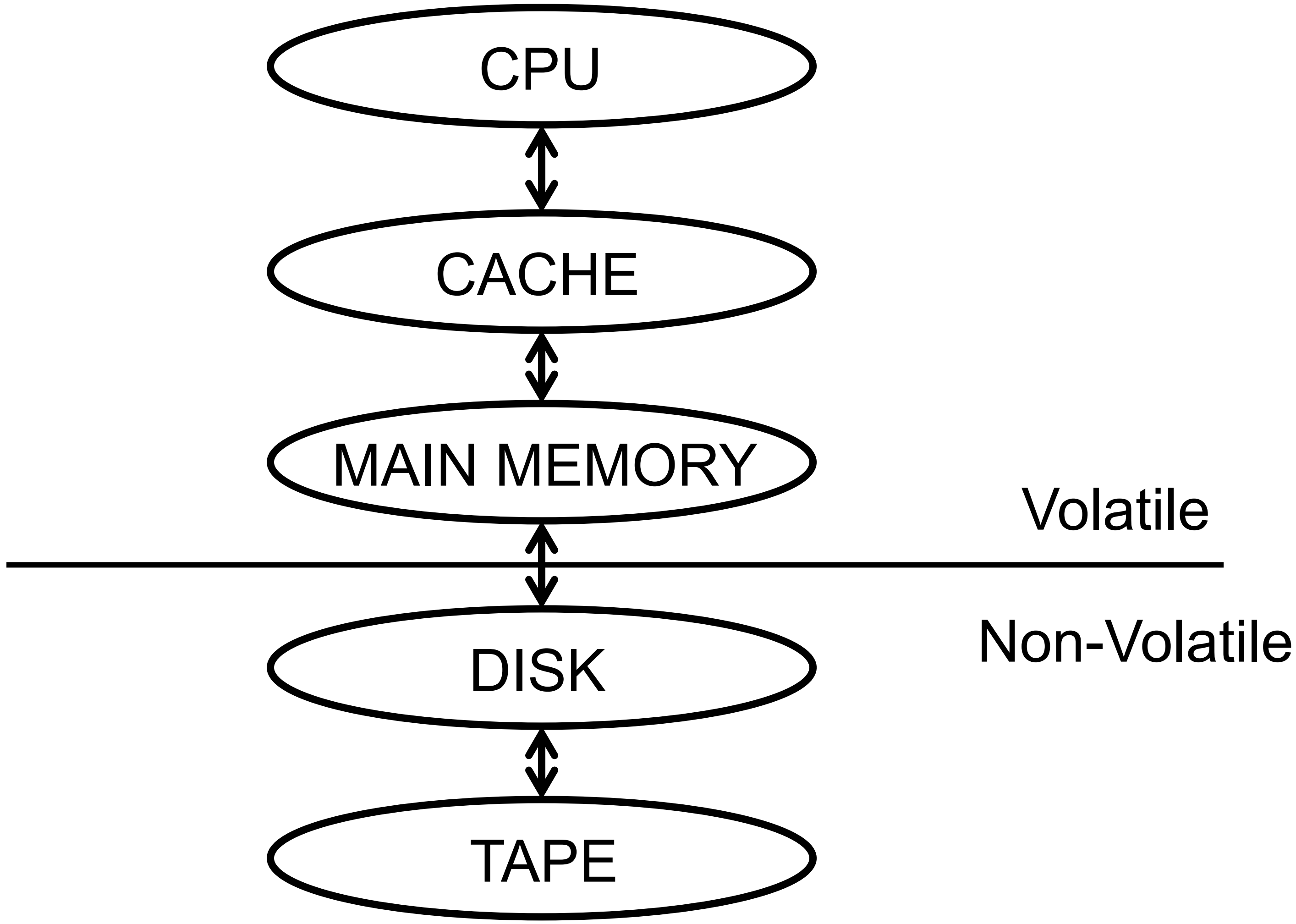
Disks and Disk Models

CS6083
CSE Department
NYU Tandon School of Engineering

NYU TANDON SCHOOL OF ENGINEERING

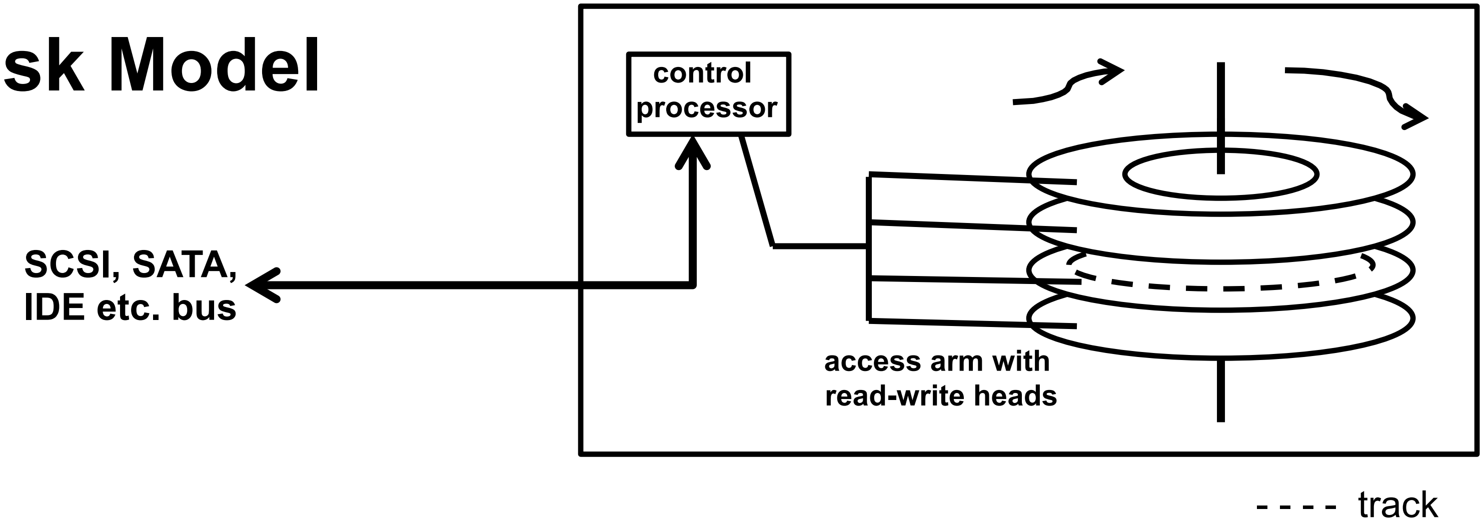
Disks and Disk Models

- Memory Hierarchy
- Hard Disks
- SSDs
- Modeling Disk Performance
- I/O – Efficient Sorting



Memory Hierarchy

Disk Model



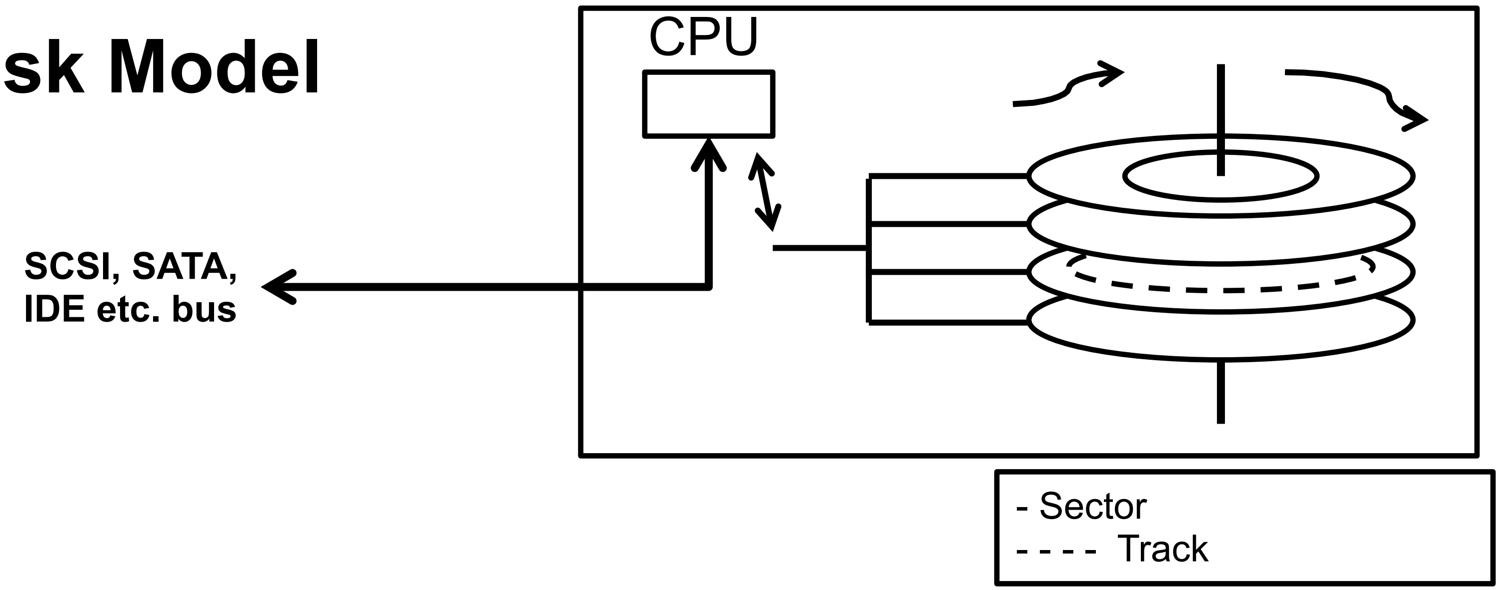
To read data, disk needs to:

- swivel access arm so head is over track holding data
- wait for start of data to rotate under head
- read all the data

Disk access latency: swivel + rotation

Max transfer rate: per rotation, all data that fits on one track

Disk Model

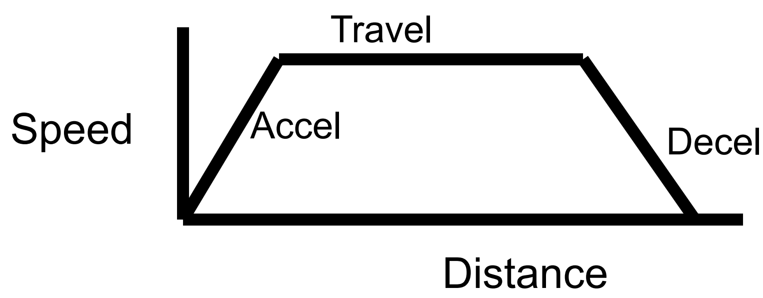


Files modeled as having sequential layout :

COST = SEEK + ROTATIONAL LATENCY + TRANSFER

More Details

Disk arm speed non-constant



Optimized track to track moves (1 – 2ms)

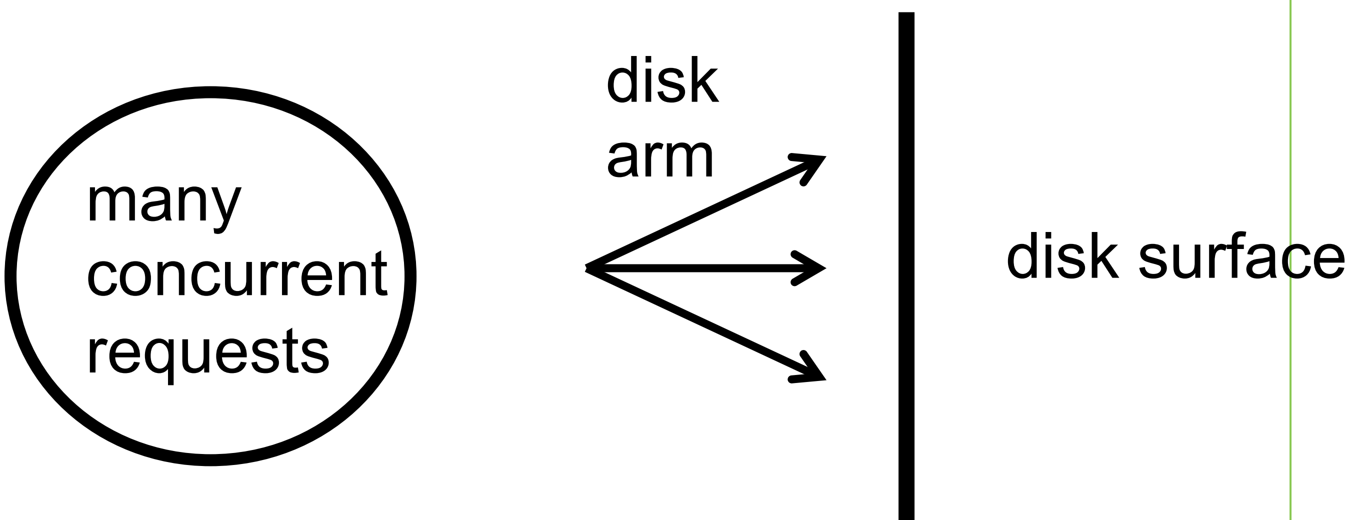
Buffering on disk, read–ahead

Bus contention (SCSI, master–slave on IDE, SATA)

More Details

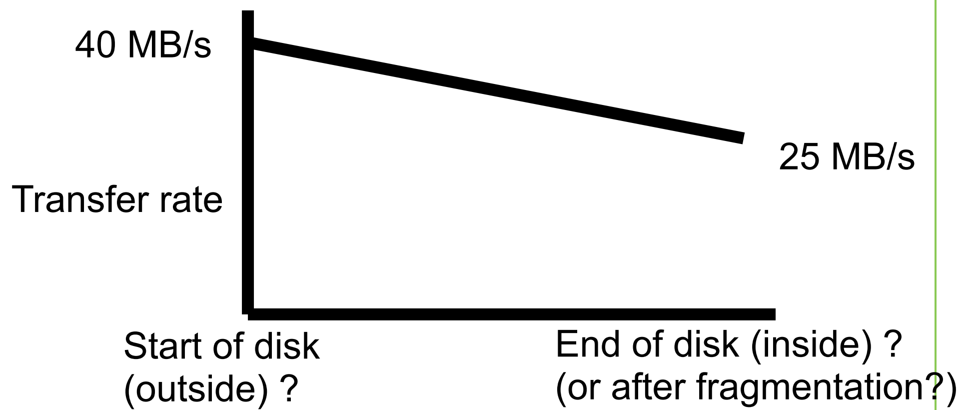
Directory lookups etc. may have significant costs

Elevator Algorithm **used under high load**



Example : SEAGATE BARRACUDA (circa 2003)

- 80GB CAPACITY
- 2 PLATTERS
- 7200 RPM → 120 RPS → 8.33 ms/rotation
- 2048 KB CACHE, IDE
- Access time 13.7ms



Today's disks:

- 70-120 MB/s (cheap SATA disks)
- 1-6TB capacity at $70-$200
- Access time between 5 and 10 ms

SSD: Solid State Drives

- Non-volatile, starting to replace hard disk in servers + laptops
- Still more expensive than hard drives (HDD)
- But getting cheaper: now as low as $200 per TB
- Much faster!!
- Transfer rate 200-500MB/s, <100 us per random access
- Big impact on large data and I/O-efficient computing

SSD: Solid State Drives (ctd)

- However, SSD drives are hard to model
- No simple model like the Block or LTR model for hard disks
- Random access still more expensive than sequential
- Also, SSD blocks only allow limited # of writes before fail
- File system needs to do smart allocation to avoid wearout
- Requires different file systems for best performance
- Some vendors combine hard disks and SSDs in one box

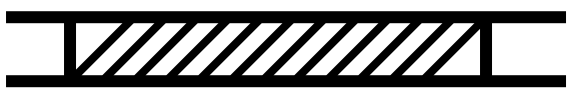
DISK PERFORMANCE MODELING

- Seek Time (5ms)
- Rotational Latency (5ms)

access time 10ms

- Transfer Rate (80MB/s)

- File of Length 400KB



Time to read : t_R = 10ms + 5ms = 15ms

- FILE of size 4KB (or 8, 16..)

  t_B = 10.05ms

NOTE: often have blocks of 4/8/16 KB

DISK MODELS

BLOCK MODEL: It takes *k* ms to read each block of size 4KB (or 8KB, or 16KB, but block size is fixed).

- **it takes 10*k* ms to read 40KB (even if in same file)**
- **access time counted for each block**
- **block oriented**

SEEK or LATENCY TRANSFER-RATE (LTR) MODEL:
It takes ACCESS TIME + TRANSFER TIME to read a file.

- **file oriented**
- **access time only counted once per file**
- **assumes file sequential on disk (sort of true)**

Seek model more precise when reading large chunks sequentially

For small random reads, no difference

- **EXAMPLE : 7200 RPM, 5ms SEEK, 80MB/s TRANSFER RATE**
  - **BLOCK MODEL : Block size 4KB**

$\Rightarrow$ **5ms + 1000/240 ms + 0.05ms**

$\Rightarrow$ **9.216ms to read block**

$\Rightarrow$ **100 * 9.21666 = 921.666ms to read 400KB file.**

B_1	B_2	B_3	B_4	. . . .	B_{100}

Note: Vast overestimate of the actual time needed!

Transfer time for
400KB at 80 MB/s

- **SEEK MODEL :**

⟹ **5ms + 1000/240 ms + 5ms**

⟹ **14.16ms to read 400KB file**

Much more realistic!

Why is the block model still popular?

- **Simpler? (no reasoning about data layout involved)**
- **OK if mostly small random reads and writes (transactions)**
- **Not good for search engines and data mining workloads**

DISK WORKLOAD EXAMPLES

- **TRANSACTIONS: Many small reads and writes (typical DB transaction workload): block or seek model fine**
- **OLAP, SEs, and Scientific: reading and writing large files, block model not good at all → use seek model**
- **INTERACTIVE (e.g., UNIX users) : Many repeated reads, few writes. Caching works. Motivation for log structured file systems**

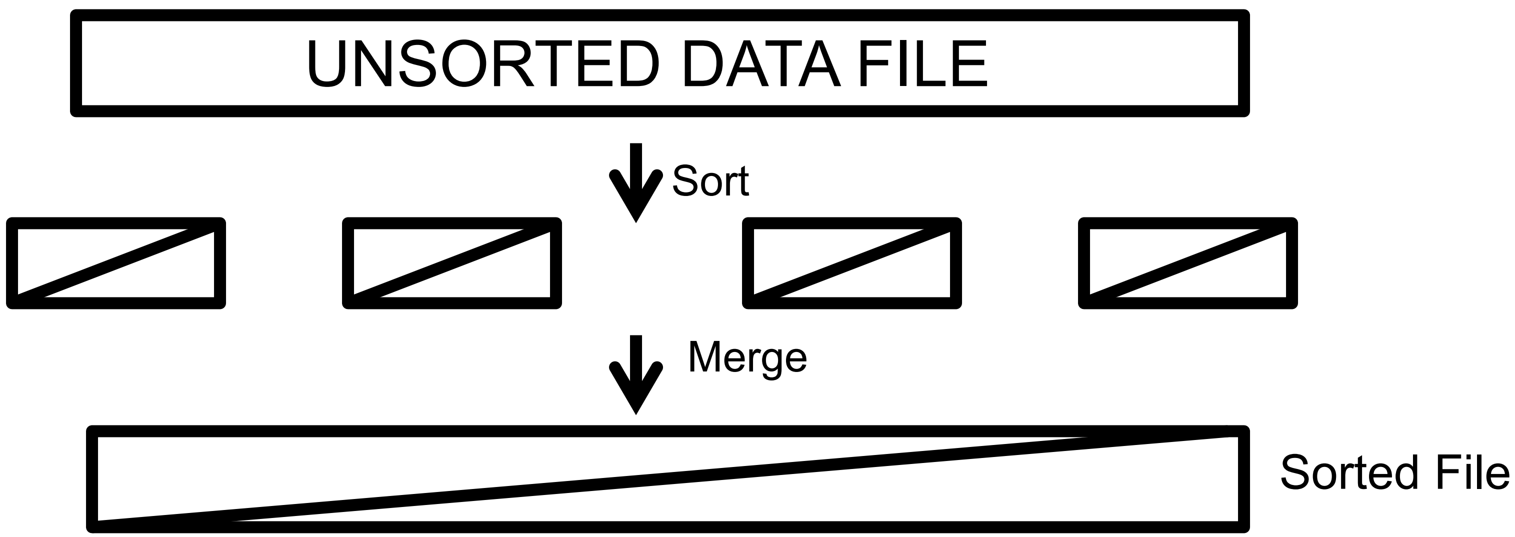
I/O – Efficient Sorting

Sorting needed in many cases:

- Inverted index construction
- Output in sorted order
- Sort – Based join
- Offline B-tree index construction
- Duplicate elimination
- Group by

I/O – Efficient Sorting

- Data may not fit in main memory
- Many algorithms will be inefficient if data on disk
- Most popular I/O-efficient method: Merge Sort



MERGE SORT EXAMPLE

- Data file (256 million records of 100 bytes each) → 25.6 GB

- Main memory available for sorting: 100 MB

Algorithm:

- Load pieces of size 100 MB, sort each piece, and write to a file
- Use d-way merging to merge d sorted files into one larger sorted file, until only one file left

→

- How to choose d?
- How to merge d lists?

IN MORE DETAIL

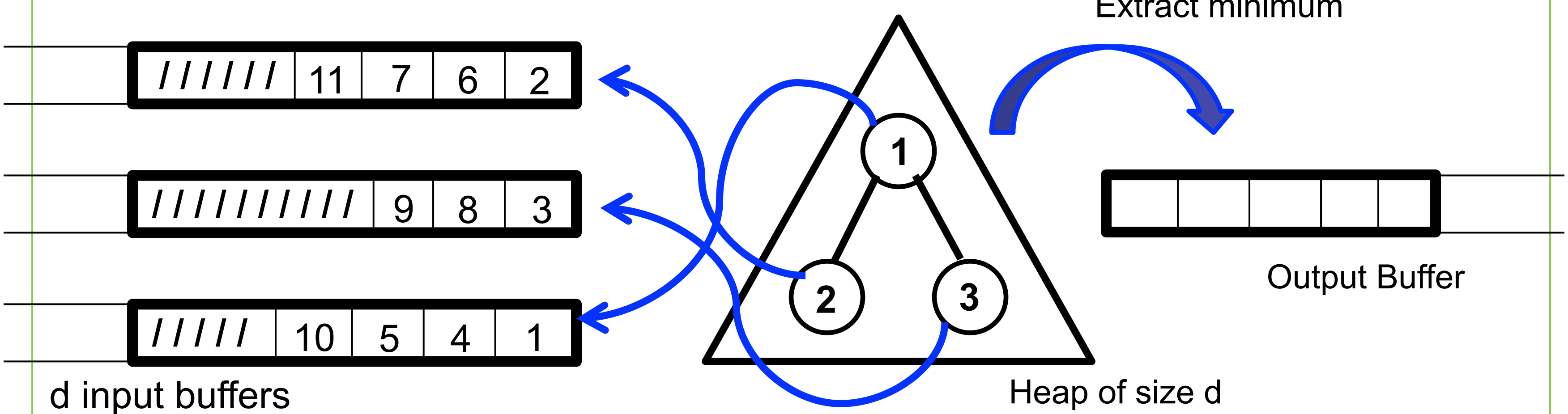
25.6 GB of data (256 million records of 100 Bytes)
100 MB of work space in main memory

Phase 1 : Repeat:
- read 100 MB data
- sort in main memory using any sorting algo
- write into a new file

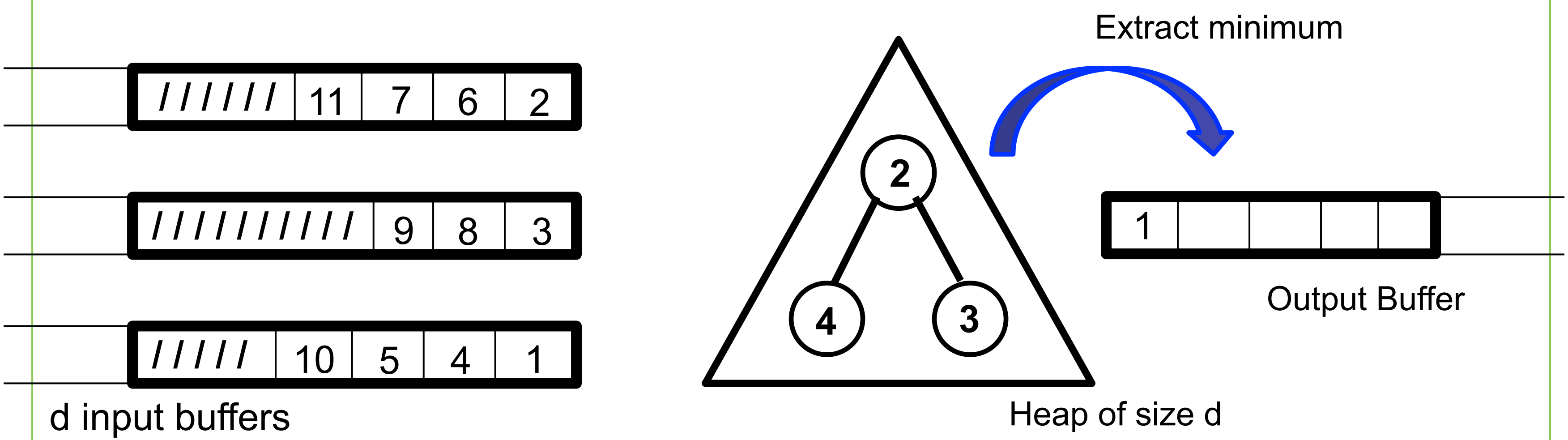
Until all data read

Phase 2: Merge the 256 files created in Phase 1
- in 1 pass: merge 256 files into one
- in 2 passes: merge 256 files into 16, then 16 into 1

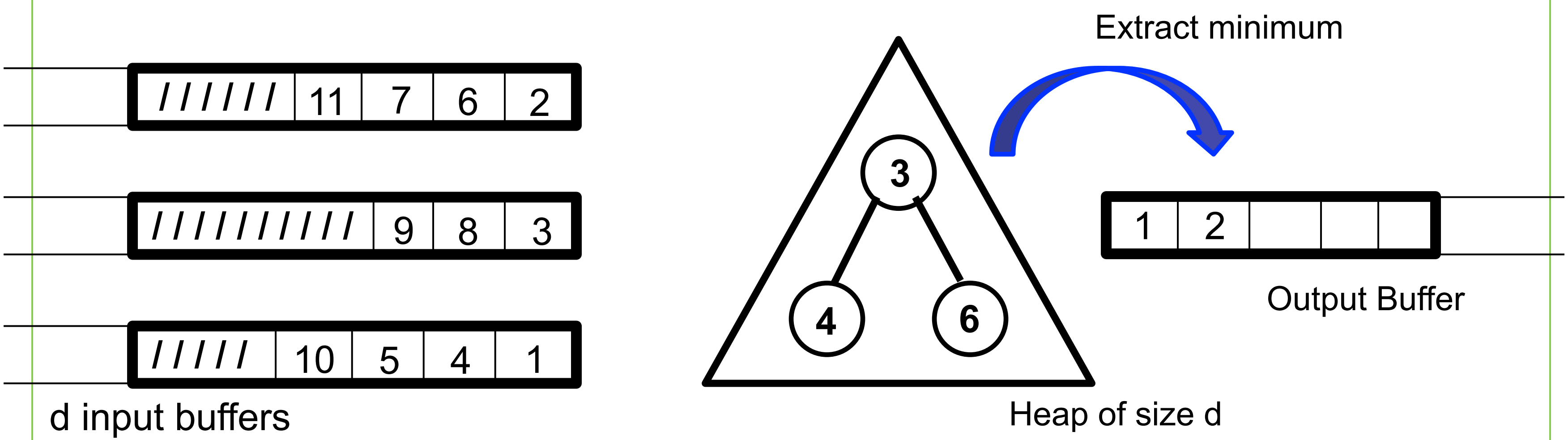
HOW TO MERGE d LISTS



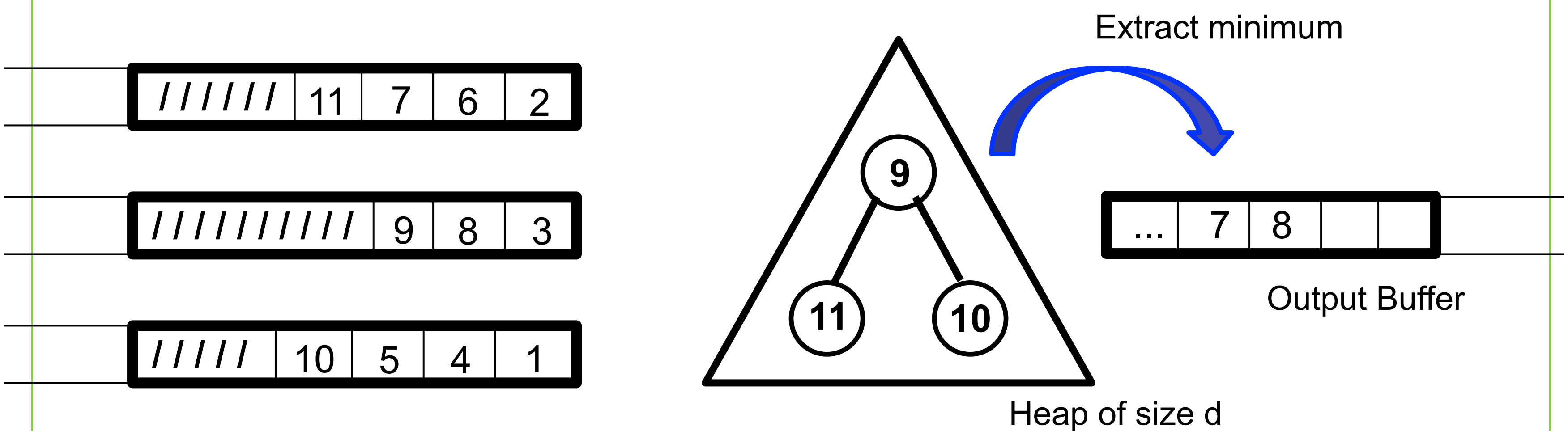
HOW TO MERGE d LISTS



HOW TO MERGE d LISTS



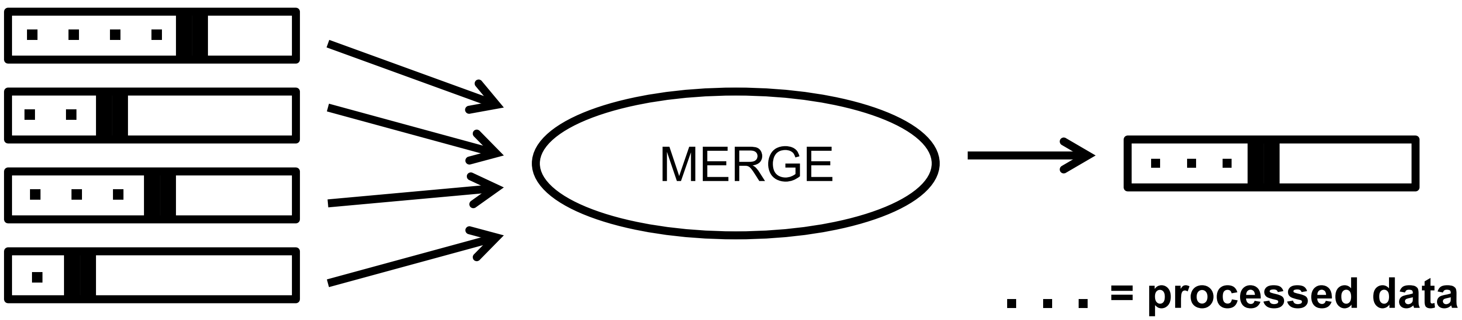
HOW TO MERGE d LISTS



- **Initially insert first (smallest) element from each list into heap**
- **Extract minimum and write out to output buffer**
- **Replace extracted element with the next element from the list where the minimum came from, then heapify again**
- **Repeat steps until heap is empty → all d lists are merged**

DATA ACCESS/MOVEMENT DURING MERGE

d-way merge: need d input buffers and 1 output buffer



- Files now sorted in ascending order (left to right)
- Note: heap-based merge makes pass from left to write
- If output buffer full, write it out: append to output file
- If input buffer empty, read next chunk of data from that file

Larger d: fewer passes but smaller buffers, thus slower disk I/O

Back to our Example:

25.6 GB of data to be sorted

100 MB of main memory available for sorting

⇒ after sort phase, we need to merge 256 sorted files of size 100 MB each

Disk with 10ms access time (seek time plus rotational latency) and 50 MB/s maximum transfer rate

Thus it takes 10 + x/50 ms to read x KB of data

What is the best choice of d?

d = 2 (8 passes since 2^8 = 256)

d = 16 (2 passes since 16^2 = 256)

d = 256 (1 pass)

d = 2:

- 2 input and 1 output buffer of 33.33MB each
- Reading/writing one buffer of data takes :
  10 + 33333/50 = 676.66 ms

$\Rightarrow$ Reading all 25.6 GB in 768 pieces of 33.33 MB takes:
768 * 676.66 ~ 520 s

$\Rightarrow$ Each pass (read in + write out)
1040 s

$\Rightarrow$ All 8 passes
8320 seconds = 2.3 hours

d = 16:

- 16 input and 1 output buffer of 5.88 MB each
- Reading/writing one buffer of data takes :
  10 + 5.888/50 = 127.6 ms

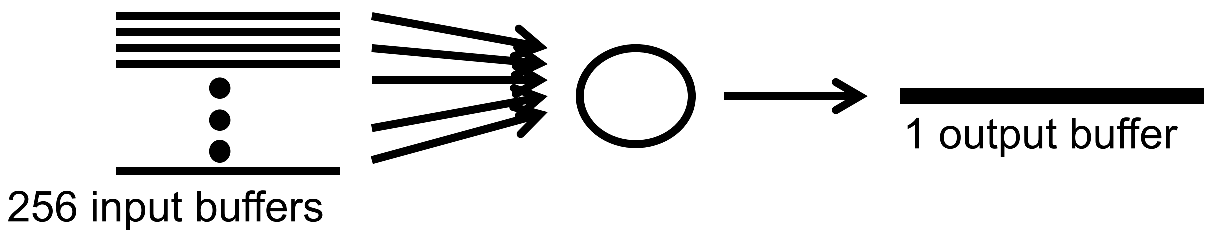
⇒ Reading all 25.6 GB in 4354 pieces of 5.88 MB takes:
4354 * 127.6 ms = 555.6 s

⇒ Each pass (read in + write out)
1111.2 sec

⇒ Total time for 2 passes
2222.4 sec or about 40 minutes

d = 256:

- **257 buffers of 389 KB each**



- **Reading/writing one buffer of data takes:**
  10 + 389/50 ~ 17.78 ms **(more than half of time on seeks)**

⇒ **Reading all 25.6 GB takes:**
1170 s

⇒ **Total (read in + write out)**
2340 seconds, or slightly slower than d=16

$\Longrightarrow$ Choose d:

- Not too small (Few passes)
- Not too large (Fast disk access)
- Typically 1 or 2 passes for current machines

<u>Also:</u>

- Use double buffering if CPU time counts
- Make the output buffer larger than the input buffer

E.g.:

- Choose 16 input buffers of 5 MB each
- 1 output buffer of 20 MB

I/O – Efficient Algorithms

- I/O-Efficient Algorithms: area dealing with theory and practice of designing algorithms for disk-resident data
- Many algorithms for many different problems
- Sorting: merge (mergesort) vs. split (quicksort, postal sort)
  - d-way merge and split, not binary
- Graph algorithms based on repeated sorting of edges
  - E.g., Pagerank algorithm
- Important operations: scan, split, merge, sort over the data
- I/O-efficient data structures: e.g., B+-tree
  - also degree d > 2 (but for slightly different reasons)

Conclusions

- Hard disks and SSDs are much slower than memory
- High cost of random accesses, esp. for HDDs
- When data does not fit in RAM, need to redesign algorithms to avoid random accesses to data (instead, stream/scan data)
- Area of I/O-efficient computing
- Design algorithms by repeatedly scanning, merging, splitting, sorting large data sets
- Reading and writing/appending to new files
- Also relevant to SSDs, and even RAM (a little)
  - E.g., optimized in-memory merge sorts may merge 8 sorted lists at a time