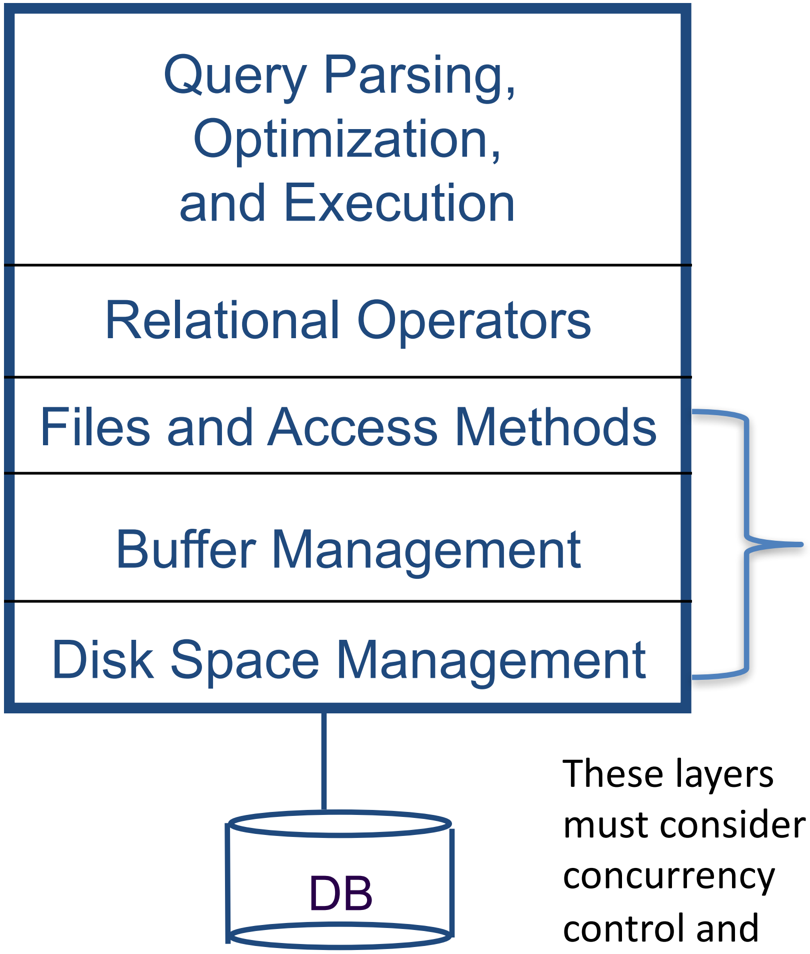
Database Storage & File System

Easy Database Internals Explaination

DBMS Architecture

- A typical DBMS has a layered architecture
- The figure does not show the concurrency control and recovery components
  – to be done in “transactions”
- This is one of several possible architectures
  – each system has its own variations



These layers must consider concurrency control and recovery

Data on External Storage

- Data must persist on disk across program executions in a DBMS
  - Data is huge
  - Must persist across executions
  - But has to be fetched into main memory when DBMS processes the data

- The unit of information for reading data from disk, or writing data to disk, is a page

- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order

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
- Size of a page = size of a disk block

  = data unit
- Request for a sequence of pages often satisfied by allocating contiguous blocks on disk
- Space on disk managed by Disk-space Manager
  - Higher levels don’t need to know how this is done, or how free space is managed

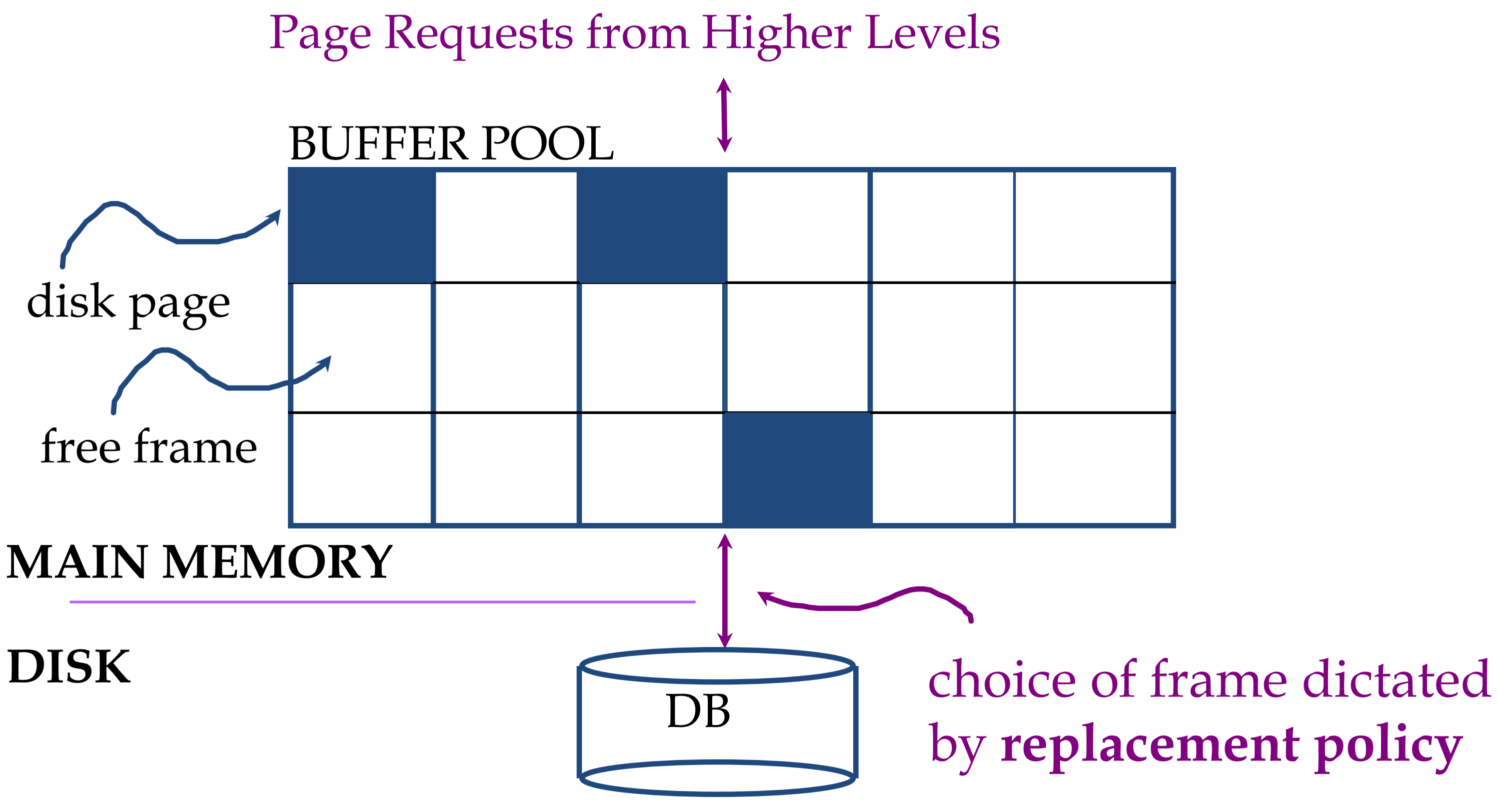
Buffer Management

Suppose

- 1 million pages in db, but only space for 1000 in memory
- A query needs to scan the entire file
- DBMS has to
  - bring pages into main memory
  - decide which existing pages to replace to make room for a new page
  - called Replacement Policy
- Managed by the Buffer manager
  - Files and access methods ask the buffer manager to access a page mentioning the “record id” (soon)
  - Buffer manager loads the page if not already there

Buffer Management

Buffer pool = main memory is partitioned into frames
either contains a page from disk or is a free frame



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

When a Page is Requested ...

For every frame, store

- a dirty bit:
  - whether the page has been modified since it has been brought to memory
  - initially 0 or off

- a pin-count:
  - the number of times a page has been requested but not released (and no. of current users)
  - initially 0
  - when a page is requested, the count in incremented
  - when the requestor releases the page, count is decremented
  - buffer manager only reads a page into a frame when its pin-count is 0
  - if no page with pin-count 0, buffer manager has to wait (or a transaction is aborted -- later)

When a Page is Requested …

- Check if the page is already in the buffer pool
- if yes, increment the pin-count of that frame
- If no,
  - Choose a frame for replacement using the replacement policy
  - If the chosen frame is dirty (has been modified), write it to disk
  - Read requested page into chosen frame
- Pin (increase pin-count of) the page and return its address to the requestor

- If requests can be predicted (e.g., sequential scans), pages can be pre-fetched several pages at a time
- Concurrency Control & recovery may entail additional I/O when a frame is chosen for replacement
  - e.g. Write-Ahead Log protocol : when we do Transactions

Buffer Replacement Policy

- Frame is chosen for replacement by a replacement policy

- Least-recently-used (LRU)
  - add frames with pin-count 0 to the end of a queue
  - choose from head

- Clock
  - an efficient implementation of LRU
  - Assign 1 to N (=#frames) to frames
  - choose next frame with pin-count 0

- First In First Out (FIFO)
- Most-Recently-Used (MRU) etc.

Buffer Replacement Policy

- Policy can have big impact on # of I/O's
- Depends on the access pattern
- Sequential flooding: Nasty situation caused by LRU + repeated sequential scans
  - What happens with 10 frames and 9 pages?
  - What happens with 10 frames and 11 pages?
  - # buffer frames < # pages in file means each page request in each scan causes an I/O
  - MRU much better in this situation (but not in all situations, of course)

DBMS vs. OS File System

- Operating Systems do disk space and buffer management too:
- Why not let OS manage these tasks?

- DBMS can predict the page reference patterns much more accurately
  - can optimize
  - adjust replacement policy
  - pre-fetch pages – already in buffer + contiguous allocation
  - pin a page in buffer pool, force a page to disk (important for implementing Transactions concurrency control & recovery)

- Differences in OS support: portability issues

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

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 (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)

File Organization

- File organization: Method of arranging a file of records on external storage
  - One file can have multiple pages
  - Record id (rid) is sufficient to physically locate the page containing the record on disk
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields

- NOTE: Several uses of "keys" in a database
  - Primary/foreign/candidate/super keys
  - Index search keys

Alternative File Organizations

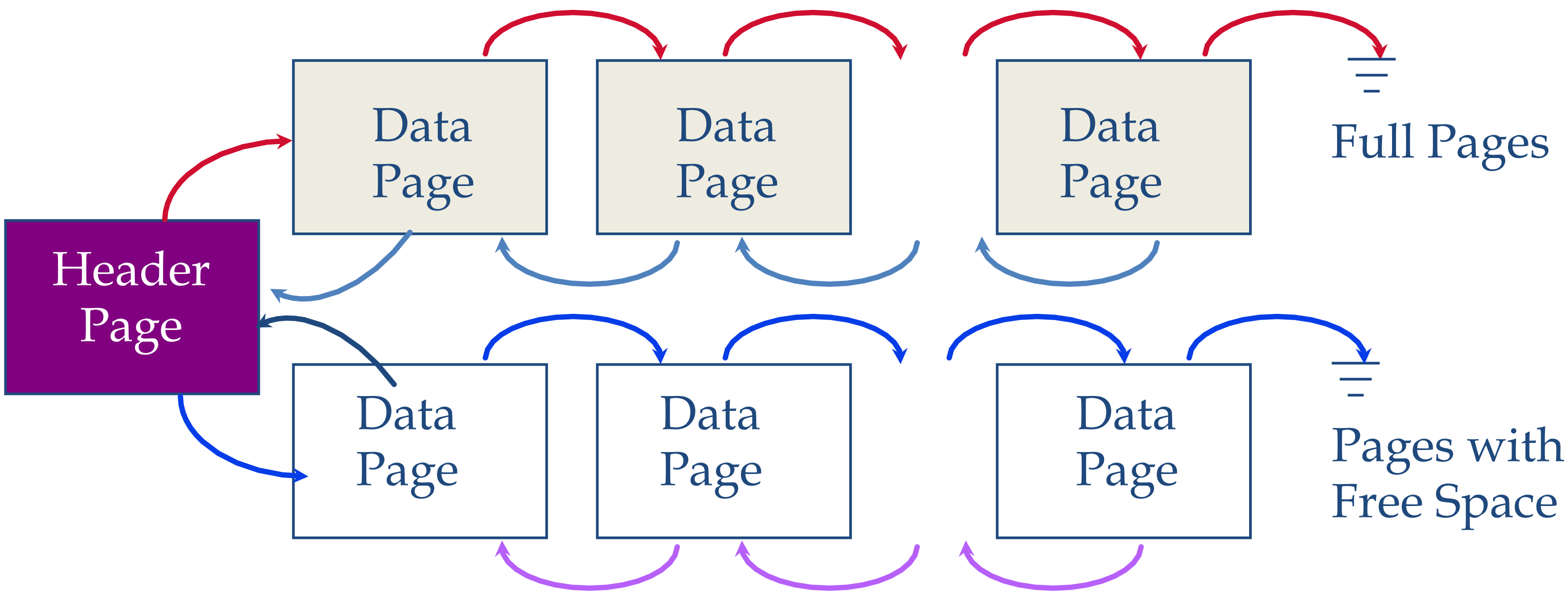
Many alternatives exist, each ideal for some situations, and not so good in others:

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records
- Sorted Files: Best if records must be retrieved in some order, or only a "range" of records is needed.
- Indexes: Data structures to organize records via trees or hashing
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files

Unordered (Heap) Files

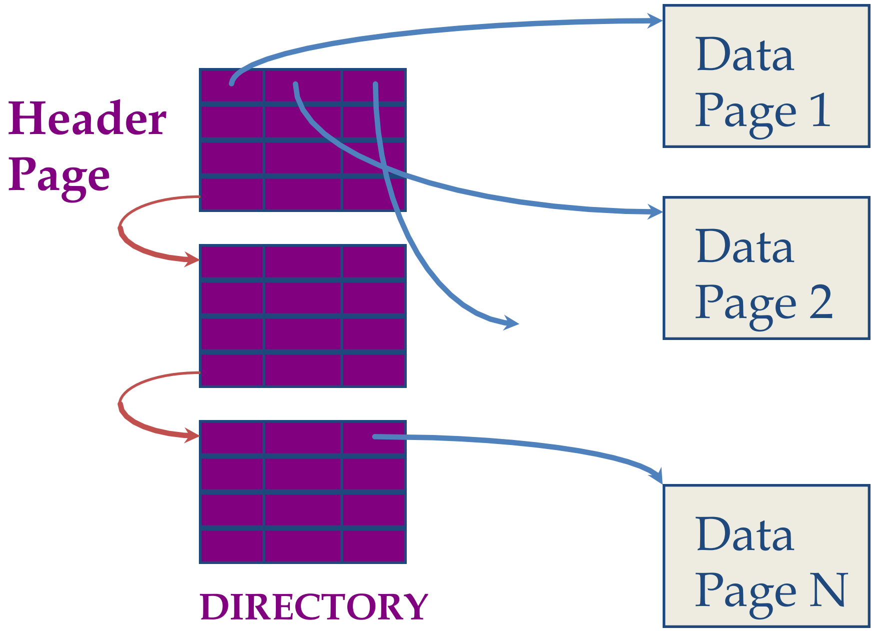
- Simplest file structure contains records in no particular order
- As file grows and shrinks, disk pages are allocated and de-allocated
- To support record level operations, we must:
  - keep track of the pages in a file
  - keep track of free space on pages
  - keep track of the *records* on a page
- 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
- Each page contains 2 `pointers' plus data
- Problem:
  - to insert a new record, we may need to scan several pages on the free list to find one with sufficient space

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 of directory is just one alternative
  - Much smaller than linked list of all heap file pages!

How do we arrange a collection of records on a page?

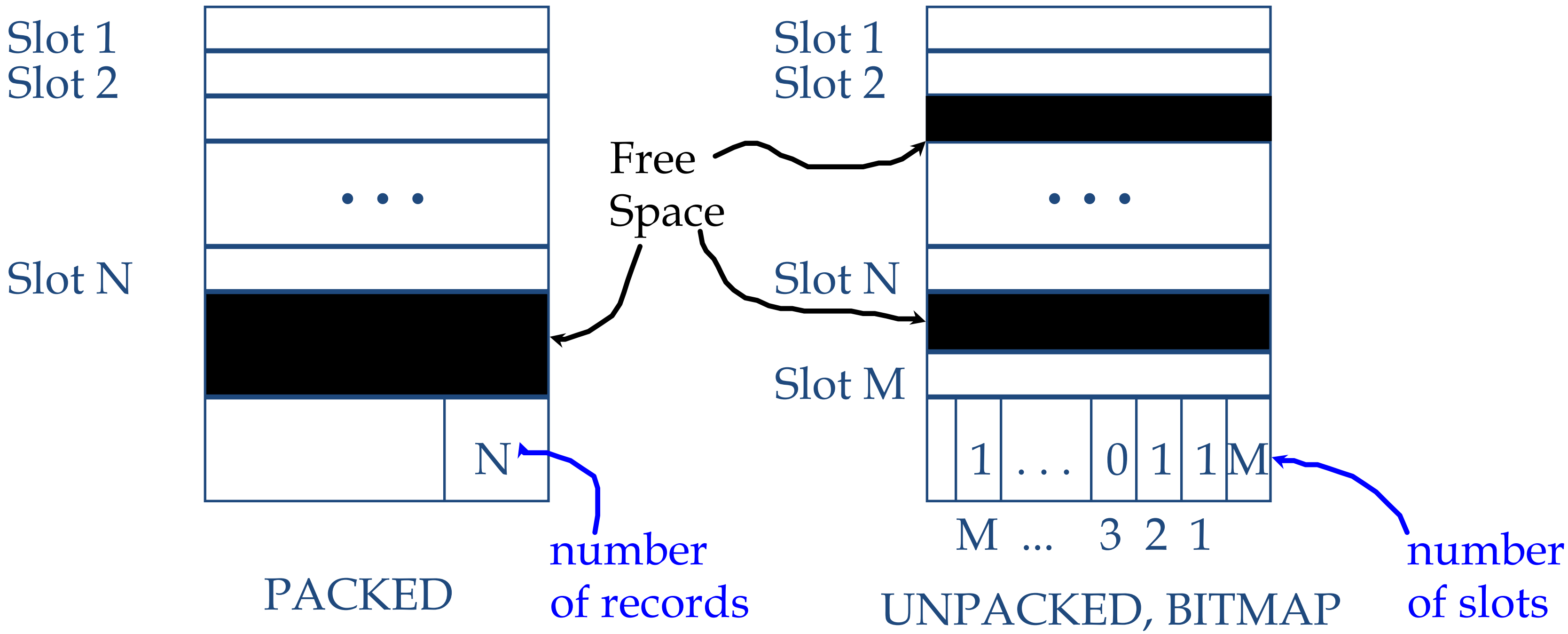
- Each page contains several slots
  - one for each record

- Record is identified by <page-id, slot-number>

- Fixed-Length Records
- Variable-Length Records

- For both, there are options for
  - Record formats (how to organize the fields within a record)
  - Page formats (how to organize the records within a page)

Page Formats: Fixed Length Records



- Record id = <page id, slot #>
- Packed: moving records for free space management changes rid; may not be acceptable
- Unpacked: use a bitmap – scan the bit array to find an empty slot
- Each page also may contain additional info like the id of the next page (not shown)

Page Formats: Variable Length Records

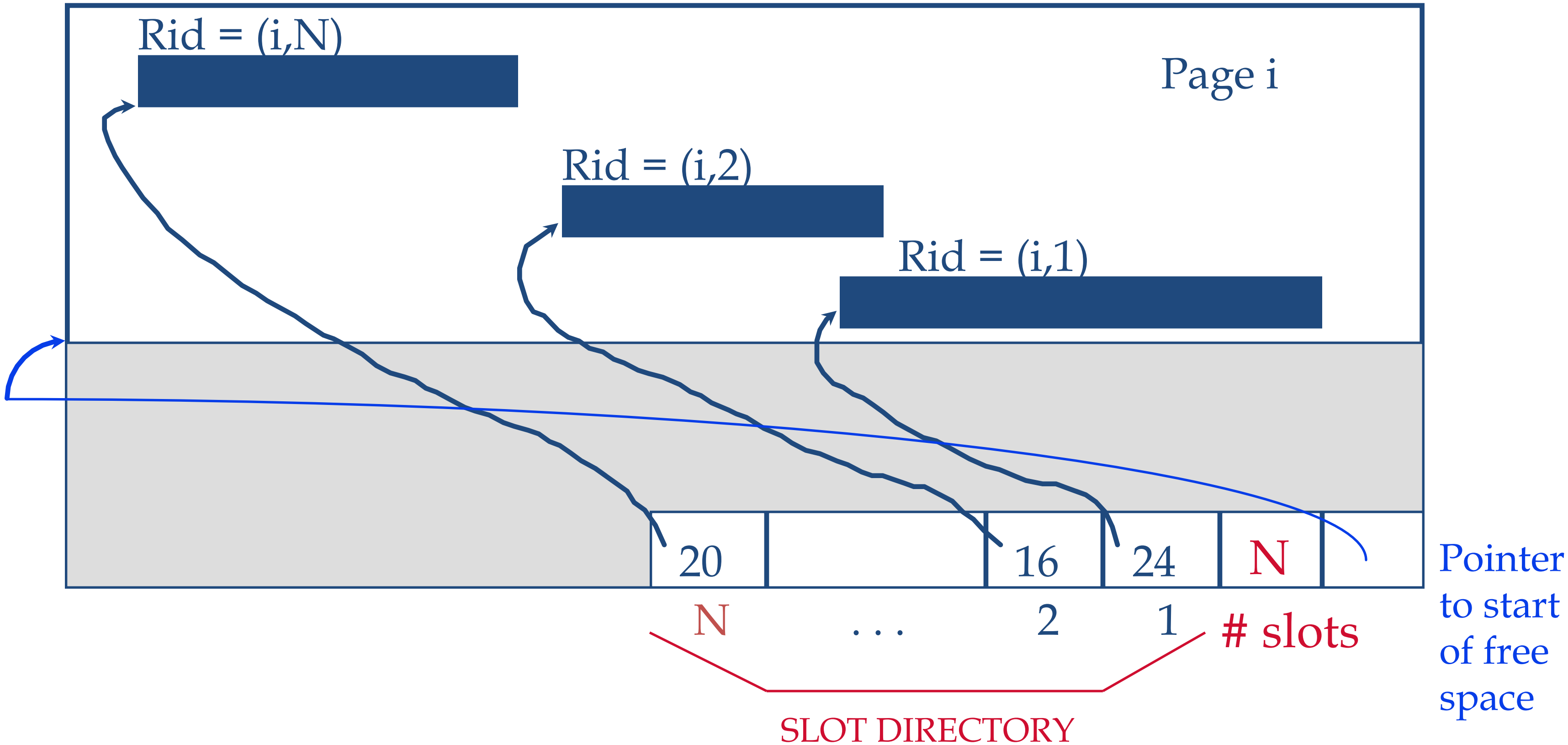
- Need to find a page with the right amount of space
  - Too small – cannot insert
  - Too large – waste of space

- if a record is deleted, need to move the records so that all free space is contiguous
  - need ability to move records within a page

- Can maintain a directory of slots (next slide)
  - <record-offset, record-length>
  - deletion = set record-offset to -1

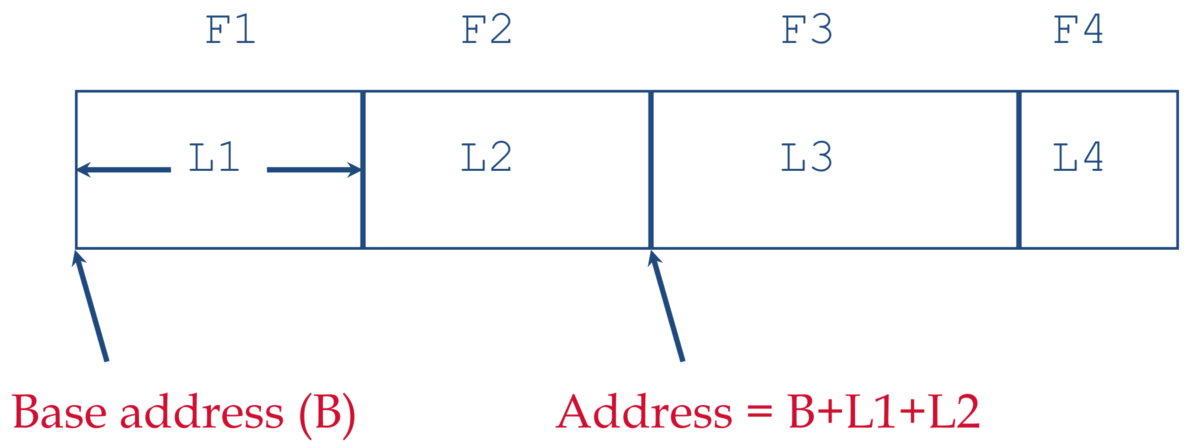
- Record-id rid = <page, slot-in-directory> remains unchanged

Page Formats: Variable Length Records



- Can move records on page without changing rid
  - so, attractive for fixed-length records too
- Store (record-offset, record-length) in each slot
- rid-s unaffected by rearranging records in a page

Record Formats: Fixed Length

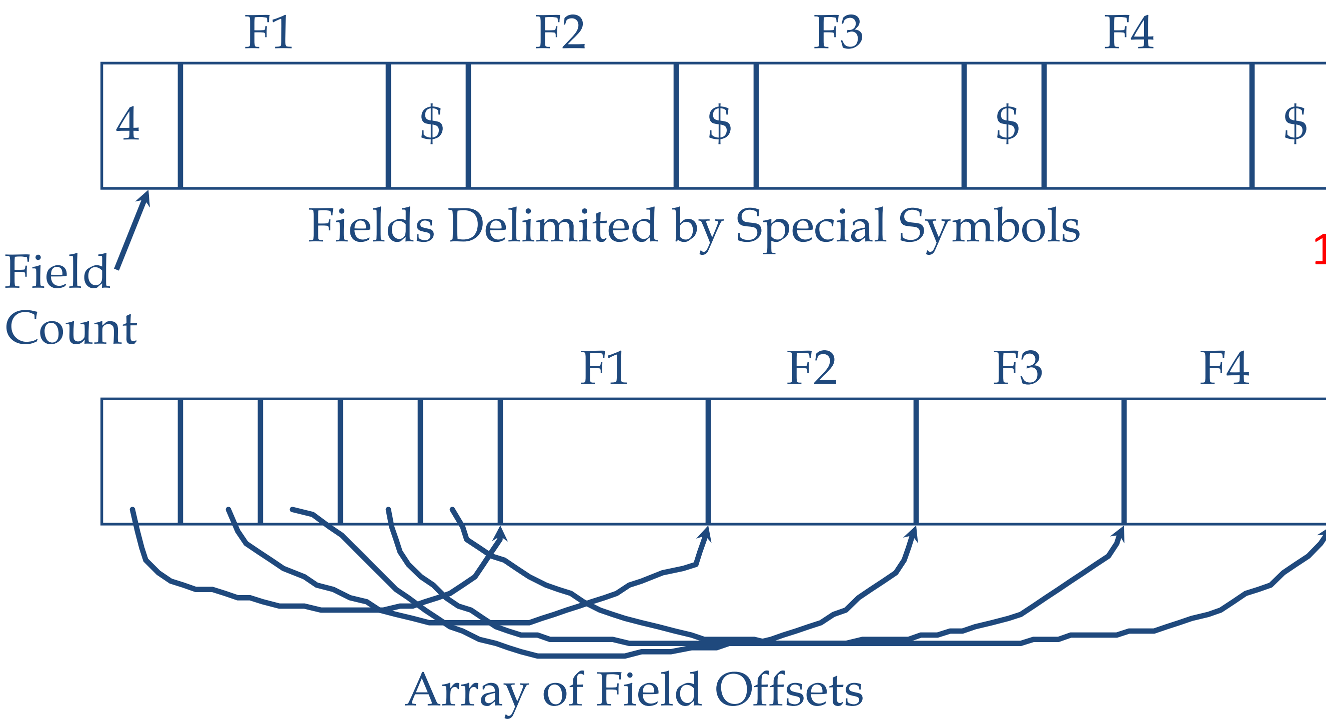


- Each field has a fixed length
  - for all records
  - the number of fields is also fixed
  - fields can be stored consecutively
- Information about field types same for all records in a file
  - stored in system catalogs
- Finding i-th field does not require scan of record
  - given the address of the record, address of a field can be obtained easily

Lecture 5 ends here

Record Formats: Variable Length

- Cannot use fixed-length slots for records
- Two alternative formats (# fields is fixed):



1. use delimiters

2. use offsets at the start of each record

- Second offers direct access to i-th field, efficient storage of nulls (special don't know value); small directory overhead
- Modification may be costly (may grow the field and not fit in the page)