Query Processing 2: I/O-Efficient Sorting

Yufei Tao

https://www.cse.cuhk.edu.hk/~taoyf

This lecture will introduce an algorithm for sorting **disk-resident** data.

(Disk-Based) Sorting

R = a set of elements drawn from a total order.
B = the number of disk blocks that R occupies.
M = the number of memory blocks (a.k.a., the buffer blocks).

Goal: Produce a sorted list of the elements in R on disk.

Think: Why not use a "memory-sorting" algorithm such as merge sort or quick sort?

Next, we will take a "detour" to discuss a stand-alone problem called **merging**. As we will see, a fast algorithm for merging implies a fast algorithm for sorting.

(Disk-Based) Merging

M = the number of memory blocks (a.k.a., the buffer blocks).
We have sets $R_1, R_2, ..., R_{M-1}$ where

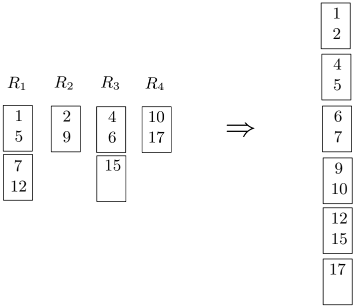
- all the elements in $R_1, ..., R_{M-1}$ are drawn from a total order;
- $R_1, ..., R_{M-1}$ are mutually disjoint;
- each R_i $(1 \le i \le M-1)$ is **sorted**.

For each $i \in [1, M-1]$, denote by B_i the number of disk blocks that R_i occupies.

Goal: Produce a sorted list of $R_1 \cup R_2 \cup ... \cup R_{M-1}$ on disk.

(Disk-Based) Merging

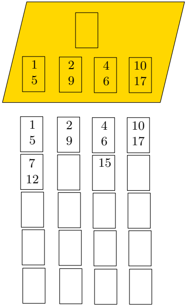
Example: $M = 5$.



(Disk-Based) Merging

To solve the problem, we allocate one memory block to each R_i ($1 \leq i \leq M-1$) as its **input buffer**. This consumes $M-1$ memory blocks. The remaining memory block is allocated as the **output buffer**.

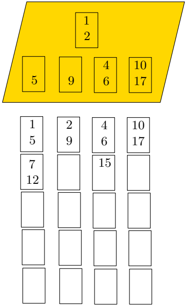
Load the first page of each R_i ($1 \leq i \leq M-1$) into its input buffer.



(Disk-Based) Merging

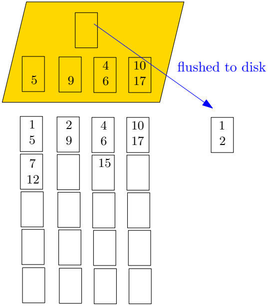
Move the smallest integer in the input buffers to the output buffer **until**

- **either** the output buffer is full
- **or** an input buffer becomes empty.



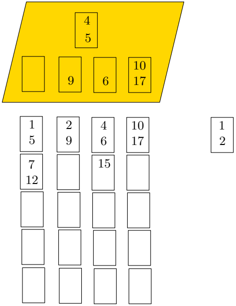
(Disk-Based) Merging

If the output buffer is full, output it to the sorted file on disk.



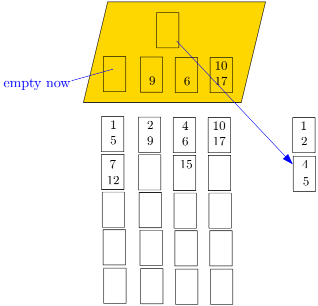
(Disk-Based) Merging

Move the smallest integer in the input buffers to the output buffer.



(Disk-Based) Merging

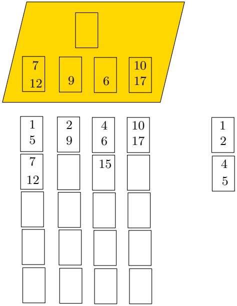
Here, the output buffer is full again and is flushed to the disk.



An input buffer is now empty.

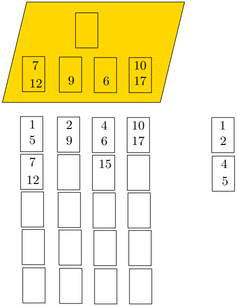
(Disk-Based) Merging

Whenever an input buffer is empty — say the buffer is for R_i — fill it with the next block of R_i (if it exists).



(Disk-Based) Merging

Repeat the above steps until all of $R_1, ..., R_{M-1}$ have been exhausted.



(Disk-Based) Merging

I/O cost:

- Number of read I/Os $= \sum_{i=1}^{M-1} B_i$
  (every input block read once)
- Number of write I/Os $\leq \sum_{i=1}^{M-1} B_i$
  (the sorted file cannot have more blocks than the input).

Total I/O cost $\leq 2\sum_{i=1}^{M-1} B_i$.

We now return to the sorting problem.
The algorithm we will introduce is called **external sort**.

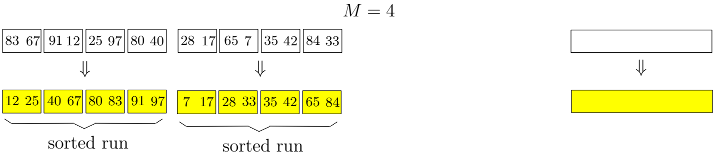
(Disk-Based) Sorting

Recall that the input is R, which occupies B blocks.

The Initial Step

Chop R into $n_0 = \lceil B/M \rceil$ **runs**, each consisting of M consecutive blocks (except possibly the last run).

For each run: load its elements into memory, sort them, and write them back to the disk, replacing the original run — this produces a **sorted run**.



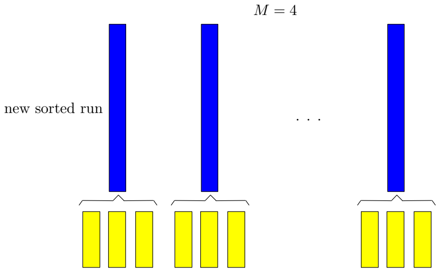
I/O cost $= 2 \cdot B$ (**think:** why 2? **hint:** apply our merging algorithm).

Merging Step 1

For every $M-1$ sorted runs: merge them into one sorted run. The number of new sorted runs is

$$n_1 = \lceil n_0/(M-1) \rceil.$$

A new sorted run has $M(M-1)$ blocks (except possibly for the last one).



I/O cost $\leq 2 \cdot B$ (**think:** why?)

(Disk-Based) Sorting

Merging Step $i \geq 1$

For every $M - 1$ sorted runs from the last step: merge them into one sorted run. The number of new sorted runs is

$$n_i = \lceil n_{i-1}/(M-1) \rceil.$$

A new sorted run has $M(M-1)^i$ blocks (except possibly for the last one).

I/O cost $\leq 2 \cdot B$.

When $n_i = 1$, we are done.

(Disk-Based) Sorting

h = the total number of merging steps.

The total I/O cost of sorting is at most $2B \cdot (h+1)$, which is $O(B \log_M B)$.

Remark: In practice, we typically have $B \leq M(M-1)$, in which case there is only one merging step and the I/O cost is $4B$.

The following questions are left to you.

We have assumed that R has no duplicate elements. How to adapt the algorithm to sort without this assumption?

Hint: no need to change the algorithm — what would be a clever "total order" here?

If R has duplicate elements, how do we remove duplicates with the cost of sorting?

Suppose that we have a relation $R(A, B)$. How to use sorting to answer the query below?

SELECT A, COUNT(B) FROM R GROUP BY A