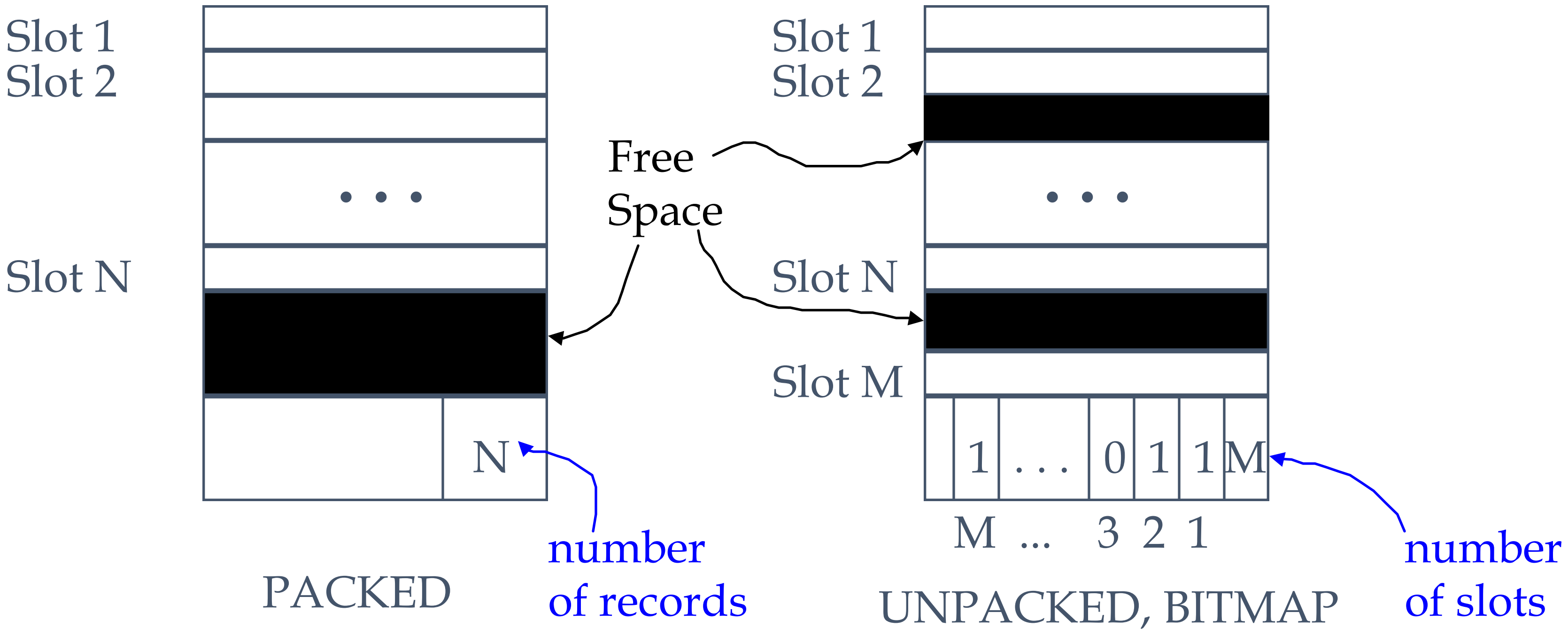
Variable Length Records(Cont.)

- In first
  - All previous fields must be scanned to access the desired records
- In Second
  - Second offers direct access to i^{th} field
  - Pointers to begin and end of the field
  - Efficient storage for nulls
  - Small directory overhead

Disadvantage of Variable Length

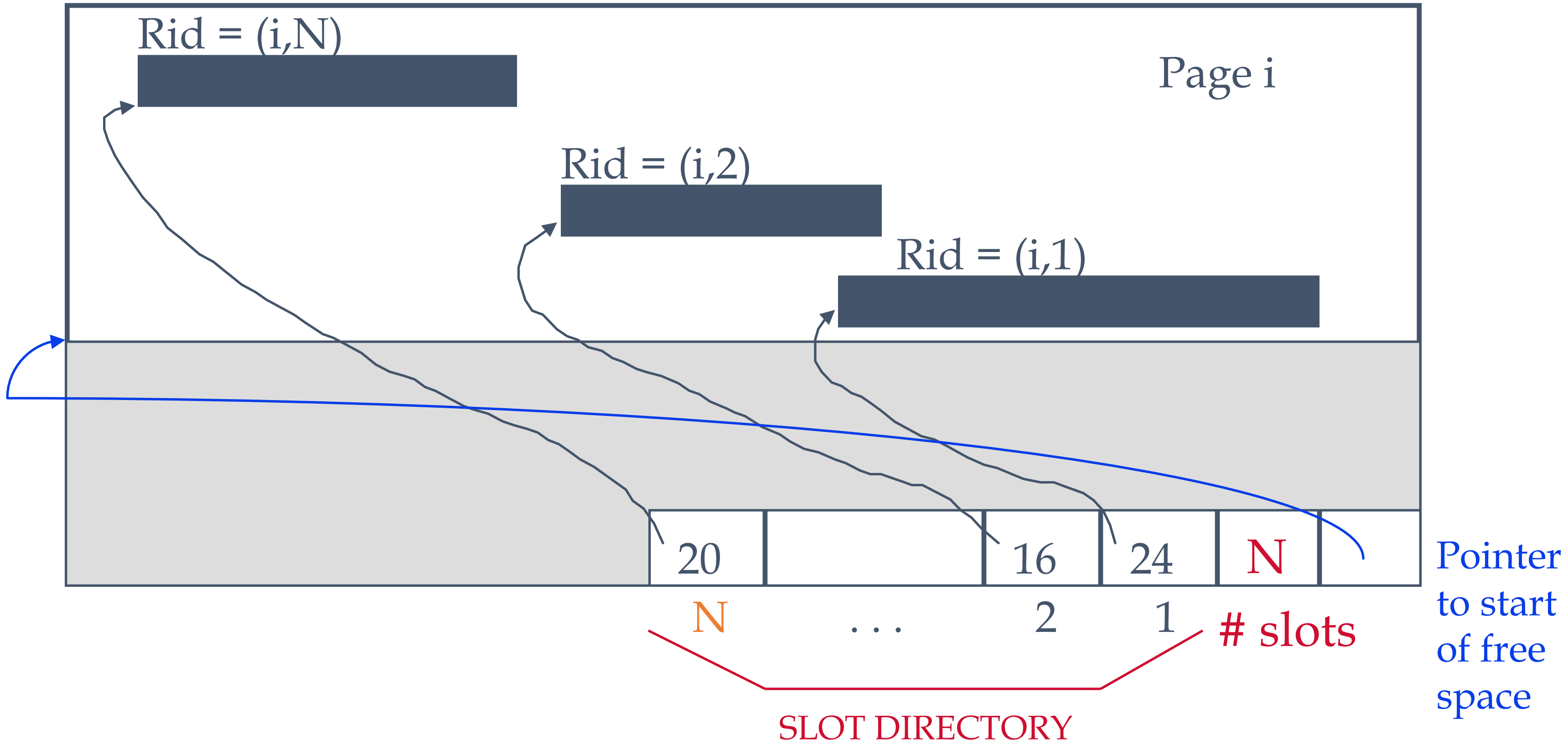
- If field grows to larger size:
  - Subsequent fields must be shifted
  - Offsets must be updated
- If after update, record does not fit in its current page:
  - memory address of the page is changed
  - references to old address must be updated
- If record does not fit in any page:
  - Record must be broken down to smaller records
  - Chaining must be set up for the smaller records

Page Formats: Fixed Length Records



- In first alternative, moving records for free space management changes memory address of record ; may not be acceptable.

Page Formats: Variable Length Records



- Can move records on page without changing memory address of records; so, attractive for fixed-length records too.

Page Formats: Variable Length Records

- Keep a directory for slots that show <record offset, record length>
- Keep a pointer to point free space
- For placement of a record
  - If it is possible, insert in free space
  - Reorganize page to combine wasted space then insert
  - Insert another page
- For deleting a record
  - Put –1 to record offset information in directory

Files of Records

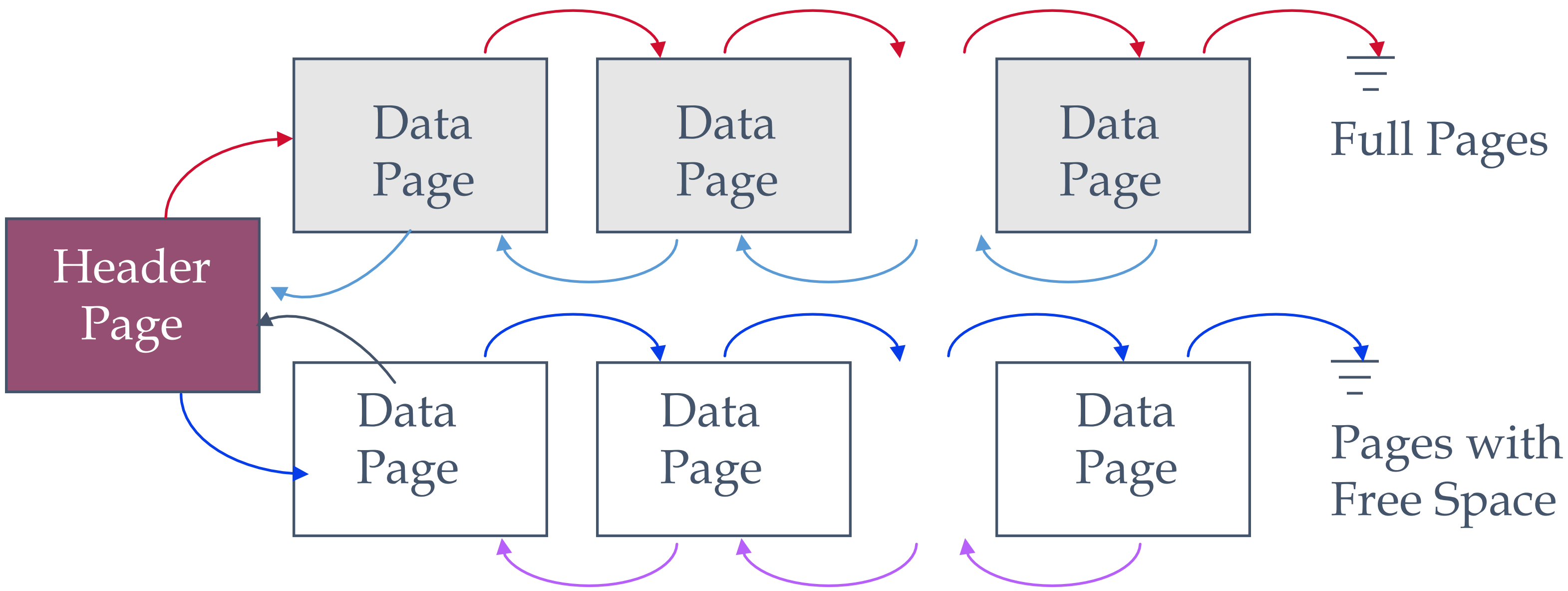
- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records.

- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

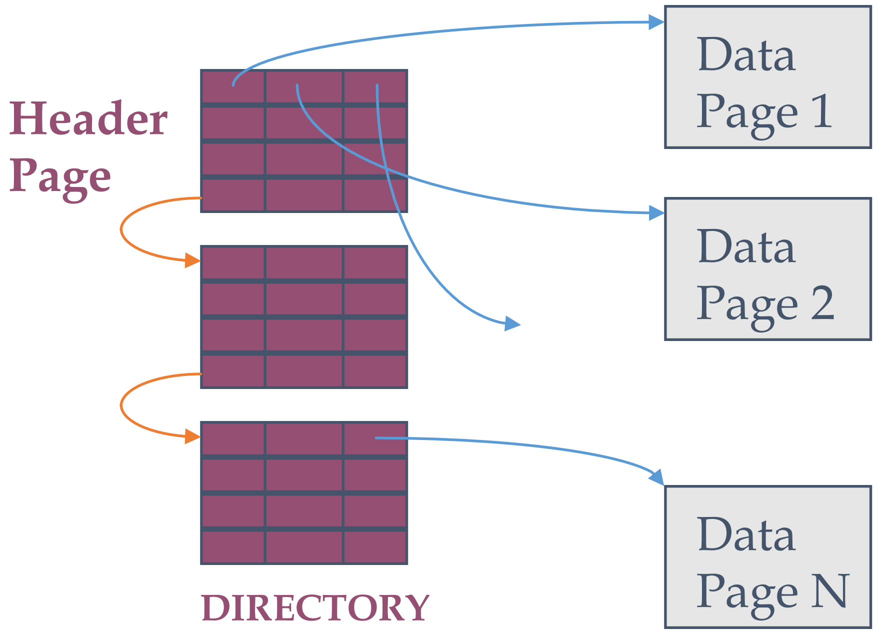
- Synoym of «Pile» and «Sequential»
- Simplest file structure as records are in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the pages in a file
  - keep track of free space on pages
  - keep track of the records on a page
- There are many alternatives for keeping track of this.

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace on disk.
- Each page contains two `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

Searching on Heap Files

- Equality search: to search a record with given value of one or more of its fields
- Range search: to find all records which satisfy given min and max values for one of fields

- We must search the whole file.
- In general, (bf is blocking factor. N is the size of the file in terms of the number of records) :
  - At least 1 block is accessed (I/O cost : 1)
  - At most N/bf blocks are accessed.
  - On average $N/2bf$

- Thus, time to find and read a record in a file is approximately :

$$\text{Time to fetch one record} = (N/2bf) * \text{time to read one block}$$

Time to read one block = seek time + rotational delay + block transfer time

More and more ...

- Time to read all records = `N/bf * time to read per block`

- Time to add new record
  - = time to read one block (for last block) + Time to write one block (for last block)
  - if the last block is full
    - = time to read one block (for last block) + time to write new one block (for new last block)

More and more ...

- Time to update one fixed length record = Time to fetch one record + time to write one block

- Time to update one variable length record = Time to delete one record + time to add new record

- Time to delete one record = ??

  You can mark the record (replace the first character with $)

Exercise

- FileA: 10000 records , BF = 100, 4 extents
- File B: 5000 records, BF = 150, 3 extents

- Time to find the number of common records of FileA and B

  Time to read FileA= 4 * (seek time + rotational delay) + (10000/100) * block transfer time

  Time to read FileB = 3 * (seek time + rotational delay) + (5000/150) * block transfer time

  = Time to read FileA + 100 * Time to read FileB

  (imagine you've got only two frames in the buffer pool.)
- Read FileA and compare each record of FileA with whole records in FileB

Sorted (Sequential) Files

- A sorted file should stay in order, but it is impossible.
  - Additions/deletions

- A sorted file uses an overflow pages list for newly added records
  - Overflow pages list does not have an ordering

- For equality search:
  - Search on sorted area
  - And then search on overflow area

***If there are too many overflow areas, the access time increase up to that of a sequential file.

Searching for a record

- We can do binary search (assuming fixed-length records) in the sorted part.

m records	k records	
Sorted part	overflow	(m + k = N)

- Worst case to fetch a record :

$$T_F = \log_2 (m/bf) \ * \text{ time to read per block.}$$

- If the record is not found, search the overflow area too. Thus total time is:

$$T_F = \log_2 (m/bf) \ * \text{ time to read per block } + k/bf * \text{ time to read per block}$$