Today:

- Disk I/O model of computation → efficient use of disk.
- Sorting example.

Soon:

- RAID for coping with disk failures.
- Representing tuples/objects and relations/classes as records on blocks and as files, respectively.
  - Index techniques.

Using Disk Efficiently

- *Disk I/O* = read or write of a block is very expensive compared with what is likely to be done with the block once it arrives in main memory.
  - Perhaps 1,000,000 machine instructions in the time to do one random disk I/O.
- Random block accesses is the norm if there are several processes accessing disks, and the disk controller does not schedule accesses carefully.
- Reasonable model of computation that requires secondary storage: count *only* the disk I/O's.

Good DBMS algorithms

a) Try to make sure that if we read a block, we use much of the data on the block.

b) Try to put blocks that are accessed together on the same cylinder.

c) Try to *buffer* commonly used blocks in main memory.

Sorting Example

Setup:

- 10M records of 100 bytes = 1Gb file.
  - ❖ Stored on Megatron 747 disk, with 4Kb blocks, each holding 40 records + header information.
  - ❖ Entire file takes 250,000 blocks = 1000 cylinders.
- 50M available main memory = 12,800 blocks = 1/20th of file.
- Sort by primary key field.

Merge Sort

Common main-memory sorting algorithms don't look so good when you take disk I/O's into account. Variants of Merge Sort do better.

- *Merge* = take two sorted lists and repeatedly chose the smaller of the "heads" of the lists (head = first of the unchosen).
- Example: merge 1,3,4,8 with 2,5,7,9 = 1,2,3,4,5,7,8,9.
- Merge Sort based on recursive algorithm: divide records into two parts; recursively mergesort the parts, and merge the resulting lists.

Two-Phase, Multiway Merge Sort

Merge Sort still not very good in disk I/O model.

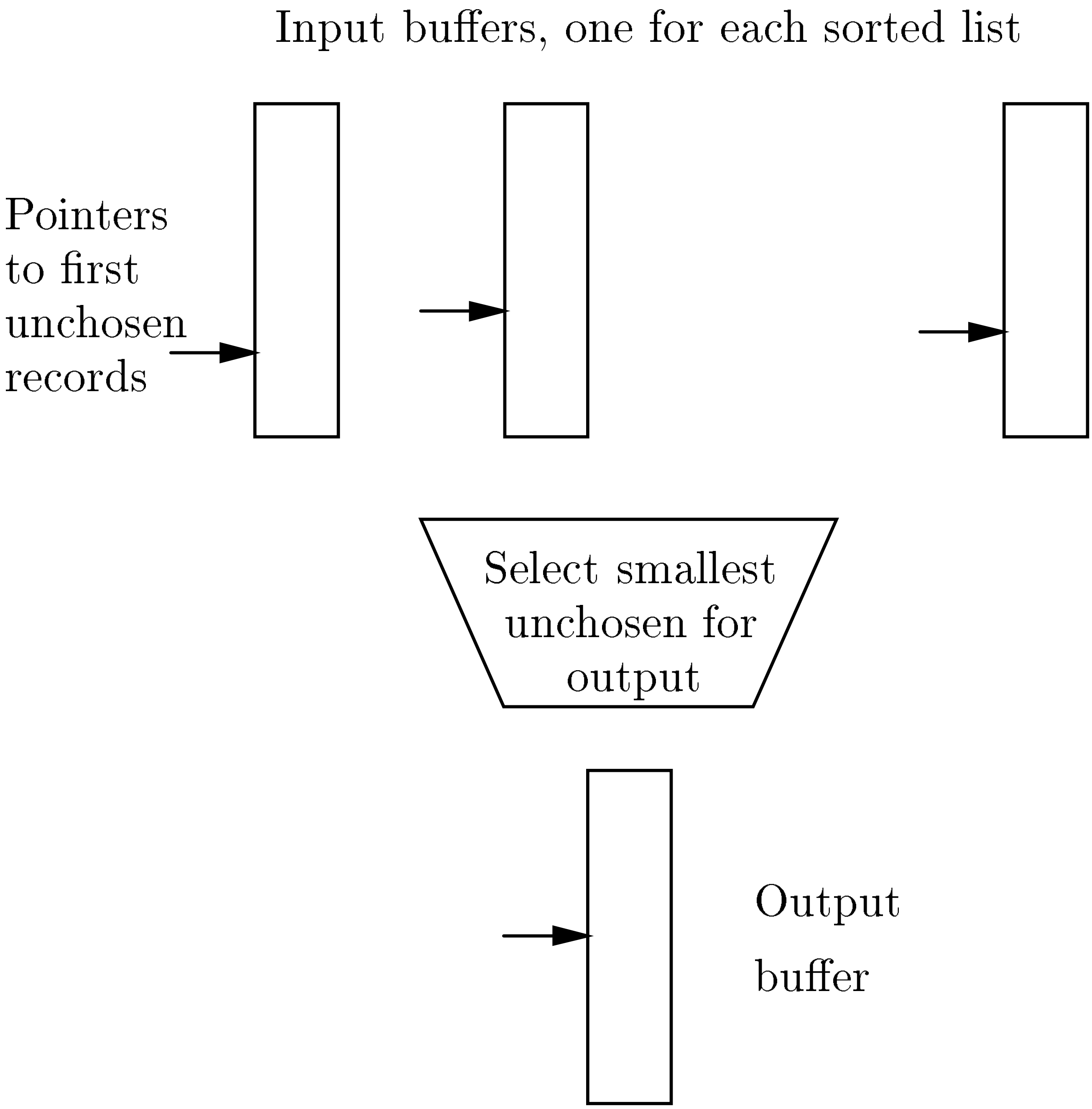
- $\log_1 n$ passes, so each record is read/written from disk $\log_2 n$ times.
- 2PMMS: 2 reads + 2 writes per block.

Phase 1

1. Fill main memory with records.
2. Sort using favorite main-memory sort.
3. Write sorted sublist to disk.
4. Repeat until all records have been put into one of the sorted lists.

Phase 2

Use one buffer for each of the sorted sublists and one buffer for an output block.



- Initially load input buffers with the first blocks of their respective sorted lists.
- Repeatedly run a competition among the first unchosen records of each of the buffered blocks.
  - Move the record with the least key to the output block; it is now "chosen."
- Manage the buffers as needed:
  - If an input block is exhausted, get the next block from the same file.
  - If the output block is full, write it to disk.

Analysis of Naive Implementation

Assume blocks are stored at random, so average access time is about 15 ms.

- File stored on 250,000 blocks, read and written once in each phase.
- 1,000,000 disk I/O's $\times$ 0.015 seconds $=$ 15,000 seconds $=$ 250 minutes $=$ 4+ hours.

Problem and Interlude

How many records can you sort with 2PMMS, under our assumptions about records, main memory, and the disk? What would you do if there were more?

Improving the Running Time of 2PMMS

Here are some techniques that sometimes make secondary-memory algorithms more efficient:

1. Group blocks by cylinder.
2. One big disk $\rightarrow$ several smaller disks.
3. Mirror disks = multiple copies of the same data.
4. "Prefetching" or "double buffering."
5. Disk scheduling; the "elevator" algorithm.

Cylindrification

If we are going to read or write blocks in a known order, place them by cylinder, so once we reach that cylinder we can read block after block, with no seek time or rotational latency.

Application to Phase 1 of 2PMMS

1. Initially, records on 1000 consecutive cylinders.

2. Load main memory from 50 consecutive cylinders.

   - Order of blocks read is unimportant, so only time besides transfer time is one random seek and 49 1-cylinder seeks (neglect).

   - Time to transfer 12,800 blocks at 0.5 ms/block = 6.4 sec.

3. Write each sorted list onto 50 consecutive cylinders, so write time is also about 6.4 sec.

But in Phase 2 $\cdots$

Blocks are read from the 20 sublists in a data-dependent order, so we cannot take much advantage of cylinders.

Multiple Disks

The Megatron 737 disk is just like a 747, but it has only one platter instead of four.

- We could replace one 747 by four 737's.
- If we divide sublists among the four 737's, then we can read or write up to four blocks at once.
    - ✧ But we must be very careful if we exploit this idea in Phase 2. As we begin to load an input block, we must start using it in the contest, but make sure we do not consume more of that block than has been read in.
    - ✧ Even so, we may not be able to get all four disks busy at once, because of contention for the same disk.

Mirror Disks

If we use n Megatron 747's and *mirror* the original disk with $n-1$ copies, then we could read n blocks at once, no matter which two they were.

- Writing requires a write on each disk, thus not sped up.
- Obvious minus: extra disk cost.

Prefetching

If we have extra space for main-memory buffers, consider loading buffers in advance of need. Similarly, keep output blocks buffered in main memory until it is convenient to write them to disk.

Example: Phase 2 of 2PMMS

With 50Mb of main memory, we can afford to buffer two cylinders for each sublist and for the output (one cylinder = 1Mb).

- Consume one cylinder for each sublist while the other is being loaded from main memory.
- Similarly, write one output cylinder while the other is being constructed.
- Thus, seek and rotational latency are almost eliminated, lowering total read and write times to about two minutes each.

Elevator Algorithm

Assume that there are many processes generating disk I/O requests.

- Disk controller (or OS, or DBMS) decides order in which requests are honored.
- Elevator algorithm has heads move from center to rim, then back again.
- Idea is to honor "nearby" requests whenever possible; lowers seek time, possibly rotational latency.

Example

Assume Megatron 747, request waiting for every 500 cylinders.

- Each seek is 2 ms instead of average 6.1 ms.
- Lowers average access time from $0.5 + 6.1 + 7.8 = 14.4$ to $0.5 + 2.0 + 7.8 = 10.3$.
- But, latency from application's point of view goes *up* while disk throughput goes *down*.
  - ❖ Typical process has to wait while 8 other requests are honored, i.e., 82 ms.