EXTERNAL SORTING

CS 564- Spring 2018

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

I/O aware algorithms for sorting

- External merge
  - a primitive for sorting
- External merge-sort
  - basic algorithm
  - optimizations

WHY SORTING?

- users often want the data sorted (**`ORDER BY`**)
- first step in bulk-loading a B+ tree
- used in duplicate elimination (why?)
- the sort-merge join algorithm (later in class) involves sorting as a first step

SORTING IN DATABASES

Why don't the standard sorting algorithms work for a database system?

- merge sort
- quick sort
- heap sort

The data typically does not fit in memory!

e.g. how do we sort 1TB of data with 8GB of RAM?

External Merge

EXTERNAL MERGE PROBLEM

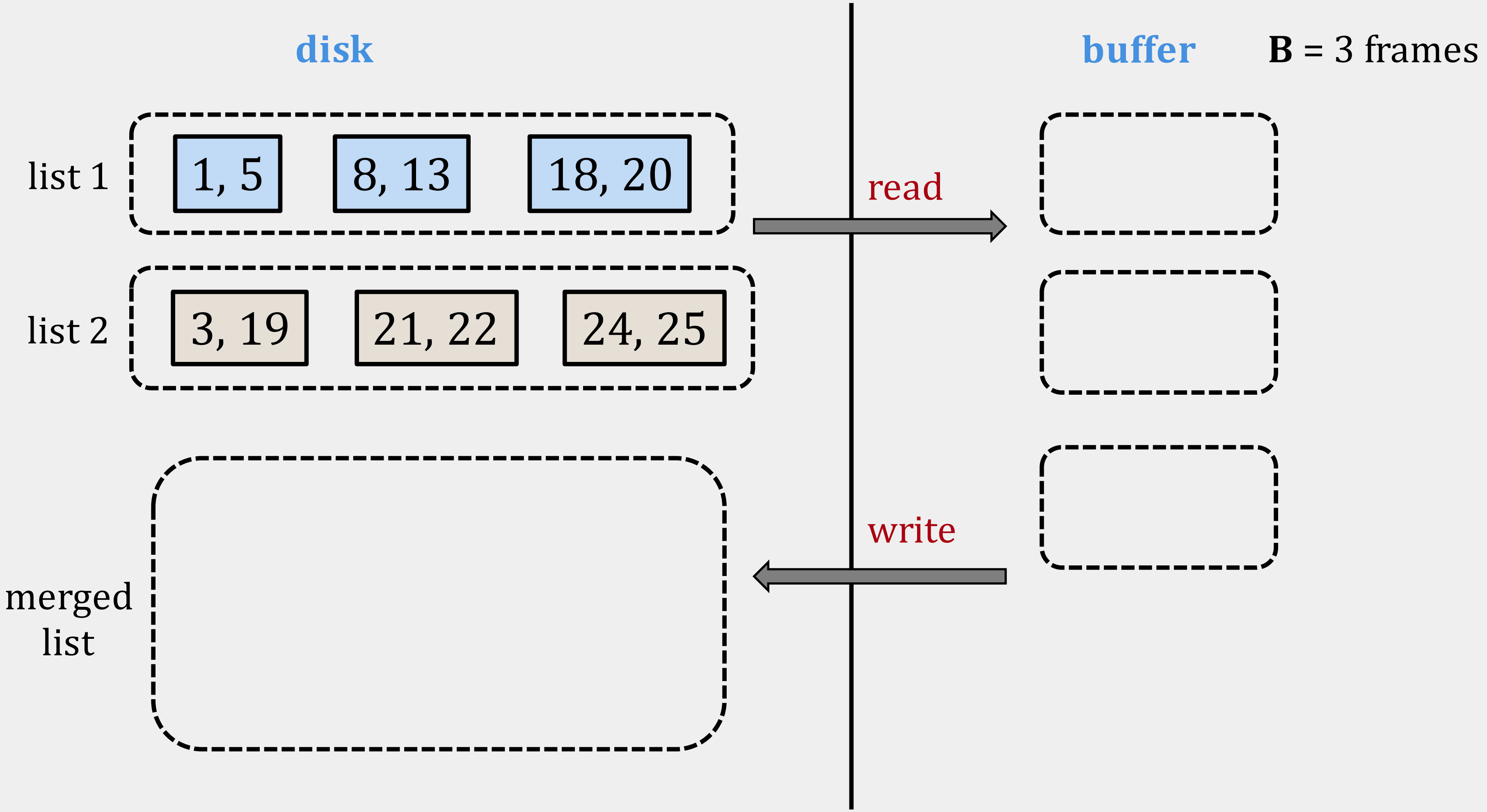
Input: 2 sorted lists (with M and N pages)

Output: 1 merged sorted list (with $M+N$ pages)

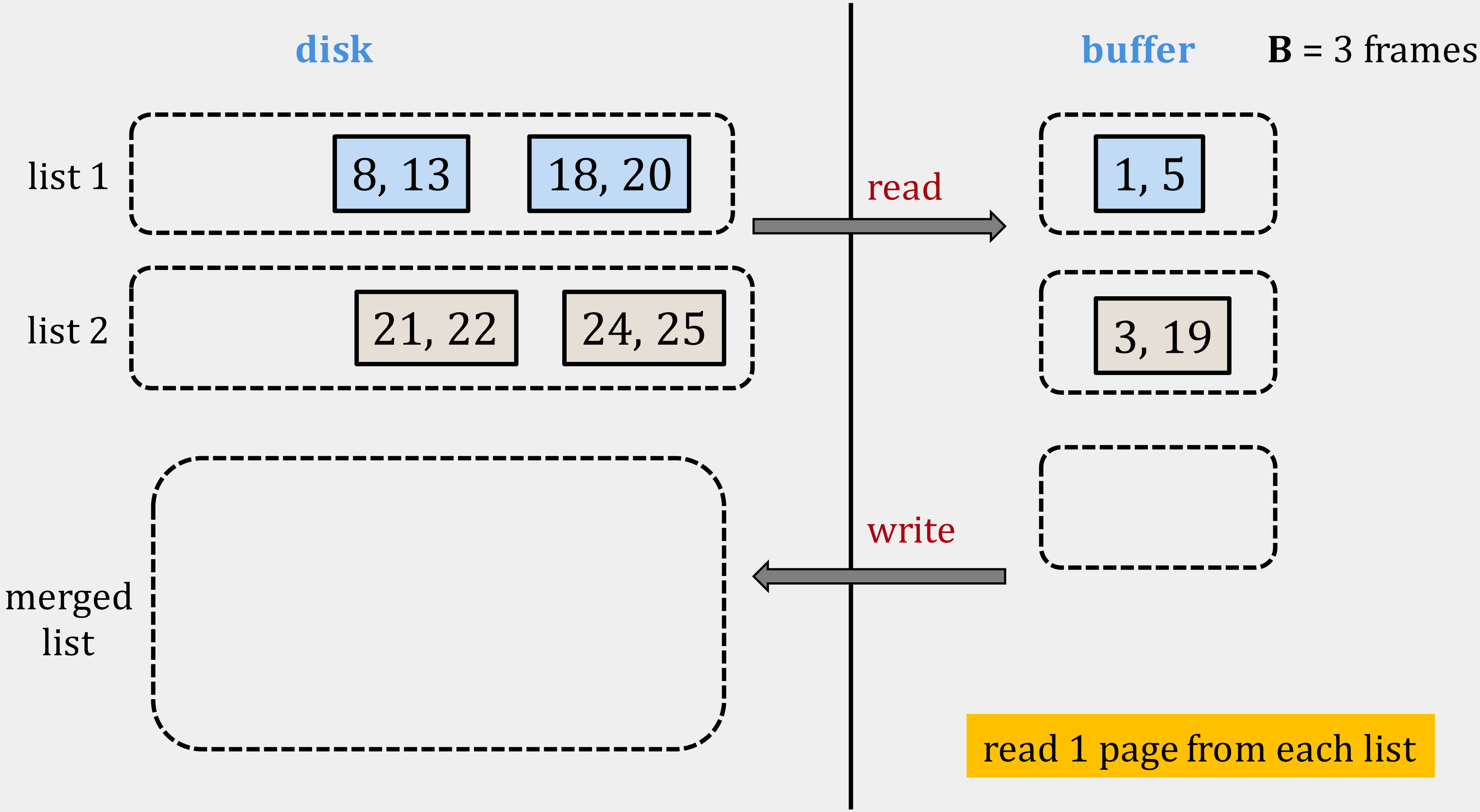
Can we efficiently (in terms of I/O) merge the two lists using a buffer of size at least 3?

Yes, using only $2(M+N)$ I/Os !

EXTERNAL MERGE ALGORITHM

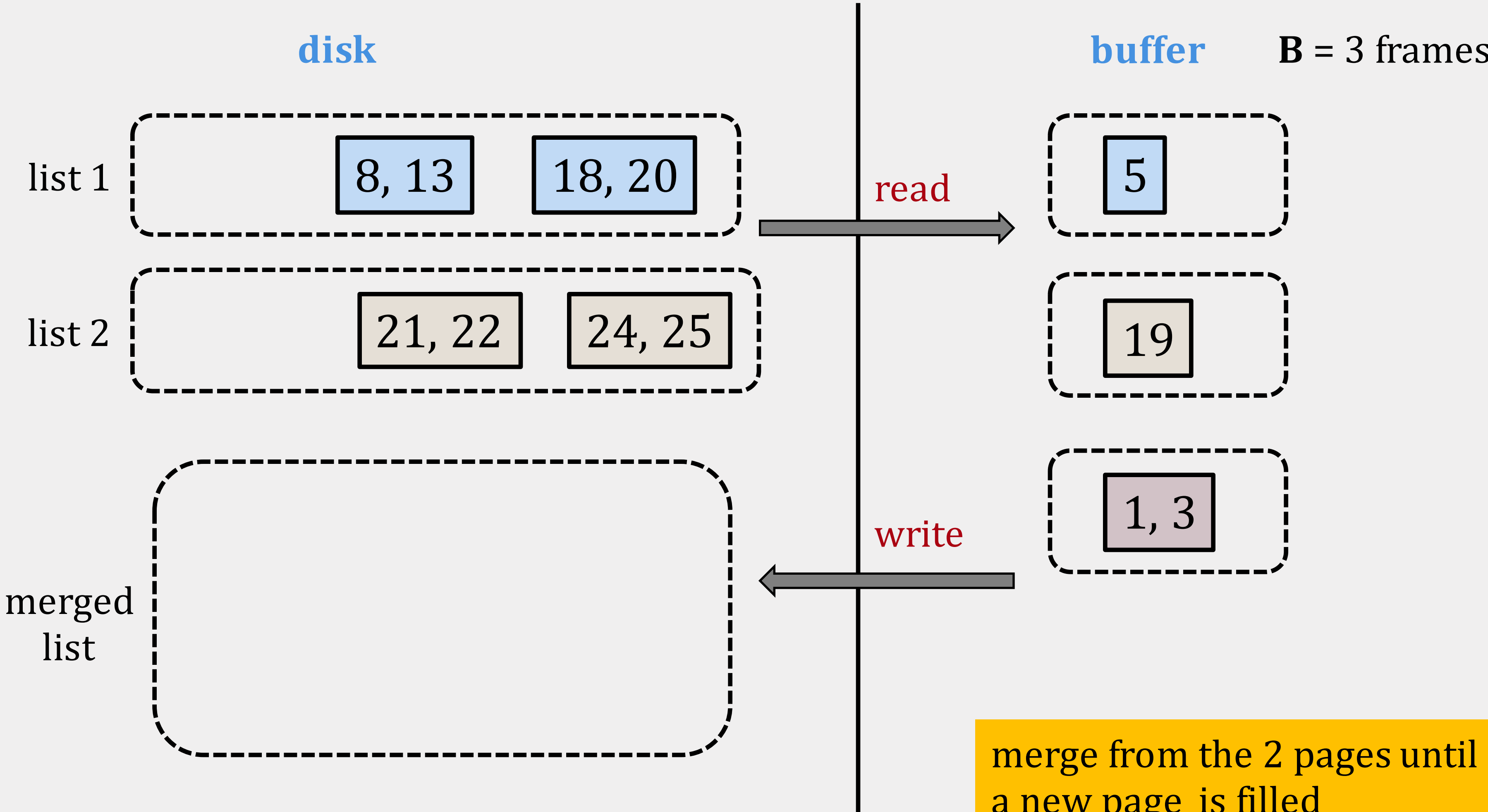


EXTERNAL MERGE ALGORITHM



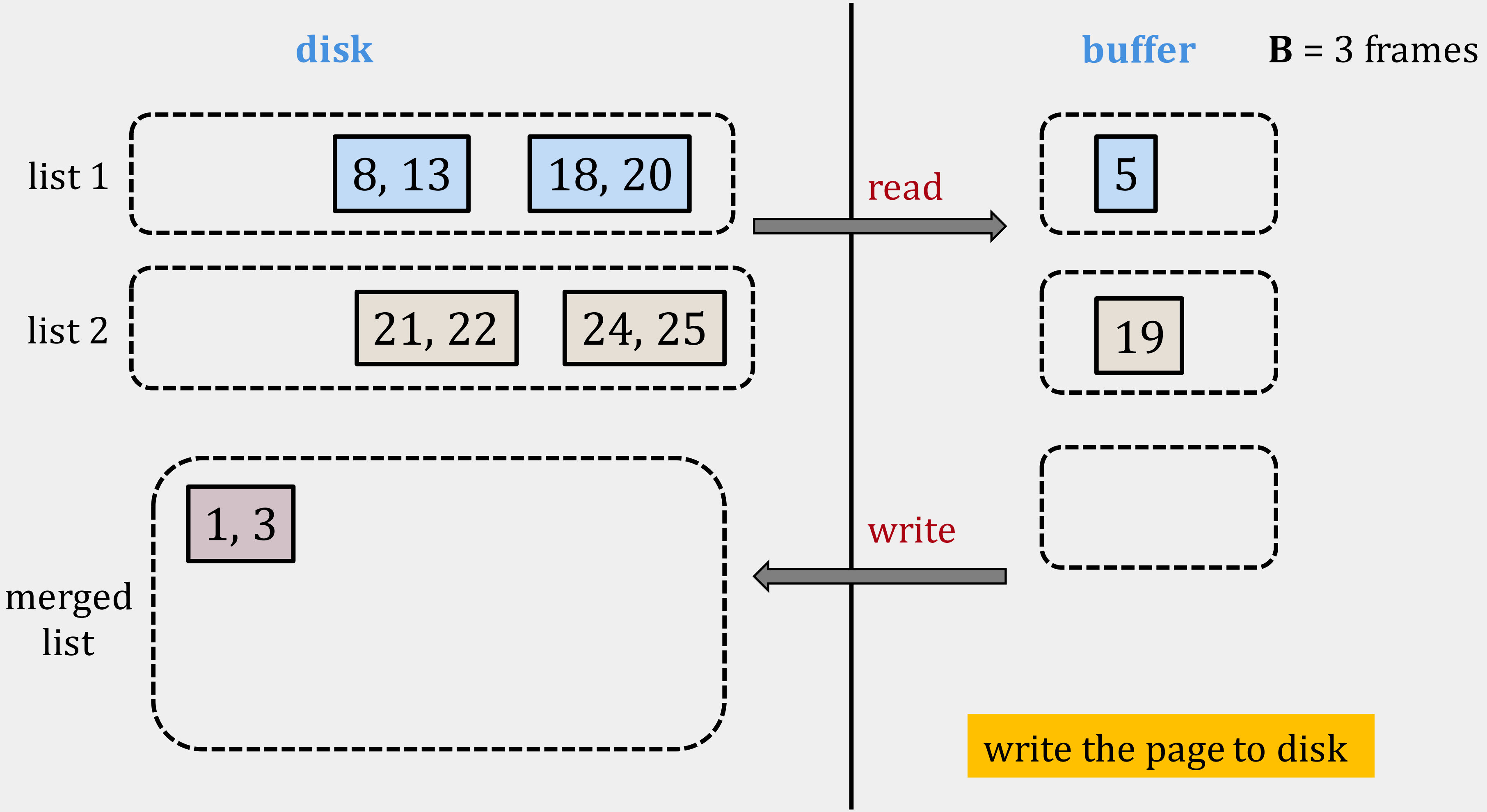
read 1 page from each list

EXTERNAL MERGE ALGORITHM

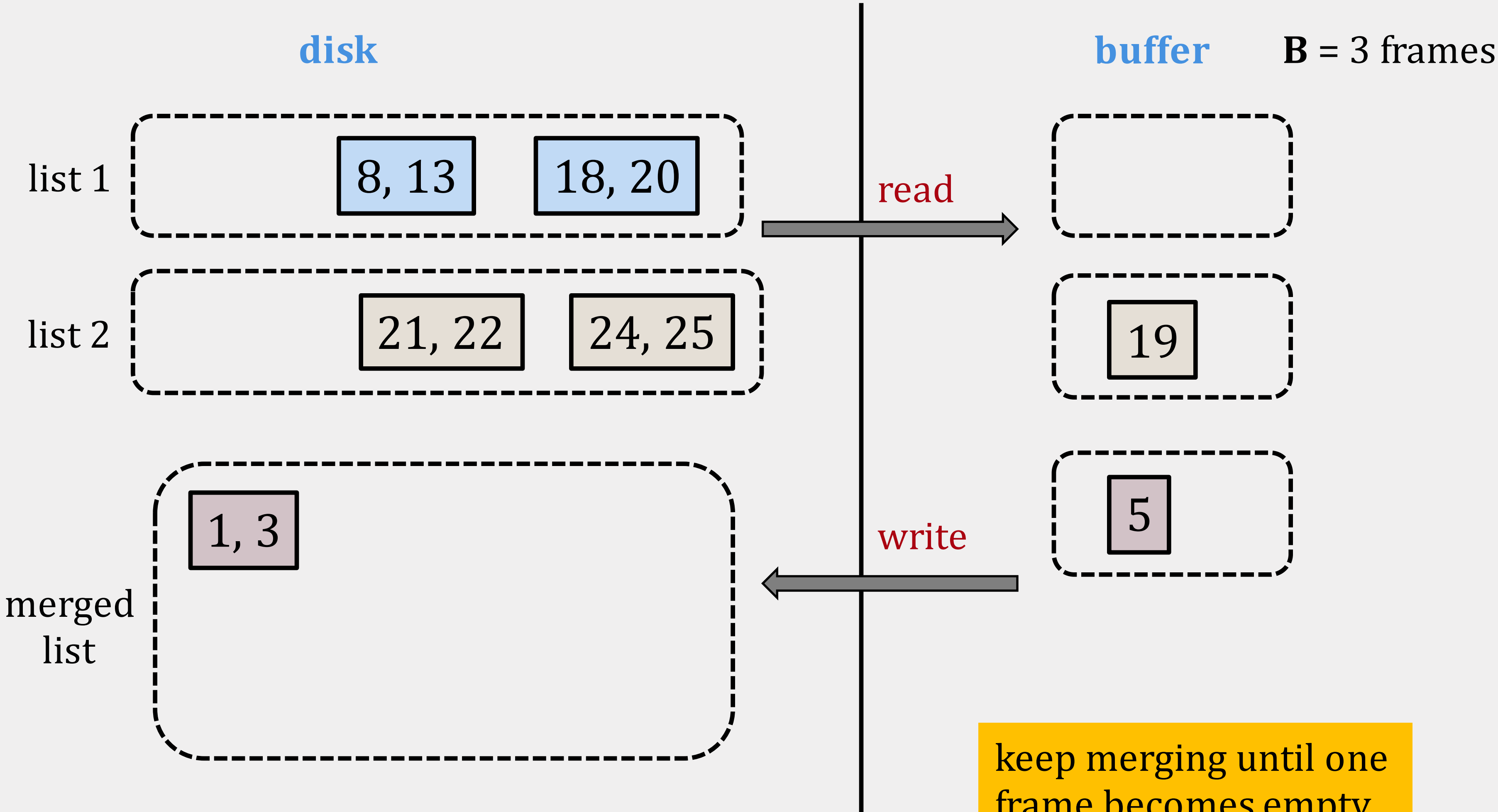


merge from the 2 pages until a new page is filled

EXTERNAL MERGE ALGORITHM

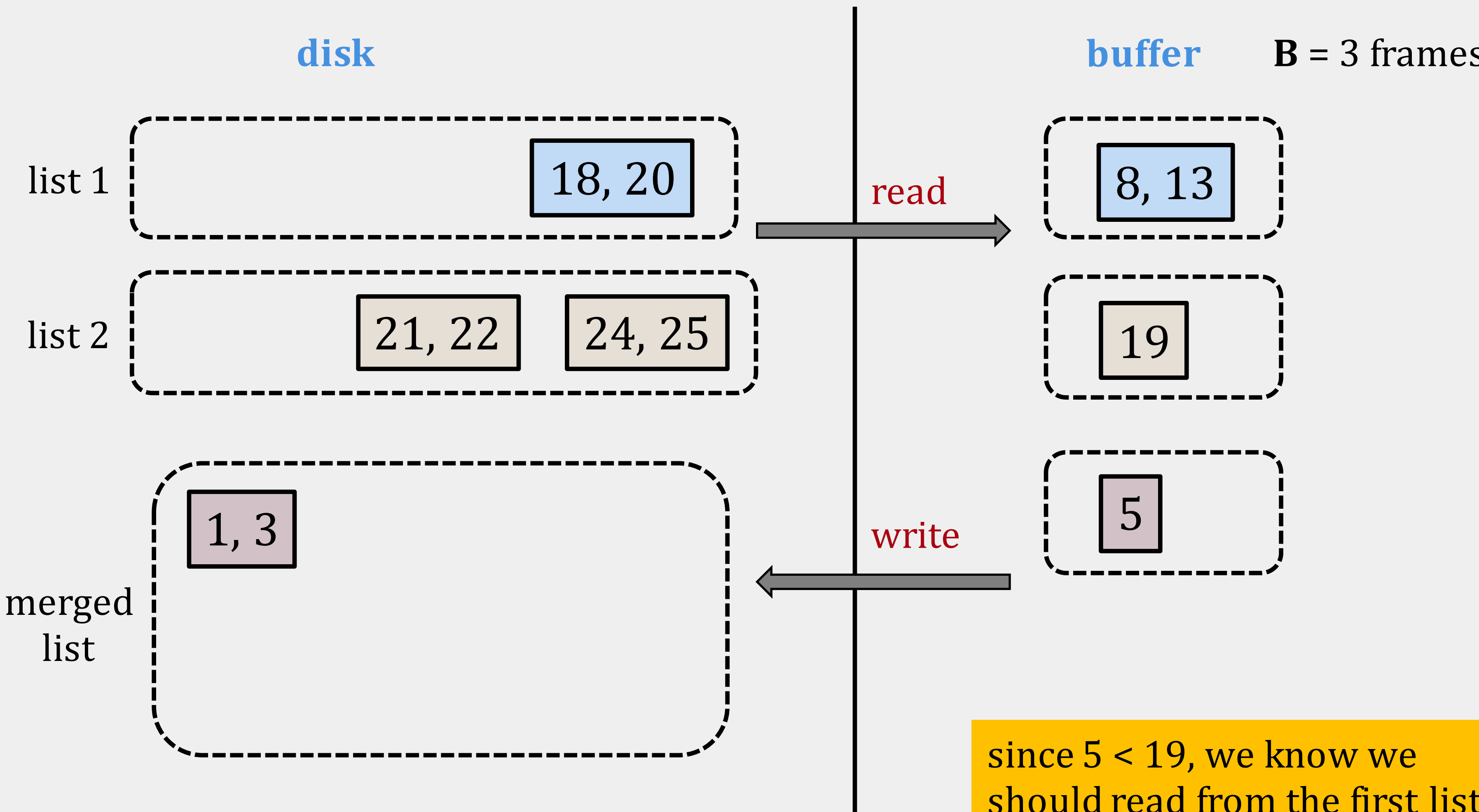


EXTERNAL MERGE ALGORITHM



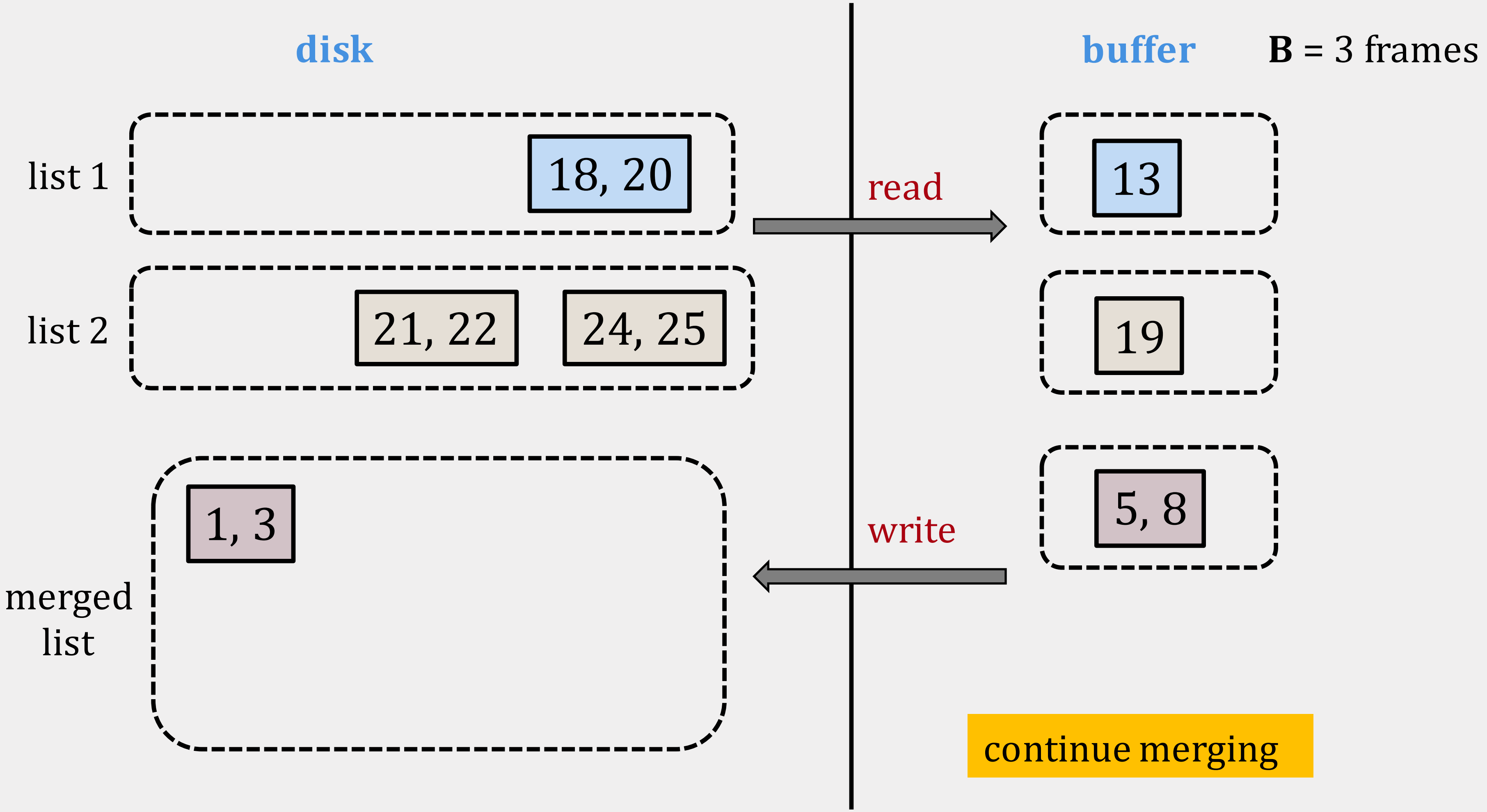
keep merging until one frame becomes empty

EXTERNAL MERGE ALGORITHM

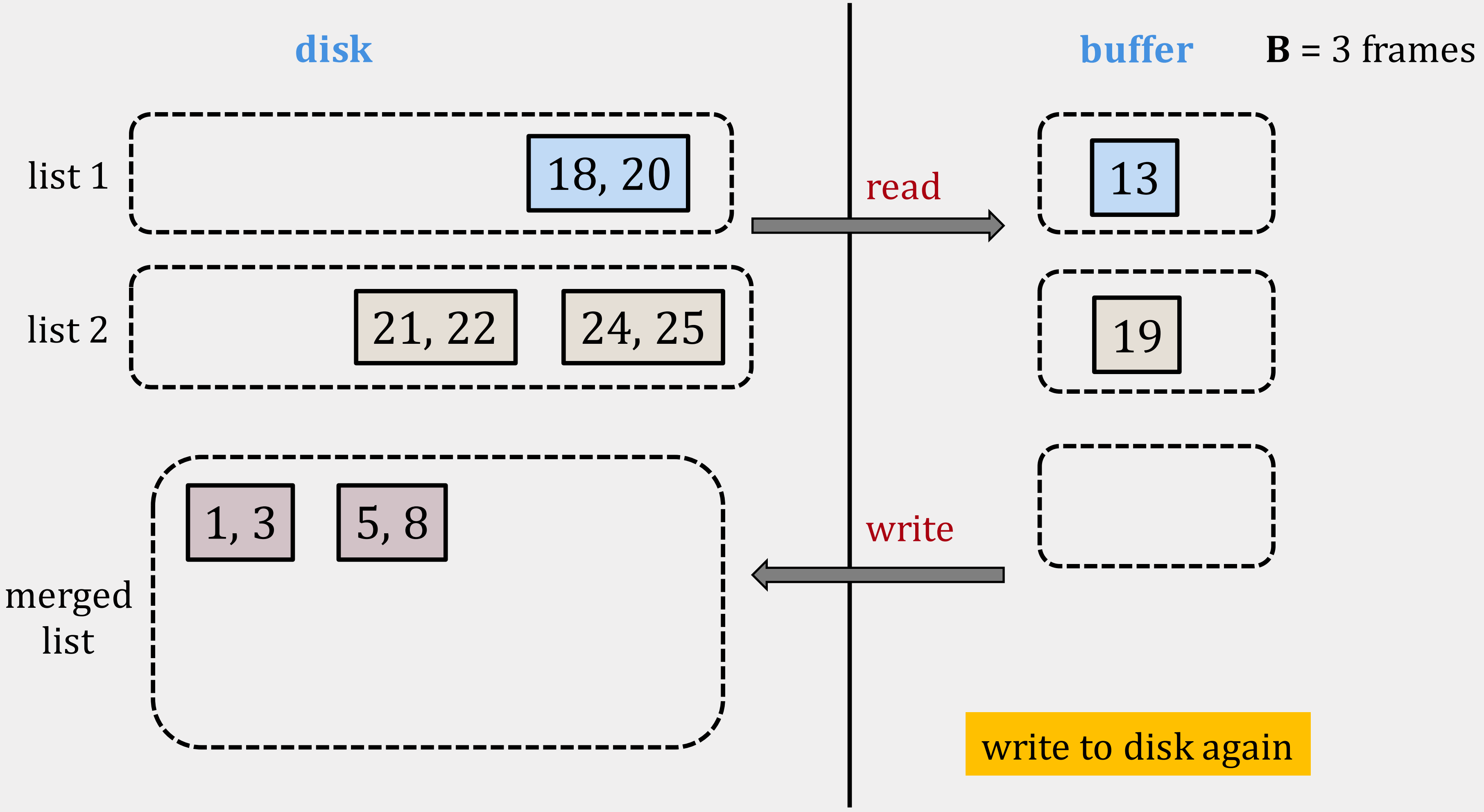


since 5 < 19, we know we should read from the first list!

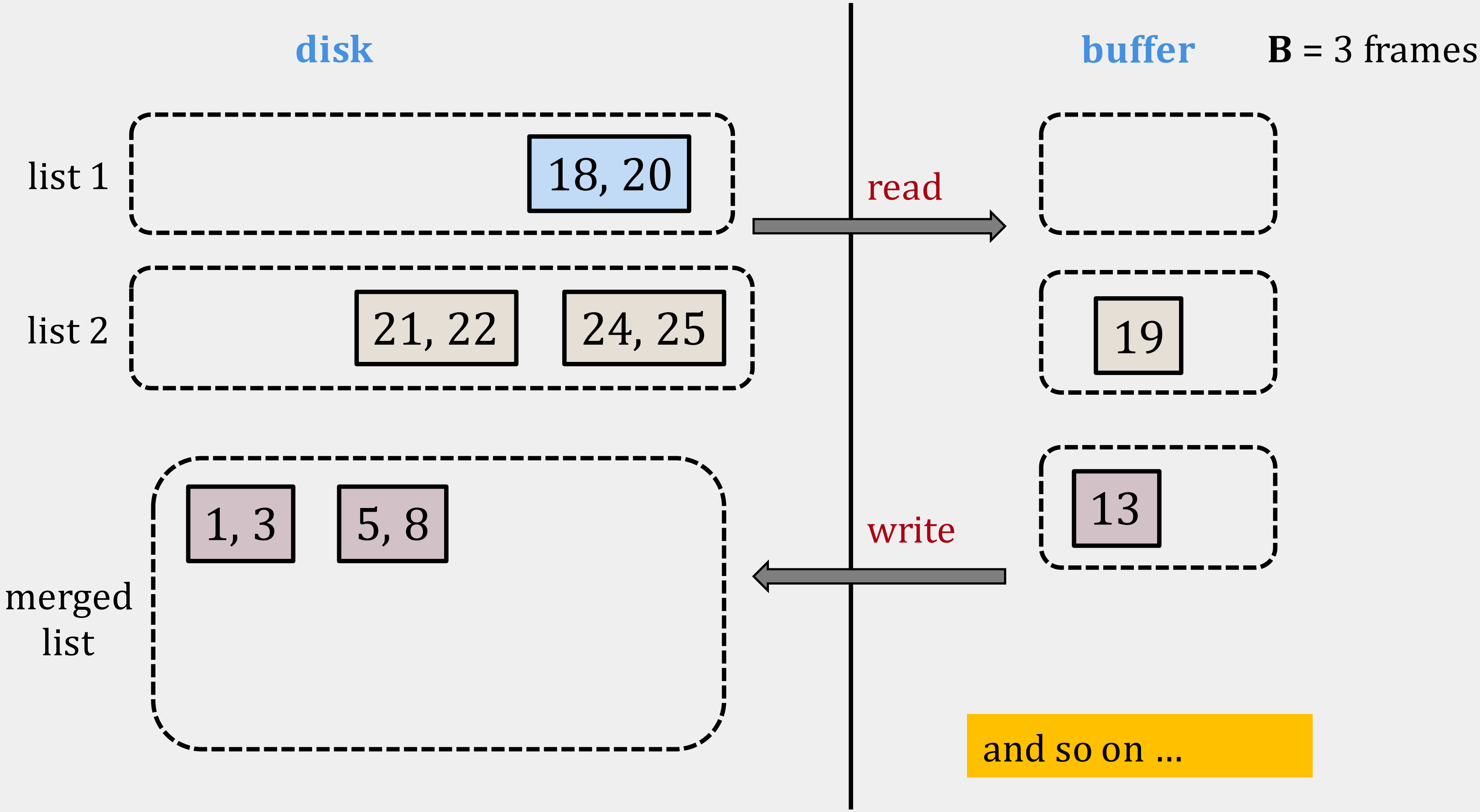
EXTERNAL MERGE ALGORITHM



EXTERNAL MERGE ALGORITHM



EXTERNAL MERGE ALGORITHM



EXTERNAL MERGE COST

We can merge 2 sorted lists of M and N pages using 3 buffer frames with

I/O cost = $2\ (M+N)$

When we have $B+1$ buffer pages, we can merge B lists with the same I/O cost

External Merge Sort

THE SORTING PROBLEM

- **B** available pages in buffer pool
- a relation R of size **N** pages (where **N** > **B**)

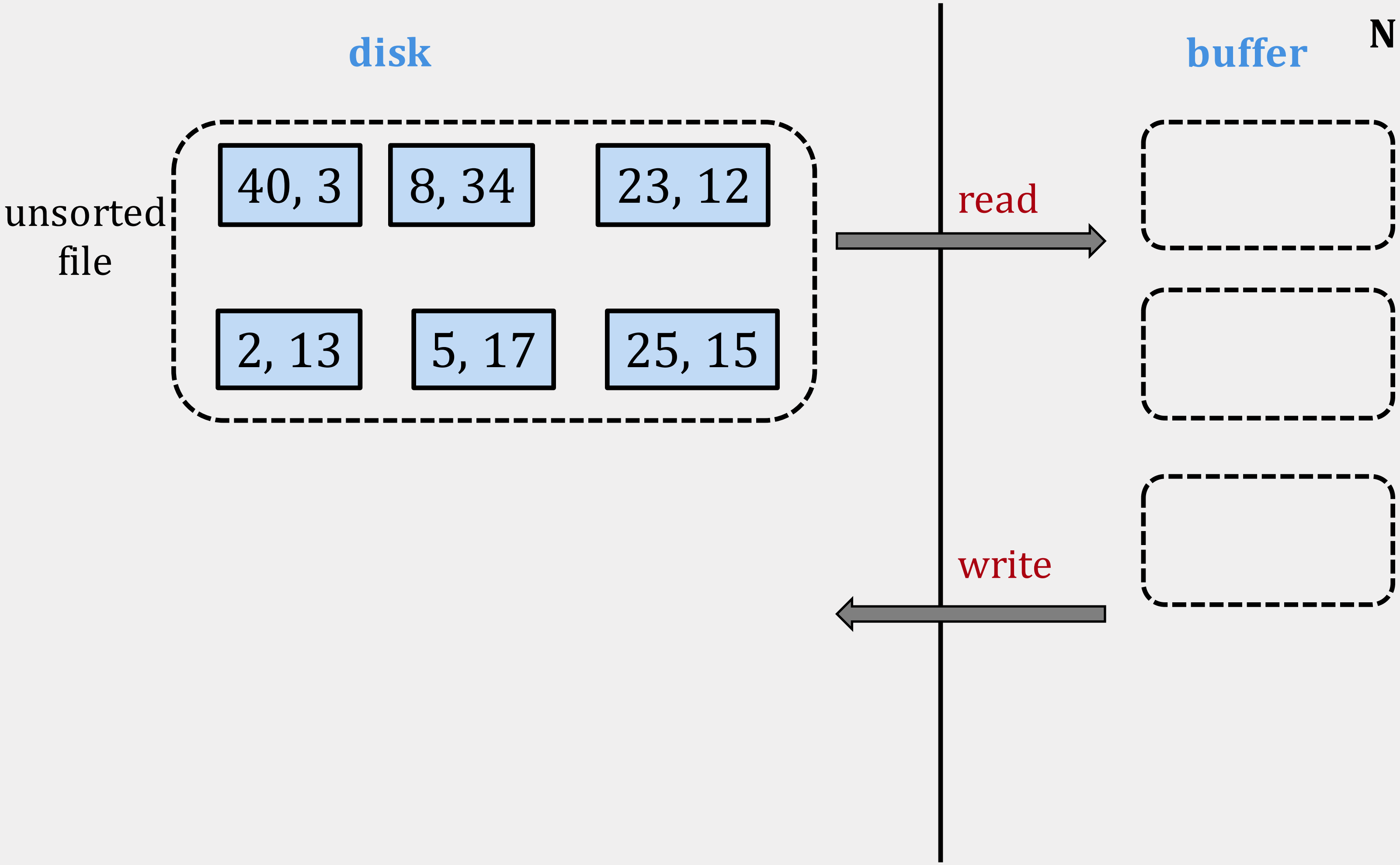
SORTING: output the same relation sorted on a given attribute

KEY IDEA

- split into chunks small enough to sort in memory (called runs)
- merge groups of runs using the external merge algorithm
- keep merging the resulting runs (each time is called a pass) until left with a single sorted file

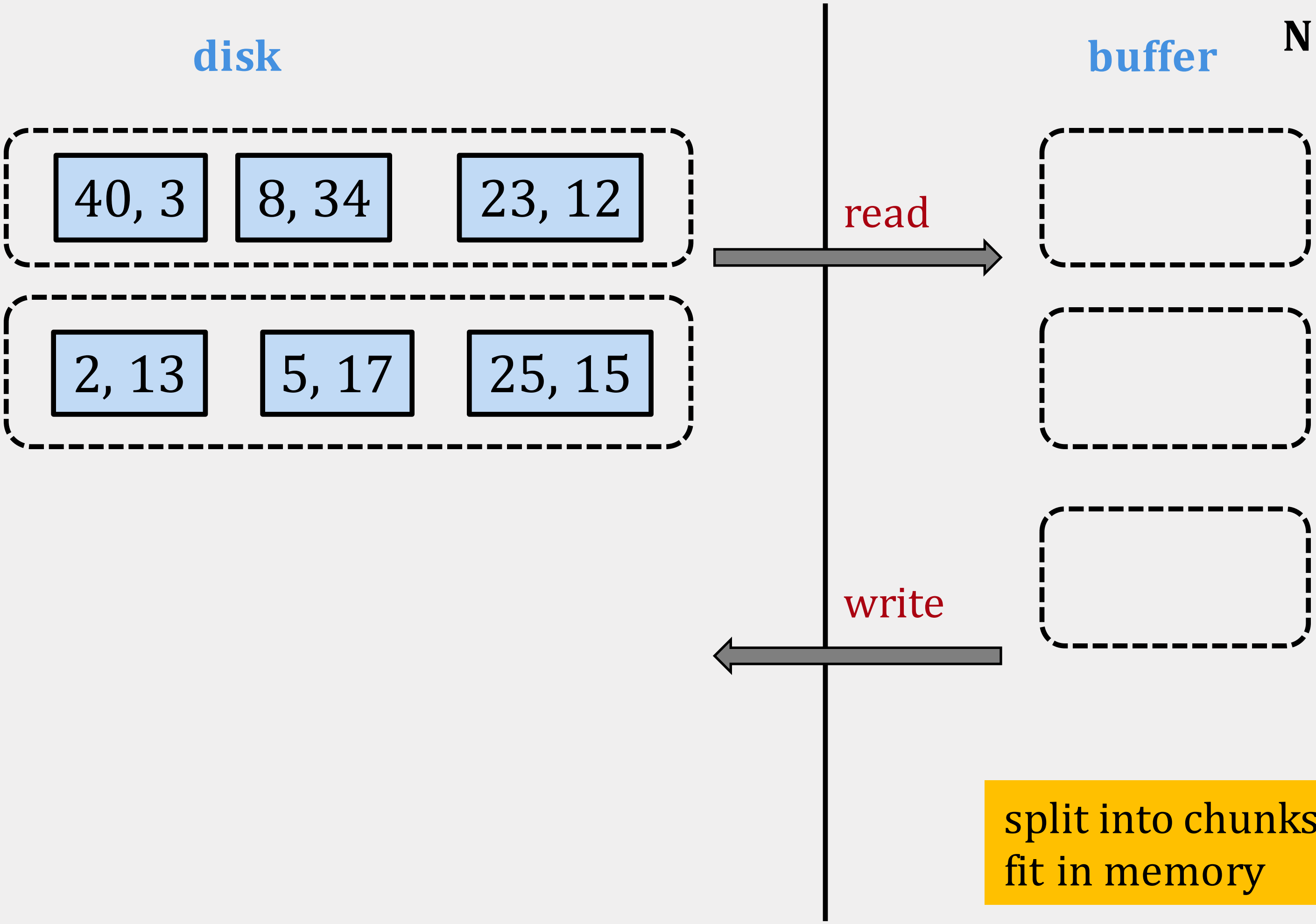
WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



WARM UP: 2-WAY SORT

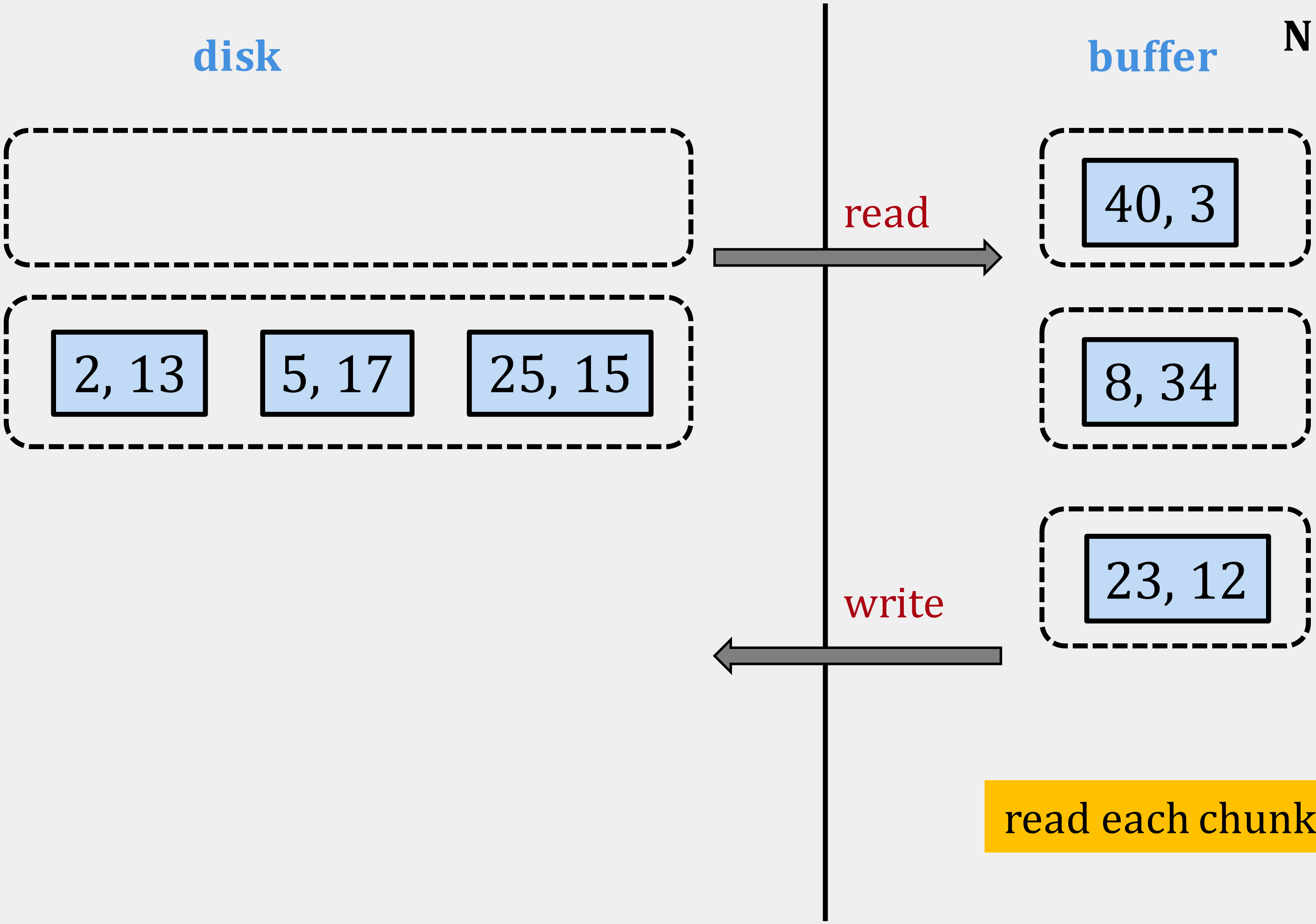
B = 3 frames
N = 6 pages



split into chunks that fit in memory

WARM UP: 2-WAY SORT

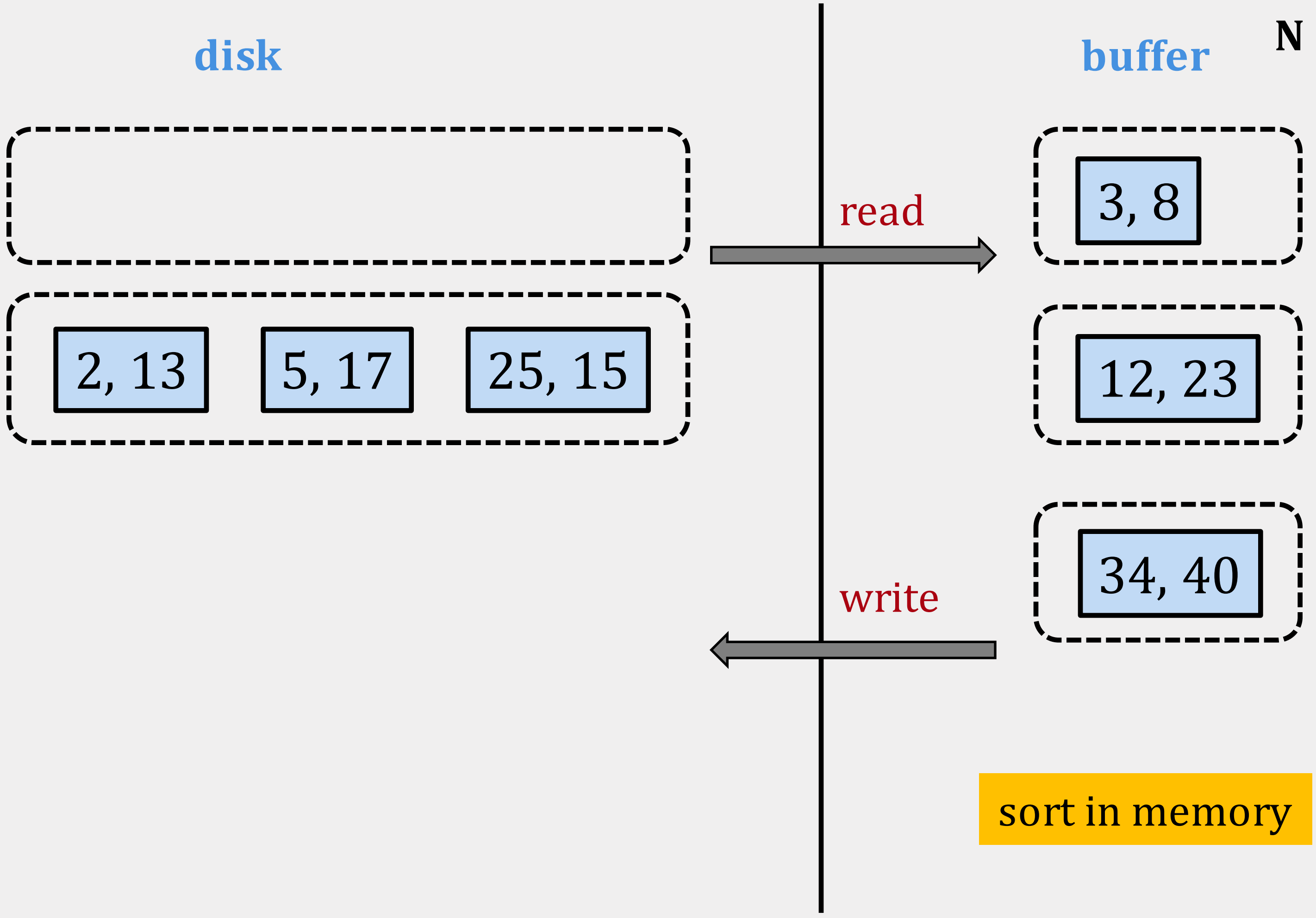
B = 3 frames
N = 6 pages



read each chunk in memory

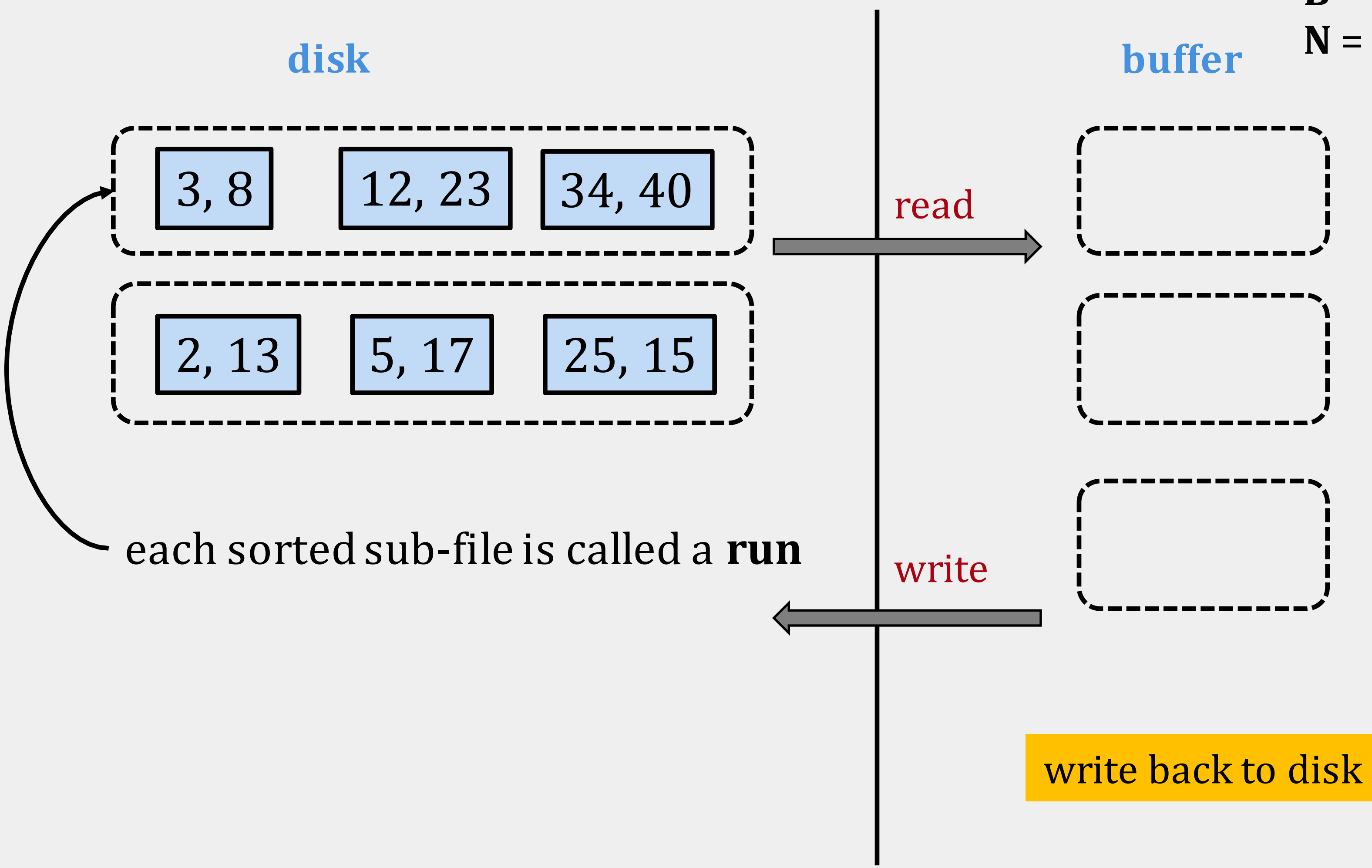
WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



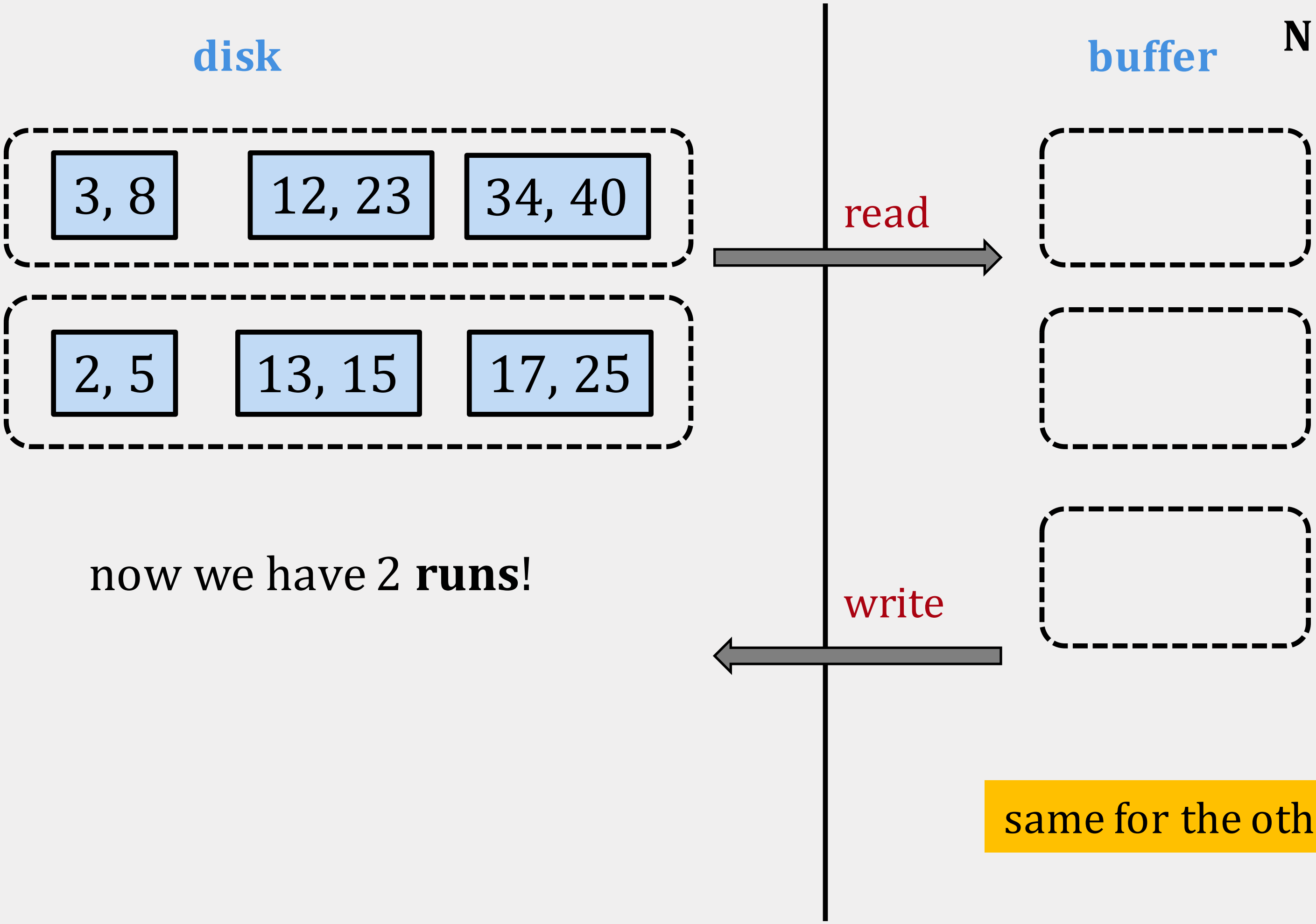
WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



WARM UP: 2-WAY SORT

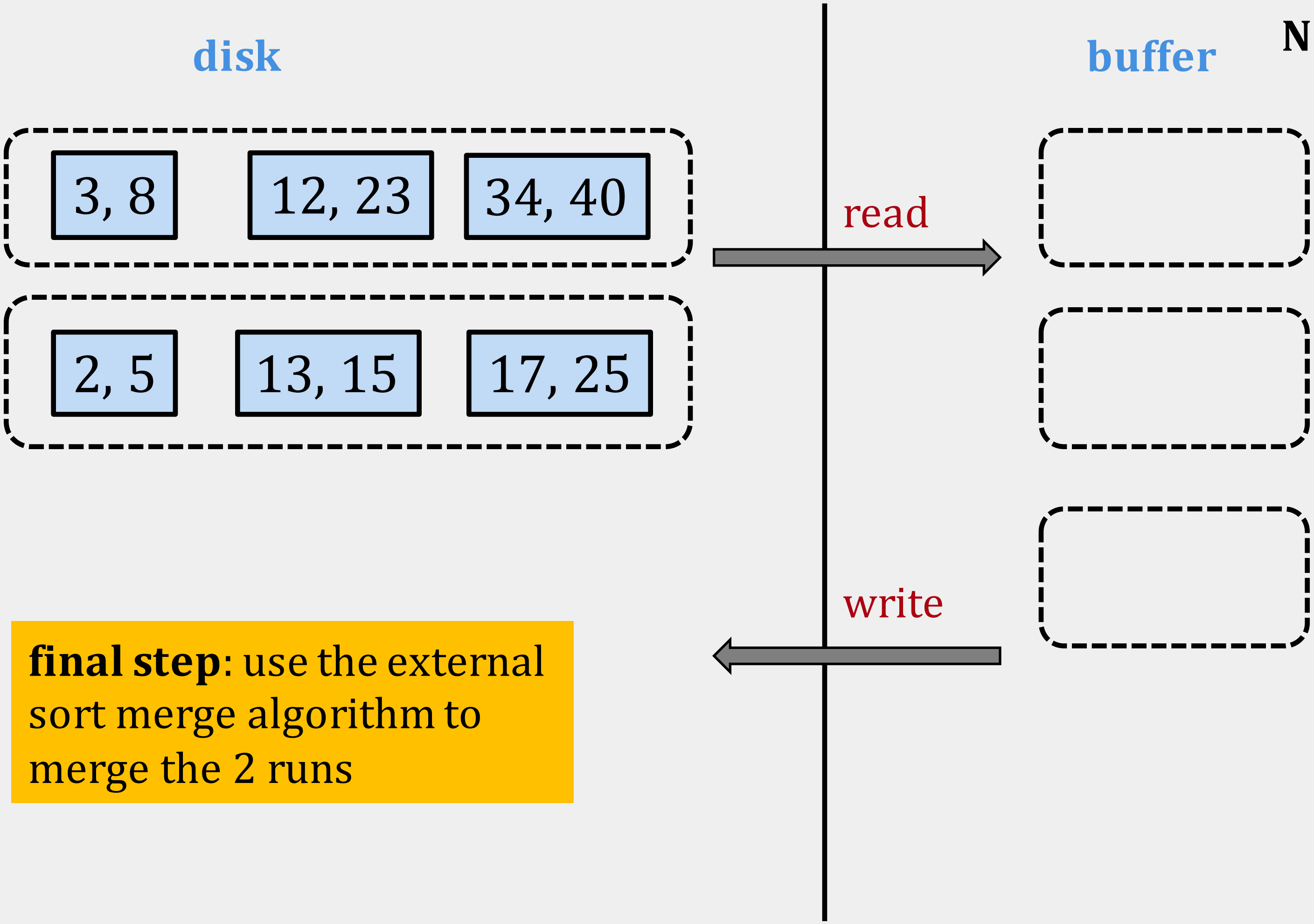
B = 3 frames
N = 6 pages



same for the other chunk

WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



final step: use the external sort merge algorithm to merge the 2 runs

CALCULATING THE I/O COST

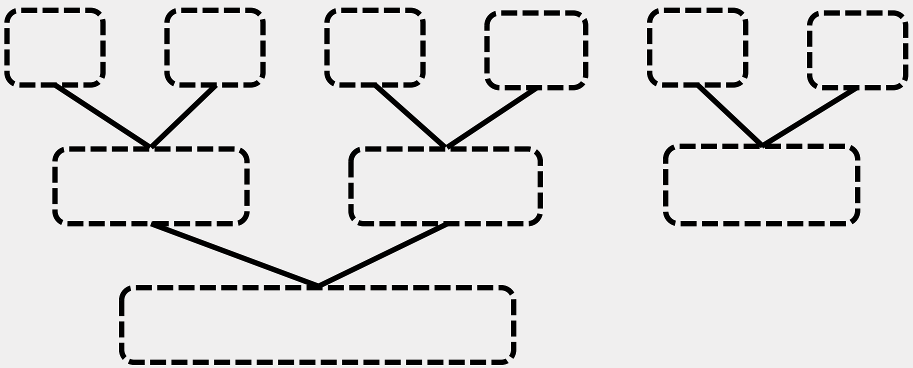
In our example, **B**= 3 buffer pages, **N** = 6 pages

- Pass **0**: creating the first runs
  - 1 read + 1 write for every page
  - total cost = $6 * (1+1) = 12$ I/Os
- Pass **1**: external merge sort
  - total cost = $2 * (3+3) = 12$ I/Os

So 24 I/Os in total

I/O COST: SIMPLIFIED VERSION

Assume for now that we initially create **N** runs, each run consisting of a single page



pass 0: N runs, each 1 page

pass 1: merge into N/2 runs

pass 2: merge into N/4 runs

- We need $\lceil log_2 N \rceil + 1$ passes to sort the whole file
- Each pass needs $2N$ I/Os

total I/O cost = $2N(\lceil log_2 N \rceil + 1)$

CAN WE DO BETTER?

- The 2-way merge algorithm only uses 3 buffer pages
- But we have more available memory!

Key idea: use as much of the available memory as possible in every pass

- reducing the number of passes reduces I/O

EXTERNAL SORT: I/O COST

Suppose we have $B \geq 3$ buffer pages available

$$2N(\lceil log_2 N \rceil + 1) \Longrightarrow 2N\left(\left\lceil log_2 \frac{N}{B} \right\rceil + 1\right) \Longrightarrow 2N\left(\left\lceil log_{B-1} \frac{N}{B} \right\rceil + 1\right)$$

- initial runs of length 1
- 3-way merge

increase the length of the initial runs to B

merge B-1 runs at a time

NUMBER OF PASSES

N	B=3	B=17	B=257
100	7	2	1
10,000	13	4	2
1,000,000	20	5	3
10,000,000	23	6	3
100,000,000	26	7	4
1,000,000,000	30	8	4

Optimizing Merge Sort

REPLACEMENT SORT

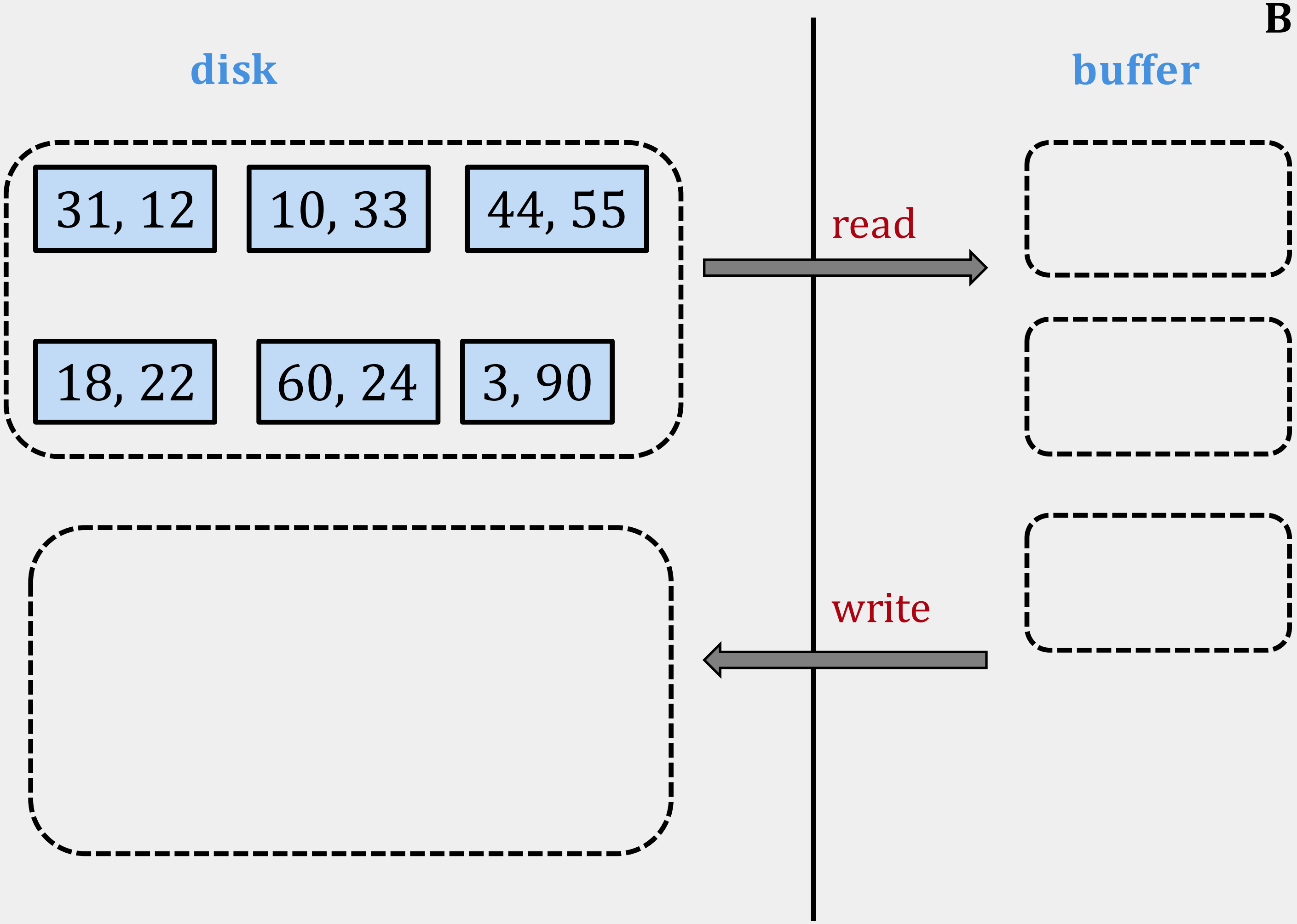
- used as an alternative for the sorting in pass 0
- creates runs of *average* size $2B$ (instead of B)

Algorithm

- read *B-1* pages in memory (keep as sorted heap)
- move smallest record (that is greater than the largest element in buffer) to output buffer
- read a new record *r* and insert into the sorted heap

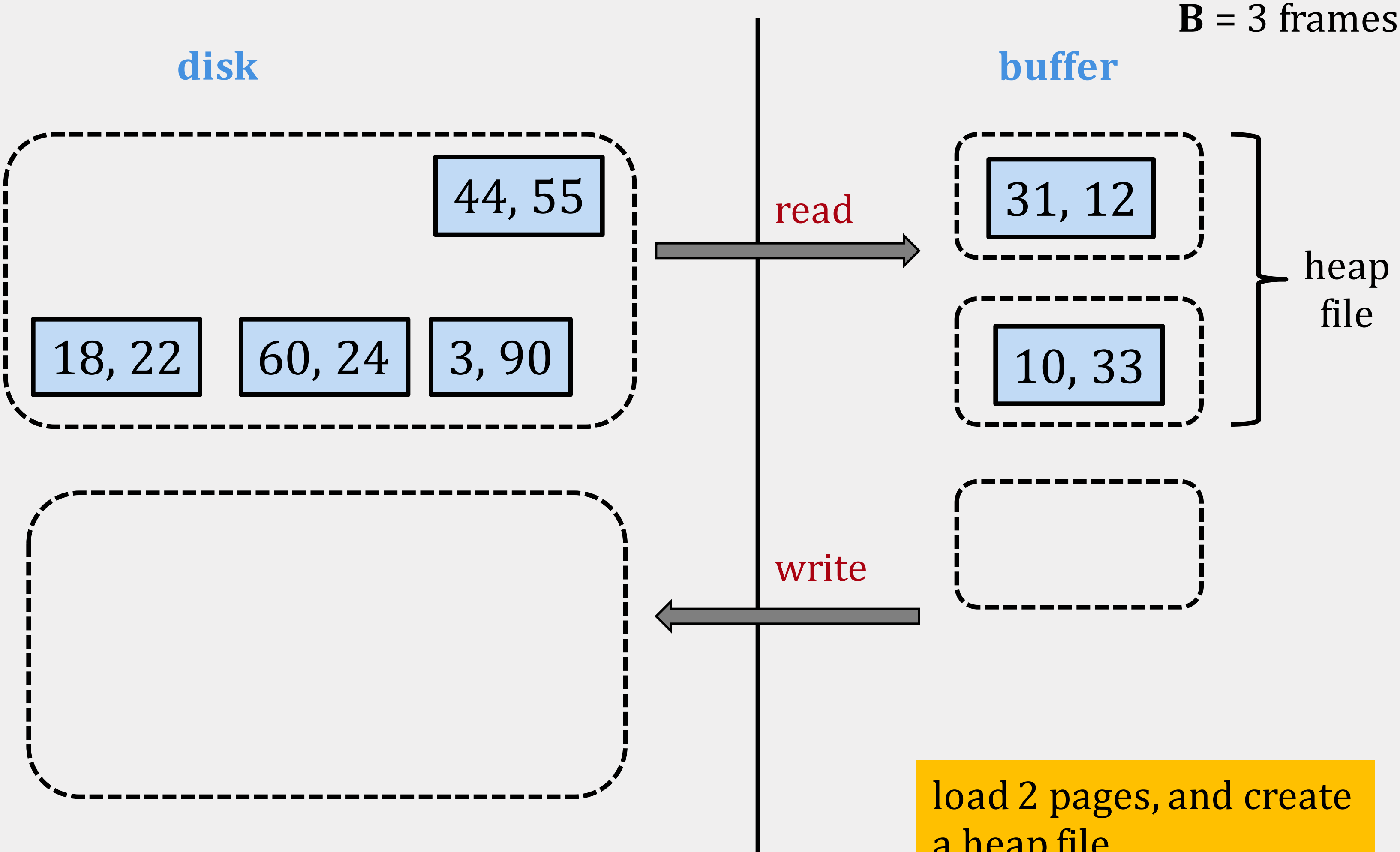
REPLACEMENT SORT: EXAMPLE

B = 3 frames



REPLACEMENT SORT: EXAMPLE

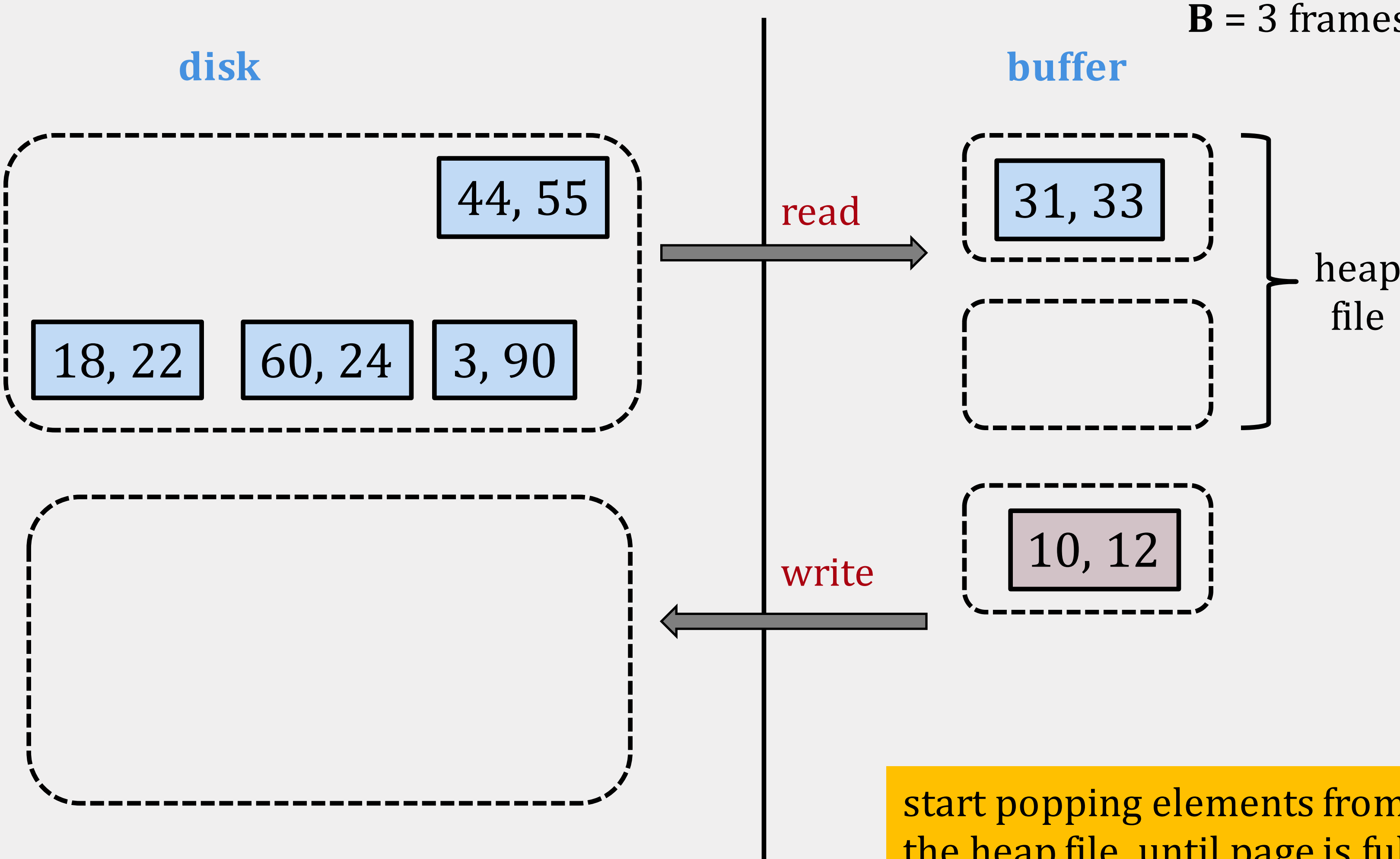
B = 3 frames



load 2 pages, and create a heap file

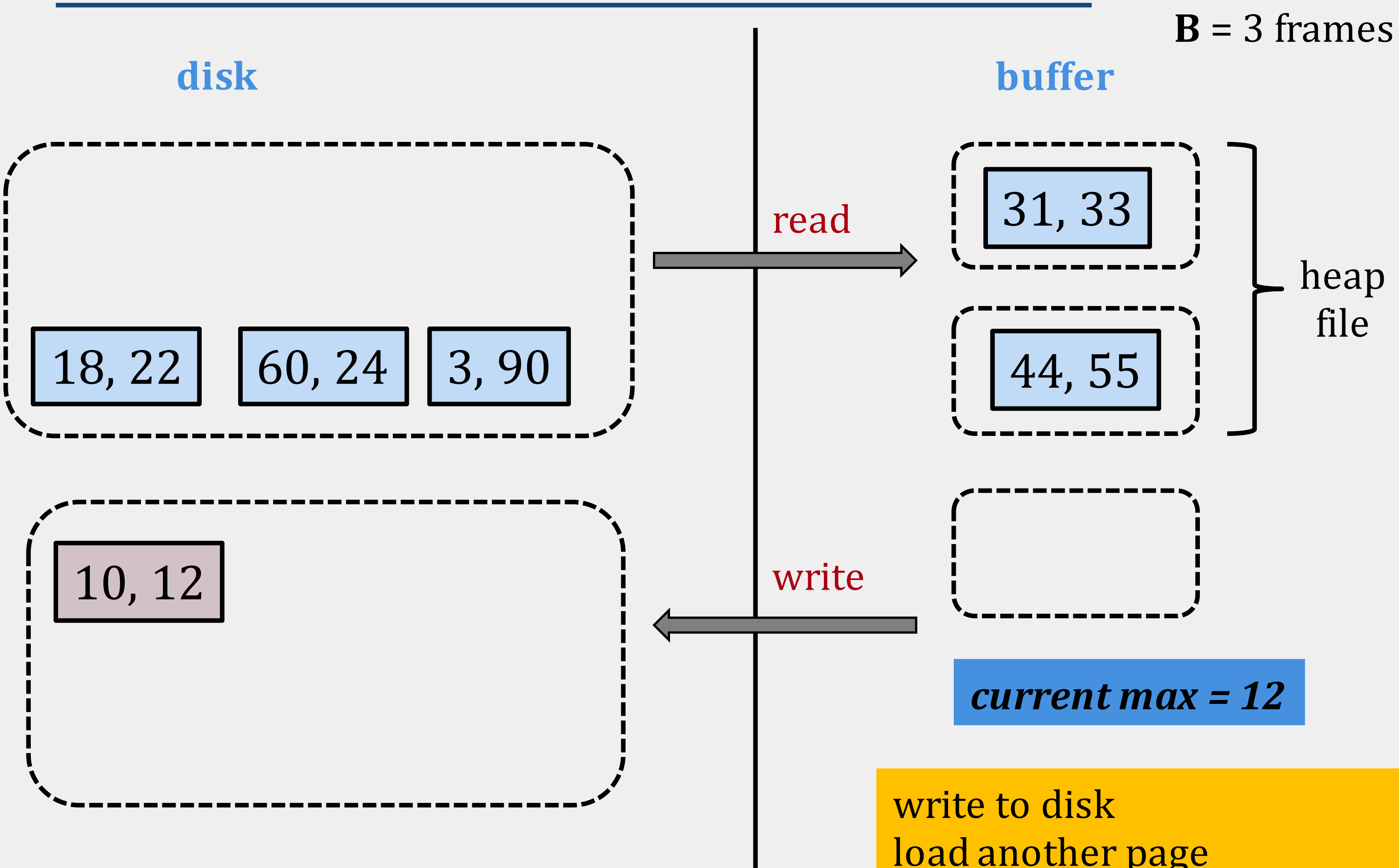
REPLACEMENT SORT: EXAMPLE

B = 3 frames



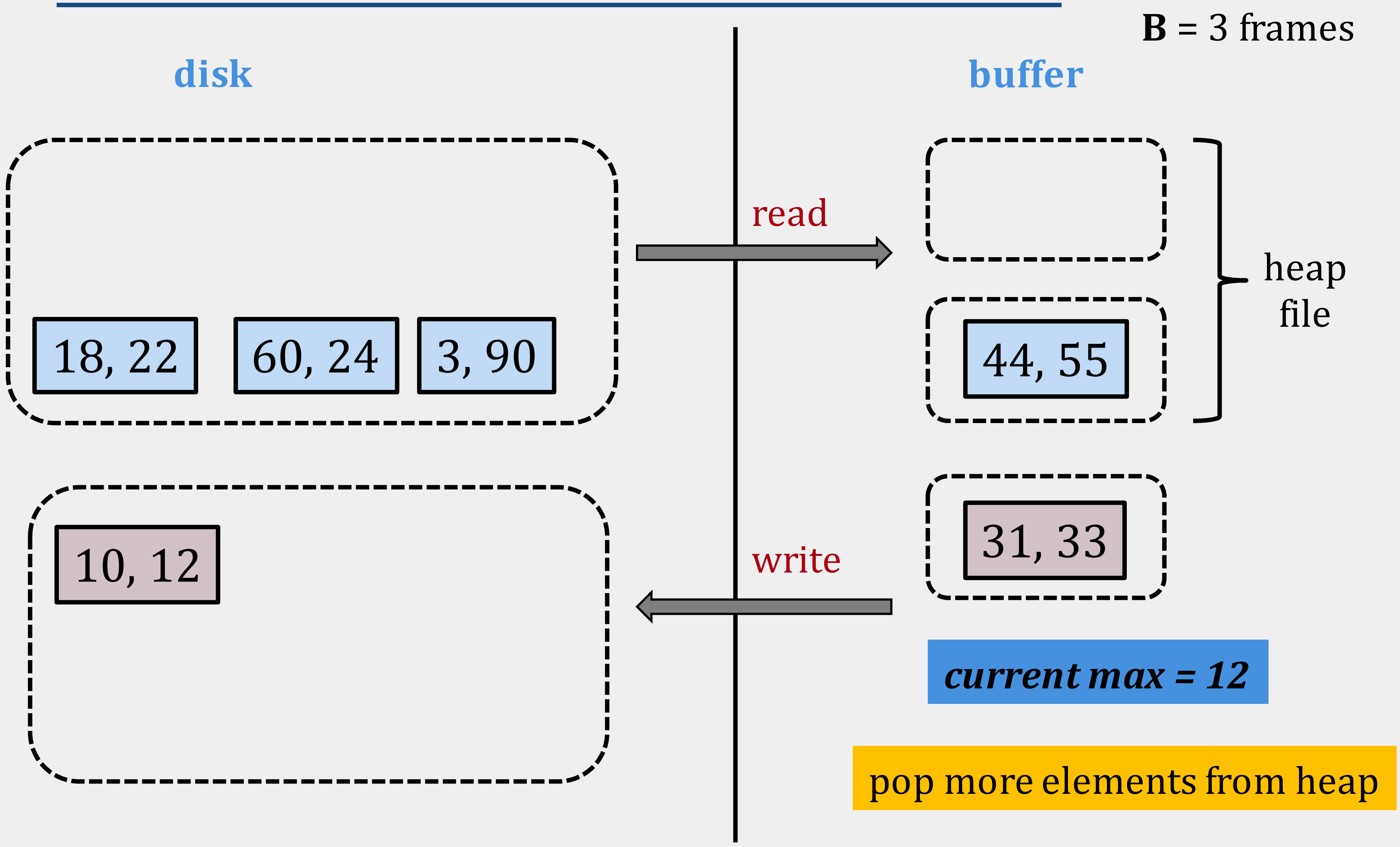
start popping elements from the heap file, until page is full

REPLACEMENT SORT: EXAMPLE

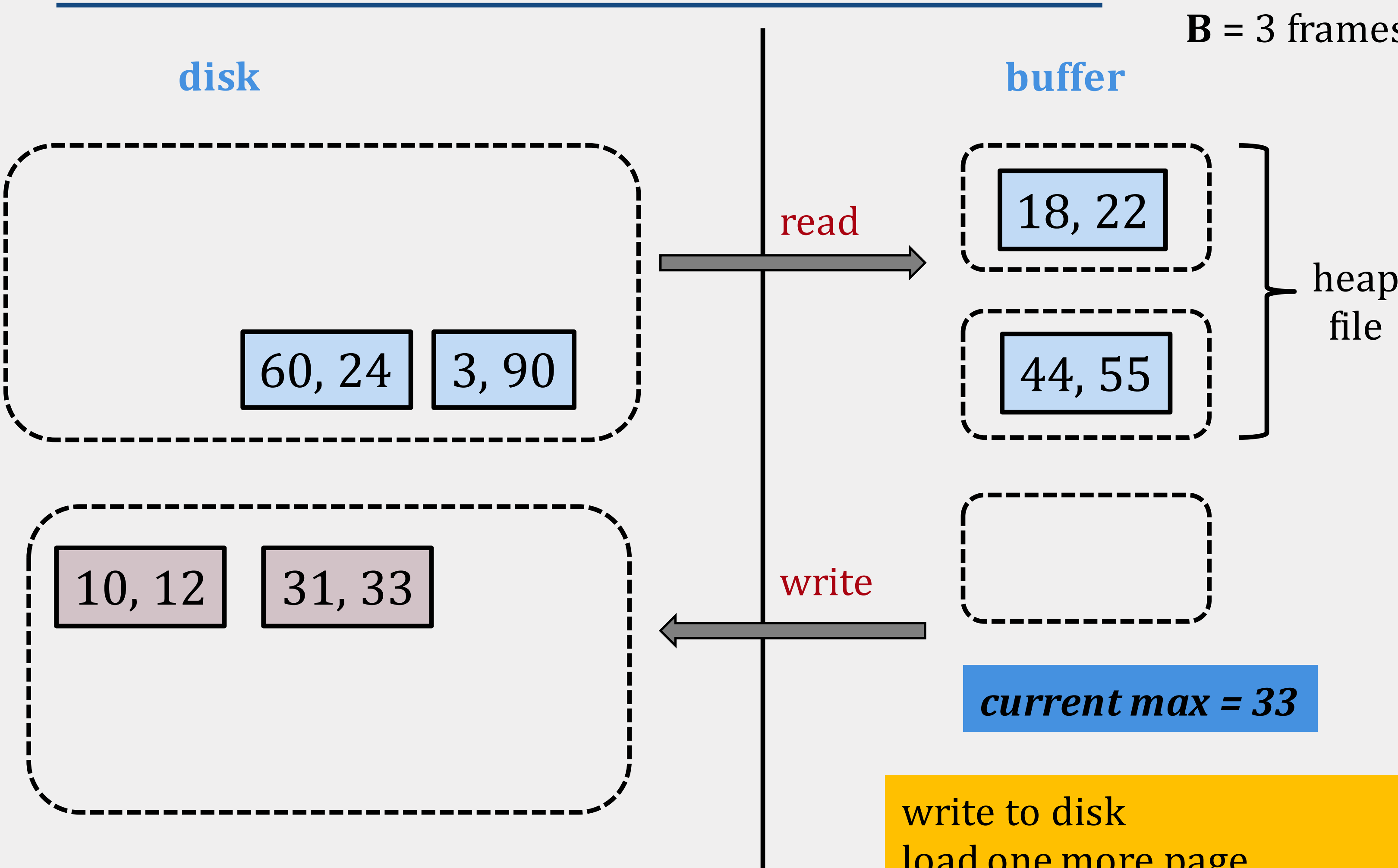


write to disk
load another page

REPLACEMENT SORT: EXAMPLE

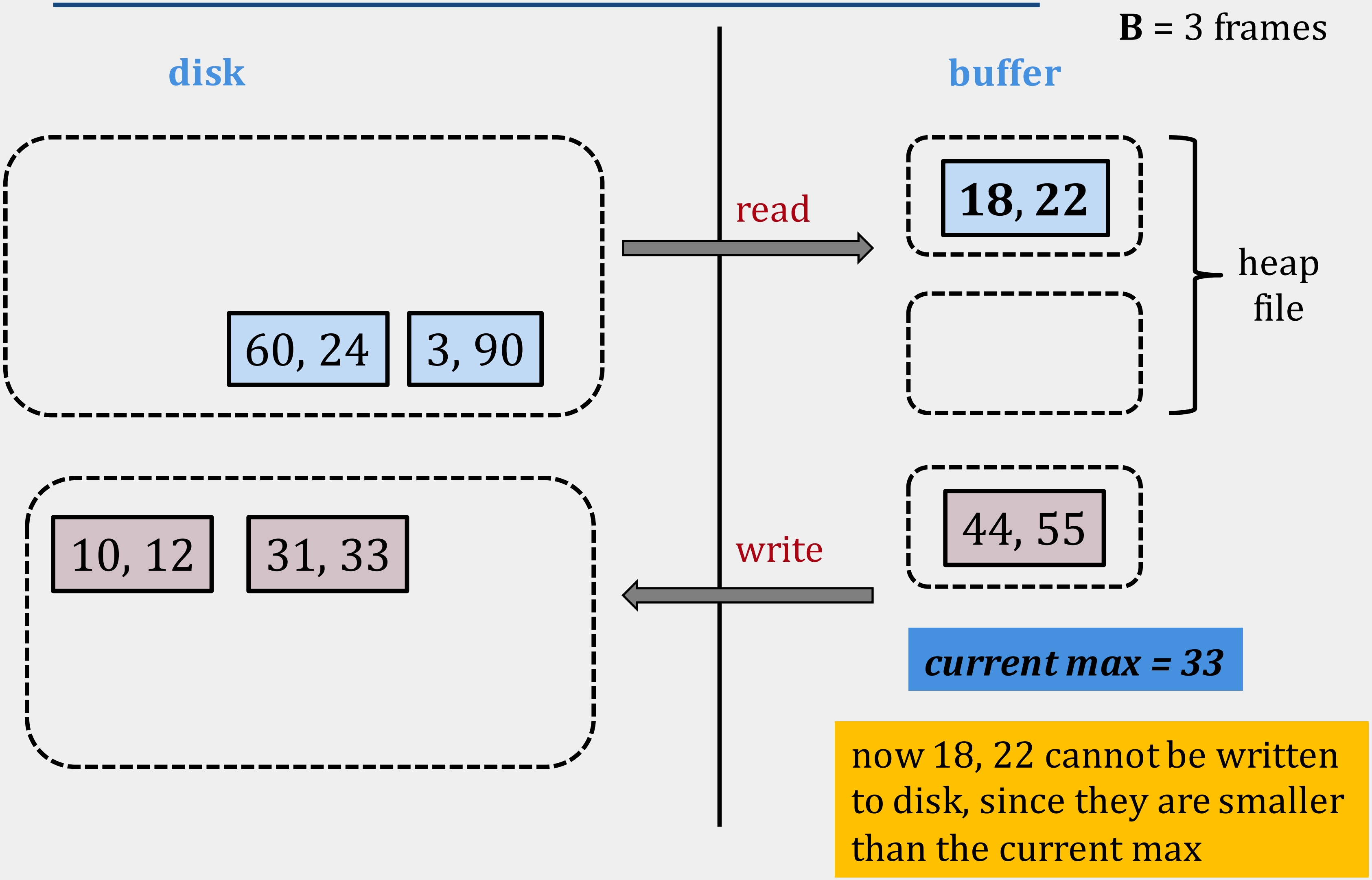


REPLACEMENT SORT: EXAMPLE

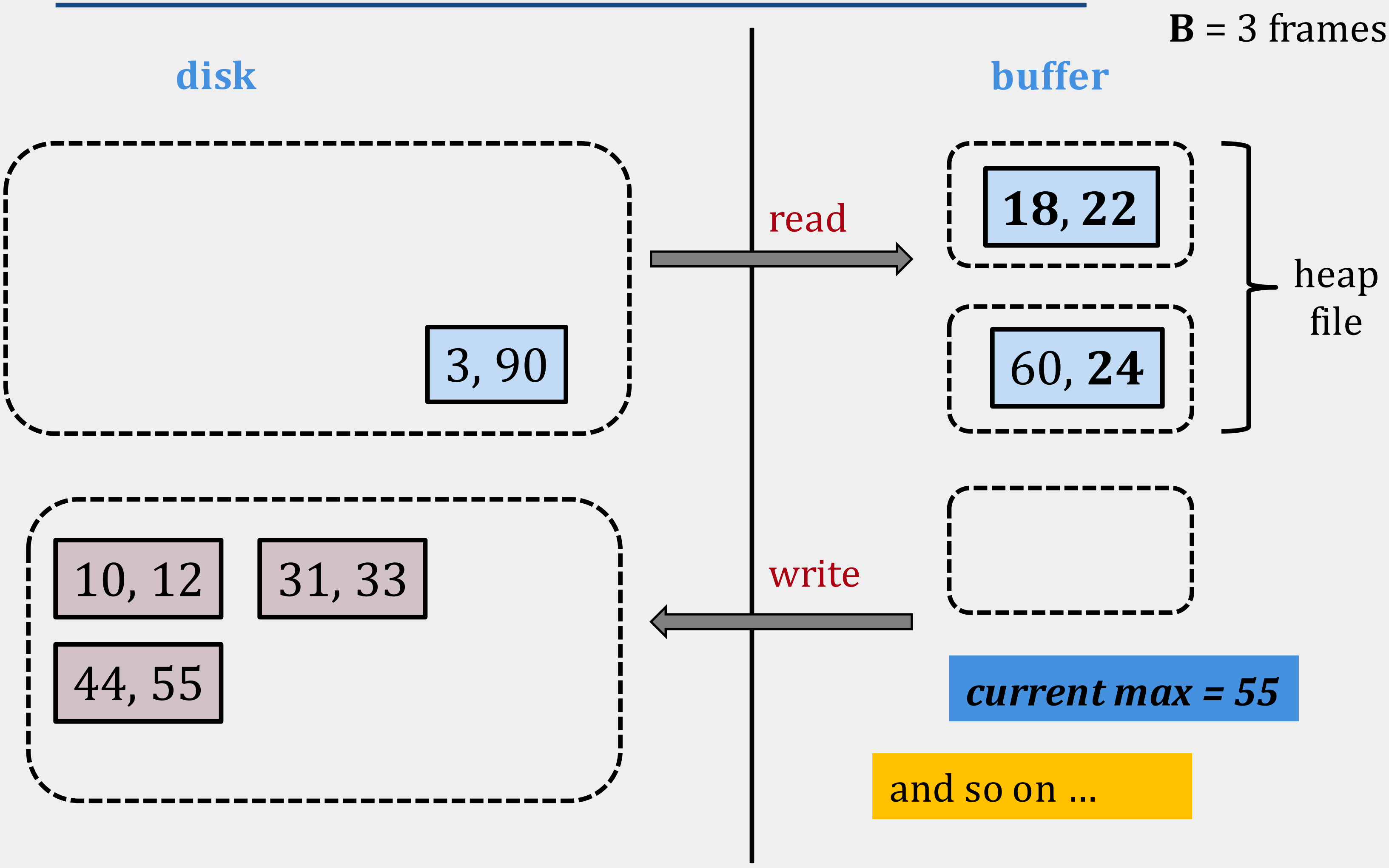


write to disk
load one more page

REPLACEMENT SORT: EXAMPLE

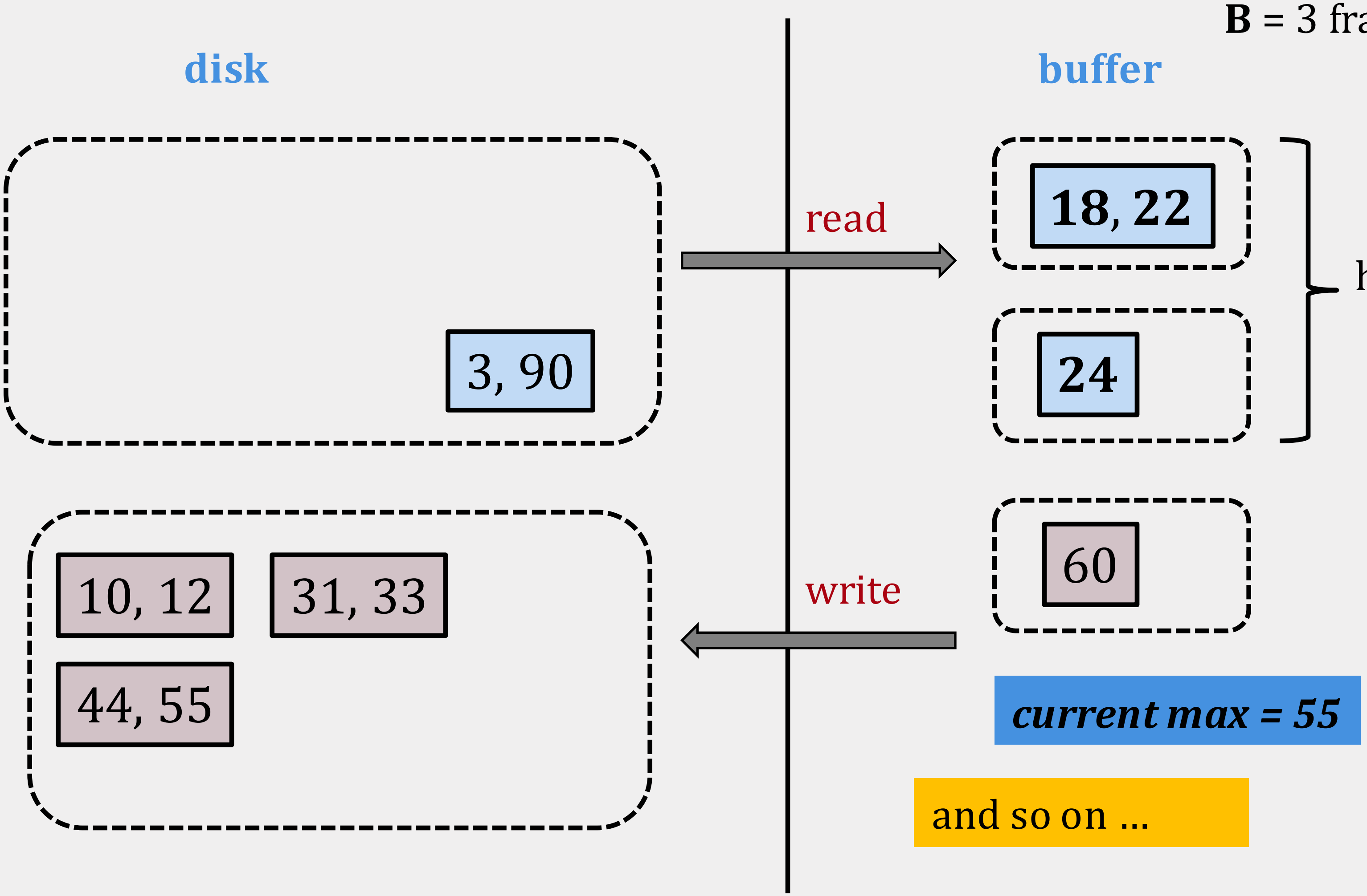


REPLACEMENT SORT: EXAMPLE



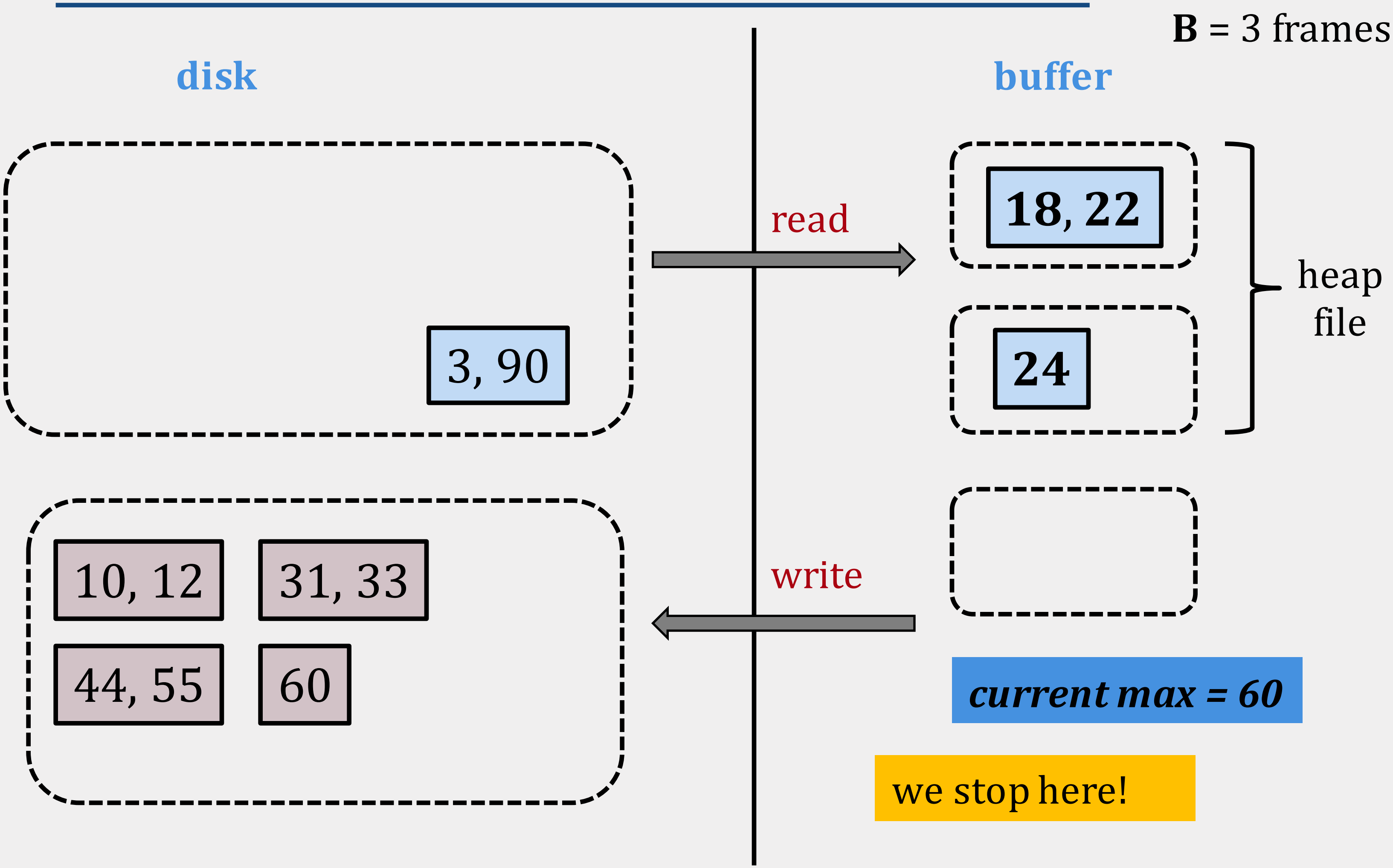
REPLACEMENT SORT: EXAMPLE

B = 3 frames



and so on …

REPLACEMENT SORT: EXAMPLE



I/O COST WITH REPLACEMENT SORT

Eacn initial run has length $\sim 2B$

$$\text{I/O cost} = 2N(\left\lceil log_{B-1} \frac{N}{2B} \right\rceil + 1)$$