ICCS240 Database Management

Storage and File Structure

Many slides in this lecture are either from or adapted from:

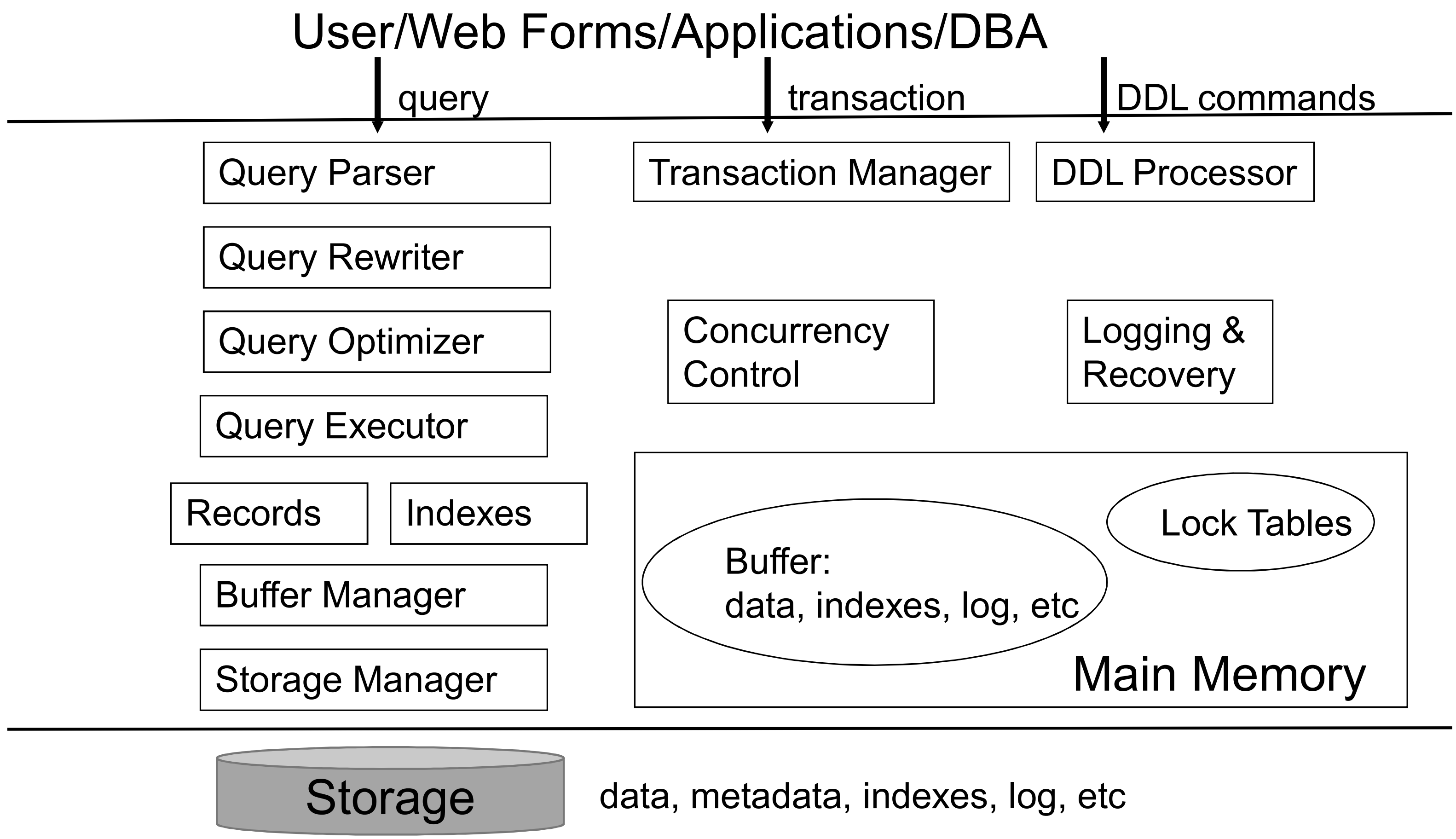
slides provided by Kazuhiro Minami, UIUC

texts from Database Management Systems by R. Ramakrishnan and J. Gehrke

Two Perspectives on DBMS

- **User** perspective
  - How to use a database system
    - Database design
    - Database programming
- **System** perspective
  - How to design and implement a database system
    - Storage management
    - Query processing
    - Transaction management

The Big Picture — DBMS Architecture

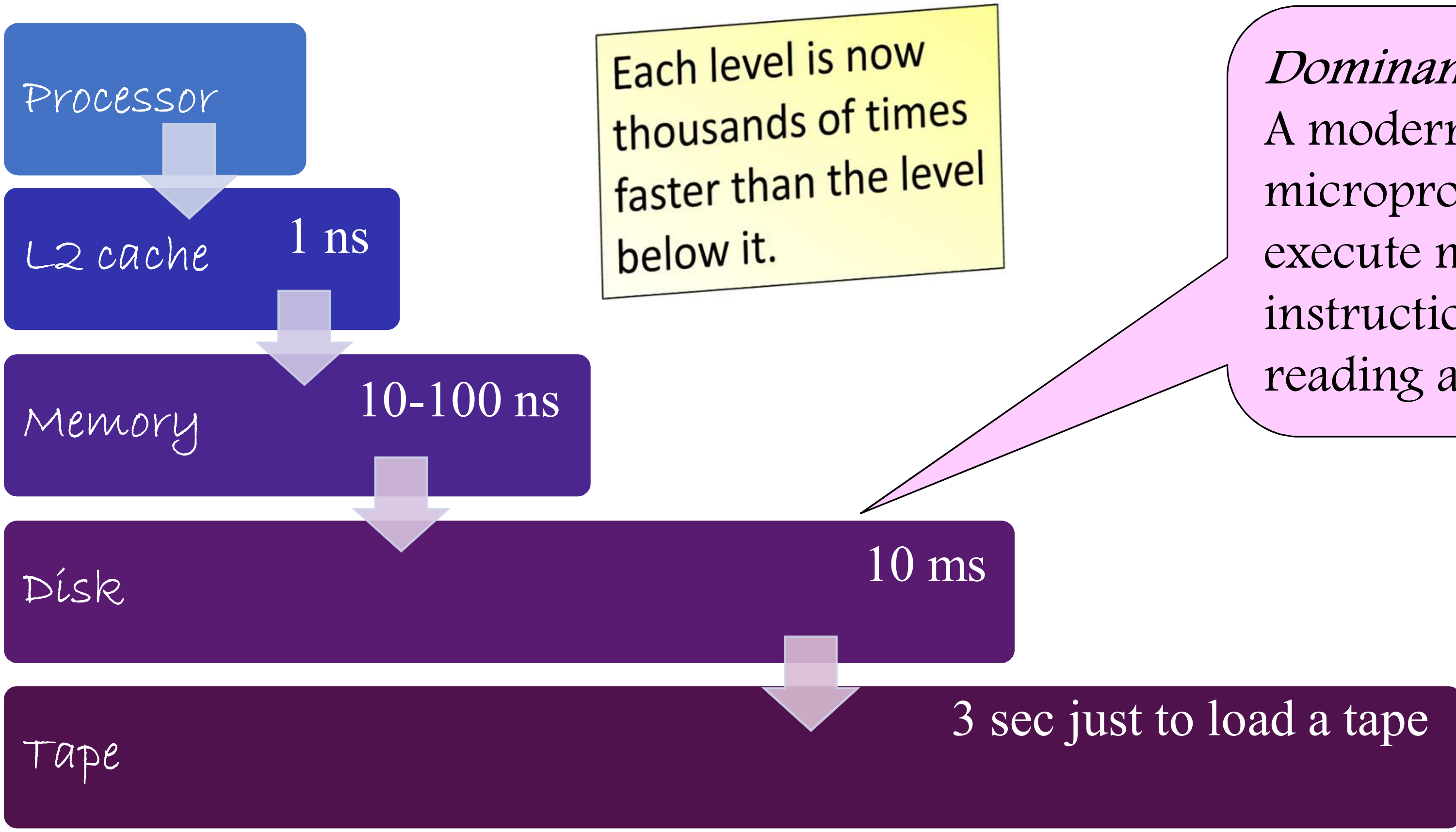


Access Times

0.5 ns	L1 Cache Ref	0.5 sec
7 ns	L2 Cache Ref	7 sec
100 ns	DRAM	100 sec
150,000 ns	SSD	1.7 days
10,000,000 ns	HDD	16.5 weeks
~30,000,000 ns	Network Storage	11.4 months
1,000,000,000 ns	Tape Archives	31.7 years

https://gist.github.com/hellerbarde/2843375

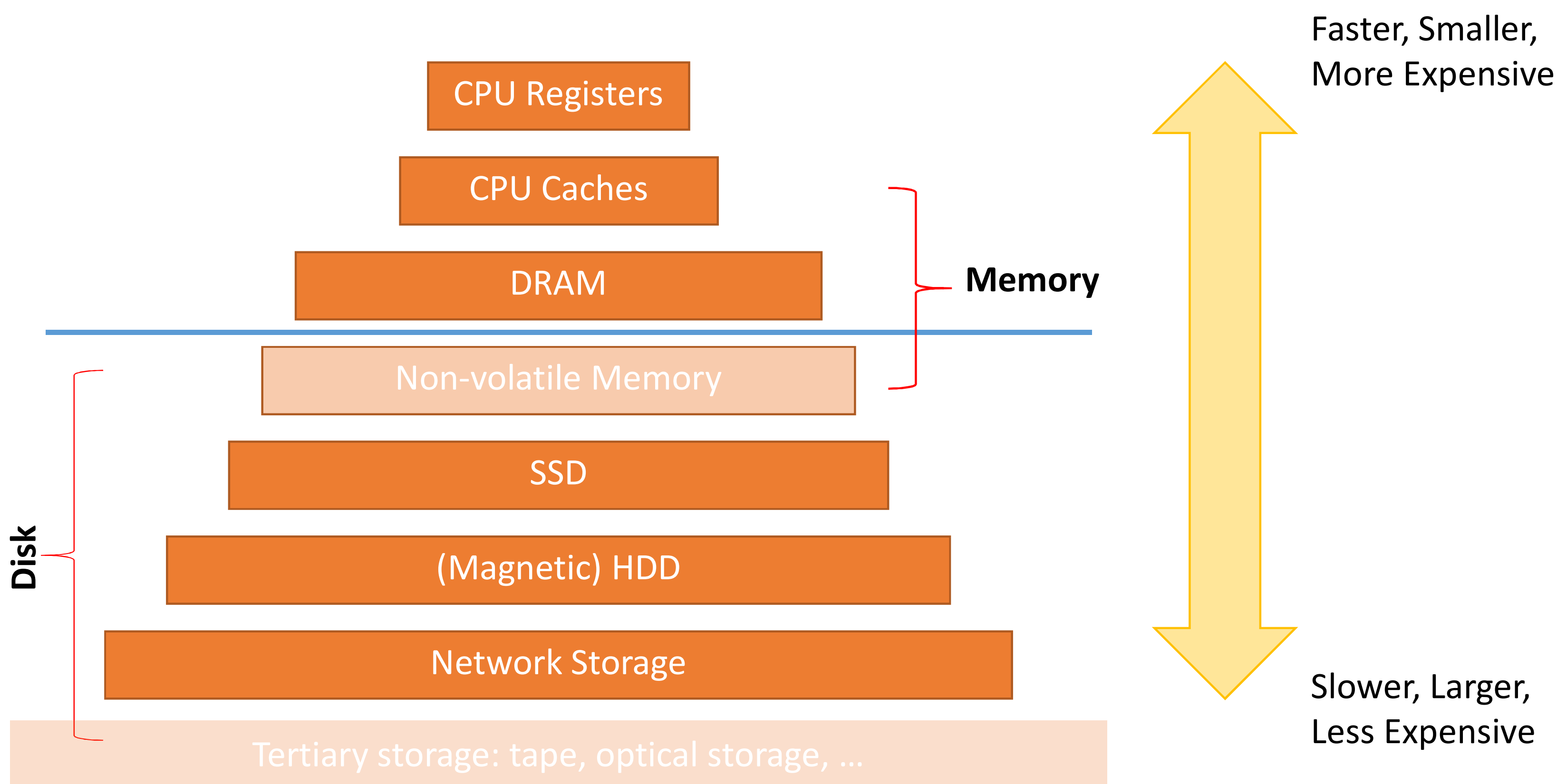
The Memory Hierarchy



Each level is now thousands of times faster than the level below it.

Dominance of I/O cost: A modern microprocessor can execute millions of instructions while reading a block.

Storage Pyramid – Where should we store data?



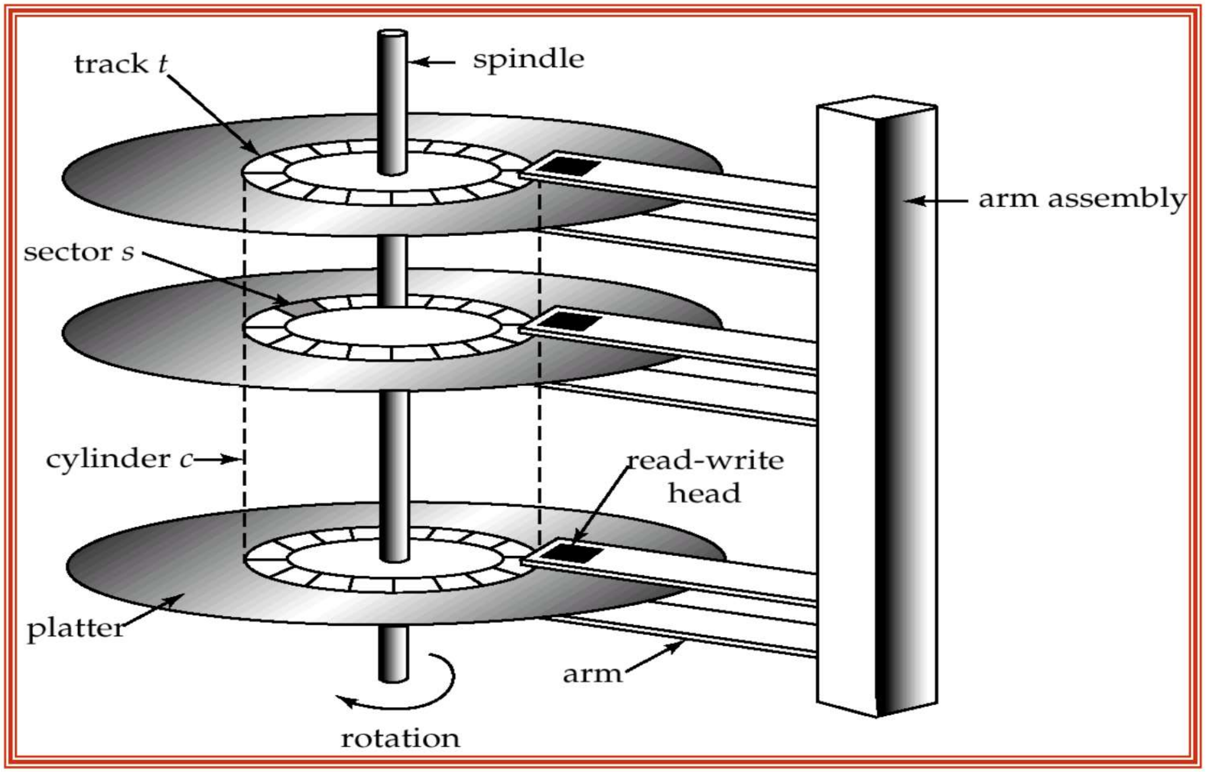
Goals

Allow the DBMS to manage databases that exceed the amount of (fast) memory available.

But reading/writing to disk is expensive, so must manage it carefully
—to minimize stalls and performance degradation.

Hard-drive Mechanism

Good for long sequential reads; bad for seek-heavy workloads



- Magnetic disks support direct access to a desired location.
- Data is stored on disk in units called disk blocks, which is the unit of reading or writing.
  - Size of disk block can be set.
  - Typically, 4K, 8K, 16K
- Surface of platter divided into circular tracks.
  - Over 50K-100K tracks per platter
- Each track is divided into sectors
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 B
  - Typically, 500-1000 sectors/track ≈ 10^5bytes/track
- Platter may have one or two surfaces
- Cylinder = set of all tracks with the same radius
  - Cylinder i^{th} consists of i^{th} track of all the platters

DBMS is primarily optimized for HDD
— implications:

- Seek time and rotational delay dominate!
- Key to lower I/O cost: **reduce seek/rotation delays!**
- How to minimize seek and rotational delays?
  - Blocks on same track, followed by
  - Blocks on same cylinder, followed by
  - Blocks on adjacent cylinder
  - Hence, sequential arrangement of blocks of a file is a big win!

Disk Organization:
Reliability & Performance

Many Disks vs. One Disk

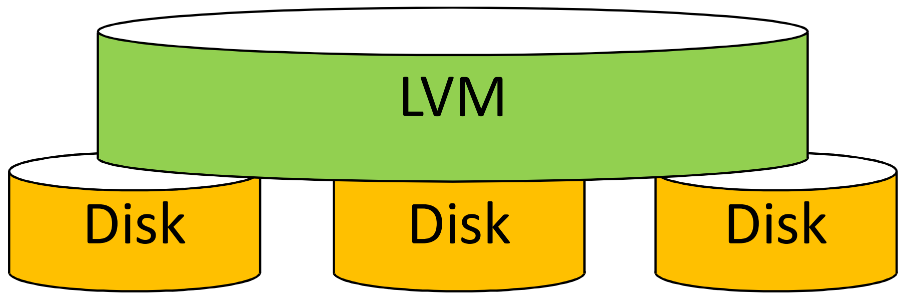
Although disks provide cheap, non-volatile storage for DBMSs,
they are usually bottlenecks for DBMSs.

Factors: **Reliability**, **Performance**, **Capacity**

How about adopting multiple disks?

- More data can be held as opposed to one disk **Capacity!**
- Data can be stored redundantly; hence, if one disk fails, data can be found on another **Reliability!**
- Data can be accessed concurrently **Performance!**

Logical Volume Managers (LVMs)

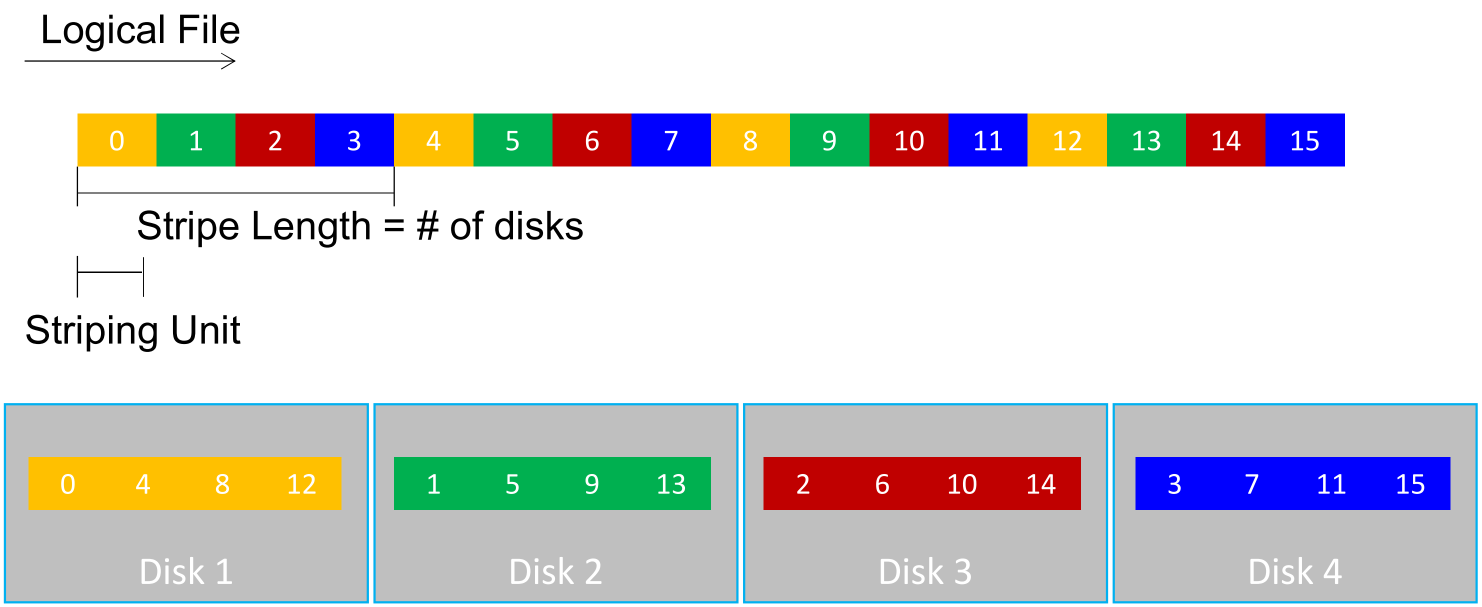


LVMs give appearance of a single logical disk.

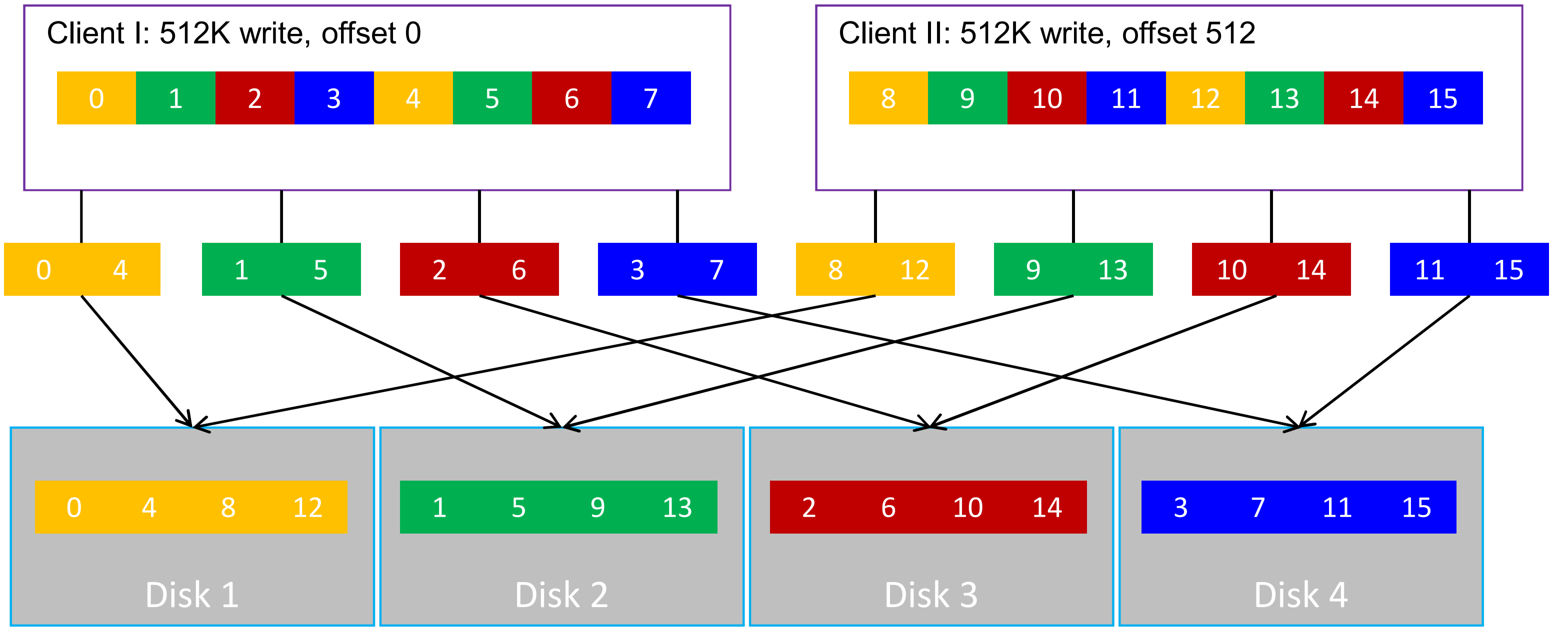
- SPANNING:
  - LVM transparently maps a larger address space to different disks
- MIRRORING:
  - Each disk can hold a separate, identical copy of data
  - LVM directs writes to the same block address on each disk
  - LVM directs a read to any disk (e.g., to the less busy one)

Data Striping

To achieve parallel accesses, we can use a technique called data striping



Data Striping (cont.)



Trade-off:

- Small striping unit values: higher parallelism, higher disk seek and rotational delays
- Large striping unit values: lower parallelism, decrease disk seek and rotational delays

RAID: Redundant arrays on independent disks

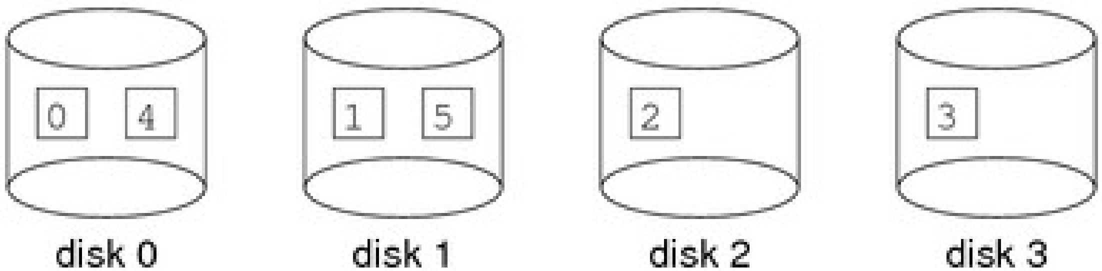
A standard techniques to improve performance and reliability when multiple disks are available.

- Improved reliability by redundant storage of data
- Reduced access cost by exploiting parallelism

(although there is obviously a trade-off between increased capacity and increased reliability via redundancy)

RAID combines mirroring and striping!

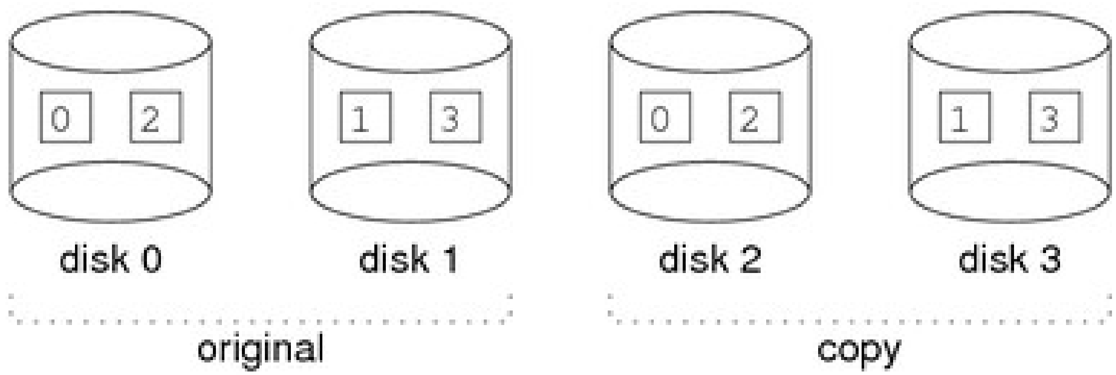
RAID0 – Striping + No Fault Tolerance



Source: https://www.cse.unsw.edu.au/~cs9315/18s2/notes/B/notes.html

RAID1 – Striping + Mirroring

Improve `READ`s; Impair `WRITE`s



Source: https://www.cse.unsw.edu.au/~cs9315/18s2/notes/B/notes.html

RAID2-6 …

The higher levels of raid incorporate various combinations of
block/bit-level striping, mirroring, and error correcting codes

The differences are primarily in

- the kind of error checking/correcting codes that are used
- where error correcting codes parity bits are stored

Source: https://www.cse.unsw.edu.au/~cs9315/18s2/notes/B/notes.html

Disk Management

Disk Space Management

- **Disk space manager** manages space on disk
- Disk space manager supports the concept of a **page** as a unit of data
- The size of a page is chosen to be the size of a disk block
- Useful to allocate a sequence of pages as a contiguous sequence of blocks to hold data frequently accessed.
- Disk space manager *keeps track* of blocks in use or location of blocks on disk

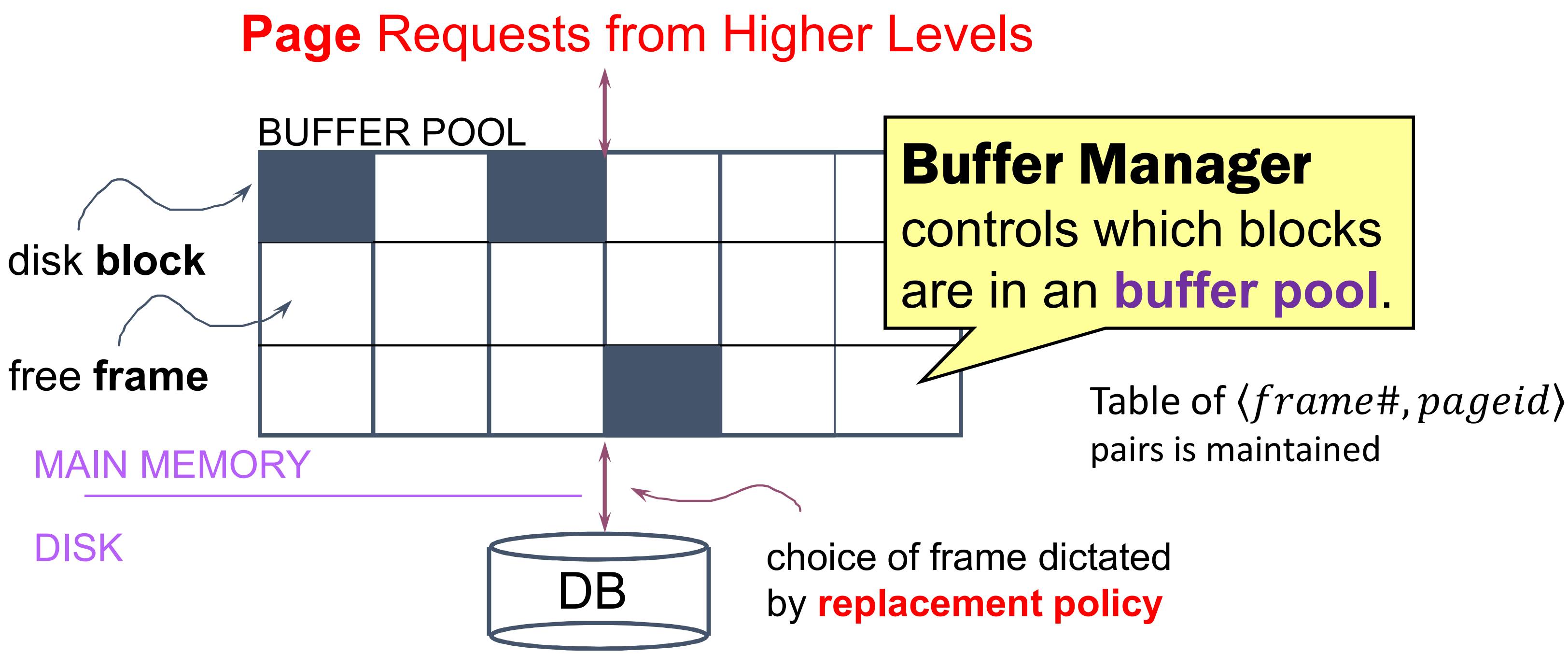
More detail in some other class …

Store in DISK, Process in (faster) MEMORY

Data must be brought into memory to be processed!

- A requested page is transferred from disk to main memory
  (if not fetched earlier & found in memory)
- Evicted pages are transferred from memory to disk
- I/O time dominates the time taken for database operations!
- To minimize I/O time, it is necessary to store and locate data strategically

Disk Buffer Management in a DBMS



- Remember "Store in DISK, Process in (faster) MEMORY"
- Files are moved between disk and main memory in **blocks**; it takes roughly 10 milliseconds
- It is vital that a disk block we are accessing is already in a buffer pool!

When a page is requested …

- A **page** is the unit of memory we request
- If *Page* is in the pool
  - Great! No need to go to disk.
- If not? Choose a frame to replace …
  - If there is a free frame, use it.

    We pin a page = it's in use.
  - If not? We need to choose a page to remove.

  How DBMS makes choice is a **replacement policy**

Goal: minimization of disk access.

Buffer Replacement Policies (cont.)

- **First-In-First-Out** (FIFO):
  - Rule: The oldest arrival page in buffer pool will be replaced first
  - Very simple, intuitive and less maintenance than LRU, but perform poorly in practice ☹
- **Clock-sweep**:
  - Keep circular list of pages in memory. Each with a 0/1 flag.
    A *hand* points to the last examined page frame in the list.
    If replacement is needed, the flag at the hand's location is inspected.
    - If 0, the new page is put in place, and advanced the hand one position.
    - Otherwise, the flag is set to 0, then the clocked move until a page is replaced.
  - A more efficient version of FIFO (no need to constantly push page to the back of the queue list.)

Buffer Replacement Policies

- **Least recently used** (LRU):
  - Blocks referenced recently are likely to be used again.
  - Maintain a table to indicate the last time the block in each buffer has been accessed
  - Rule: Throw out the block that has not been read or written for the longest time.
- **Most recently used** (MRU): …

File Organization

Recap: Data Storage Principles

- Database relations are implemented as **files** of **records**.
- The storage medium are **disks**, which consist of **pages**
- Pages are *read* from disk and *written* to disk: *high cost operations!*
- Each record has a **record identity** (rid), which identifies the *page* where it is stored and its *offset* on that page
- The DBMS reads (and writes) entire pages and stores a number of them in a **buffer pool**
- The **buffer manager** decides which pages to load into the buffer

File Organization: *issues*

- **File** format: how are the pages organized in a file
- **Page** format: how are records organized in a page
- **Record** formats: how are attributes organized in a record

Files must support operations like:

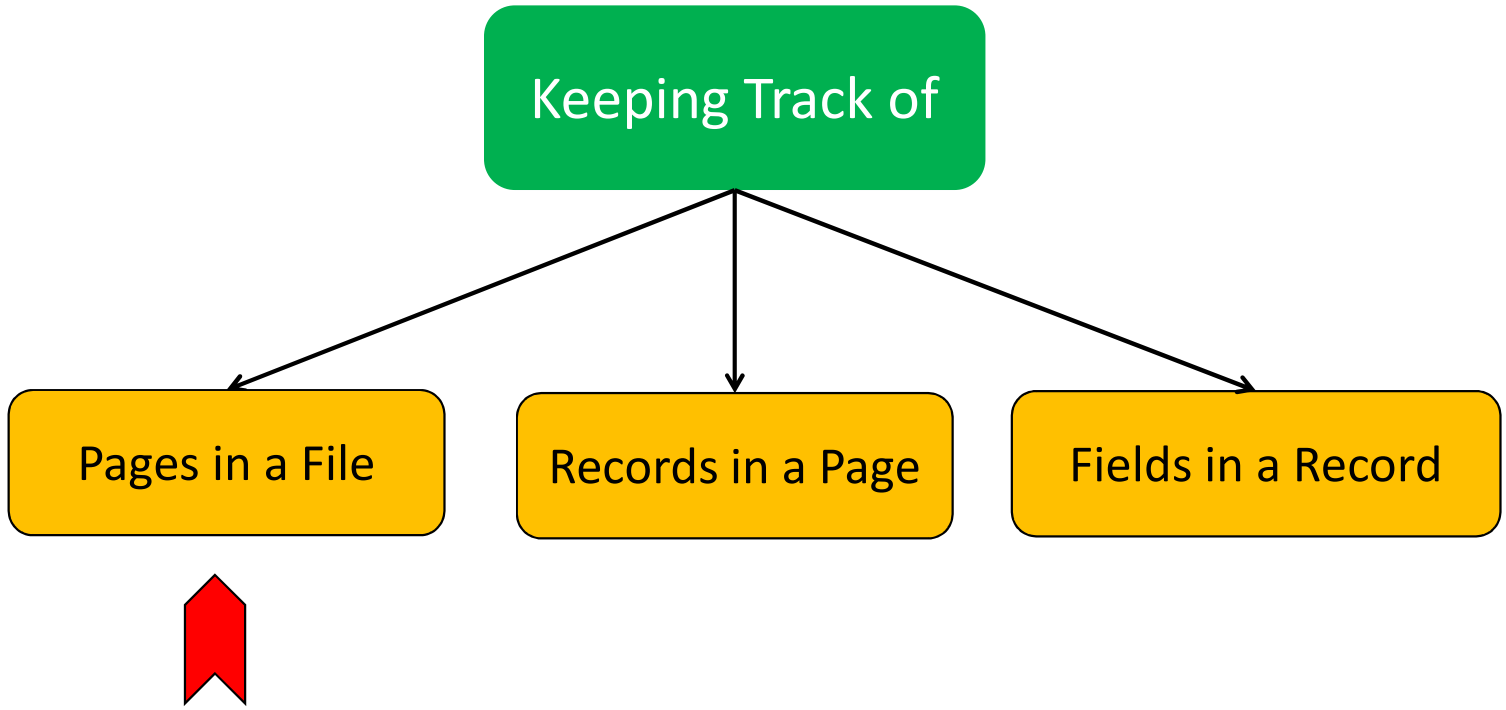
- `INSERT`/`DELETE`/`MODIFY` records
- `READ` a particular record
- `SCAN` all records (possibly with some condition)

File Format – Heap Files

- Data in heap file is **not ordered**.
- Database is stored as a collection of **files**.
- As a heap file grows and shrinks, disk pages are allocated and de-allocated.

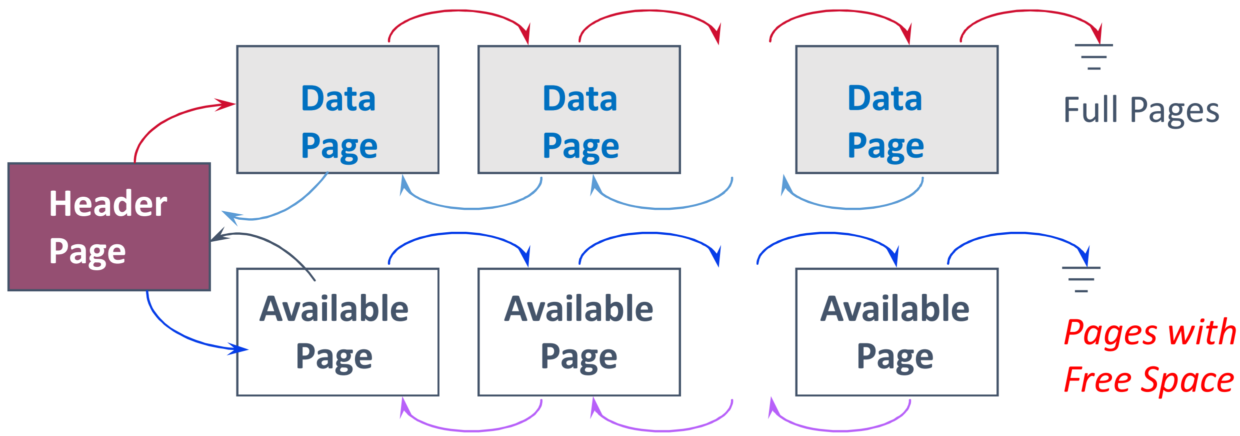
- To support record level operations, we must
  - Keep track of the pages in a file
  - Keep track of the records on each page
  - Keep track of the fields on each record

Supporting Record-Level Operations



Heap Files using *Linked-Lists* of Pages

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



The head page, $\langle heap\_file\_name, page\_addr \rangle$, is stored in a known location on disk.

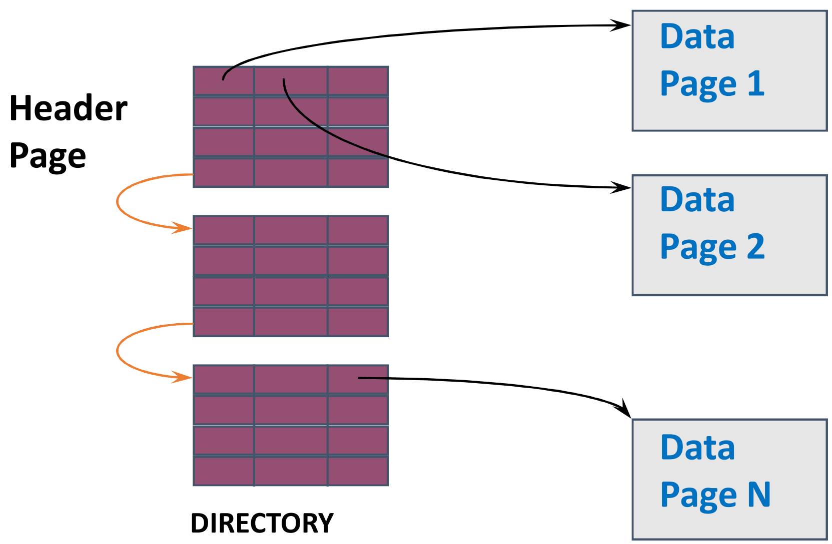
Each page contains 2 pointers plus data

Heap Files using *Linked-Lists* of Pages

- It is likely that 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:

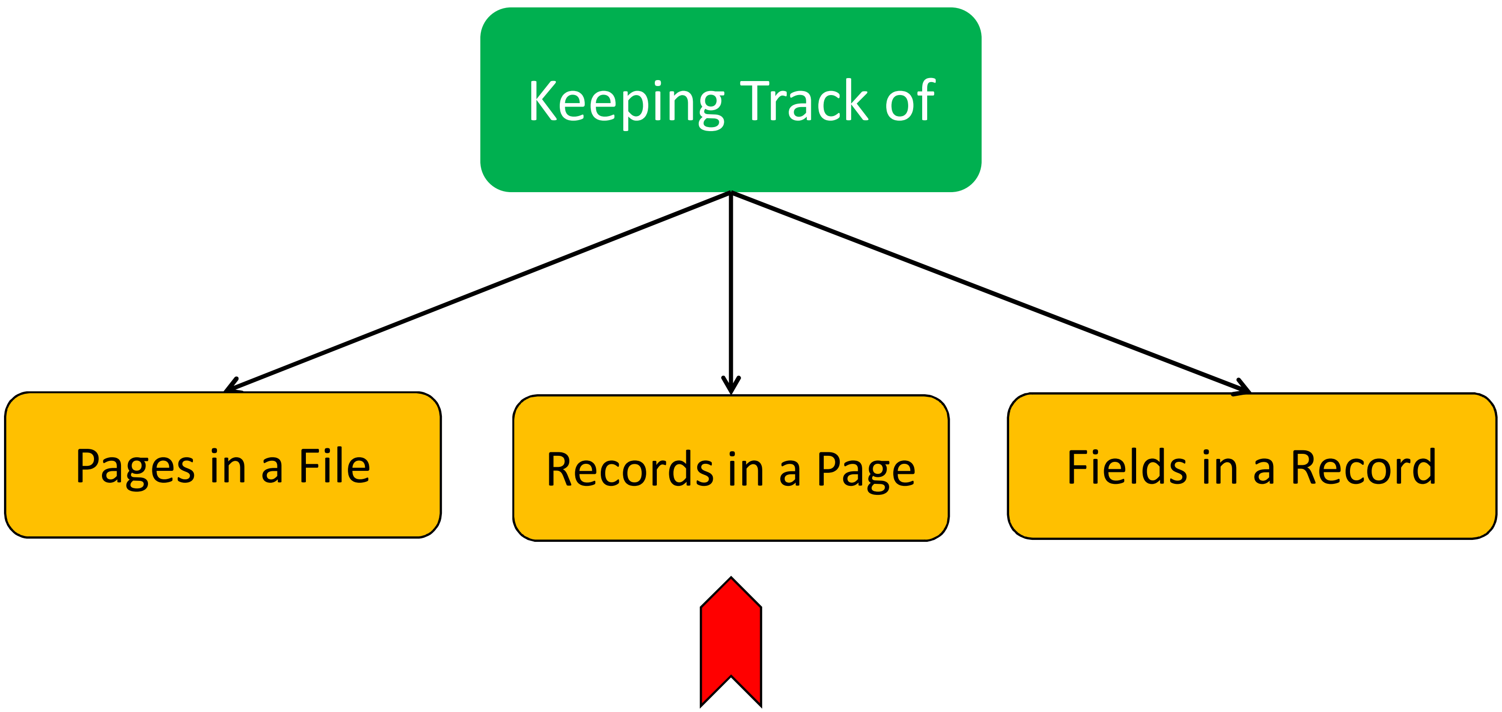
- A **bit** per entry (indicating whether the corresponding page has any free space)
- A **count** per entry (indicating the amount of free space on the page)

Alternative File Formats

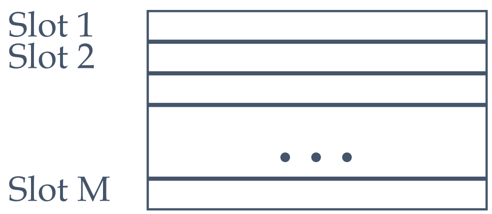
Alternatives are ideal for some situation, and not so good in others:

- **Heap Files**:
  No order on records. Suitable when typical access is a file scan retrieving all records.
- **Sorted Files**:
  Sorted by specific record field (or key).
  Best if records must be retrieved in some order, or only a range of records is needed.
- **Hashed Files**:
  File is a collection of **buckets**. (primary page + zero or more overflow pages)
  Hash Function h: compute $h(r) =$ bucket into which record r belongs.
  Good for equality search (have more trouble with range search).

Supporting Record-Level Operations



Page Formats



A page is a collection of **slots**, which contains records

Each record is identified (commonly) by

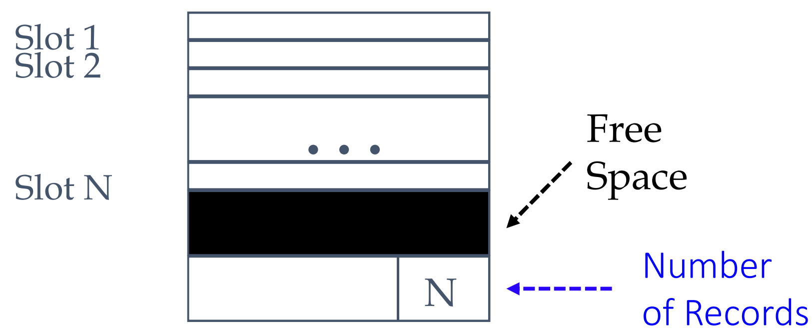
$$\mathrm{RID} = \langle page\_id, slot\_number \rangle$$

Issues to consider:

- 1 page = fixed size (e.g., 8KB)
- Records: fixed length, variable length

Fixed-length Records (cont.)

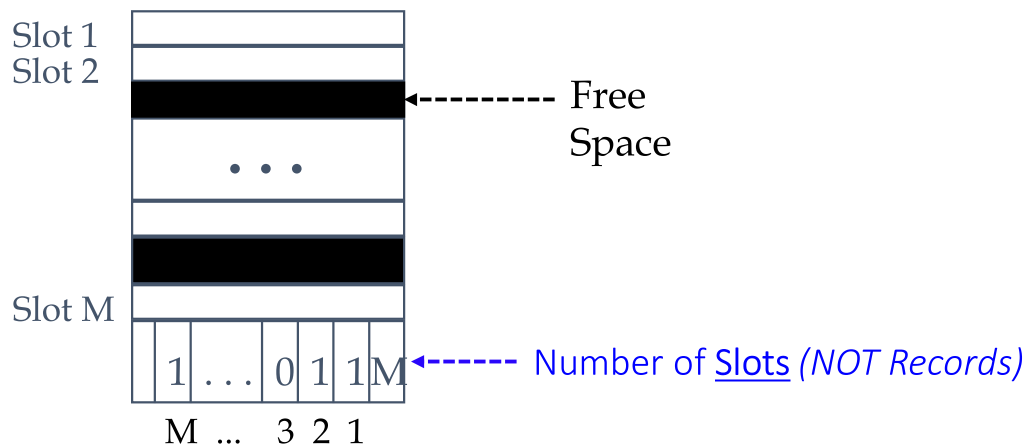
- When records are of fixed-length,
  slots become *uniform* and can be arranged *consecutively*



- Records can be located by simple offset calculations
- To delete a record, the record is moved into the vacated slot

Fixed-length Records (cont.)

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



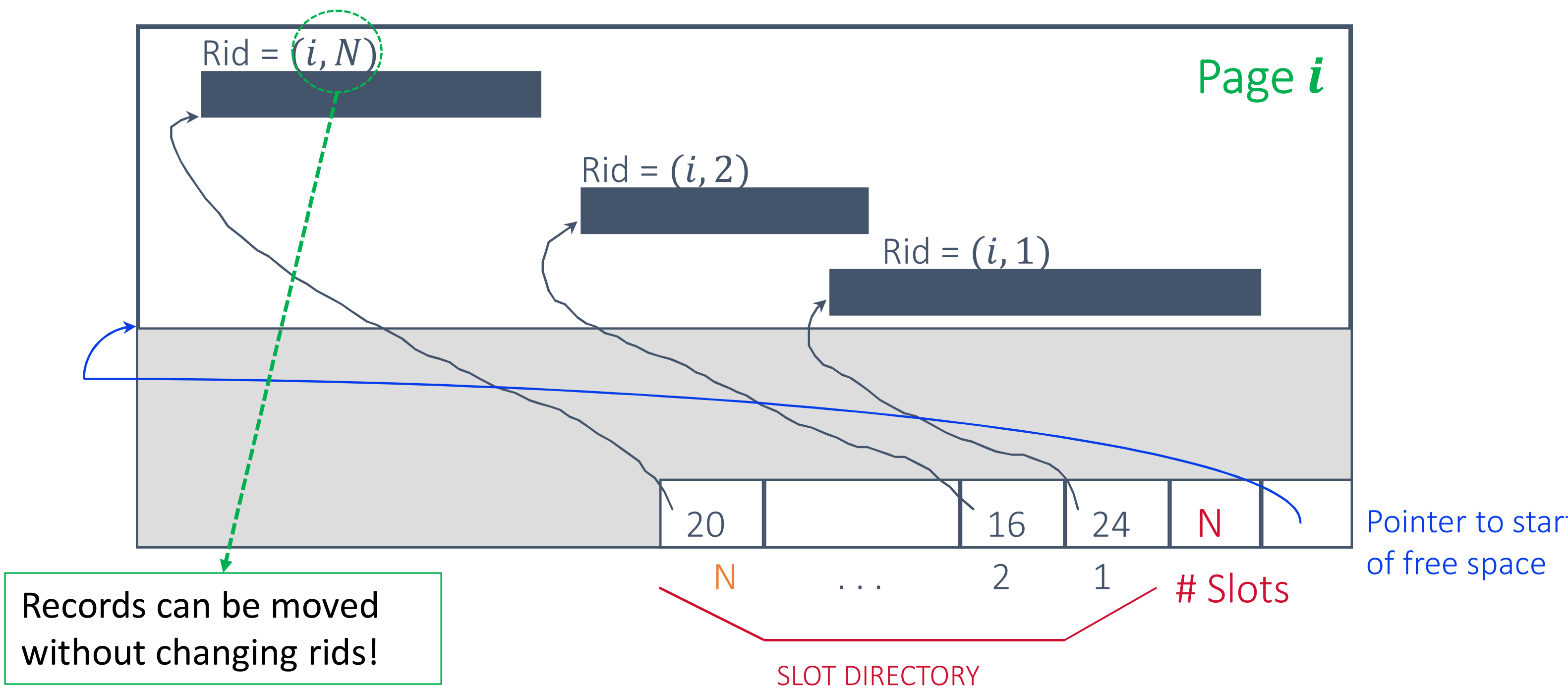
- When a record is deleted, its bit is turned off.
- Thus, the RIDs of currently stored records remain the same!

Variable-Length Records

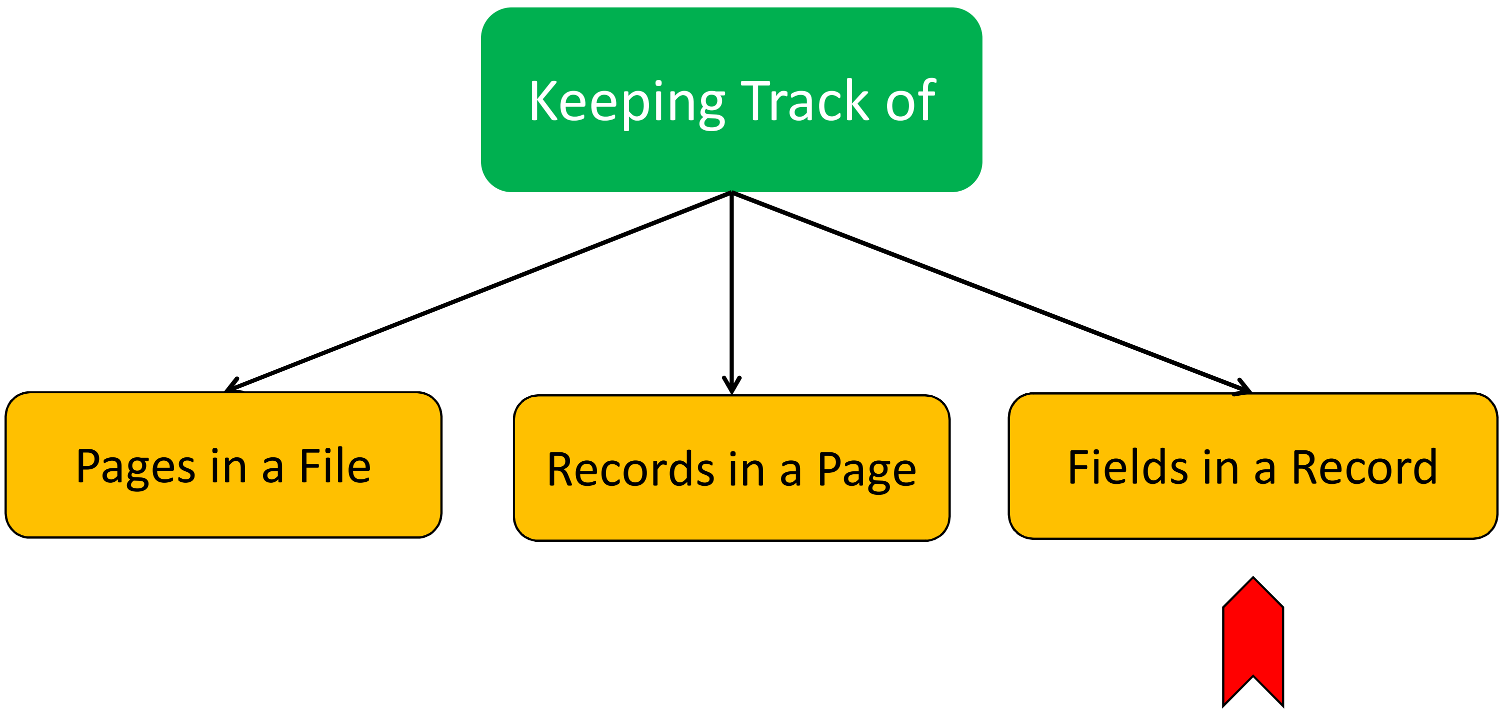
- If the records are of variable length,
  we cannot divide the page into a fixed collection of slots
- When a new record is to be inserted,
  we have to find an empty slot of “just” the right length
- Thus, when a record is deleted,
  we better ensure that all the free space is contiguous
- The ability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

Variable-Length Records (cont.)

Maintain a kind of **directory of slots** $\langle record\_offset, record\_length \rangle$ for each page



Supporting Record-Level Operations



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

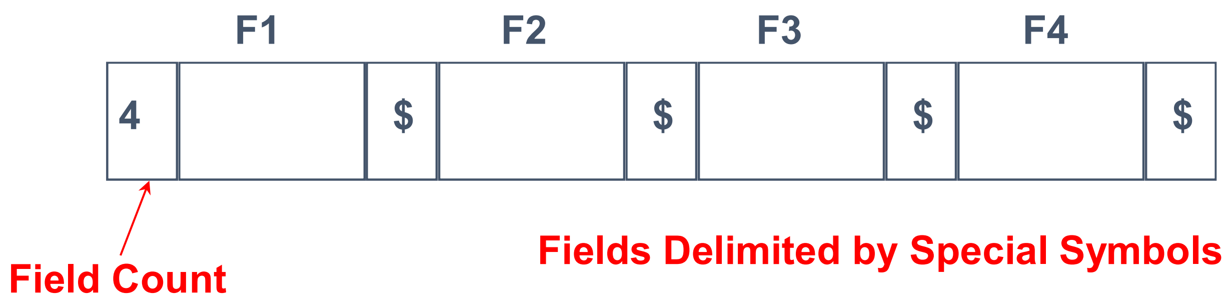
Fixed-Length Fields

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

Variable-Length Fields

There are two possible organizations to store variable-length fields

1. Consecutive storage of fields separated by delimiters

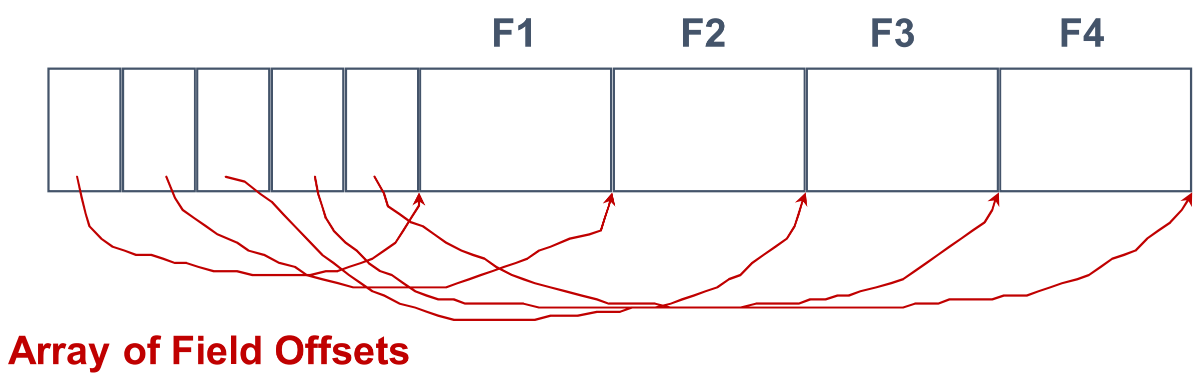


This entails a scan of records to locate a desired field!

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 more efficiently!

Representing Data

How to lay out a tuple (=record)

```
CREATE TABLE Product (
      pid INT PRIMARY KEY,
      name CHAR(20),
      wholesale BIT,
      description VARCHAR(200));
```

pid	name	wholesale	description
4 B	21 B	1 bit	200 B

First guess

How to lay out a tuple (=record)

```
CREATE TABLE Product (
      pid INT PRIMARY KEY,
      name CHAR(20),
      wholesale BIT,
      description VARCHAR(200));
```

because it is too slow to parse things that don't align with boundaries

pid	name	wholesale	empty space	description
4 B	21 B	1 bit		200 B

Second guess

How to lay out a DB page (= block)

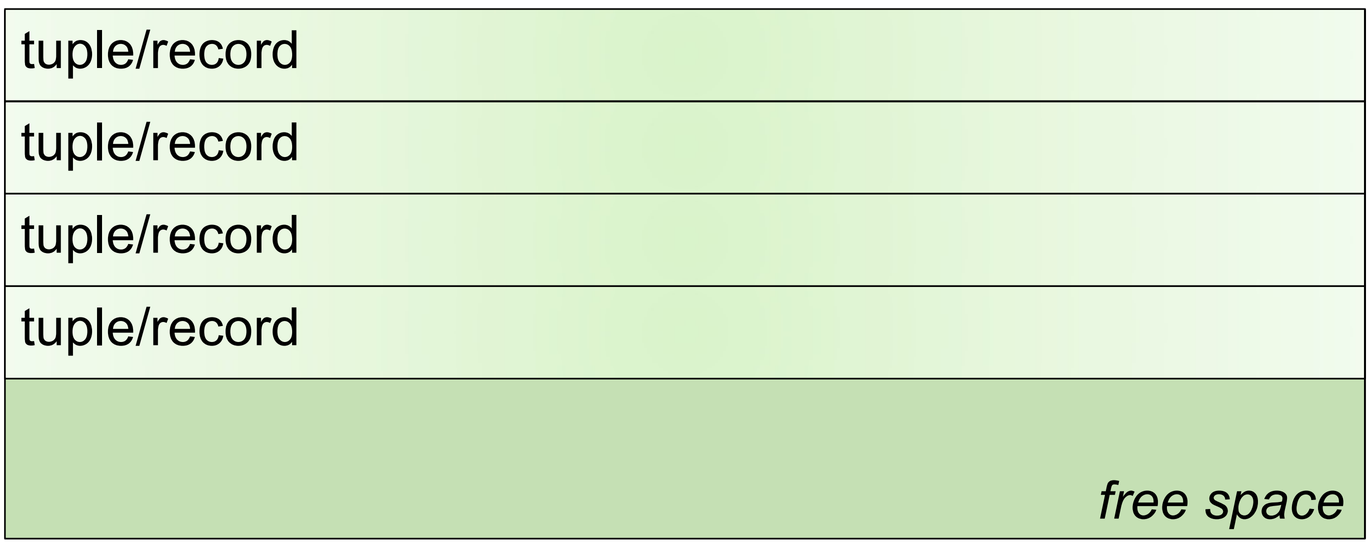
DB page/block = multiple of disk block size

In practice, 8 KB or more

How to lay out a fixed-length records

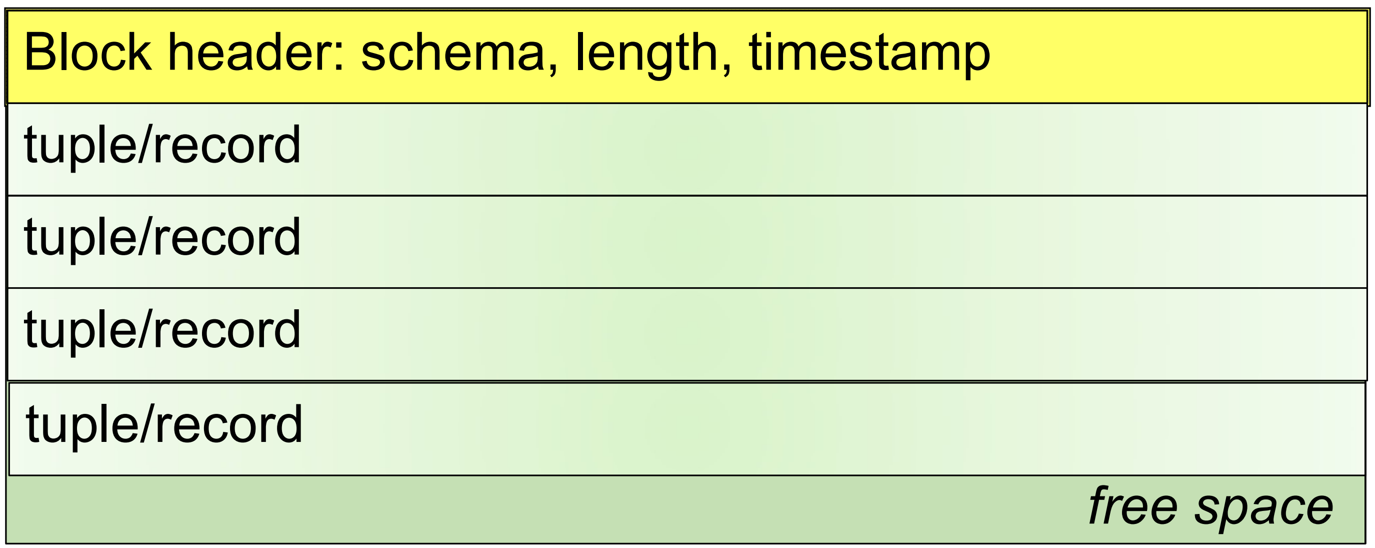
We know neither the length of each record or the size of each field in it

First attempt



How to lay out a fixed-length records

Second attempt



How to lay out a (fixed-length) records

Variable-length records can be handle using offset + record length (as mentioned earlier)

Second attempt (with detail)

Block header: schema, length, timestamp					
pid	name		wholesale	description	
pid	name		wholesale	description	
pid	name		wholesale	description	
pid	name		wholesale	description	
					free space

What about tuples bigger than a page?

spanned **tuples**