COMP322003 Introduction to Database (Fall 2018)

PHYSICAL DATABASE DESIGN

Chapters 16

- Disk Storage, Basic File Structures

Prof. Young-Kyoon Suh

Chapter Outline

- Physical Database Design
  - Storage Hierarchy
  - Disk Architecture
  - Basic File Structure: Primary File Organization
    - Heap file
    - Sorted file
    - Hash file (if time)

Introduction

- Databases typically stored on magnetic disks
  - Database are accessed using physical database file structures.
- The process of physical database design involves choosing the particular data organization techniques.
  - Best fitting the requirements of application among a variety of options.
- In today and next classes we study the following things:
  - The *primary organization of databases* in storage and
    - How to physically store records in a file
  - The *techniques for accessing* them efficiently using various algorithms.
    - How to access the records more efficiently
    - Some algorithms require auxiliary data structures, called *indexes*.

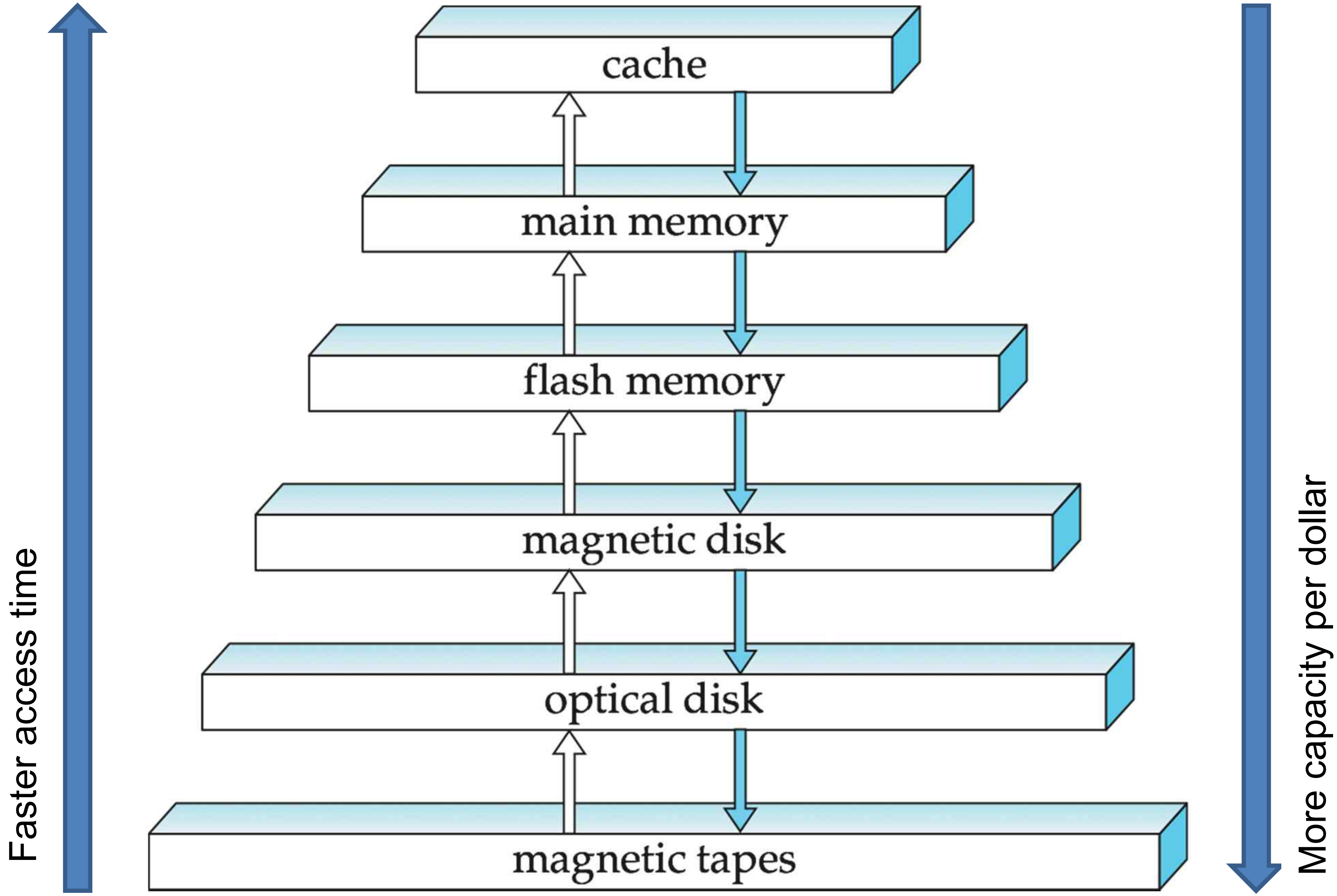
Introduction -

Storage Hierarchy

- The collection of data organizing database must be stored physically on some computer storage medium.
  - The DBMS (e.g., Oracle) software can then retrieve, update, and process this data as needed.
- Computer storage medium form a *storage hierarchy*:
  - **Primary storage**: operated by central processing unit (CPU)
    - CPU cache memory, RAM (random access memory),
  - **Secondary storage**: non-volatile, cheaper than primary
    - Hard disk drives (HDD), (USB) flash memory, solid-state drives (SSDs)
  - **Tertiary storage**: cheapest, but slowest
    - Removable media: tapes, optical disks (CD-ROMs, DVDs, …)

Introduction -

Storage Hierarchy (Cont’d)



Introduction -

Storage Types and Characteristics

Type	Capacity*	Access Time	Max Bandwidth	Commodity Prices (2014)**
(Cache memory	12MB	0.5-2.5ns	45GB/s	$500-$1000 **per GB**)
Main Memory- RAM	4GB–1TB	30ns	35GB/sec	$100–$20K
Flash Memory- SSD	64 GB–1TB	50µs	750MB/sec	$50–$600
Flash Memory- USB stick	4GB–512GB	100µs	50MB/sec	$2–$200
Magnetic Disk	400 GB–8TB	10ms	200MB/sec	$70–$500
Optical Storage	50GB–100GB	180ms	72MB/sec	$100
Magnetic Tape	2.5TB–8.5TB	10s–80s	40–250MB/sec	$2.5K–$30K
Tape jukebox	25TB–2,100,000TB	10s–80s	250MB/sec–1.2PB/sec	$3K–$1M+

*Capacities are based on commercially available popular units in 2014.

**Costs are based on commodity online marketplaces.

Magnetic tape

IBM System Storage TS3500 Tape Library,

Introduction -

Storage Organization of Databases

- *Persistent data* (nonvolatile)
  - Most databases typically store large amounts of data persisting over long periods of time.
  - Stored on secondary storage on magnetic and/or SSD disks. Why?
    1) In general, DBs are too large to fit in main memory in its entirety.
    2) Nonvolatile; don't disappear after power off
    3) The cost of storage per unit of data: 10x cheaper than primary storage.
  - Portion of data is loaded from disk into RAM for processing and rewritten to the disk.
  - The data stored disk is organized as files of records.
    - Each record is a collection of data values representing facts about ER.
    - Records should be **well-placed** on disk for **efficient access**.
- C.f. *Transient data* (volatile): contrasts with persistent data.
  - Exist only during program execution; e.g., `malloc()`

Introduction -

Storage Organization of Databases (Cont'd)

- **Primary file organizations** determine:
  - How the file records are *physically placed* on the disk, *and hence*
  - *How the records can be accessed*.

Primary file organization

- ***Heap file*** (unordered file)
  - Places records on disk in NO particular order by appending new records at the end of the file
- ***Sorted file*** (sequential file)
  - Keeps the records order by the value of the sort key field
- ***Hashed file***
  - Uses a *hash function* applied to the hash key field to determine placement on disk
- ***Tree-structured file*** (B-trees)
- ***Column-based file***

- **Secondary organization** (or *auxiliary access structure*)
  - Allows efficient access to file records based on alternate fields different from those that have been used for the primary file organization.
  - Most of these exist *indexes*.

SECONDARY STORAGE DEVICES

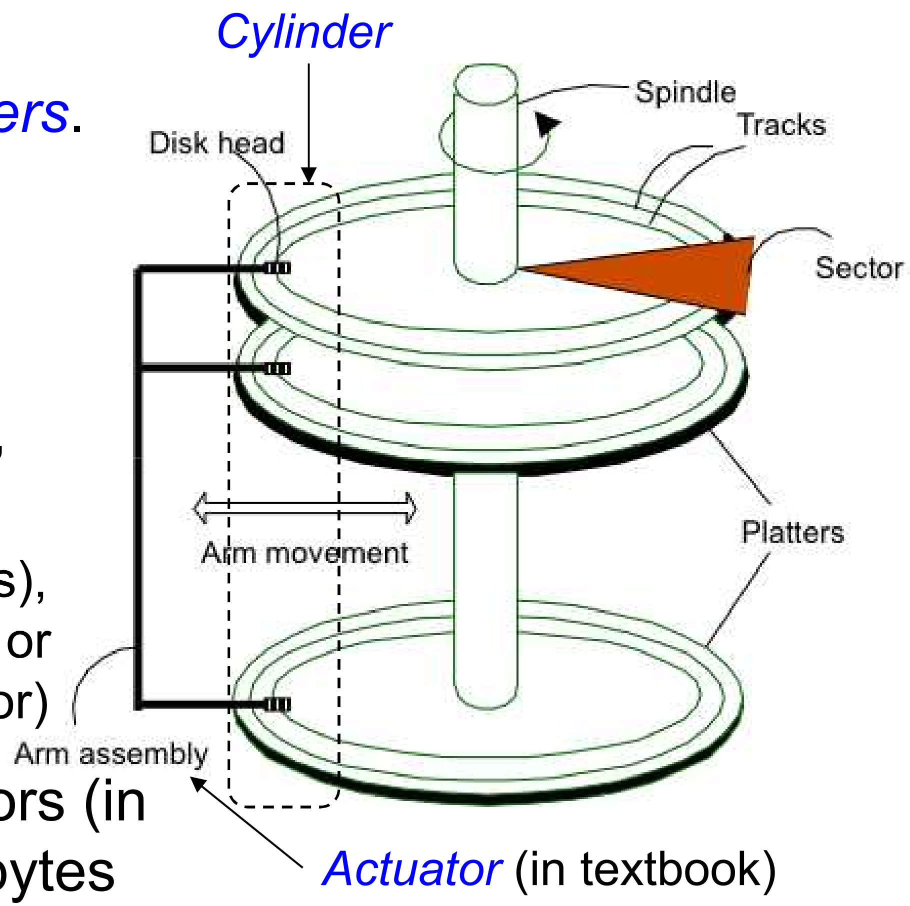
Chapter 16.2

Disk Storage: Nonvolatile, rotating magnetic storage

- A disk: a random access addressable device
  - Transfer of data from/into RAM in unit of disk blocks
  - A block (or sector) address consists of
    - A cylinder number, a track number (within the platter), a block number (within the track) (or, a sector number (within the block))
  - The block address is supplied to the disk I/O controller.
  - In many modern disk drives, a single number called LBA (logical block address) is mapped automatically to the right block by the controller.
- *Disk read*
  - The desired disk block with an LBA is copied into the disk drive *buffer*.
- *Disk write*
  - The contents of the buffer is copied into the disk block with an LBA.

Disk Storage: Internal Architecture (Cont'd)

- *Disk* is a stack of magnetic *platters*.
- The surface of each platter is organized into many *tracks*.
- Tracks divided into *sectors*.
- *Sector*: the basic unit of storage, consisting of
  - *Sector ID*, *data* (typically, 512 bytes), *Error-correction code*, sync. fields, or gaps (the sector # of the next sector)
- *Block* (or *page*): a group of sectors (in unit of read/write), typically, 4K bytes



- The disk heads move to the desired tracks (in the same *cylinder*) while the *spindle* rotates to locate desired sectors.

* Most disk controllers with *a built-in cache* for better performance (of what?).

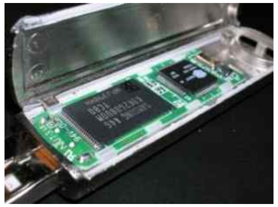
Disk Access Time

- Three major components
  - *Seek time*: taken to move the disk head to the desired track
  - *Rotational delay*: taken to wait for the desired sector to rotate until it comes under the (read/write) head
    - Usually assumed to *be half* (why is it so??) of the full rotation time
  - *Transfer time*: taken to transfer a disk block of bits
    - Function of the sector size, the rotation speed, and the recording density
    - 100 ~ 200 MB / sec in 2012
- Besides, there could be "more latencies" from *queuing delays* when other accesses exists and from *disk controller overhead*.
- Example: 512B sector, 7,200rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, <u>idle</u> disk
  - Q: What would be the expected "read" (or "write") time of a sector on this disk (즉, 한 섹터 읽(쓰)는데 걸리는 평균 시간)?

Techniques for Efficient Data Access

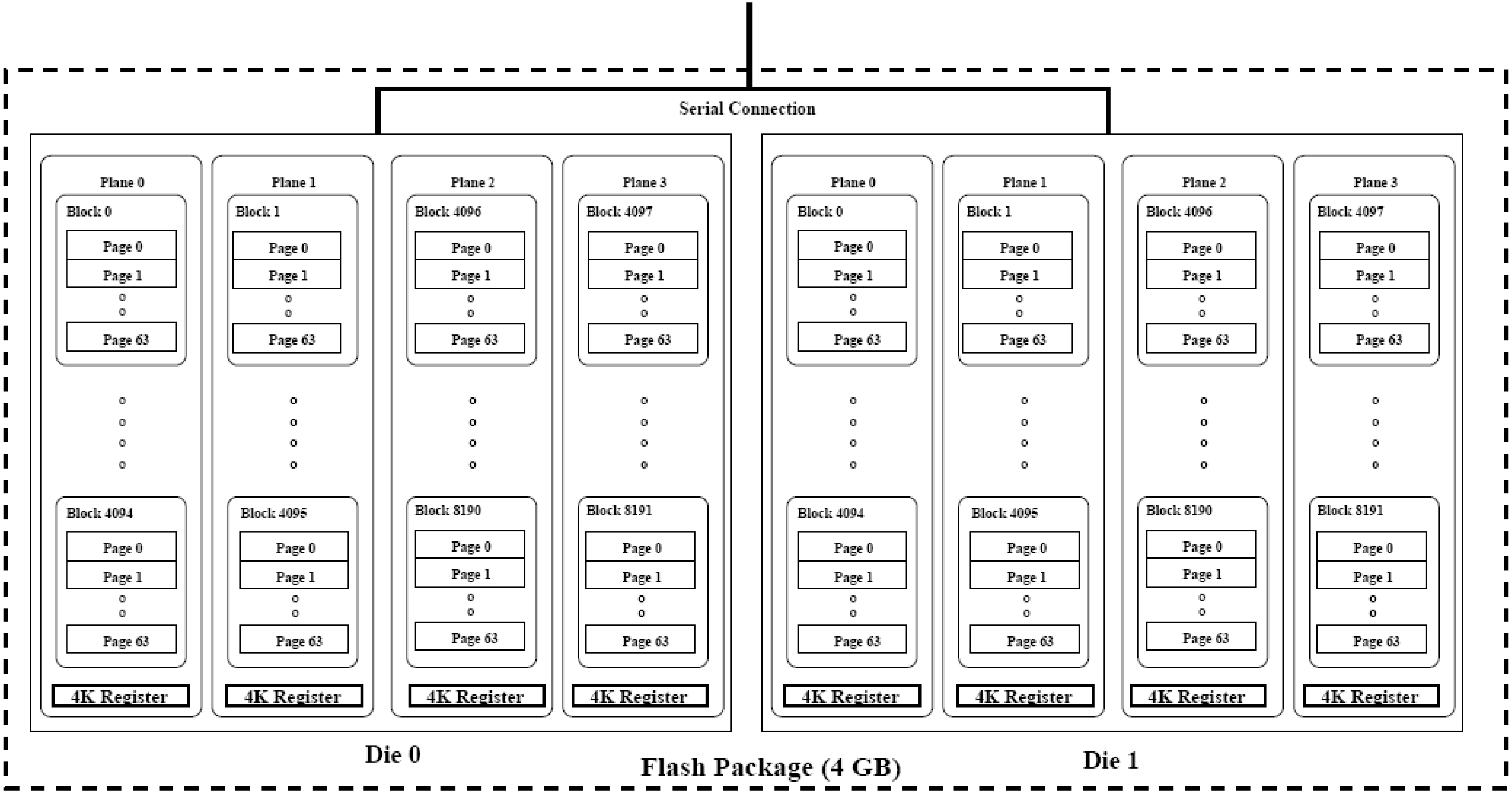
- Buffering of data (in main memory)
  - New data can be held in a buffer while old data is in processing.
- Proper organization of data on disk
  - Keep related data on contiguous blocks; place data blocks close to head
- Reading data ahead of request: called *prefetching*
  - For a disk read, blocks from the rest of the track can also be read.
- Proper scheduling of I/O requests
  - Aims at minimizing total access time;
  - Arms moves only in one direction: called *the elevator algorithm*
- Uses of log disks to temporarily hold writes
  - All blocks to be written go to one log disk to eliminate seek time.
- Use of SSDs: no latency of mechanical parts but expensive

Flash Memory


- Type of *E*lectrically *E*rasable *P*rogrammable *R*ead-Only *M*emory (EEPROM)
  - Non-volatile semiconductor storage with **NO** moving mechanical part
    - 100× – 1000× faster than disk: *msec (ms)* vs. *microsec (us)*
    - *Exceedingly fast* read speed, *smaller*, *less* power, *more* robust
      - But more $/GB (between disk and DRAM)
- Two types over a flash cell
  - <u>NOR-based</u> flash: bit cell acts like a NOR gate
    - Providing (byte-addressable) random read/write access
    - More expensive, taking longer to erase and write new data
  - <u>NAND-based</u> flash: bit cell acts like a NAND gate
    - Denser (bits/area) thus more storage, but page(block)-addressable access
    - Cheaper per GB
- Disadvantages
  - *No in-place update*: asynchronous latency between read and write
  - *Limited lifetime*: cannot be written on one cell forever => *wear-leveling*

Flash Memory: SSD (Samsung 4GB)



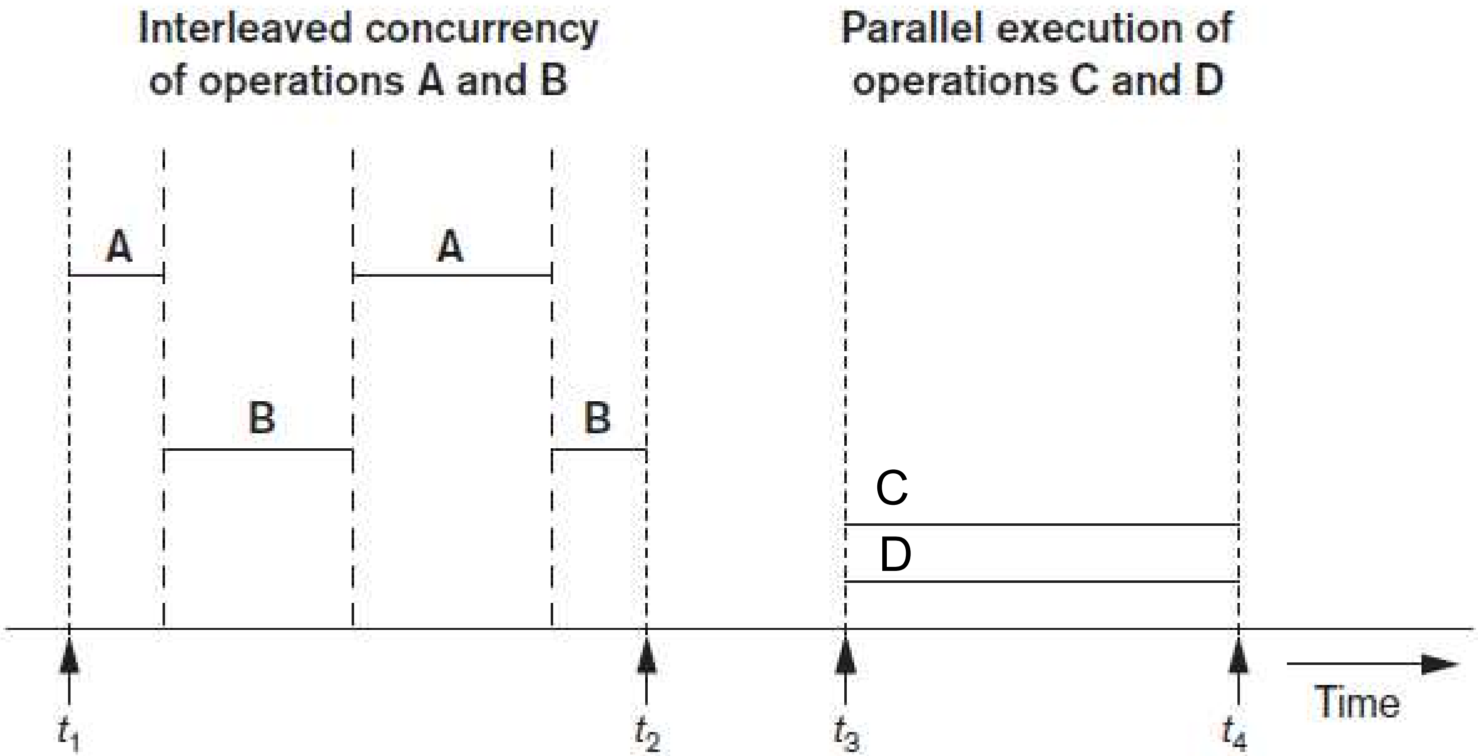
Die (4 planes) – Plane (4K blocks) – Block (64 pages) – Page (2K (currently, 4K or 8K))

BUFFERING OF BLOCKS

Chapter 16.3

- *Buffer* refers to a part of main memory available to receive disk blocks.

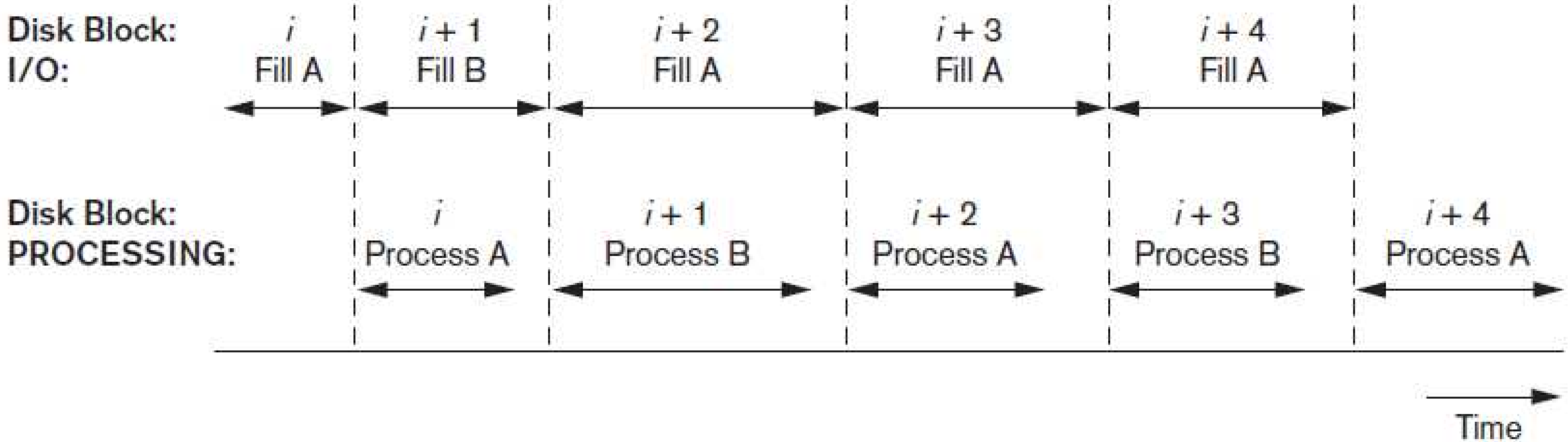
Interleaved Concurrency (on Single CPU) vs. Parallel Execution (on multi-CPUs)



- Buffering is most useful when processes can run concurrently in parallel. Why? See next.
  - Because a separate disk I/O processor available or multiple CPU processors exist

Concept of *Double Buffering*

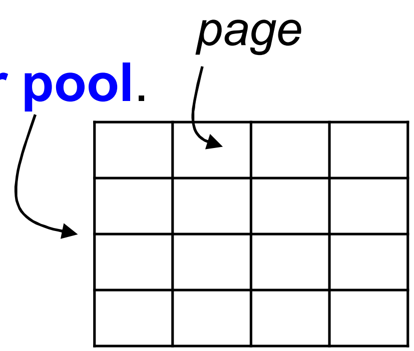
- Use of two buffers: ***A*** and ***B***, for reading disk



- Illustrates how reading and processing can proceed when the time for processing is *smaller* the time for reading the next block (I/O time)
- ***Double buffering***:
    - Once a block transfer is completed, CPU can start processing that block.
    - At the same time, the disk I/O controller (processor) can be reading and transferring the next block into a different buffer.
  - Can be used to read continuous streams of blocks.

Buffer Management

- *Buffer manager* of a DBMS
  - Responds to data request
  - Decides (i) *which buffer to use* and (ii) *what pages to replace in the buffer* to accommodate new pages* (or, disk blocks*)
  - Views the available main memory storage as a **buffer pool**.



  - Keeps <u>two types of information</u> about each page:

    1) ***Pin-count***: *# of times that page has been requested* (or *# of concurrent users on that page*); initially, 0

    - ***Pinning***: incrementing the pin-count
      - If it falls to 0, then ***unpinned***. Otherwise, its associated page cannot be evicted.

    2) ***Dirty bit***: Initially, 0, but set to 1 whenever that page is updated by program

* Page and block are interchangeably used.

When a Certain Page is Requested...

- The buffer manager checks if the requested page is already in a buffer in the buffer pool.
  - If the page exists, then the manager increments its pin-count and releases the page.
  - If NOT, then the manager does the following things:

    a) It chooses a page for replacement, using the replacement policy (to be discussed shortly), and increments its pin-count.

    b) If the dirty bit of the replacement page is ON, the manager writes that page to disk (by replacing its old copy on disk). (If the bit is OFF, then no need to write back to the disk. Why?)

    c) It reads the requested page into the space just freed up.

    d) The main memory address of the new page is passed to the requestor.
- If there is no unpinned page and the requested page is not available in the buffer pool, then the manager must wait (until a page gets available).

Buffer Replacement Strategies

- Popular buffer replacement strategies
  - *Least recently used (LRU)*
    - Throw out that page that has not been used for the longest time.
  - *Clock policy*: a round-robin variant of LRU
    - Finds a buffer with a flag with a value of 0 in round-robin fashion
      - Assume each buffer can have a value of 0 (*unused*) or 1 (*used*).
  - First-in-first-out (FIFO)
    - Will replace the page that has been occupied the longest
    - [Caution] A root block of index may be thrown out, but ….

PLACING FILE RECORDS ON DISK

Chapter 16.4

Records and *Record Type*

- **Record**: collection of "related" data values or items
  - Values correspond to *record field* (or tuple attribute)

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5

- **Record type** (**format**)
  - A collection of (i) field names and (ii) their corresponding data types

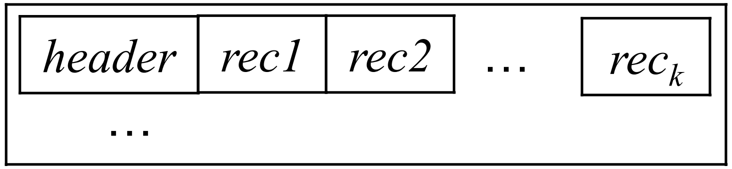
    Ex) An `EMPLOYEE` record type in C-style notation

    - Data types: numeric, string, boolean, date/time …
    - Binary large objects (BLOBs): unstructured objects (images, videos or audio stream)
      - A BLOB data item: stored separately from its record in a pool of disk blocks; a pointer to the BLOB kept in the record.
    - Character large objects (CLOBs): for storing free text

```
struct Employee{
  char fname[10];
  char minit[1];
  char lname[10];
  char ssn[9];
  char bdate[10];
  char address[50];
  char sex[1];
  int salary;
  char superssn[9];
  short int dno;
};
```

Files, *Fixed-Length Records*, and *Variable-Length Records*

- File: a *sequence* of records.



- Fixed-length records
  - Every record in a file has exactly the same size (in bytes).
- Variable-length records
  - Different records in the file have different sizes.
- Why variable-length records? Four reasons.
  1) One or more fields have variable-length: e.g., `VARCHAR`
  2) One or more fields are repeating. e.g.,
  3) One ore more fields are optional.
  4) File contains records of different types.

lastName	homePhone	workPhone	cellPhone	fax
Barnes	562-874-1234		310-999-3628	

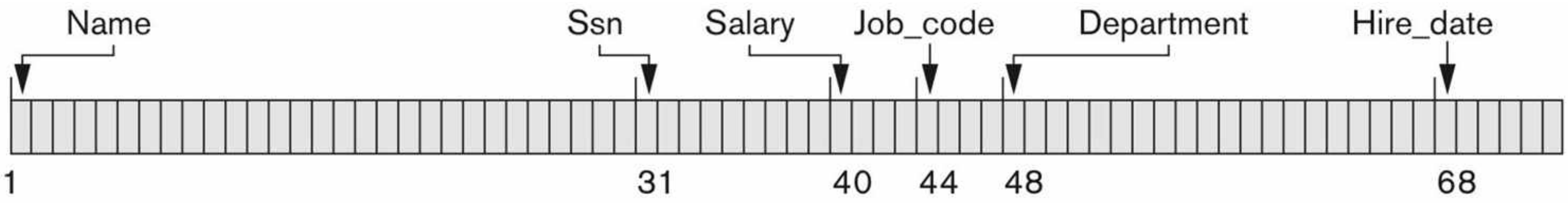
Repeating!

TRANSCRIPT

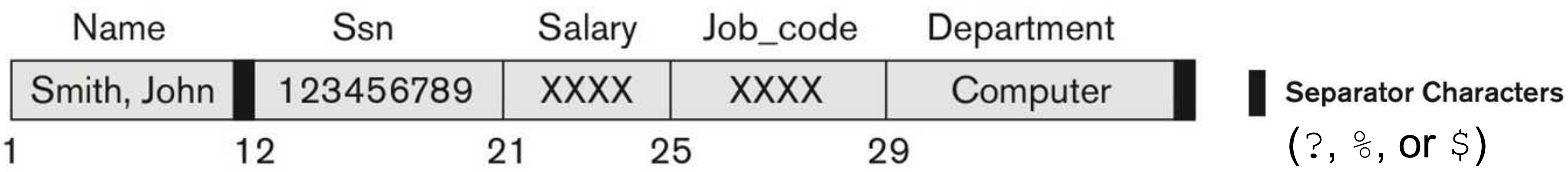
Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112

3 Options for Formatting Records of a File of Variable-length Records: EMPLOYEE

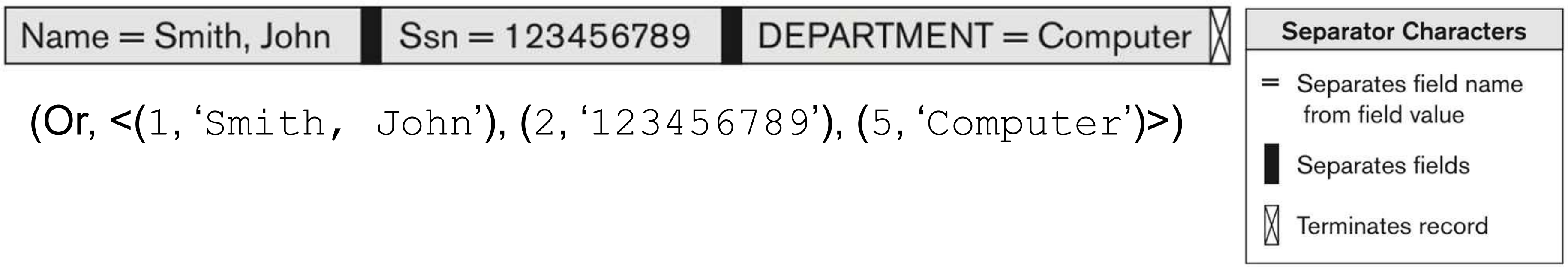
1) A fixed-length record with 6 fields and size of 71 bytes: easy location



2) A record with 2 variable-length fields and 3 fixed-length fields: for variable-length



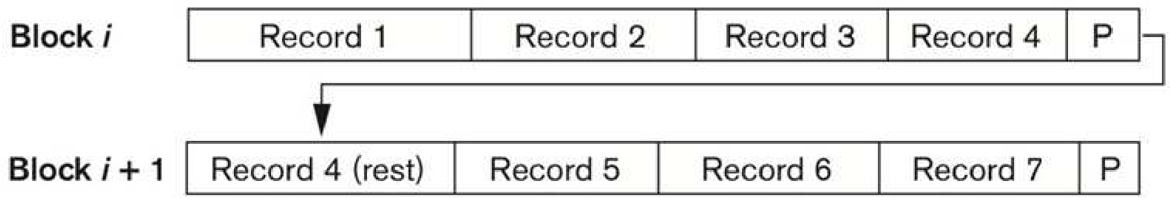
3) A variable-field record with 3 types of separator characters: with *optional fields*



(Or, <(1, 'Smith, John'), (2, '123456789'), (5, 'Computer')>)

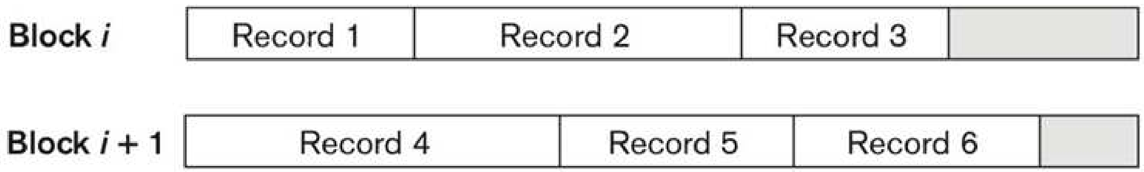
Record Blocking and Spanned vs. Unspanned Records

- The records of a file MUST be allocated to disk blocks.
  - Why? A block is the *unit of data transfer* between disk and memory.
  - When |block| >= |record|, each block will contain numerous records.
- *Spanned records*
  - Larger than a single block (page: 4K or 8K)
  - Pointer at end of first block points to block containing rest of record



Record 4: a spanned record

- *Unspanned records*
  - Used when a record is smaller than a block
  - not allowed to cross block boundaries



Record Blocking and Spanned vs. Unspanned Records (Cont'd)

- *Blocking factor*: avg. # of records per block for the file
  - Some unused space = $B - (bfr * R)$ bytes, where
    - B: block size, R: record size, and bfr: blocking factor for the file
  - bfr can be used to calculate # of blocks, say b, needed for a file of r records.
    - $b = ceiling((r / bfr))$ blocks, where
      - r: the number of records for the file
  - Q: How many blocks are needed to store a file of 1000 records?
    - Record size: 16 Bytes
    - Block size: 4 KBytes
    - Unused space: 0 Bytes (for convenience)

Allocating File Blocks on Disk

- Several standard techniques
  - Contiguous allocation: the file blocks are allocated to consecutive disk blocks
    - The whole file can be read fast by double buffering while being hard to make it expanded.
  - Linked allocation: each file block contains a pointer to the next file block.
    - Easy to expand but make it slow to read the whole file.
  - Cluster allocation: a combination of the two
    - Allocates clusters, or called *file segments* or *extents*, which are linked.
  - Indexed allocation: one or more index blocks contain pointers to the actual file blocks. (e.g., a hash file)

File Headers

- A file header (descriptor) contains information about a file needed by the system programs accessing the file records.
  - Which information?
    - To determine the disk addresses of the file blocks
    - To record format descriptions: e.g., field-lengths, the order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records

Database file

header				
record 1	123456789	John Smith	New York	5
record 2	234567891	Chris Young	San Diego	4
	..	…	…	…
record n	987654321	Christian Lee	Tucson	1

OPERATIONS ON FILES

Chapter 16.5

Operations on Files

- *Retrieval* operations
  - Do not change any data in the file
  - Only locate certain records so that their field values can be examined and processed
- *Update* operations
  - Change the file by insertion, or deletion of records or by modification of field values.
- In either case, we should select one or more records
  - Based on a selection (or filtering) condition
    - Specifying criteria that the desired record(s) must satisfy

File Organization (vs. Access Method)

- *File organization*
  - Refers to the organization of the data of a file into records, blocks, and access structures.
  - Includes the way records and blocks are placed on the disk and interlinked.
- How to organize records in files? (To be discussed in detail)
  - *Heap*: a record can be placed anywhere in the file where there is space
  - *Sequential*: stores records in sequential order, based on the value of the search key of each record
  - *Hashing*: uses a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
  - *Multitable clustering file organization*: records of each relation may be stored in a separate file.
    - A records of several different relations can be stored in the same file.
      - Motivation: store related records on the same block to minimize I/O

(File Organization vs.) **Access Method** (Cont'd)

- *Access method*
  - Provides a group of operations that can be applied to a file:
    - `Open`, `Find`, `FindNext`, `Read`, `Delete`, `Modify`, `Insert`, `Close`, `Scan`.
  - Possible to apply several access methods to a file organized using a certain organization.
  - Some access methods can be applied only to files organized in certain ways.
    - Ex) An indexed access method can't be applied to a file without an index.

FILES OF UNORDERED RECORDS (**HEAP FILES**)

Chapter 16.6

Head (or Pile) File

- Simplest and most basic type of organization
- Records are placed in the file in the order in which they are inserted.
  - New records are inserted at the end of the file.
- "Insertion" of a new record is very efficient.
  - The last disk block of the file is copied into a buffer.
  - The new record is added.
  - The block is then rewritten back to disk.
    - The address of the last file block is kept in the file buffer.
- But "searching" a record is *inefficient* due to a linear search.
  - On average, (b / 2), where b: # of blocks in a file.
  - In the worst, b file blocks will be visited.

Head (or Pile) File (Cont'd)

- One way to "delete" a record, a program must
  - First find its block,
  - Copy the block into a buffer,
  - Delete the record from the buffer, and finally,
  - Rewrite the block back to the disk.

Wasted storage space

- Another way: to use a deletion marker
  - An extra bit or byte is stored with each record.
  - A record is deleted by setting the marker to a certain value.
  - A different value for the marker indicates a valid record.
  - Search considers only valid records in a block.

$\Rightarrow$ Both approaches required periodic **reorganization** of the file to reclaim the unused space of deleted records; <u>packing</u> of existing undeleted records

- Third way: to use the space of deleted records for insertion

FILE OF ORDERED RECORDS (**SORTED FILES**)

Chapter 16.7

Sorted (Sequential) Files

- "Physically" order (sort) the records of a file on disk based on the values of one of their fields, called the ordering field.
  - *Ordering key*: if the ordering field is a key field of the file

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
Block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
⋮						
Block n–1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
Block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

Illustrates an ordered file with `Name` as the ordering key field (assuming names are distinct.)

Sorted (Sequential) Files (Cont'd)

- Advantages (over heap files)
  - Reading records by ordering key is extremely efficient.
    - **NO sorting required**; strong over a search/range condition on the key
  - Finding the next record requires **no additional block access** unless the record is the last one in the block.
  - Allows the **binary search technique** to be used for performing a search over the value of an ordering key, resulting in "faster access".

```
l = 1; u = b; //b: # of blocks
while (u >= l) {
   i = (l + u)/2;
   read block i of the file
   into the buffer.
   if K <(ordering key field
          value of the first
          record in block i)
    then u = i-1;
   else if K > (ordering key
           field value of
           the last record
           in block i)
    then l = i+1;
   else if K == (ordering key
                field value)
    then break; // found
   else break; // not found
}
```

Sorted (Sequential) Files (Cont'd)

- Deletion: utilize **pointer chains** or deletion marker
- Insertion: locate the position where the record is to be inserted
  - If there is free space, insert there
  - If no free space, insert the record in an **overflow** (or transaction) block
  - In either case, pointer chain must be updated.

Ordering key

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

32222	Verdi	Music	48000

- Need to reorganize the file from time to time to restore sequential order

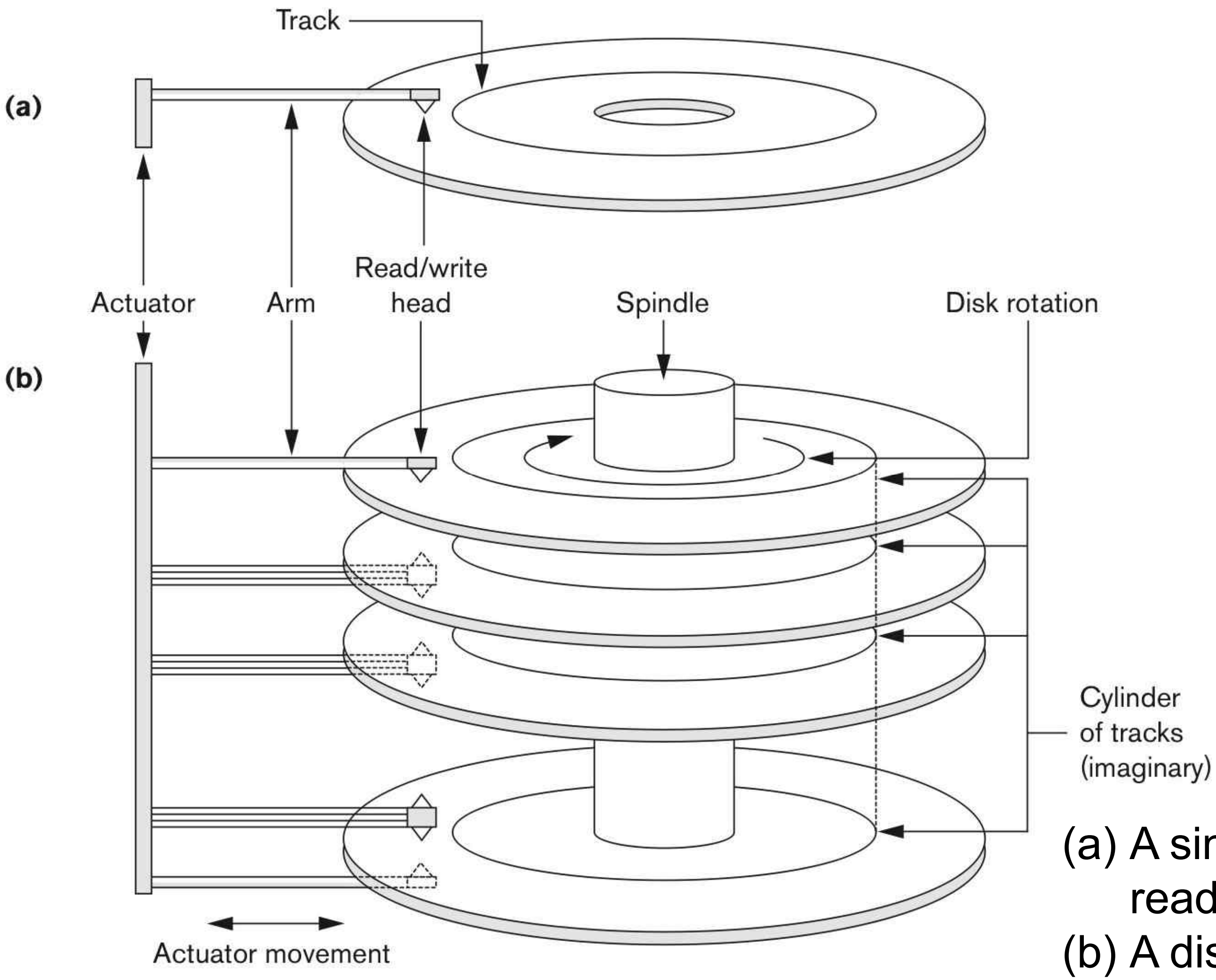
Sorted (Sequential) Files (Cont'd)

- Average access times for a file of b blocks under basic file organizations

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

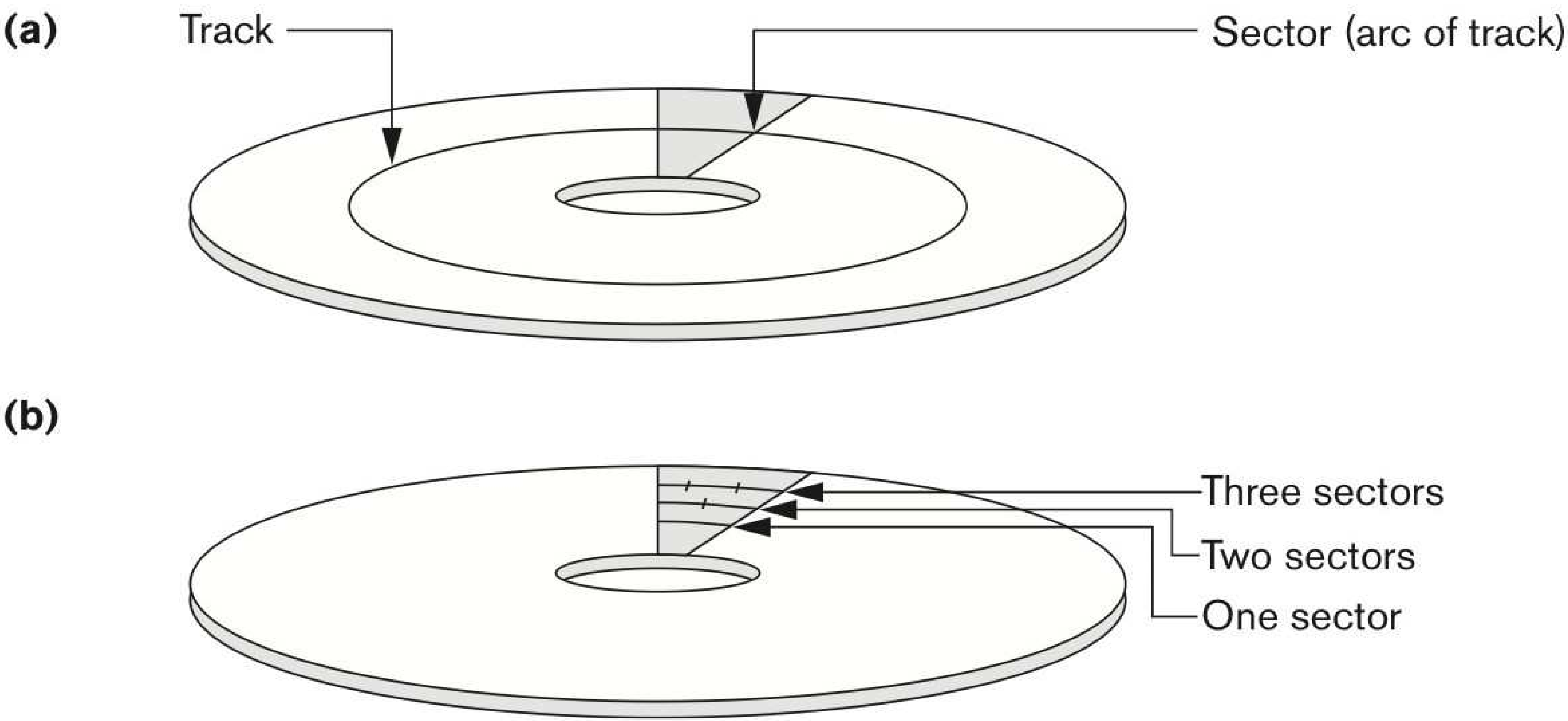
APPENDIX

Another Disk Internal



(a) A single-sided disk with read/write hardware.
(b) A disk pack with read/write hardware.

Different Sector Organizations on Disk



(a) Sectors subtending a fixed angle.
(b) Sectors maintaining a uniform recording density.