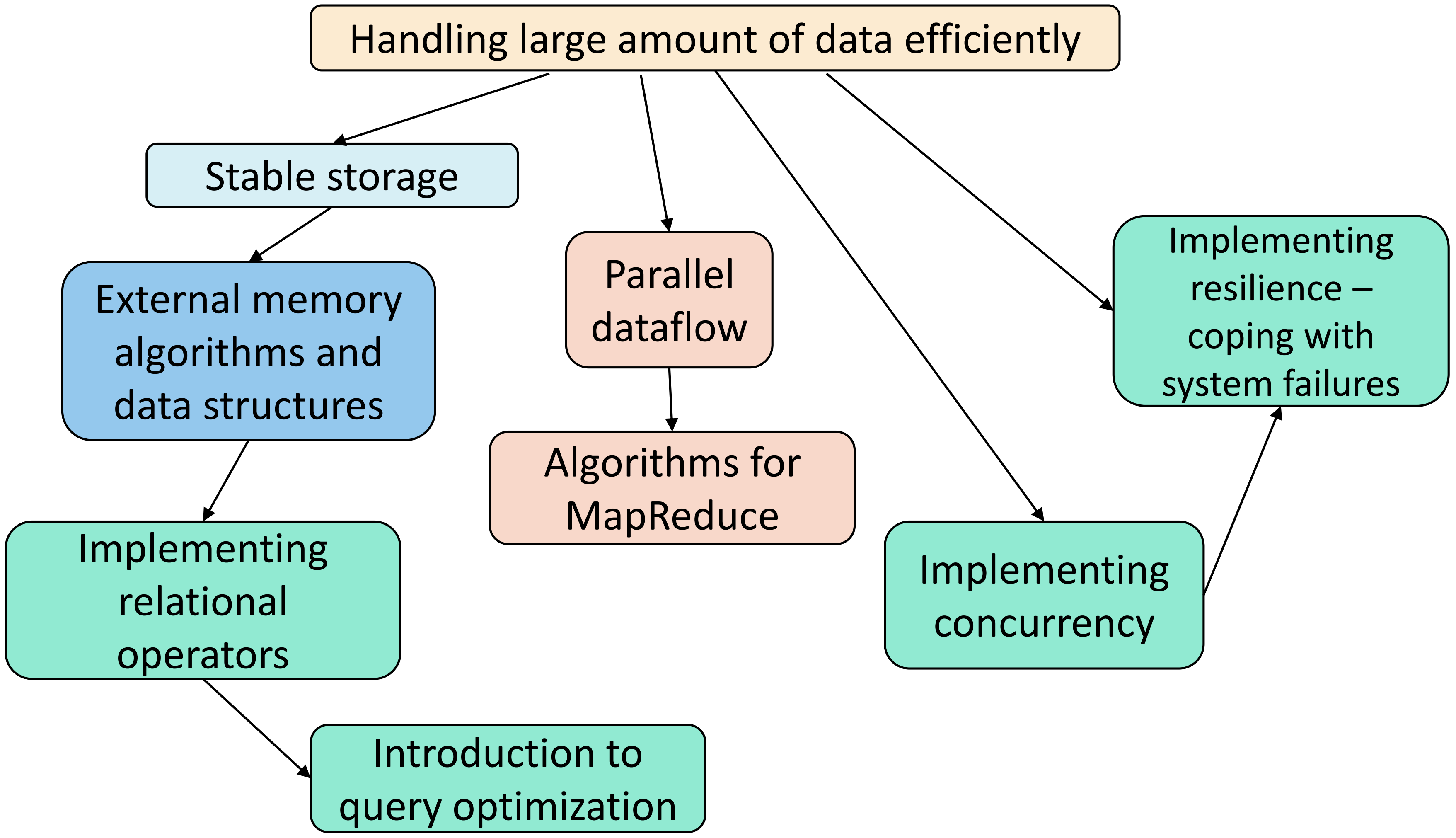
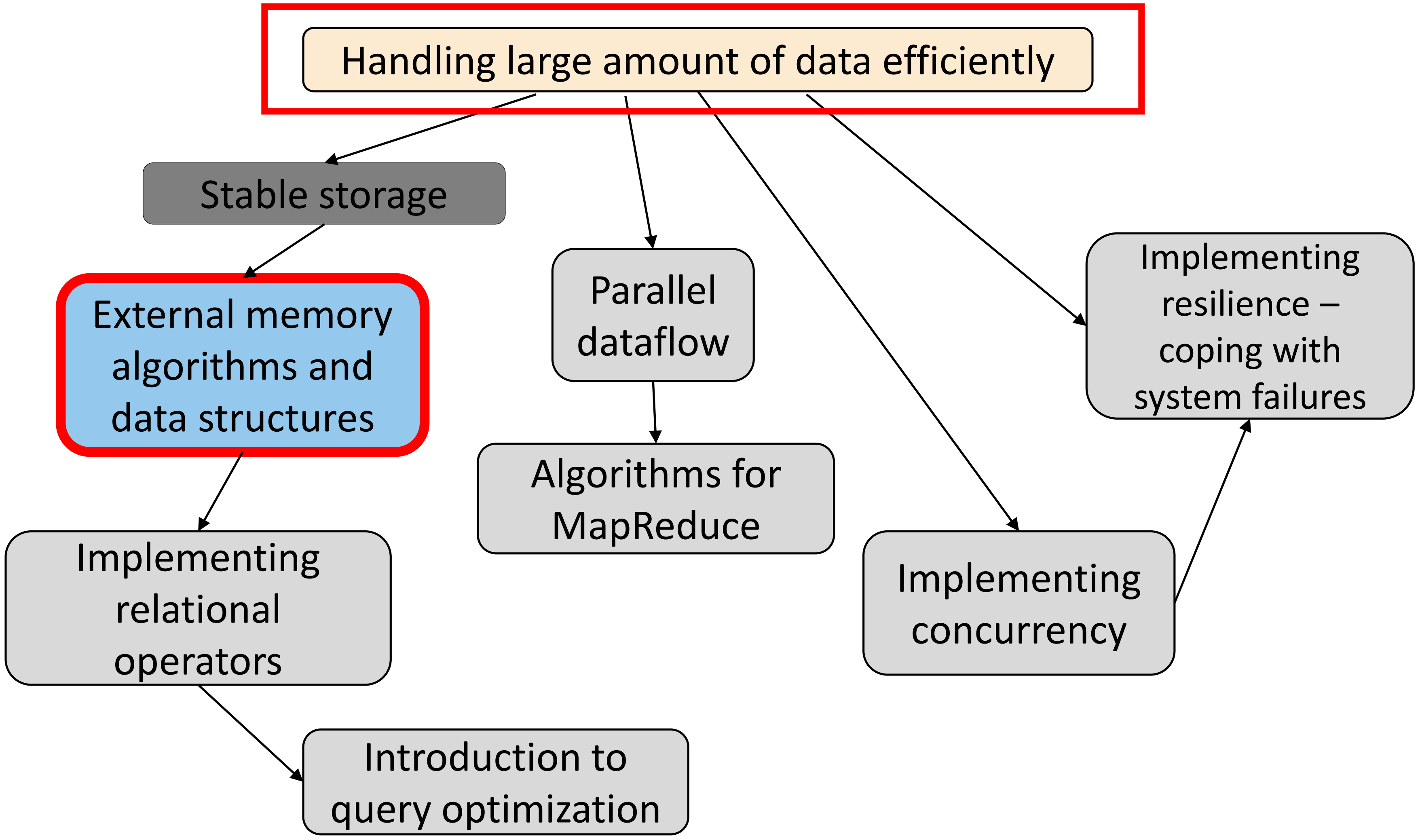
Roadmap


Handling large amount of data efficiently
Stable storage
External memory algorithms and data structures
Implementing relational operators
Introduction to query optimization
Parallel dataflow
Algorithms for MapReduce
Implementing concurrency
Implementing resilience – coping with system failures

Special algorithms for disks


Handling large amount of data efficiently
Stable storage
External memory algorithms and data structures
Parallel dataflow
Algorithms for MapReduce
Implementing resilience – coping with system failures
Implementing concurrency
Implementing relational operators
Introduction to query optimization

Lecture 02.02

2PMMS: comparative analysis

By Marina Barsky
Winter 2017, University of Toronto

Merge sort: how long does it take?

- 10,000,000 records of 200 bytes each = 2GB file
  - Stored on disk, with 20K blocks, each holding 100 records
  - Entire file occupies **100,000** blocks $(\underbrace{200*10^7}_{\text{Total in bytes}} / \underbrace{20*10^3}_{\text{1 block}})$

- 100MB available **main memory**
  - The number of blocks that can fit in 100MB of memory is $\underbrace{100 \times 10^6}_{\text{Total RAM}} / \underbrace{(20 * 10^3)}_{\text{1 block}}$, or ≈ **5,000** blocks.

I/O time per block – avg 11.5 ms:
6 ms – average seek time
5 ms – average rotational delay
0.5 ms – transfer time for 20K block

Analysis – Phase 1

- **5000** out of the **100,000** blocks will fill main memory.
  - We thus fill memory $\lceil 100{,}000/5{,}000 \rceil = 20$ times, sort the records in main memory, and write 20 sorted runs out to disk.

- How long does this phase take?
- We read each of the 100,000 blocks once, and we write 100,000 new blocks. Thus, there are 200,000 disk I/O's for 200,000***11.5** ms = 2300 seconds, or **38 minutes**.

I/O time per block – avg 11.5 ms:
6 ms – average seek time
5 ms – average rotational delay
0.5 ms – transfer time for **20K** block

Analysis – Phase 2

- Every block holding records from one of the sorted lists is read from disk exactly once.
  - Thus, the total number of block reads is 100,000 in the second phase, just as for the first.
- Likewise, each record is placed once in an output block, and each of these blocks is written to disk once.
  - Thus, the number of block writes in the second phase is also 100,000.

- We conclude that the second phase takes another **38 minutes**.

Total time

- Total: Phase 1 + Phase 2 = 38 + 38 = **76 minutes!** (Ratio of disk data size to main memory is 20)

Advantage of larger blocks

- In our analysis block size is 20K.
- What would happen if the block size is 40K?

- The number of blocks to read and write is now 50,000 (half of 100,000)

- Each read and write takes longer, so each I/O is more expensive
- Now, the only change in the time to access a block would be that the transfer time increases to **0.50*2=1** ms, the average seek time and rotational delay remain the same
- The time for block size **40K**: (2 * 50,000 disk I/Os for phase I + 2 * 50,000 disk I/Os for phase II)*12 ms = 2400 ms for **both** phases = **40 min (vs. 76)!**

I/O time per block – avg 12 ms:

6 ms – average seek time

5 ms – average rotational delay

1 ms – transfer time for **40K** block

Another example: block size = 500K

- For a block size of 500K the transfer time per block is 0.5***25**=12.5 milliseconds.
- Average block access time would be

  11 + 12.5 approx. 24 ms, (as opposed to 11ms we had)
- However, now a block can hold 100***25** = 2500 records and the whole table will occupy 10,000,000 / 2500 = 4000 blocks (as opposed to 100,000 blocks we had before).
- Thus, we would need only 4000 * 2 * 2 disk I/Os for 2PMMS for a total time of 4000 * 2 * 2 * 24 = 384,000ms or about **6.4 min**
- Speedup:

  76 / 6.4 = 12 fold !

I/O time per block – avg 25 ms:

  6 ms – average seek time

  5 ms – average rotational delay

  12.5 ms – transfer time for **500K** block

Reasons to limit the block size

1. We cannot use blocks that cover *several tracks* effectively
2. Small relations would occupy only a fraction of a block, so large blocks would waste space on disk
3. The larger the blocks are, the fewer records we can sort by 2PMMS (less runs we can merge in Phase II)

```
$ sudo hdparm -I /dev/sda

/dev/sda:
ATA device, with non-removable media
        Model Number:       ST3500630AS
        Serial Number:      9XXYZ845YZ
        Firmware Revision:  3.AAK
Standards:
        Supported: 7 6 5 4
        Likely used: 7
Configuration:
        Logical         max     current
        cylinders       16383   16383
        heads           16      16
        sectors/track   63      63
        --
        CHS current addressable sectors:
 …
```

Find out parameters of a hard drive in Unix

Nevertheless, as machines get more memory and disks more capacious, there is a tendency for block sizes to grow.

Max number of records we can sort in 2 passes

Phase 2

- How many input buffers we can have at most in Phase II?

  M/B **-1**

  at least 1 block per buffer

  one separate block for output buffer

B - block size in bytes.
M - main memory in bytes.
R – size of one record in bytes.

Phase 1

- How many sorted sublists max?

  (*M/B*)-1

- How many records max we can sort in 1 sublist?

  M/R

Hence, we are able to sort **(M/R)*[(M/B)-1]** ≈ $\mathbf{M^2/RB}$ records

Example: Max number of records we can sort in 2 passes

M=100MB = 100,000,000 Bytes = 10^8 Bytes

B = 20,000 Bytes

R = 100 Bytes

So, $M^2/RB = (10^8)^2 / (100 * 20,000)$ $= 6 * 10^9$ records,

100 bytes each

or relation of 600 GB

- just with 100MB of memory!

B - block size in bytes.
M - main memory in bytes.
R – size of one record in bytes.

$\boxed{M^2/RB \text{ records}}$

$\boxed{M^2/B \text{ bytes}}$

How many runs to have

B - block size in bytes.
M - main memory in bytes.
N – size of input in bytes

- In general, we want to have a larger buffer for each run where we would buffer several blocks at a time, taking advantage of sequential access, and thus we want to have a comparatively small number of big runs ***k:***
- Phase I : $k > N/M$, because we can sort at most M bytes of input in RAM
- Phase II: $k < M/B - 1$, because we cannot have more input buffers of size at least 1 block

$$M/B - 1 > \boldsymbol{k} > N/M$$

Space for experiments: Assignment 1.2

Sorting larger relations

B - block size in bytes.
M - main memory in bytes.
N – size of input in bytes

$$M/B - 1 > \boldsymbol{k} > N/M$$

- What if N/M > M/B -1?

i.e. we need to produce so many runs in Phase I, that we cannot allocate at least 1 block for each run in Phase II?

- Then we first produce much longer runs using 2PMMS: each run will be of max length $M/B * M$
- In Phase III we merge this runs, the maximum we can merge is M/B runs of size M^2/B each.
- With **3** passes we can sort relation of size $\mathbf{M^3/B^2}$ bytes!

Example: max size we can sort in 3 passes

$$\boxed{M^3/B^2}$$

1 KB	10^3
1 MB	10^6
1 GB	10^9
1 TB	10^{12}
1 PB	10^{15}

- Memory M = 100 MB = 10^8 bytes
- Block size = 20 KB = $2*10^4$ bytes

- In 3 passes we can sort

$$(10^8)^3 / 4* 10^8 = 2.5*10^{15} \text{ bytes} =$$

2.5 PB!

Sorting efficiency is crucial:

- Duplicate elimination
- Grouping and aggregation
- Union, intersection, difference
- Join

Performance tricks for improving the Running Time of 2PMMS

- Blocked I/O

  Reading into buffer *P* blocks (pages) at a time, instead of one block

- Double-buffering ("Prefetching")

  2 block-buffers per run, once the first is processed, it gets refilled while CPU is working on the second

- Cylindrification and multiple disks

Cylindrification

Idea: Place all input blocks by cylinder, so once we reach that cylinder we can read block after block, with no seek time or rotational latency (*Assuming this is the only process running in the machine that needs disk access*).

Application to Phase 1 of 2PMMS

- Because we have only transfer time, we can do Phase I for sorting 100,000 blocks (2 I/Os – read and write) in:

  2*100,000*1ms = 200 sec = 3.3 min

But, Phase 2 …?

I/O time per block – avg 12 ms:
6 ms – average seek time
5 ms – average rotational delay
1 ms – transfer time for 40K block

I/O time per block – **1 ms**:

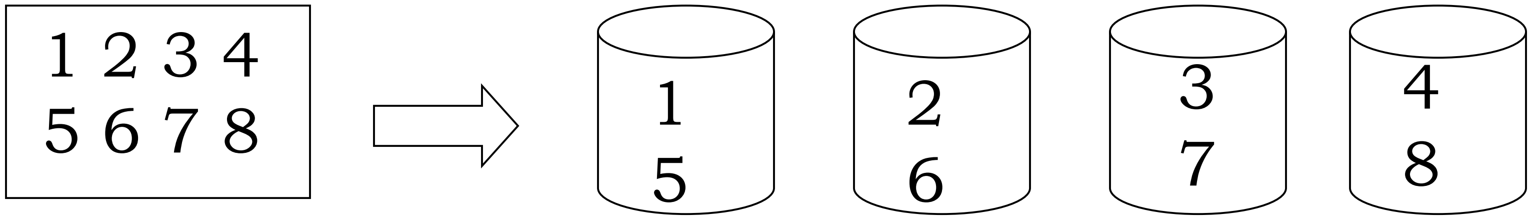
Cylindrification – Phase 2

- Storage by cylinders **does not help in Phase II**
  - Blocks are read from the fronts of the sorted lists in **an order that is determined by which list next exhausts its current block.**
  - Output blocks are written one at a time, interspersed with block reads
- Thus, the second phase will still take **38 min**.
- Total: 38+3 = 41 min vs 76 min

- We have cut the sorting time almost half, but cannot do better by cylindrification alone.

Multiple Disks and Cylindrification

- Use several disks with independent heads
- **Example**: Instead of a large disk of 1TB, let's use 4 smaller disks of 250GB each
  - We divide the given records among the four disks; the data will occupy adjacent cylinders on each disk.
  - We distribute each sorted list onto the four disks, occupying several adjacent cylinders on each disk.



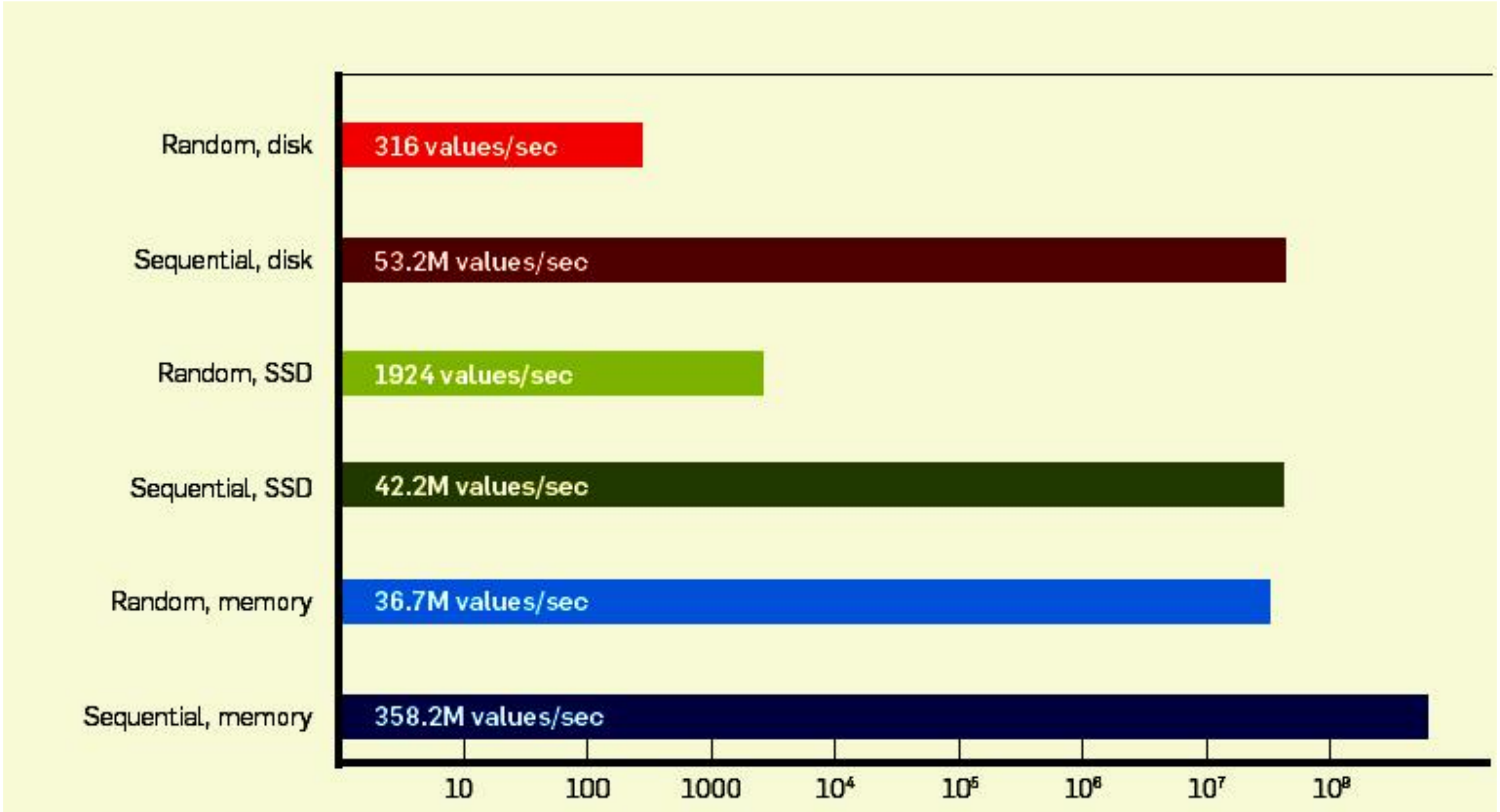
Multiple Disks – Phase 2

- **Phase 2**:
  - Use **4 output buffers,** one per disk, and cut writing time in about 1/4.

  - When we need to fill an input buffer, we can perform several block reads in parallel. Potential for a 4-fold speedup.
  - In practice, we get 2-3 fold speedup for Phase 2.

- Total time $\approx$ 3 + 38/3 $\approx$ **15 min**

Remember?



* Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64GB RAM and eight 15,000RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest generation Intel high-performance SATA SSD.

Improvement over the tape…

Jacobs:

“A further point that’s widely under-appreciated: in modern systems, as demonstrated in the figure, random access to memory is typically slower than sequential access to disk. Note that random reads from disk are more than 150,000 times slower than sequential access; SSD improves on this ratio by less than one order of magnitude.

In a very real sense, **all of the modern forms of storage improve only in degree, not in their essential nature, upon that most venerable and sequential of storage media: the tape.**”

A Tape algorithm (Knuth, Vol. 3)

- Balanced 2-way merge with 4 "working tapes"
  - During the first phase, sorted runs produced by the main-memory sorting are placed alternately on Tapes 1 and 2,
  - Then Tapes 1 and 2 are *rewound* to their beginnings, and we merge the runs from these tapes, obtaining new runs which are twice as long as the original ones;
  - The new runs are written alternately on Tapes 3 and 4 as they are being formed.
  - Then all tapes are rewound, and the contents of Tapes 3 and 4 are merged into quadruple-length runs recorded alternately on Tapes 1 and 2.

A Tape algorithm (cont.)

- The process continues, doubling the length of runs each time, until only one run is left (namely the entire sorted file).
- If *S* runs were produced during the internal sorting phase this balanced 2-way merge procedure makes *log S* merging passes over all the data (not log N)

	T1	T2	T3	T4	
Pass 1	$A_1A_1A_1A_1$	$A_1A_1A_1A_1$	—	—	Initial distribution
Pass 2	—	—	D_2D_2	D_2D_2	Merge to T3 and T4
Pass 3	A_4	A_4	—	—	Merge to T1 and T2
Pass 4	—	—	D_8	—	Final merge to T3

From Knuth, “The art of computer programming”, Vol 3, p.299

Tape algorithm example. Phase I

- 5000 records are to be sorted with a main memory capacity of 1000

  Sort $R_1 \ldots R_{1000}$; $R_{1001} \ldots R_{2000}$; $R_{2001} \ldots R_{3000}$; $R_{3001} \ldots R_{4000}$ $R_{4001} \ldots R_{5000}$ and distribute into tape 1 and tape 2:

  Tape 1: $R_1 \ldots R_{1000}$; $R_{2001} \ldots R_{3000}$; $R_{4001} \ldots R_{5000}$

  Tape 2: $R_{1001} \ldots R_{2000}$; $R_{3001} \ldots R_{4000}$

  Tape 3: (empty)

  Tape 4: (empty)

Tape algorithm example. Phase II

- Input to Phase II:

  Tape 1: R_1 . . . R_{1000}; R_{2001} . . . R_{3000}; R_{4001} . . . R_{5000}

  Tape 2: R_{1001} . . R_{2000}; R_{3001} . . . R_{4000}

  Tape 3: (empty)

  Tape 4: (empty)

- Output of Phase II:

  Tape 3: R_1 . . . R_{2000}; R_{4001} . . . R_{5000}

  Tape 4: R_{2001} . . . R_{4000}

  Tape 1: to erase

  Tape 2: to erase

Tape algorithm example. Phase III

- Input to Phase III:
  Tape 3: $R_1 \ldots R_{2000}$; $R_{4001} \ldots R_{5000}$
  Tape 4: $R_{2001} \ldots R_{4000}$

- Output of Phase II:
  Tape 1: $R_1 \ldots R_{4000}$
  Tape 2: $R_{4001} \ldots R_{5000}$

- Finally – merge Tape 1 and Tape 2

Tape algorithm: toy example

- Sorted runs are written alternately to tapes 1 and 2:

1, 4, 6, 3, 8, 10, 2, 7, 11, 5, 9, 12

Tape 1: **1, 4, 6, 2, 7, 11**

Tape 2: **3, 8, 10, 5, 9, 12**

Empty tapes:

Tape 3:

Tape 4:

Tape algorithm: toy example – rewind 2 tapes

Tape 1: **1, 4, 6, 2, 7, 11**

Tape 2: **3, 8, 10, 5, 9, 12**

Empty tapes:

Tape 3:

Tape 4:

Tape algorithm: toy example - merge

Tape 1: **1, 4, 6, 2, 7, 11**

Tape 2: **3, 8, 10, 5, 9, 12**

Merge into:

Tape 3: **1, 3, 4, 6, 8, 10**

Tape 4: **2, 5, 7, 9, 11, 12**

Tape algorithm: toy example – rewind 4 tapes

Tape 1: 1, 4, 6, 2, 7, 11

Tape 2: 3, 8, 10, 5, 9, 12

Merge into:

Tape 3: 1, 3, 4, 6, 8, 10

Tape 4: 2, 5, 7, 9, 11, 12

Tape algorithm: toy example - final merge

Input:

Tape 3: 1, 3, 4, 6, 8, 10

Tape 4: 2, 5, 7, 9, 11, 12

Merge:

Tape 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Tape algorithm: while rewinding the tape

- Sorted runs:

1, 4, 6, 3, 8, 10, 2, 7, 11, 5, 9, 12

Tape 1: **1, 4, 6, 2, 7, 11**

Tape 2: **3, 8, 10, 5, 9, 12**

While rewinding Tape 1 and Tape 2, merge backwards – and write on Tapes 3, 4 in descending order:

Tape 3:

Tape 4:

Tape algorithm: while rewinding the tape

- Sorted runs:

1, 4, 6, 3, 8, 10, 2, 7, 11, 5, 9, 12

Tape 1: **1, 4, 6, 2, 7, 11**

Tape 2: **3, 8, 10, 5, 9, 12**

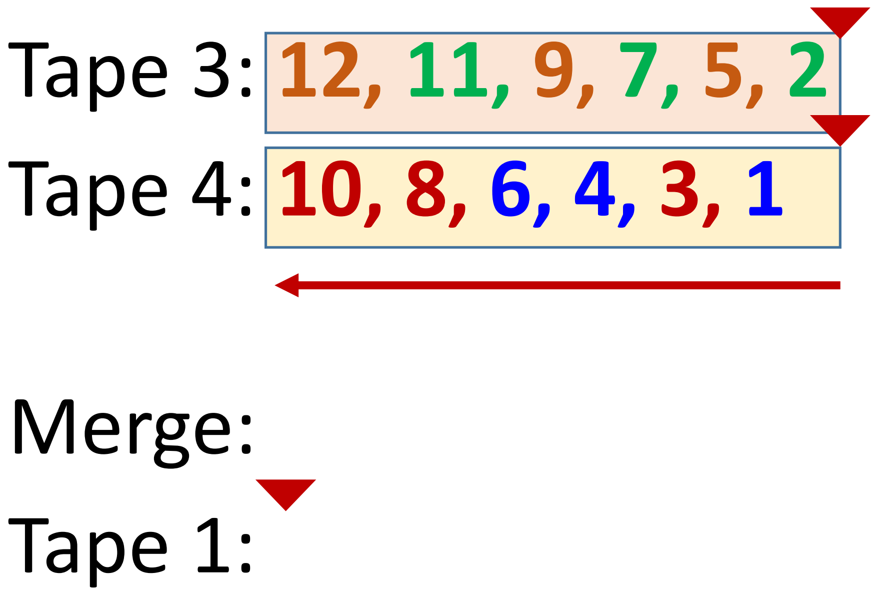
While rewinding Tape 1 and Tape 2, merge backwards – and write on Tapes 3, 4 in descending order:

Tape 3: **12, 11, 9, 7, 5, 2**

Tape 4: **10, 8, 6, 4, 3, 1**

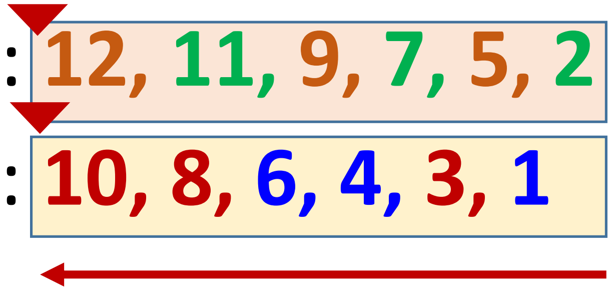
Tape algorithm: while rewinding the tape

Inputs tapes:



Tape algorithm: while rewinding the tape

Inputs tapes:


Tape 3: 12, 11, 9, 7, 5, 2

Tape 4: 10, 8, 6, 4, 3, 1

Merge:

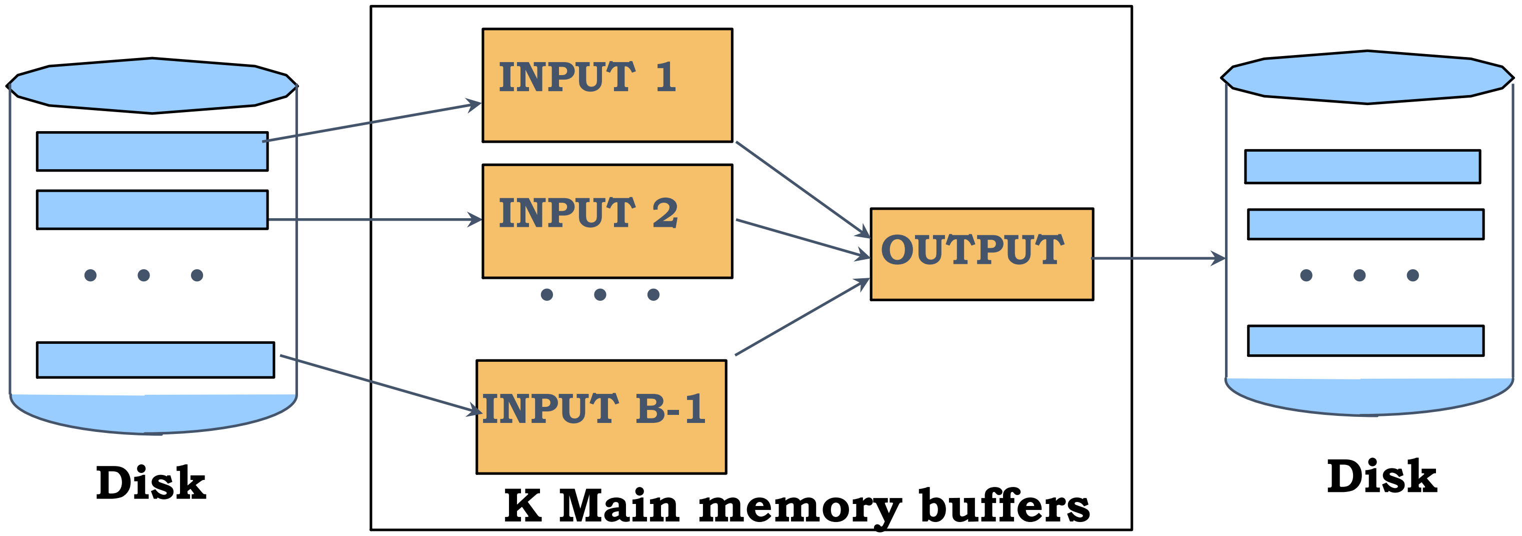
Tape 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Unfortunately, we cannot read from the hard disk backwards!

But we have a random access!

Summary

- Disk constraints require different algorithms which reduce the number of disk I/Os
- The fastest and widely used sorting algorithm for large inputs is 2PMMS



Reflect about:

- How would you implement efficient random shuffling of very large inputs?
- How would you incorporate duplicate elimination into the sorting algorithm?
- What data structure to use best for the selection of the smallest key under current pointer in all input buffers?