Database Applications (15-415)

DBMS Internals: Part II

Lecture 11, February 17, 2015

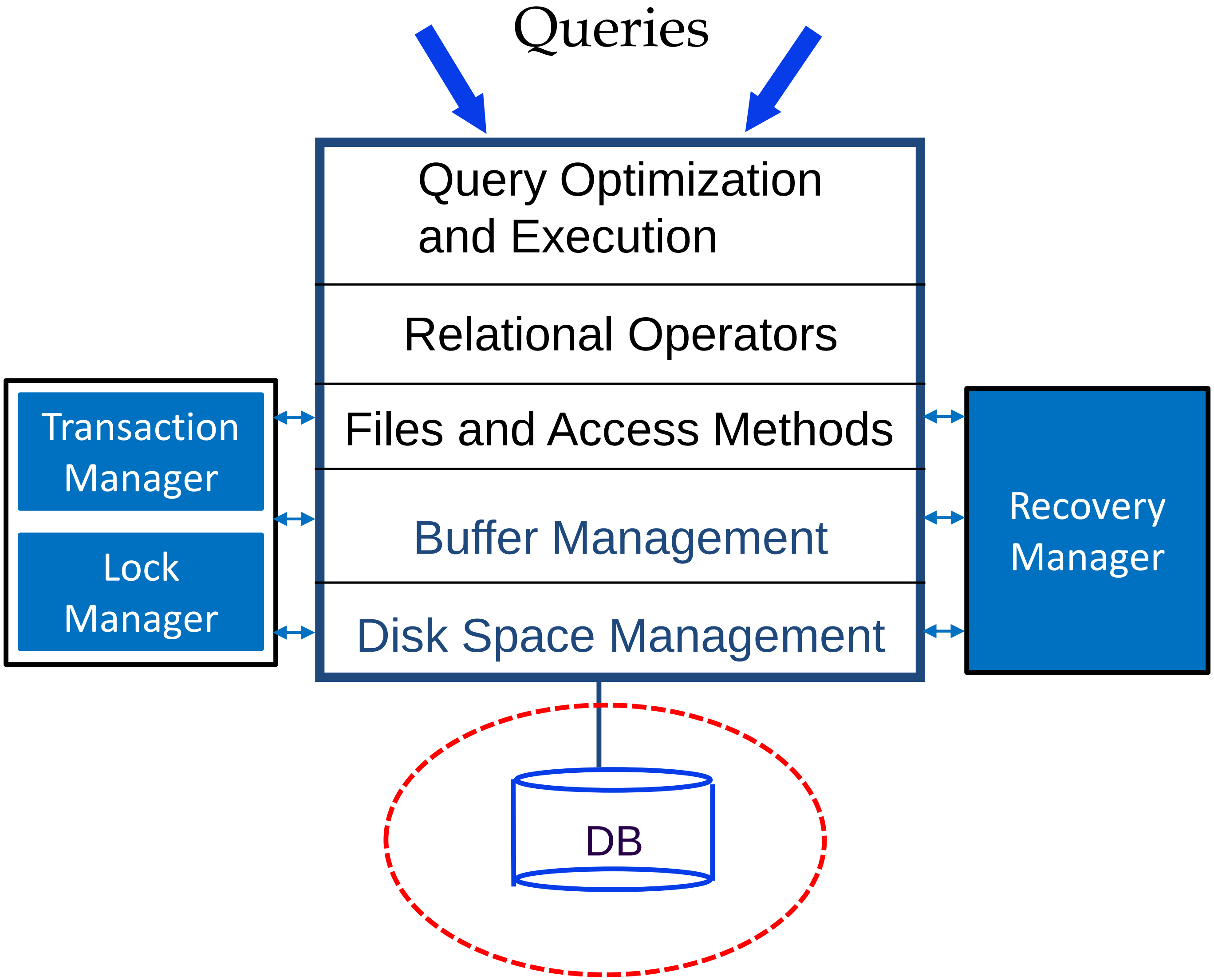
Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Today…

- Last Session:
  - DBMS Internals- Part I
- Today's Session:
  - DBMS Internals- Part II
    - A Brief Summary on Disks and the RAID Technology
    - File Organizations
- Announcements:
  - Project 1 is due today by midnight
  - The midterm exam is on Tuesday Feb 24 (*all* materials are included)

DBMS Layers



Queries

Query Optimization
and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

Lock Manager

Recovery Manager

DB

Disks: A “Very” Brief Summary

- DBMSs store data in disks
    - Disks provide large, cheap and non-volatile storage
- I/O time dominates!
- The cost depends on the locations of pages on disk (*among others*)
- It is important to arrange data *sequentially* to minimize *seek* and *rotational* delays

Disks: A "Very" Brief Summary

- Disks can cause reliability and performance problems

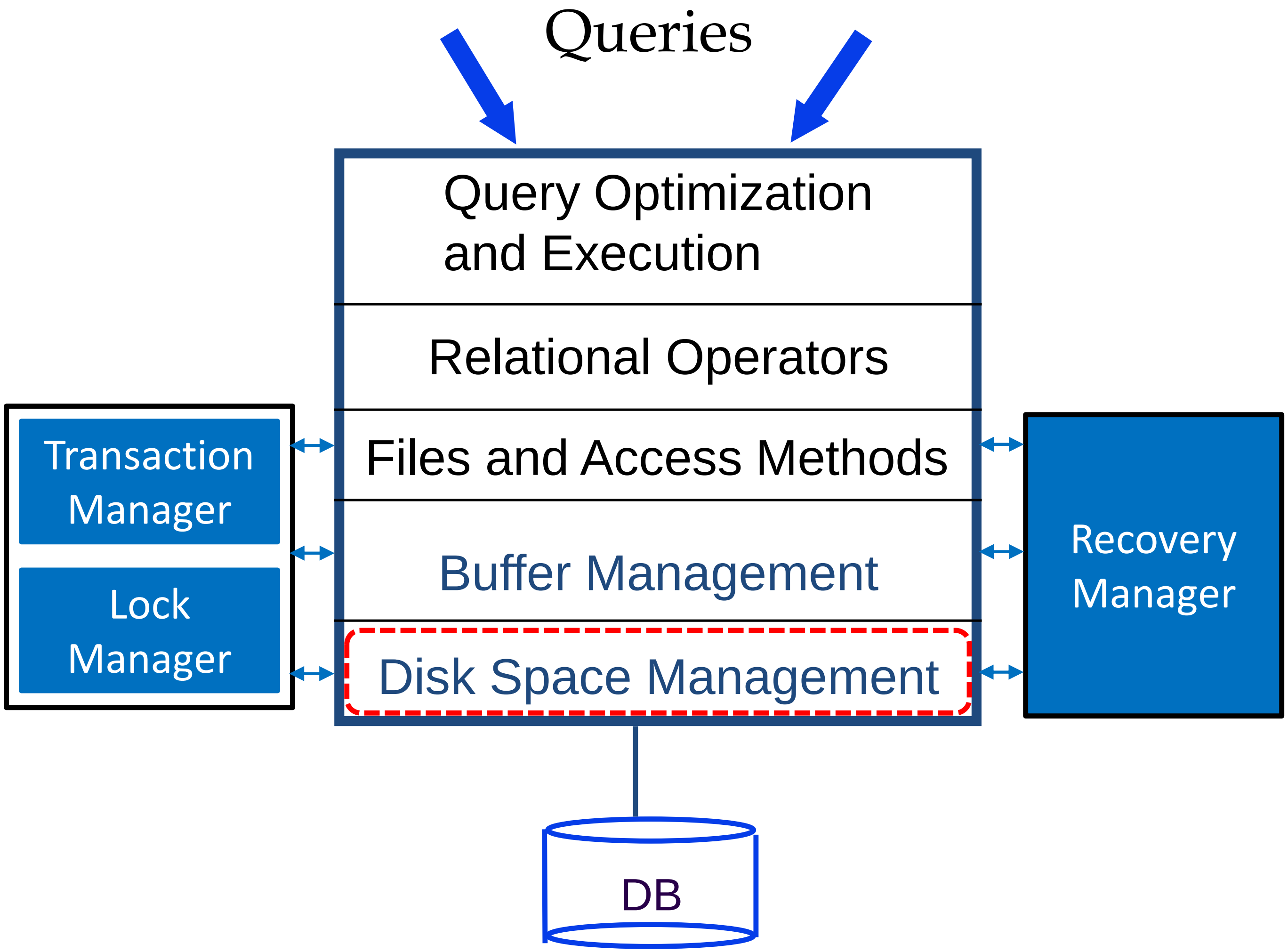
- To mitigate such problems we can adopt "multiple disks" and accordingly gain:
  1. More capacity
  2. Redundancy
  3. Concurrency

- To achieve only redundancy we apply mirroring

- To achieve only concurrency we apply striping

- To achieve redundancy *and* concurrency we apply RAID levels 2, 3, 4 or 5

DBMS Layers



Queries

Query Optimization
and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

Lock Manager

Recovery Manager

DB

Disk Space Management

- DBMSs disk space managers
  - Support the concept of a page as a unit of data
    - Page size is usually chosen to be equal to the block size so that reading or writing a page can be done in 1 disk I/O
  - Allocate/de-allocate pages as a *contiguous* sequence of blocks on disks
  - Abstracts hardware (and possibly OS) details from higher DBMS levels

What to Keep Track of?

- The DBMS disk space manager keeps track of:
  - Which disk blocks are in use
  - Which pages are on which disk blocks

- Blocks can be initially allocated contiguously, but allocating and de-allocating blocks usually create *“holes”*

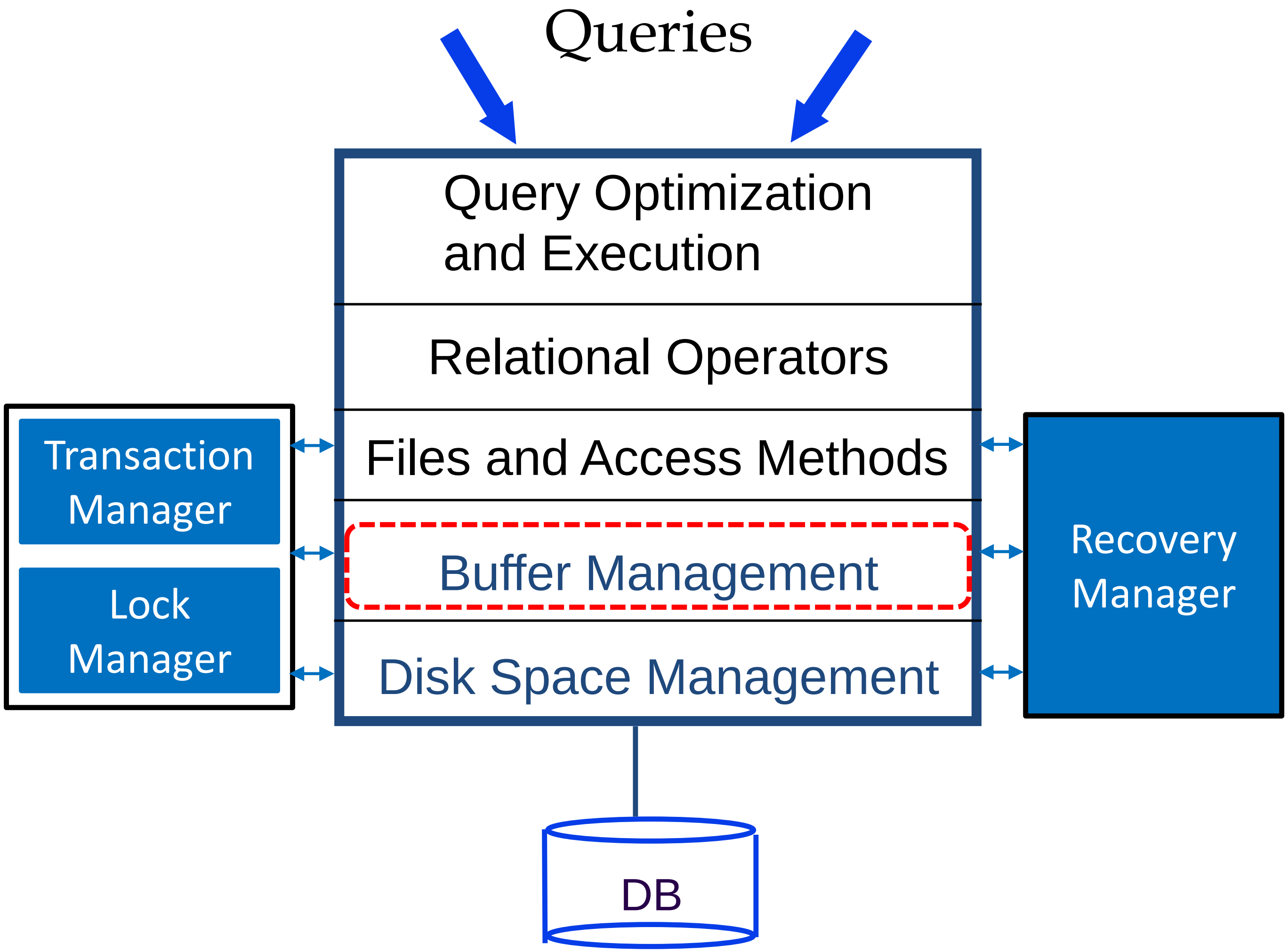
- Hence, a mechanism to keep track of *free blocks* is needed
  - A list of free blocks can be maintained (*storage could be an issue*)
  - Alternatively, a bitmap with one bit per each disk block can be maintained (*more storage efficient and faster in identifying contiguous free areas!*)

OS File Systems vs. DBMS Disk Space Managers

- Operating Systems already employ disk space managers using *their* “file” abstraction
  - “Read byte ***i*** of file ***f***” ➔ “read block ***m*** of track ***t*** of cylinder ***c*** of disk ***d***”

- DBMSs disk space managers usually pursue their own disk management without relying on OS file systems
  - Enables portability
  - Can address larger amounts of data
  - Allows *spanning* and *mirroring*

DBMS Layers



Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

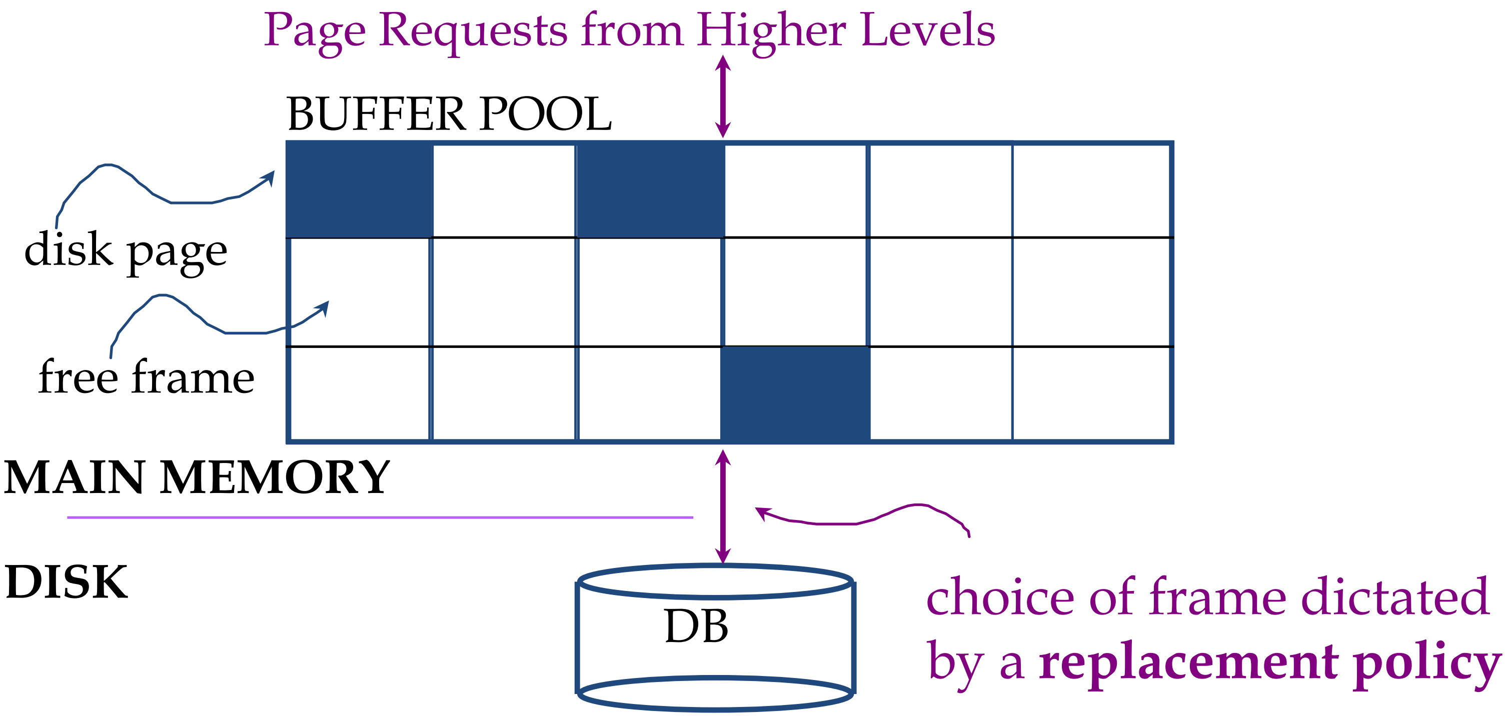
Lock Manager

Recovery Manager

DB

Buffer Management

- What is a DBMS buffer manager?
  - It is the software responsible for fetching pages in and out from/to disk to/from RAM as needed
  - It hides the fact that not all data is in the RAM



Satisfying Page Requests

- For each frame in the pool, the DBMS buffer manager maintains
  - The *pin_count* variable: # of users of a page
  - The *dirty* variable: whether a page has been modified or not

- If a page is requested and not in the pool, the DBMS buffer manager
  - Chooses a frame for *replacement* and increments its pin_count (a process known as pinning)
  - If frame is dirty, writes it back to disk
  - Reads the requested page into chosen frame

Satisfying Page Requests (Cont’d)

- A frame is not used to store a *new* page until its pin_count becomes 0
  - I.e., until all requestors of the *old* page have unpinned it (a process known as unpinning)
- When *many* frames with pin_count = 0 are available, a replacement policy is applied
- If no frame in the pool has pin_count = 0 and a page which is not in the pool is requested, the buffer manager must *wait* until some page is released!

Replacement Policies

- When a new page is to be placed in the pool, a resident page should be evicted first

- Criterion for an optimum replacement [*Belady, 1966*]:
  - The page that will be accessed the farthest in the future should be the one that is evicted

- Unfortunately, optimum replacement is not implementable!

- Hence, most buffer managers implement a different criterion
  - E.g., the page that was accessed the farthest back in the past is the one that is evicted
  - Or: MRU, Clock, FIFO, and Random, among others

Replacement Policies

- When a new page is to be placed in the pool, a resident page should be evicted first

- Criterion for an optimum replacement [*Belady, 1966*]:
    - The page that will be accessed the farthest in the future should be the one that is evicted

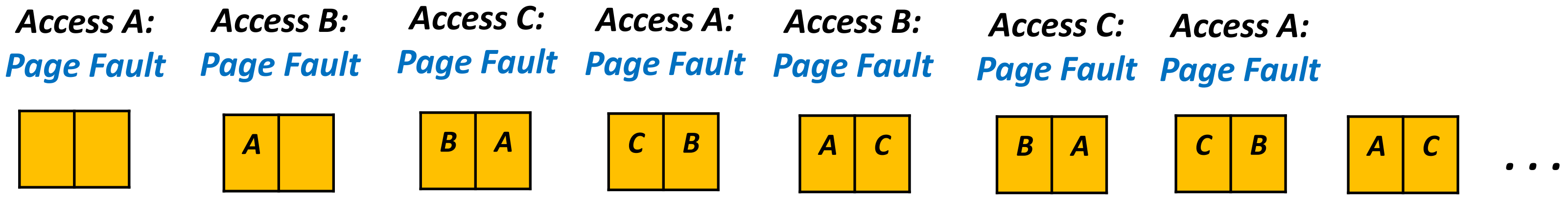
- Unfortunately, optimum replacement is not implementable!

- Hence, most buffer managers implement a different criterion
    - *This policy is known as the **Least Recently Used** (**LRU**) policy!*
    - Or: MRU, Clock, FIFO, and Random, among others

The LRU Replacement Policy

- Least Recently Used (LRU):
  - For each page in the buffer pool, keep track of the time it was *unpinned*
  - Evict the page at the frame which has the *oldest* time
- But, what if a user requires *sequential scans* of data which do not fit in the pool?

Assume an access pattern of ***A***, ***B***, ***C***, ***A***, ***B***, ***C***, etc.



Access A: *Page Fault*

Access B: *Page Fault*

Access C: *Page Fault*

Access A: *Page Fault*

Access B: *Page Fault*

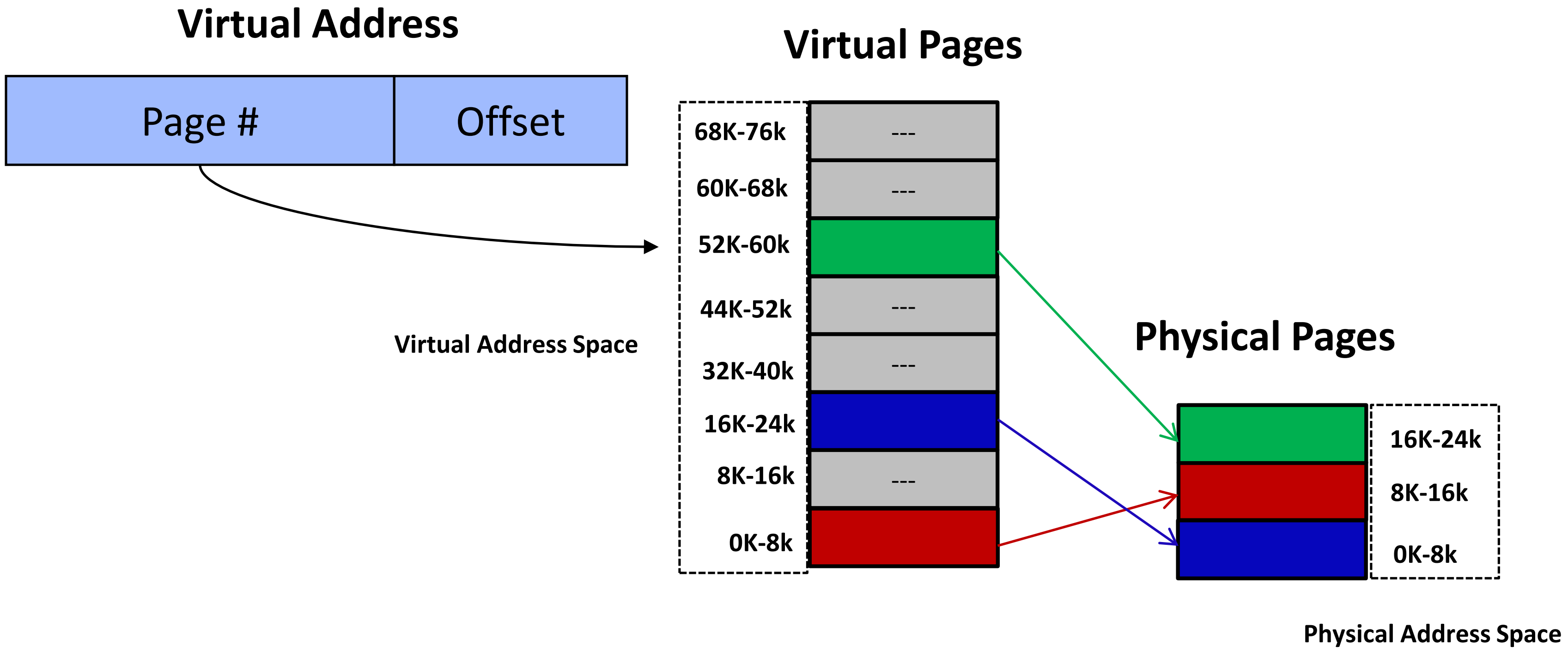
Access C: *Page Fault*

Access A: *Page Fault*

This phenomenon is known as "sequential flooding" (*for this, MRU works better!*)

Virtual Memory vs.
DBMS Buffer Managers

- Operating Systems already employ a buffer management technique known as virtual memory



Virtual Address

Page #

Offset

Virtual Pages

Virtual Address Space

68K-76k ---

60K-68k ---

52K-60k

44K-52k ---

32K-40k ---

16K-24k

8K-16k ---

0K-8k

Physical Pages

16K-24k

8K-16k

0K-8k

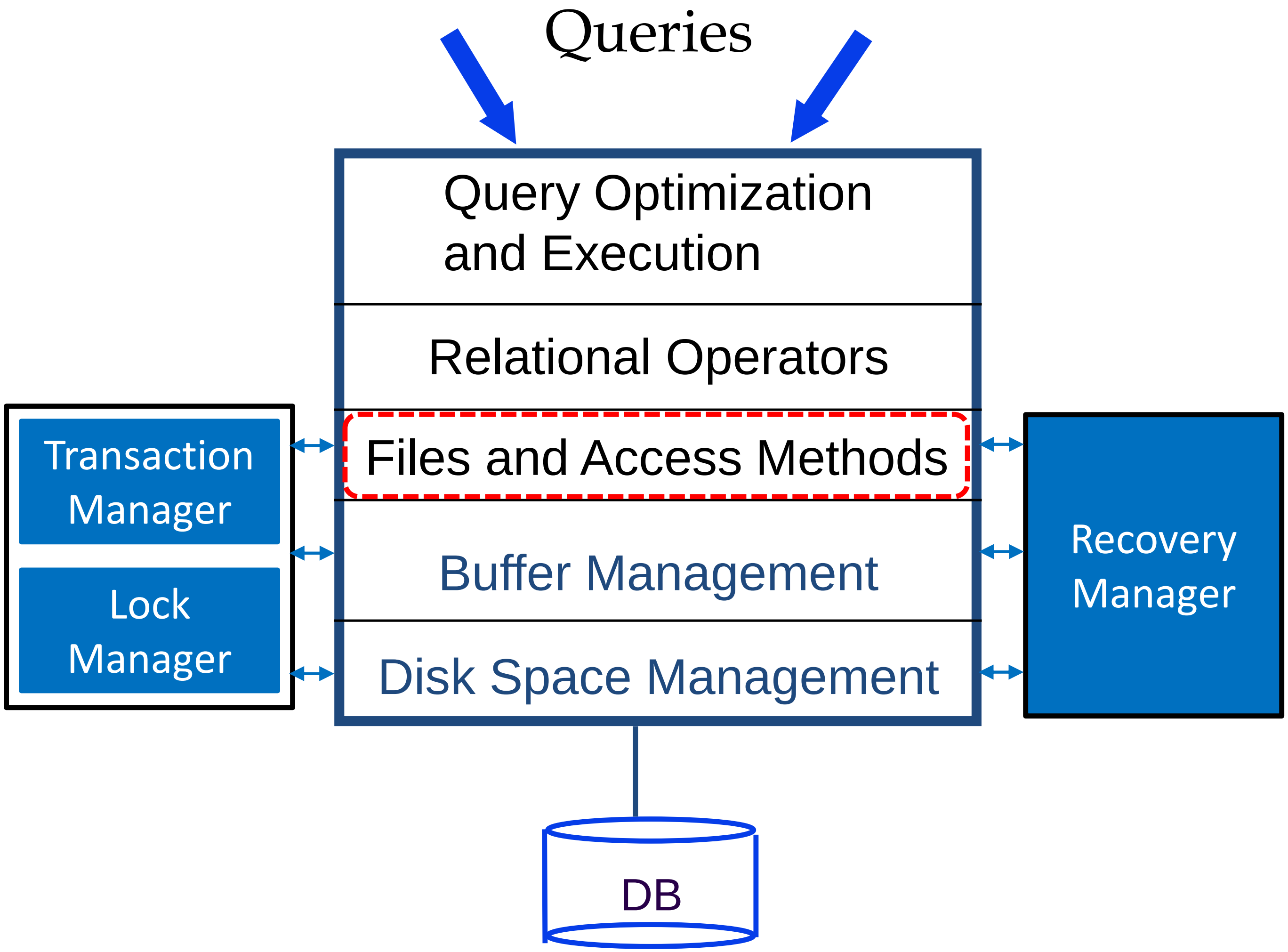
Physical Address Space

Virtual Memory vs. DBMS Buffer Managers

- Nonetheless, DBMSs pursue their own buffer management so that they can:
    - Predict page reference patterns more accurately and applying effective strategies (e.g., page prefetching for improving performance)

    - F*orce* pages to disks (needed for the WAL protocol)
        - Typically, the OS cannot guarantee this!

DBMS Layers



Queries

Query Optimization
and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

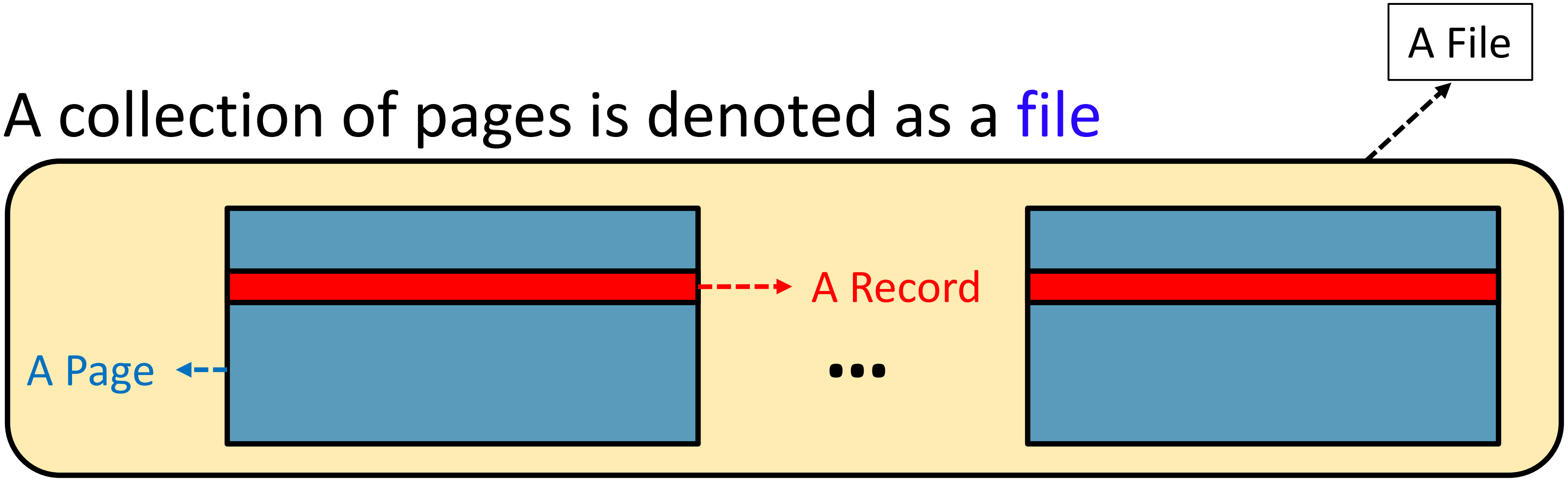
Lock Manager

Recovery Manager

DB

Records, Pages and Files

- Higher-levels of DBMSs deal with records (not pages!)
- At lower-levels, records are stored in pages
- But, a page might not fit all records of a database
  - Hence, multiple pages might be needed
- A collection of pages is denoted as a file



File Operations and Organizations

- A file is a collection of pages, each containing a collection of records

- Files must support operations like:
  - Insert/Delete/Modify records
  - Read a particular record (specified using a *record id*)
  - Scan all records (possibly with some conditions on the records to be retrieved)

- There are several organizations of files:
  - Heap
  - Sorted
  - Indexed

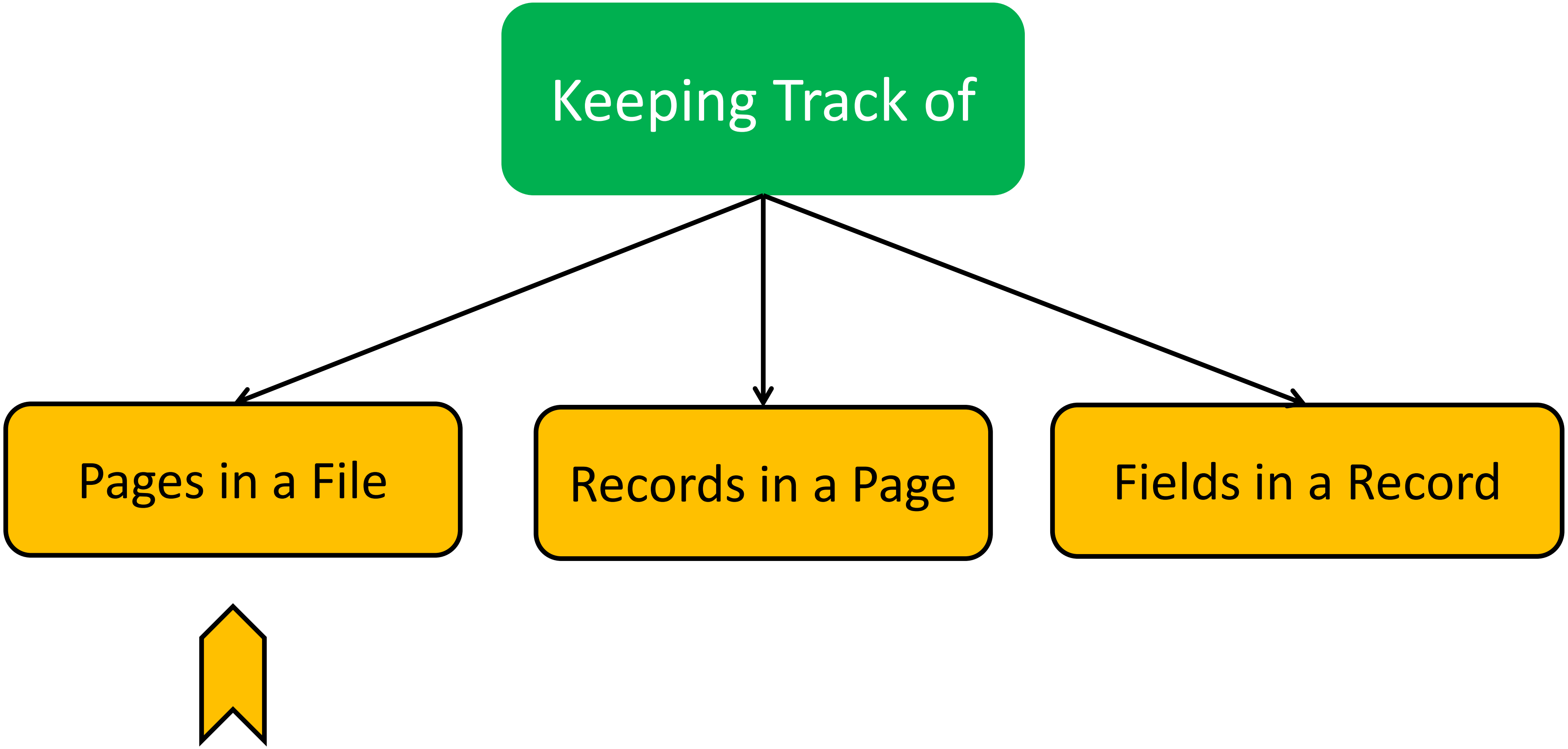
Heap Files

- Records in heap file pages do not follow any particular order

- As a heap 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 the *records* on a page
  - Keep track of the *fields* on a record

Supporting Record Level Operations



Keeping Track of

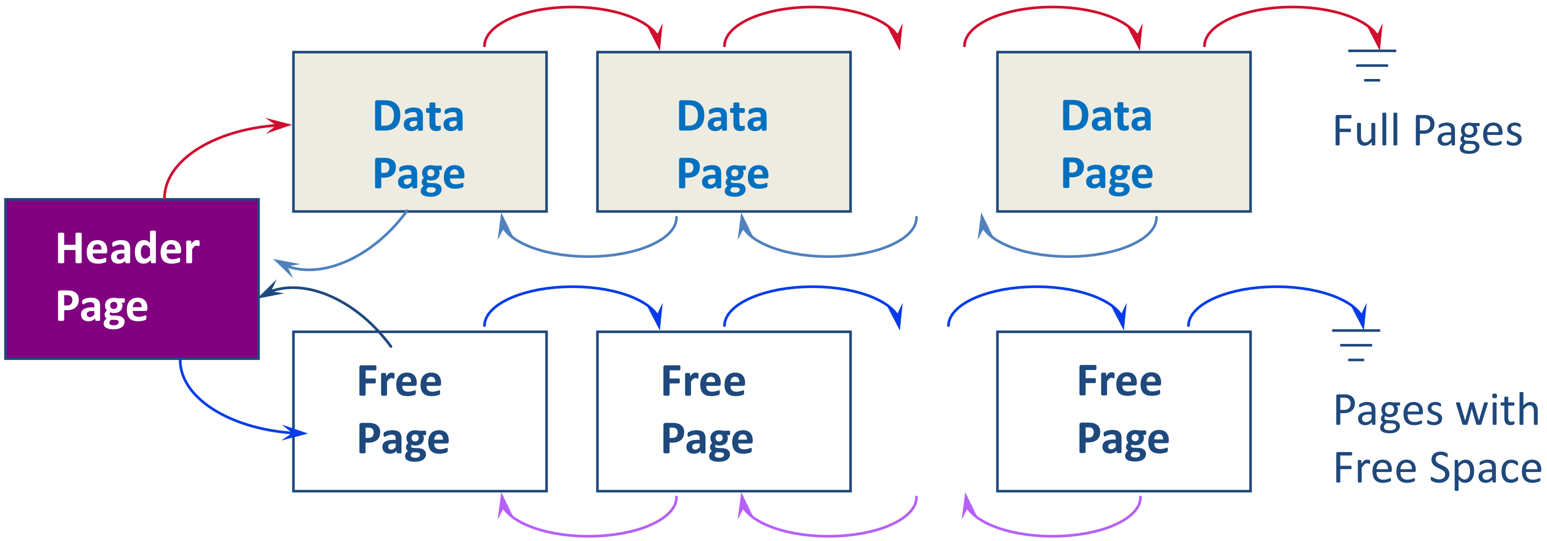
Pages in a File

Records in a Page

Fields in a Record

Heap Files Using *Lists* of Pages

- A heap file can be organized as a *doubly linked list* of pages



- The Header Page (i.e., <*heap_file_name*, *page_1_addr*> is stored in a known location on disk
- Each page contains 2 ‘pointers’ plus data

Heap Files Using *Lists* of Pages

- It is likely that every page has at least a few free bytes

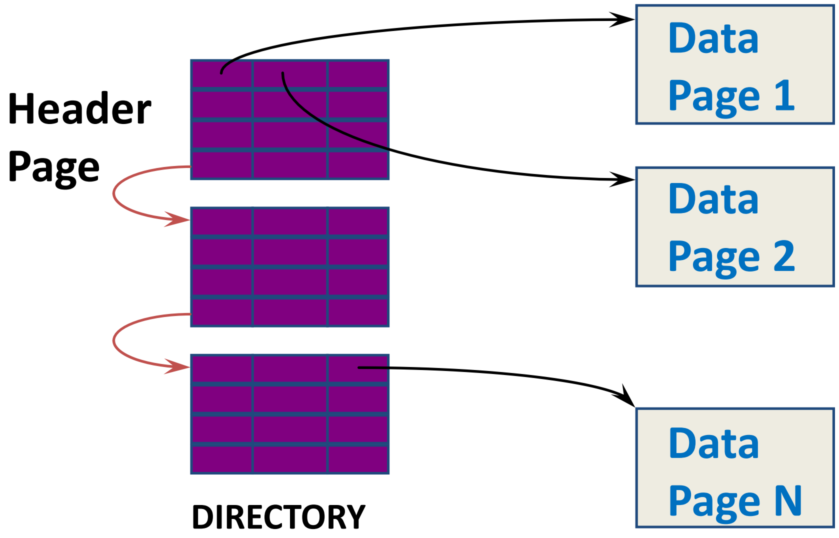
- Thus, virtually all pages in a file will be on the free list!

- To insert a typical record, we must retrieve and examine several pages on the free list before one with *enough* free space is found

- This problem can be addressed using an alternative design known as the directory-based heap file organization

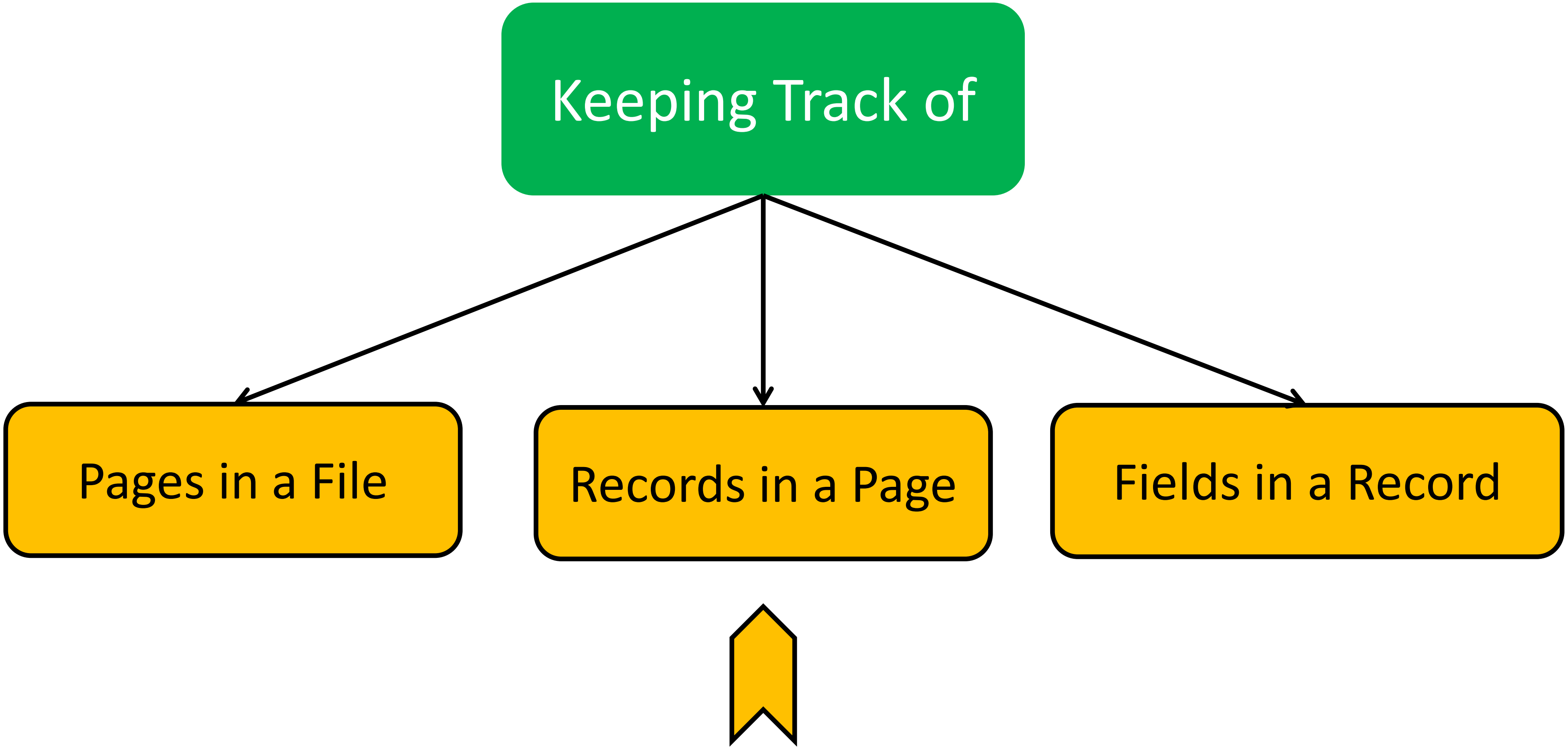
Heap Files Using *Directory* of Pages

- A directory of pages can be maintained whereby each directory entry identifies a page in the heap file



- Free space can be managed via maintaining:
  - A *bit* per entry (indicating whether the corresponding page has any free space)
  - A *count* per entry (indicating the amount of free space on the page)

Supporting Record Level Operations



Keeping Track of

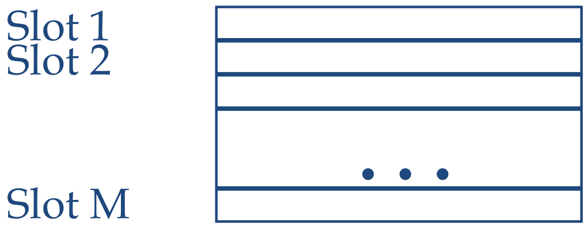
Pages in a File

Records in a Page

Fields in a Record

Page Formats

- A page in a file can be thought of as a collection of slots, each of which contains a record

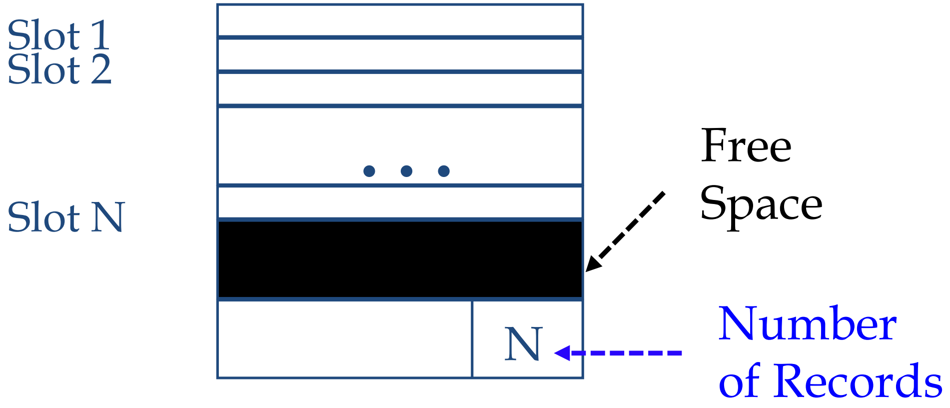


- A record can be identified using the pair <page_id, slot_#>, which is typically referred to as record id (rid)

- Records can be either:
  - Fixed-Length
  - Variable-Length

Fixed-Length Records

- When records are of fixed-length, slots become *uniform* and can be arranged *consecutively*

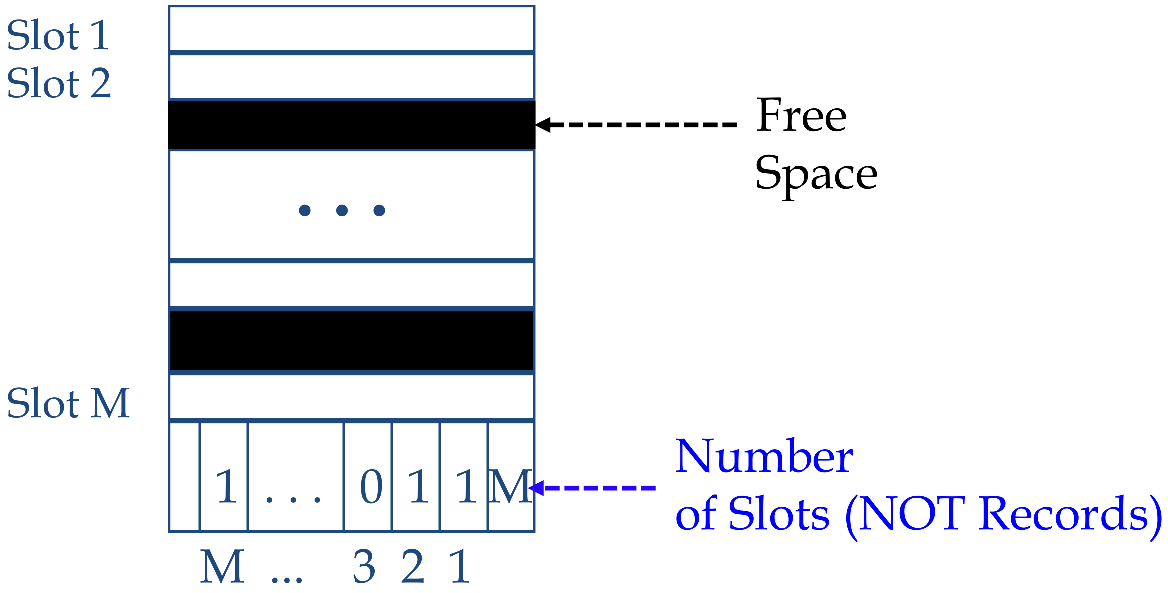


- Records can be located by simple offset calculations

- Whenever a record is *deleted*, the last record on the page is *moved* into the vacated slot
  - This changes its rid <page_id, slot_#> (*may not be acceptable!*)

Fixed-Length Records

- Alternatively, we can handle deletions by using an array of bits



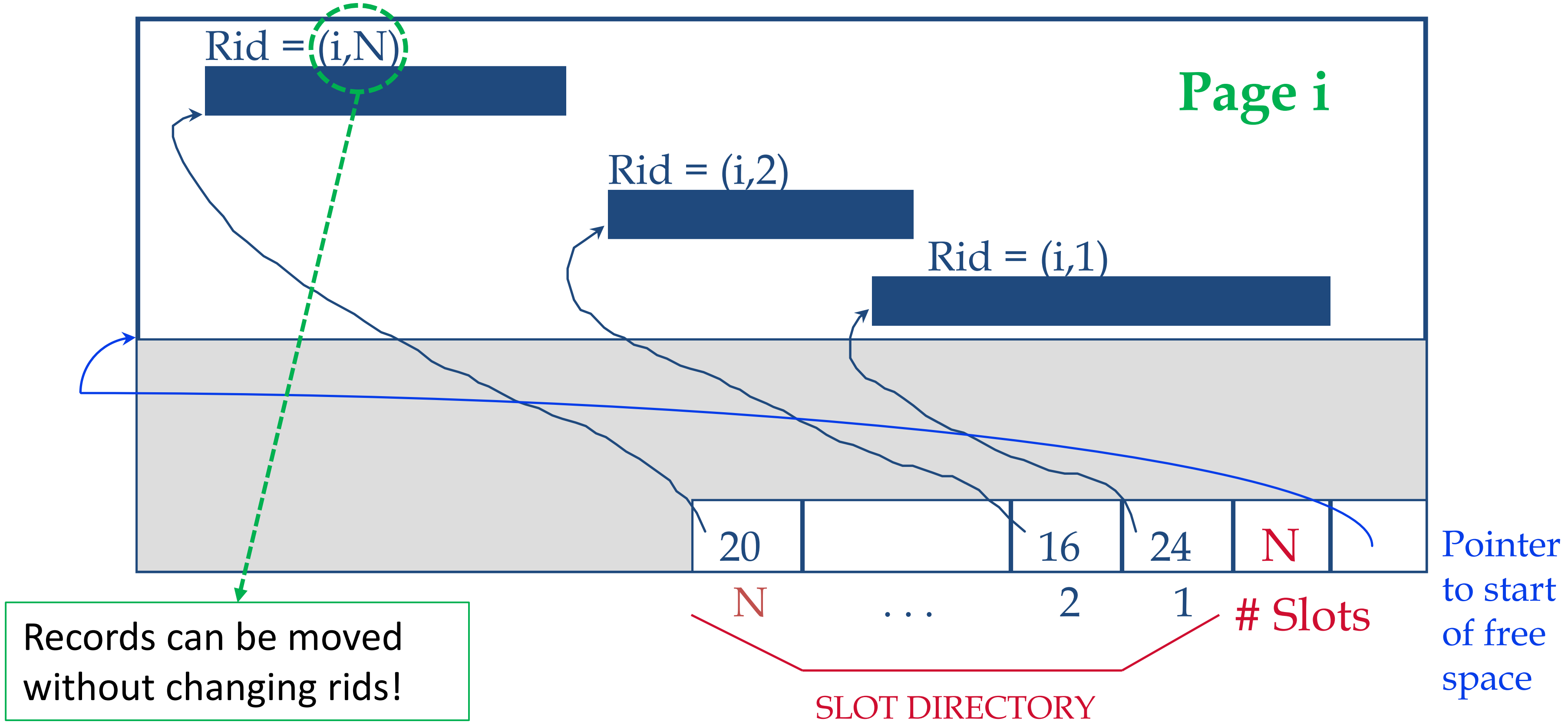
- When a record is deleted, its bit is turned off, thus, the rids of currently stored records remain the same!

Variable-Length Records

- If the records are of variable length, we cannot divide the page into a fixed collection of slots
- When a new record is to be inserted, we have to find an empty slot of “just” the right length
- Thus, when a record is deleted, we better ensure that all the free space is contiguous
- The ability of moving records “*without changing rids*” becomes crucial!

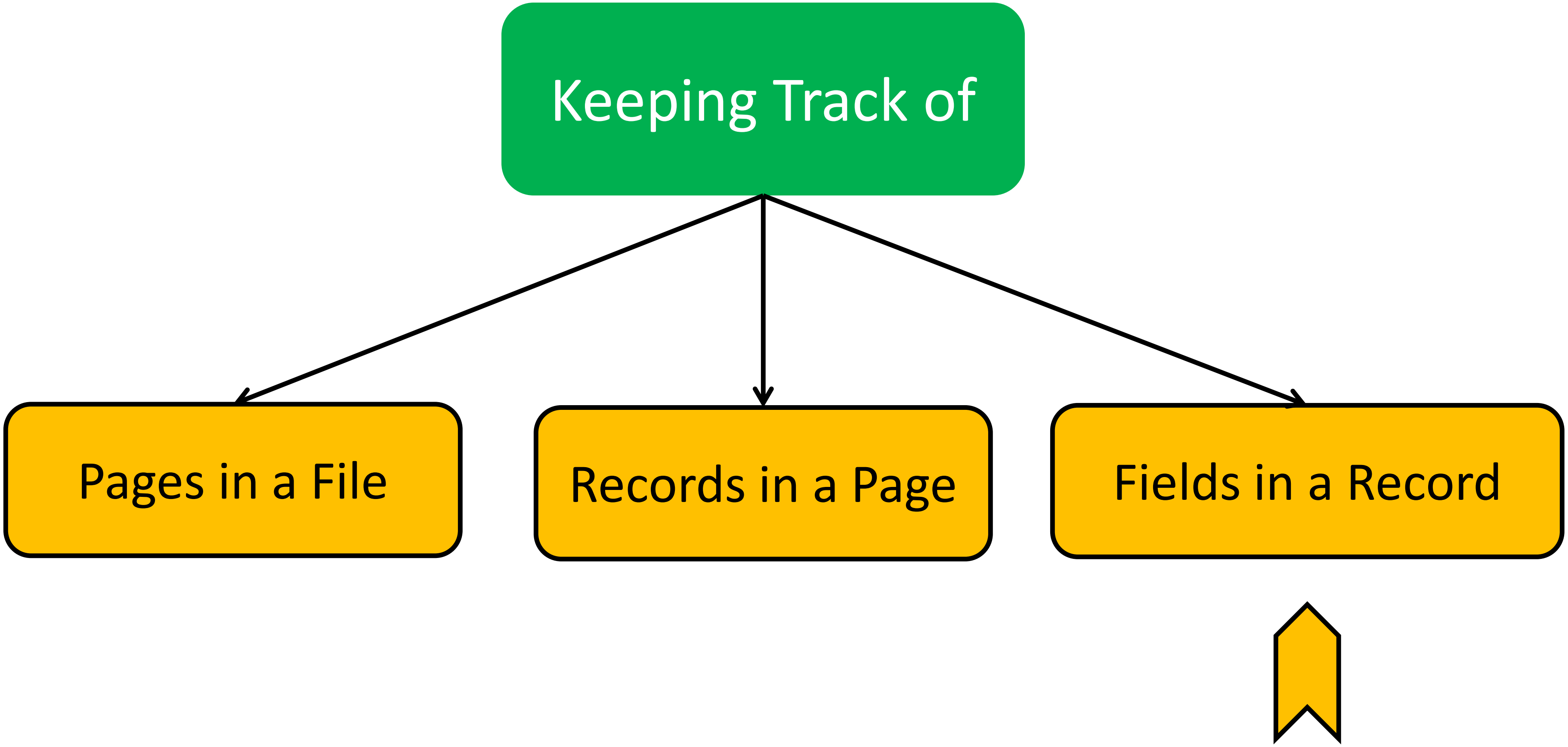
Pages with Directory of Slots

- A flexible organization for variable-length records is to maintain a directory of slots with a *<record_offset, record_length>* pair per a page



Records can be moved without changing rids!

Supporting Record Level Operations



Keeping Track of

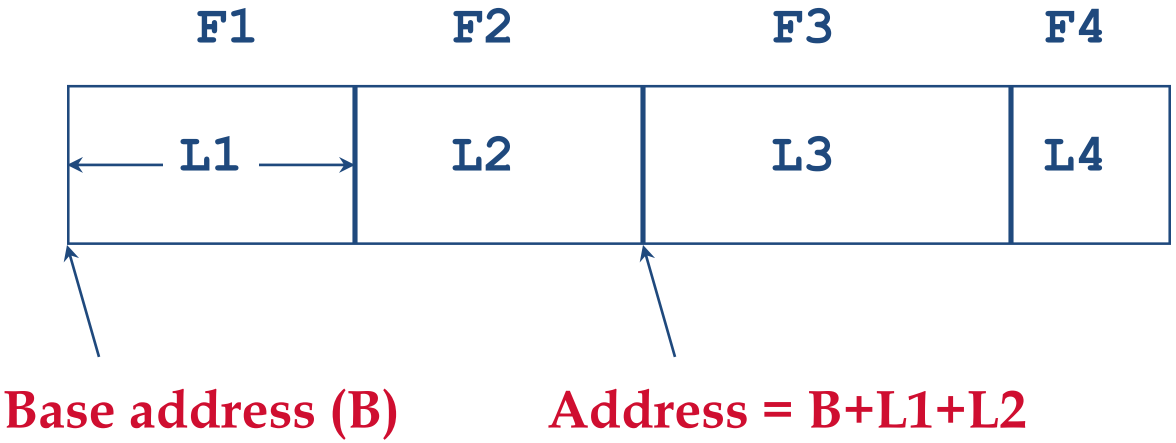
- Pages in a File
- Records in a Page
- Fields in a Record

Record Formats

- Fields in a record can be either of:
  - Fixed-Length: each field has a fixed length and the number of fields is also fixed
  - Variable-Length: fields are of variable lengths but the number of fields is fixed
- Information common to all records (e.g., number of fields and field types) are stored in the system catalog

Fixed-Length Fields

- Fixed-length fields can be stored consecutively and their addresses can be calculated using information about the lengths of preceding fields



F1 F2 F3 F4

L1 L2 L3 L4

Base address (B)

Address = B+L1+L2

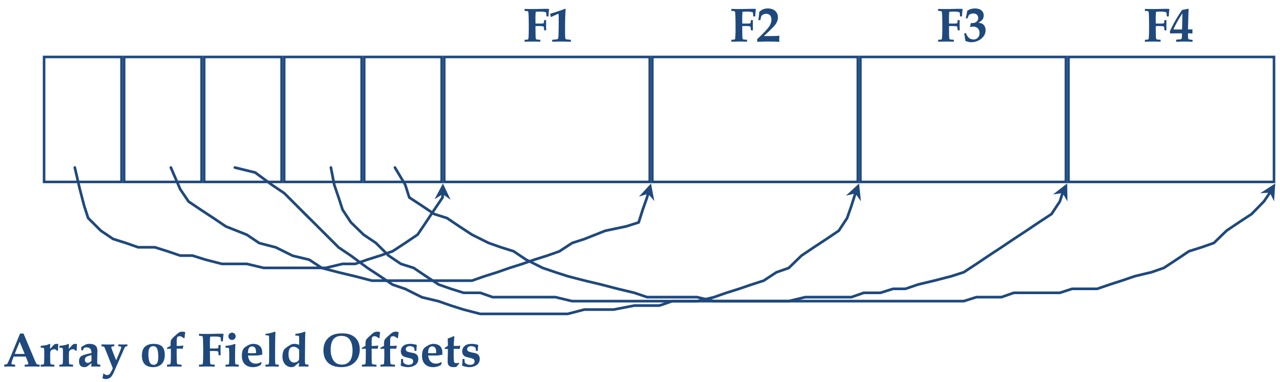
Variable-Length Fields

- There are two possible organizations to store variable-length fields
  1. Consecutive storage of fields separated by delimiters

This entails a scan of records to locate a desired field!

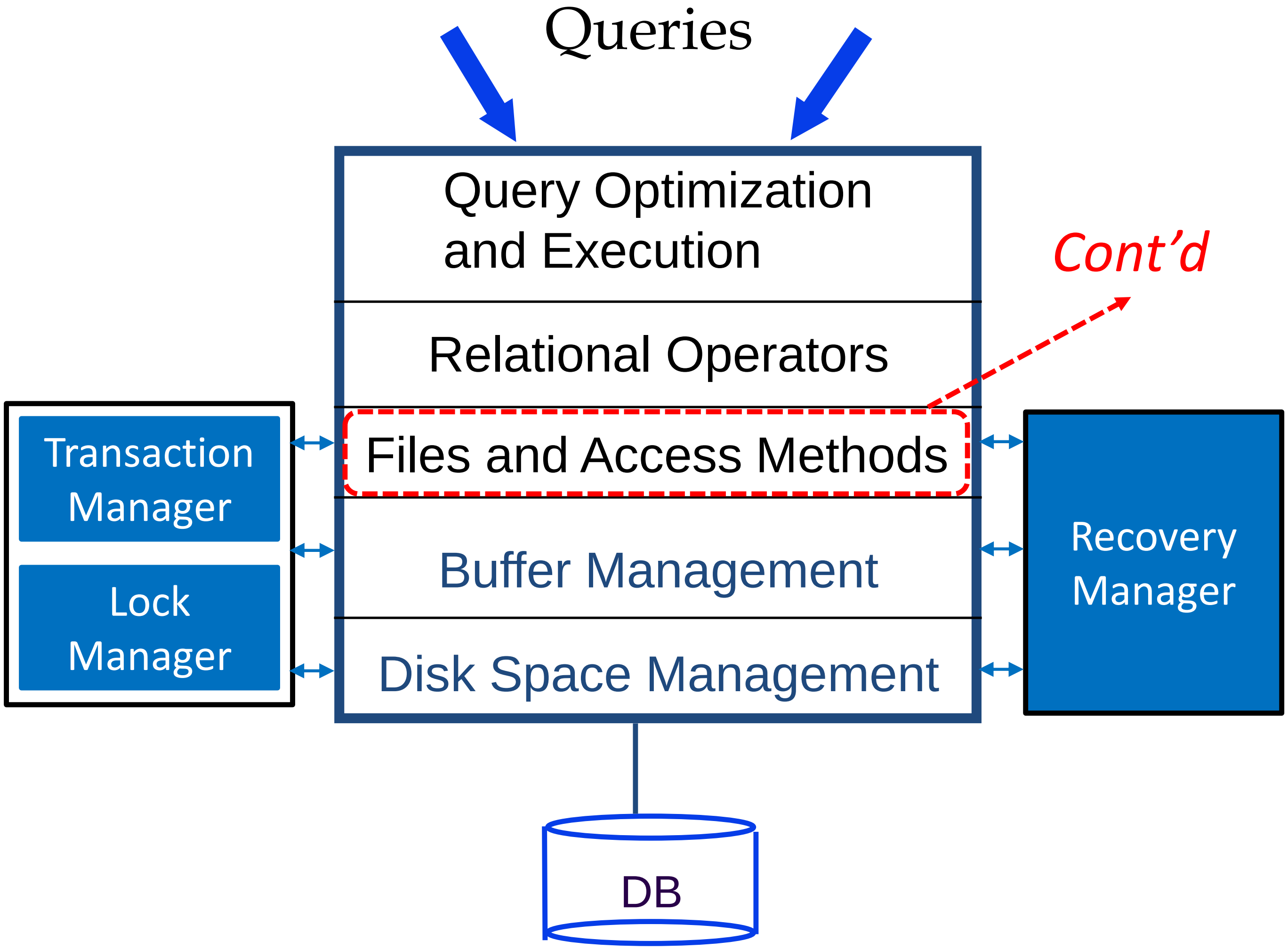
Variable-Length Fields

- There are two possible organizations to store variable-length fields
  1. Consecutive storage of fields separated by delimiters
  2. Storage of fields with an array of integer offsets



This offers *direct access* to a field in a record and stores NULL values efficiently!

Next Class



Queries

Query Optimization
and Execution

Relational Operators

Files and Access Methods

Cont'd

Buffer Management

Disk Space Management

Transaction
Manager

Lock
Manager

Recovery
Manager

DB