Overview of Storage and Indexing
Storing Data: Disks and Files

Chapters 8-9

Buffer Replacement Policy

- A frame is chosen for replacement by a *replacement policy.*

- *Least Recently Used (LRU):*
  - Can be implemented using a queue of pointers for frames with zero pin count
  - A frame is added to end of the queue when it becomes candidate for replacement (i.e., pin count becomes zero)
  - The page chosen for replacement is the one in the frame at the head of the queue.

Buffer Replacement Policy

- LRU + Clock:
  - A variable, named *current*, is set from 1 to N (no. buffers)
  - The *current* frame is considered for replacement, if it does not qualify, the *current* is incremented
  - Each frame has a reference bit, initially set to 0, turned to 1 once the *pin count* becomes 0
  - If the *current* frame has reference bit 1, the clock algorithm turns the bit to 0 and increments *current*
    - This way, a recently referenced page is less likely to be replaced
  - If the *current* frame has pin count 0 and reference count 0, it is chosen for replacement

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.
  - # buffer frames < # pages in file means each page request causes an I/O.

- Other replacement techniques can be used, e.g., Random, FIFO, and MRU (Most recently used)

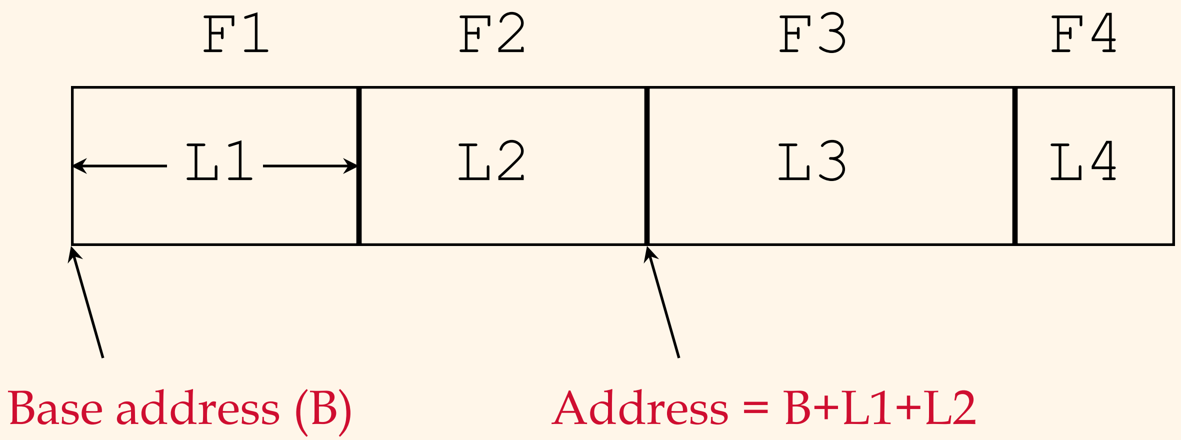
DBMS vs. OS File System

OS does disk space & buffer management:

Why not let OS manage these tasks?

- Differences in OS support: portability issues
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.

Record Formats: Fixed Length



- 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.

System Catalogs

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

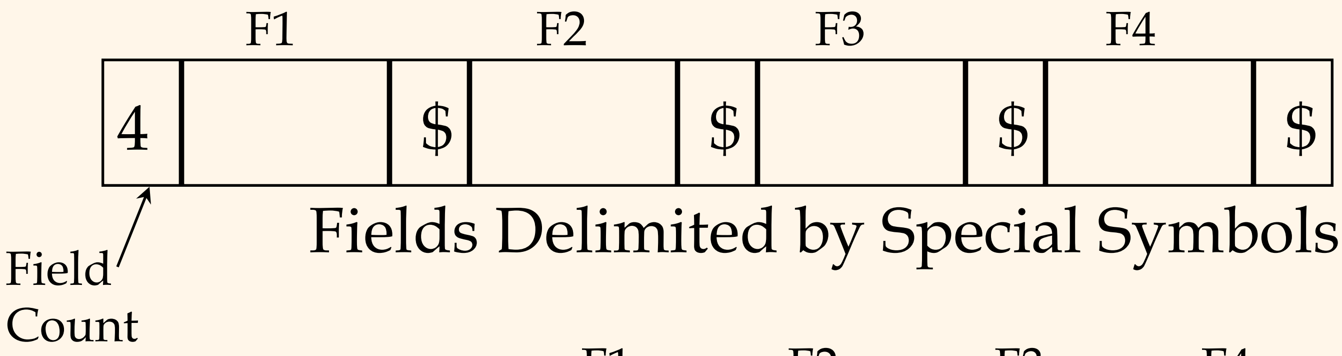
  ☛ *Catalogs are themselves stored as relations*!

Attr_Cat(attr_name, rel_name, type, position)

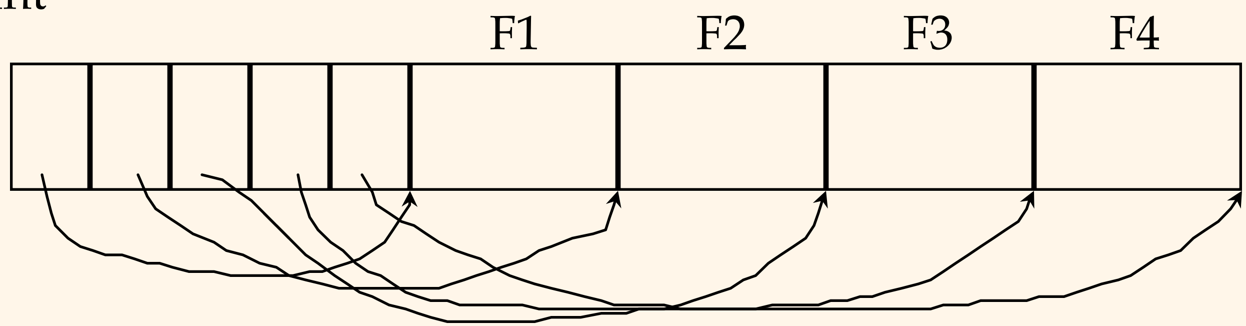
attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



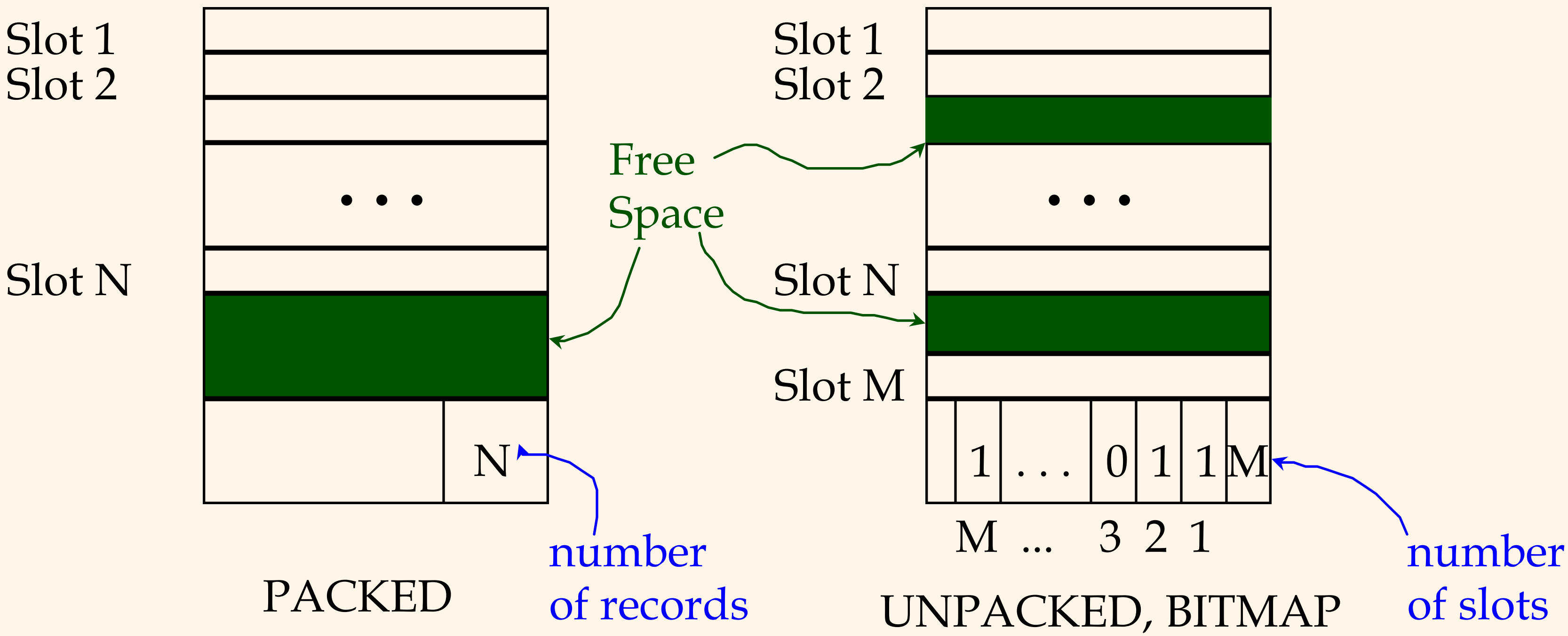
Fields Delimited by Special Symbols



Array of Field Offsets

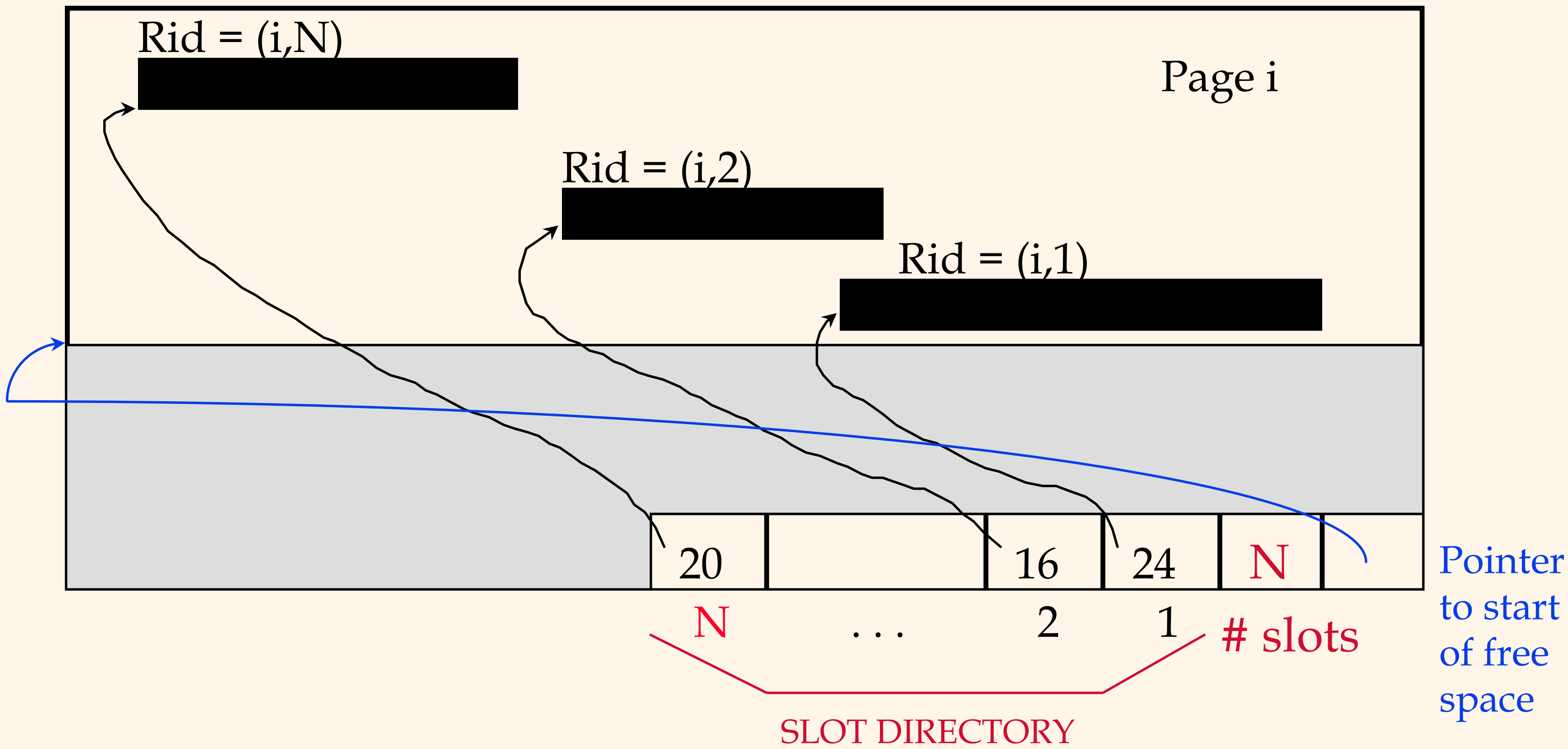
☛ Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

Page Formats: Fixed Length Records



- *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Page Formats: Variable Length Records



- *Can move records on page without changing rid; so, attractive for fixed-length records too.*

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)

Alternative File Organizations

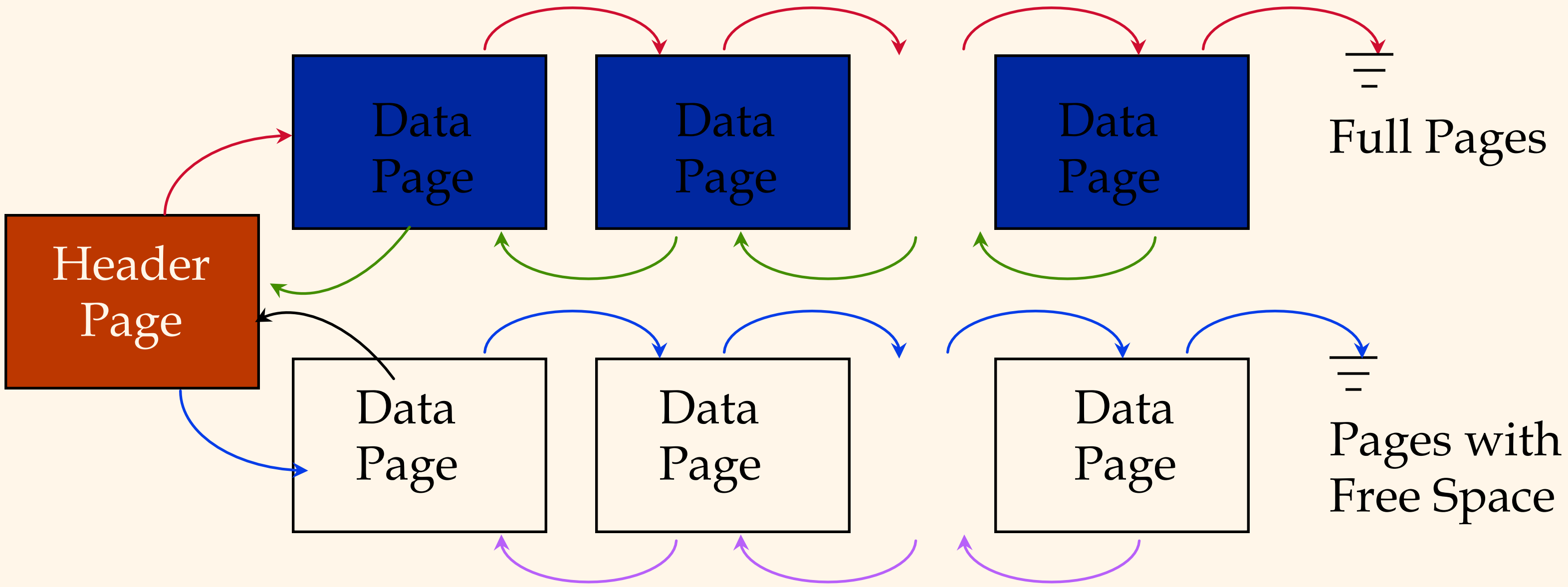
Many alternatives exist, *each ideal for some situations, and not so good in others:*

- 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

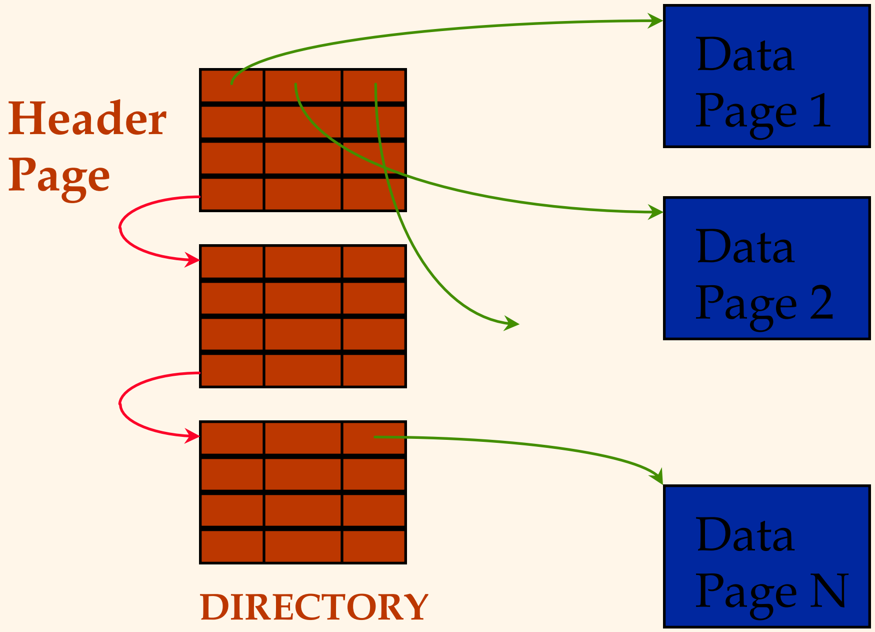
- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on 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.

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.