Discussion #6

Diwanshu, Elena

July 15, 2020

Logistics

- Checkpoint 3 due EOD.
- Checkpoint 4 due 7/19, 11:59 PM.
- Final Project Presentation: Sign up here

Outline

- Dynamic Hashing: Extendible Hashing
- External Sorting

Why Dynamic Hashing?

- The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks.
- Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand.

Dynamic Hashing: Extendible Hashing

→ **Extendible Hashing** is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Main features

- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.
  - *A bucket may contain more than one pointers to it.*

Main features

- **Global depth** of directory
  - Max # of bits needed to tell which bucket an entry belongs to

- **Local depth** of a bucket
  - # of bits used to determine if an entry belongs to this bucket

Q. Compare:

Local depth ? Global depth

Main features

- **Global depth** of directory
  - Max # of bits needed to tell which bucket an entry belongs to

- **Local depth** of a bucket
  - # of bits used to determine if an entry belongs to this bucket

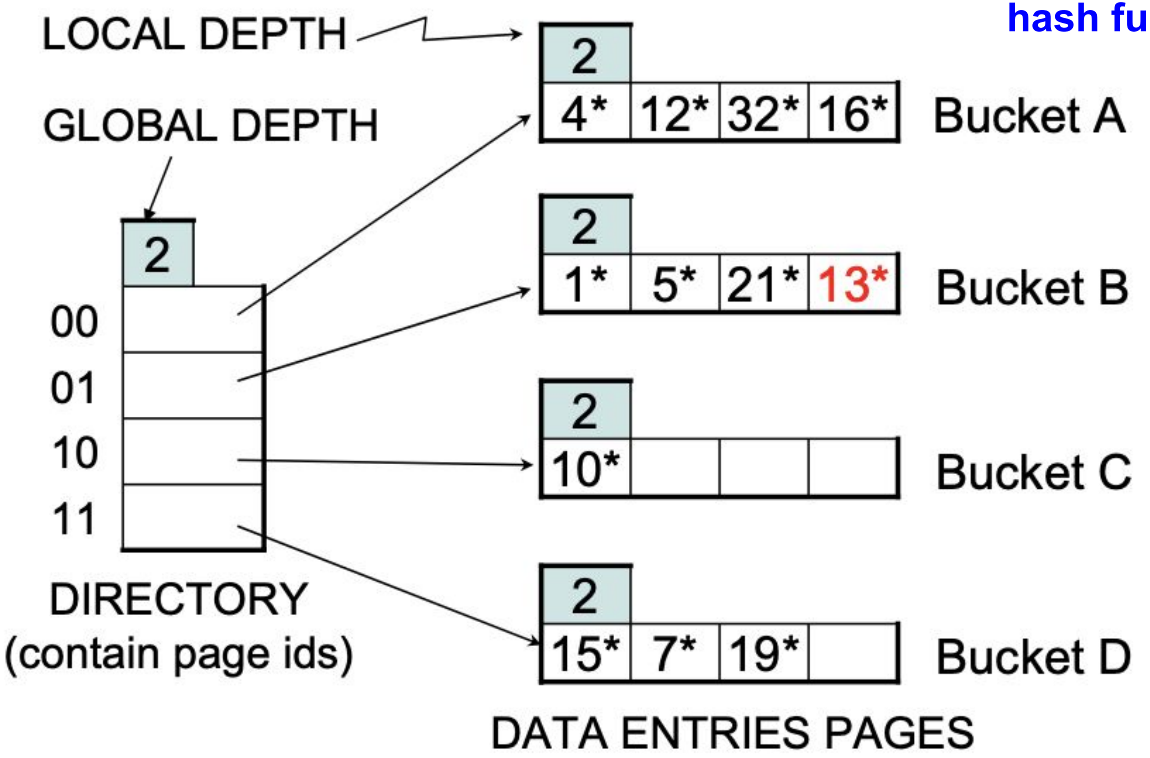
Q. Compare:

Local depth ≤ Global depth

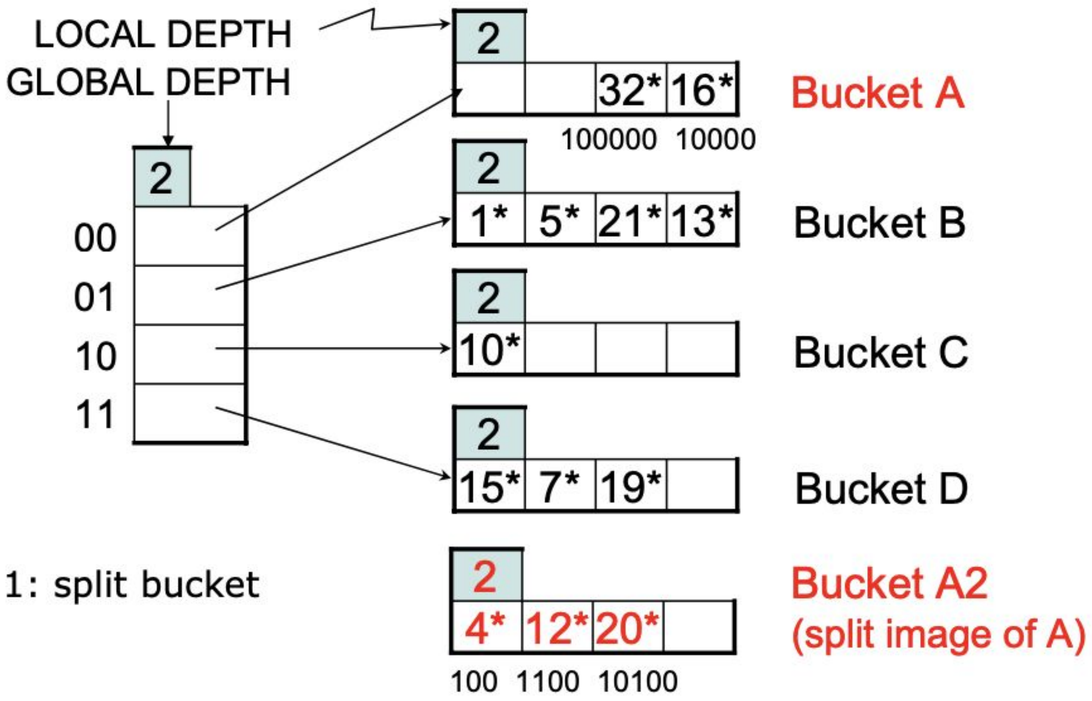
Example

Insert entry x with h(x) = 13

hash function: h(x) = x mod 4



Example



Step 1: split bucket

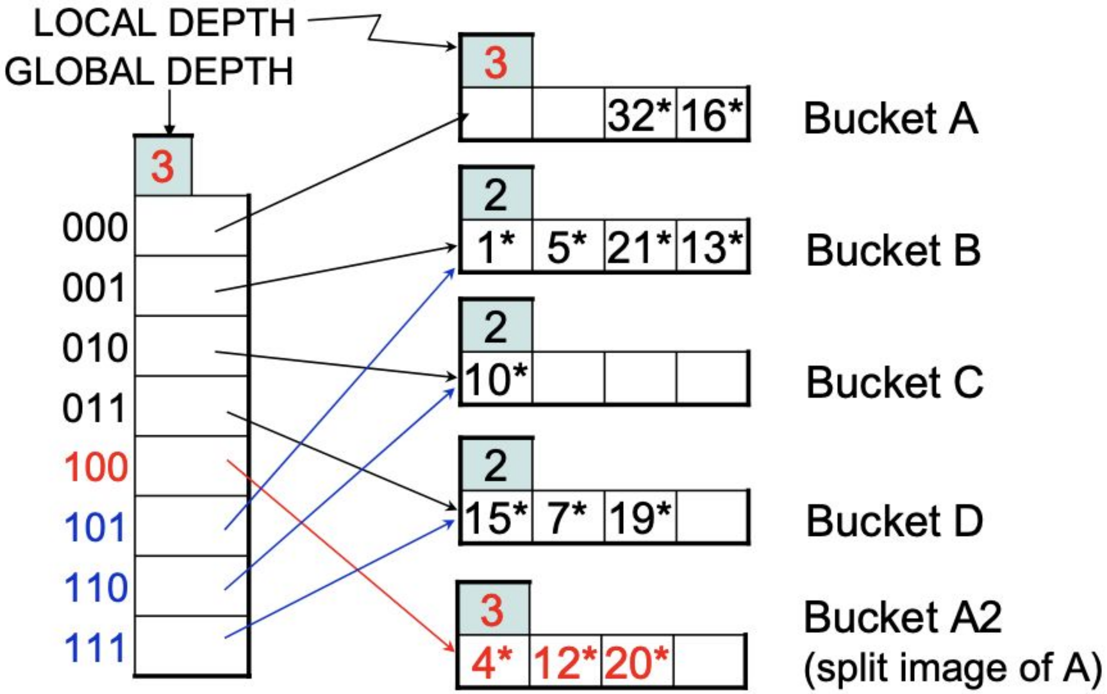
Insert entry x with h(x) = 20

hash function: h(x) = x mod 8

<u>OVERFLOW?</u>

- **If local depth = global depth**
  - We need to double the size of table, add new bucket, redistribute data, update pointers
- **If local depth < global depth**
  - Add new bucket, redistribute data, update pointers

Example



Step 2: double directory and increase depth

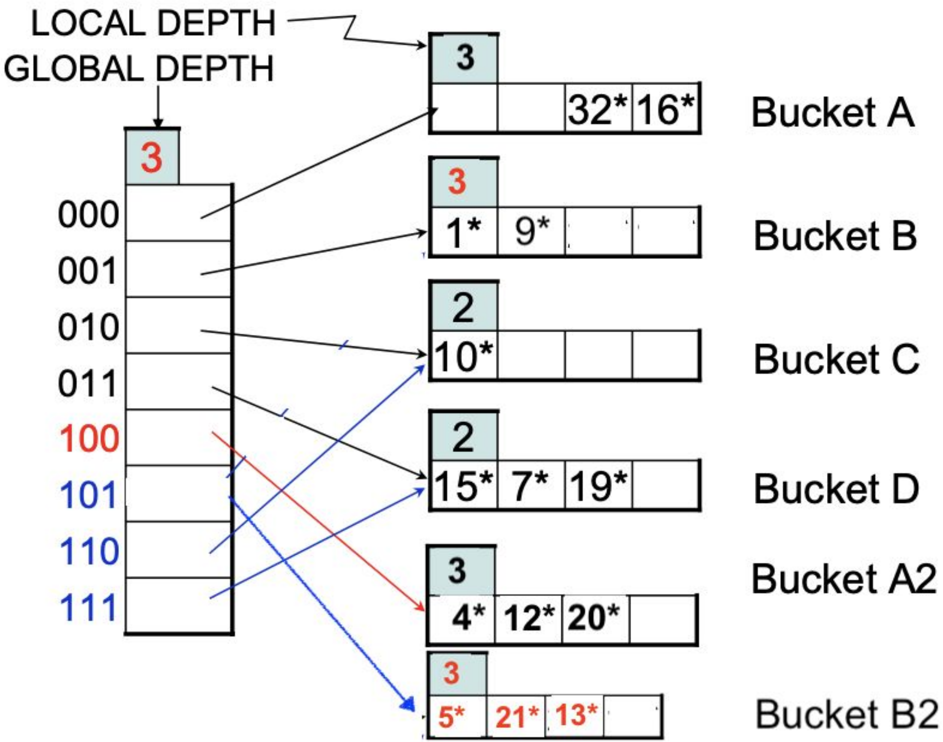
Insert entry x with h(x) = 20

hash function: h(x) = x mod 8

OVERFLOW?

- **If local depth = global depth**
  - We need to double the size of table, add new bucket, redistribute data, update pointers
- **If local depth < global depth**
  - Add new bucket, redistribute data, update pointers

Example



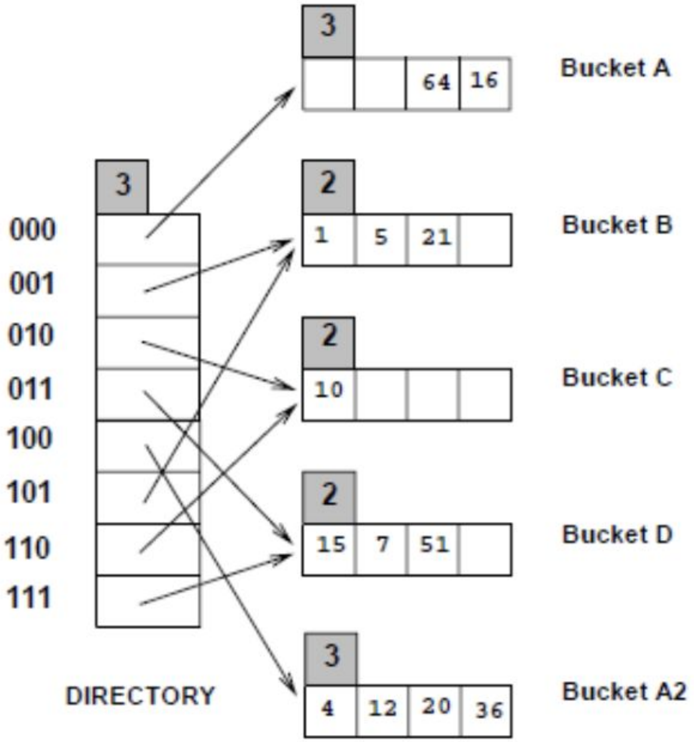
Insert entry x with h(x) = 9

hash function: h(x) = x mod 8

OVERFLOW?

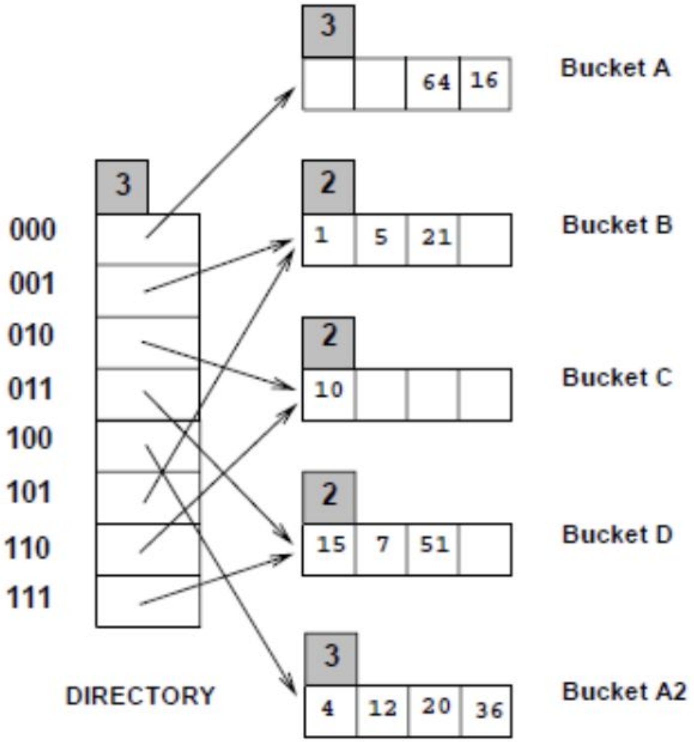
- **If local depth = global depth**
  - We need to double the size of table, add new bucket, redistribute data, update pointers
- **If local depth < global depth**
  - Add new bucket, redistribute data, update pointers

Practice Question



Q. What is the current hash function?

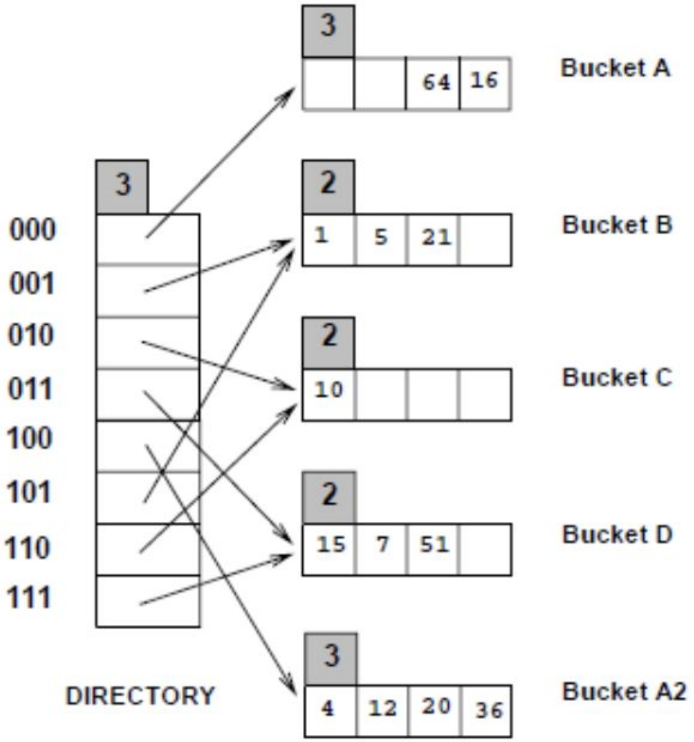
Practice Question



Q. What is the current hash function?

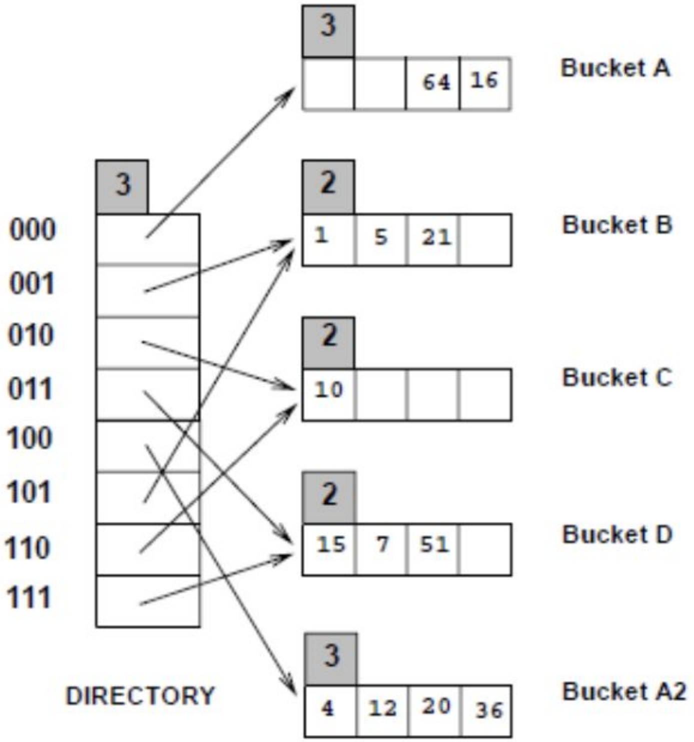
A. hash(x) = the most right three digits in binary representation of x.
Or
hash(x) = x mod 8

Practice Question



Q. What can you say about the last entry whose insertion into the index caused a split?

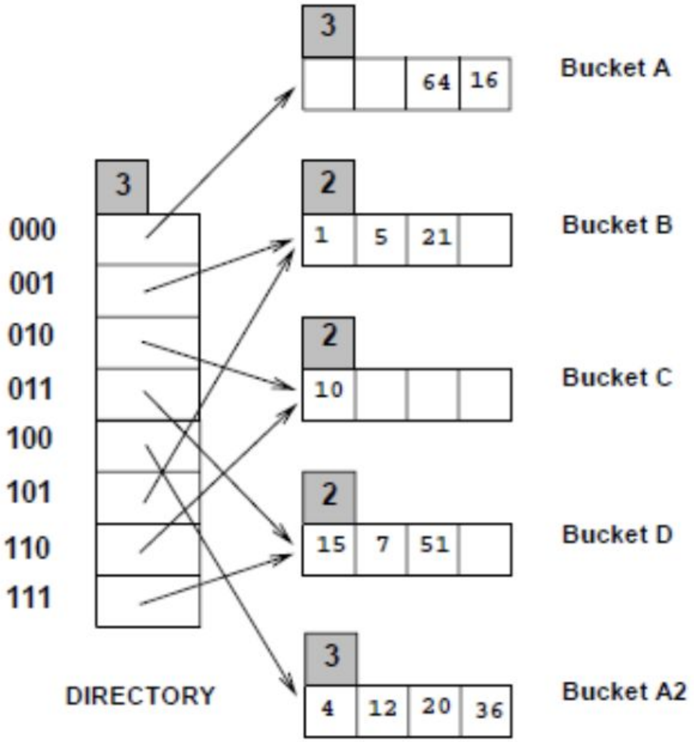
Practice Question



Q. What can you say about the last entry whose insertion into the index caused a split?

A. From the observation, we could conclude that the entry the last two digits of binary representation of which is '00' caused a split since the global depth is currently 3 and there are only two buckets the local depth of which are 3 corresponding to the entries the last two digits of the binary representation of which are 2. Thus the last entry whose insertion caused the split must be in {4, 12, 16, 20, 36, 64}.

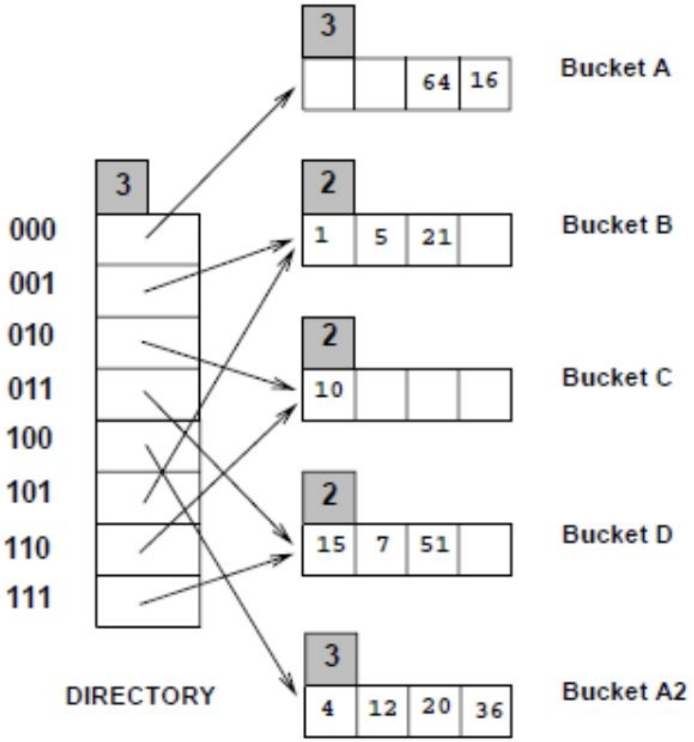
Practice Question



Q. Which of the following insertion will cause |global_depth - local_depth| to be maximum among all possible cases?

1. 13
2. 50
3. 55
4. 44

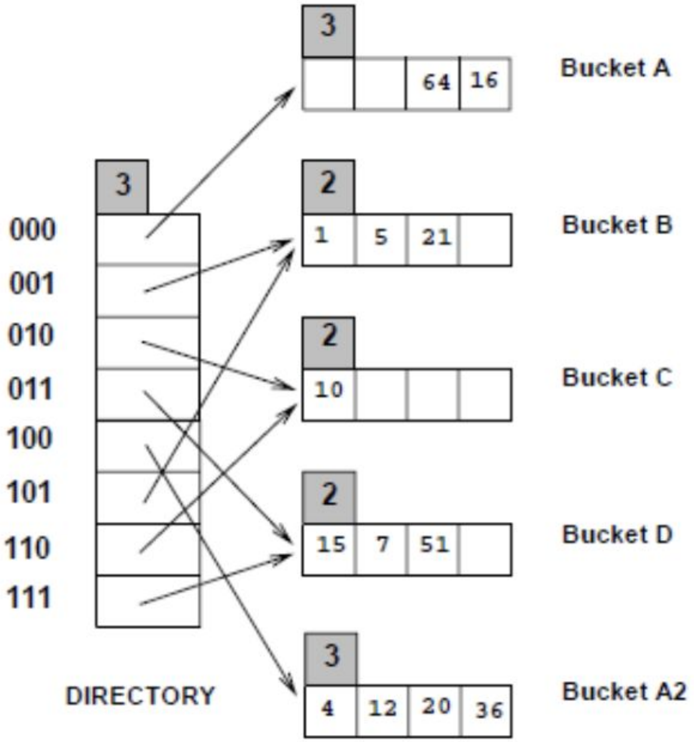
Practice Question



Q. Which of the following insertion will cause |global_depth - local_depth| to be maximum among all possible cases?

1. 13
2. 50
3. 55
4. 44

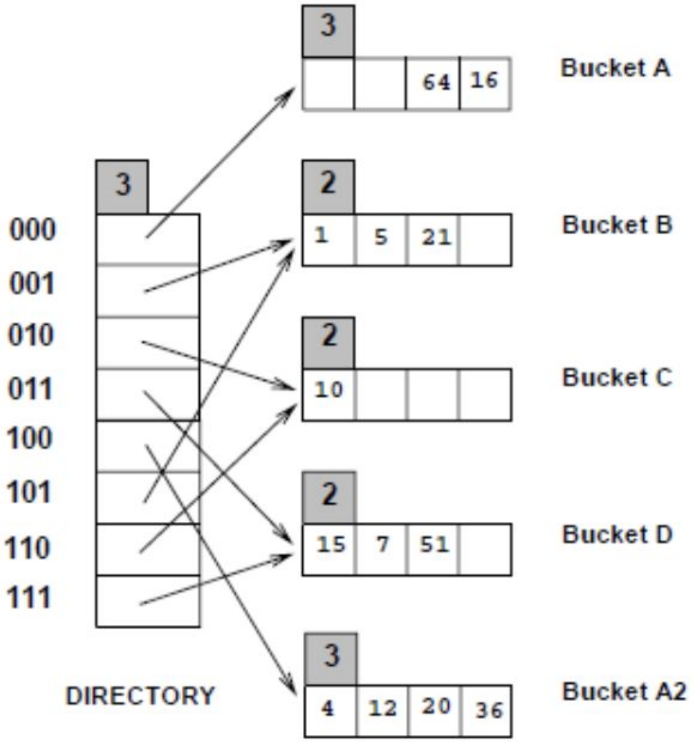
Practice Question



Q. Let L[i] = local_depth[i], where i denotes Bucket i. Let M = max(L). What is minimum possible size of the directory?

1. 2^{M-1}
2. $\log_2(M)$
3. $\log_2(M-1)$
4. 2^M

Practice Question



Q. Let L[i] = local_depth[i], where i denotes Bucket i. Let M = max(L). What is minimum possible size of the directory?

1. 2^{M-1}
2. $\log_2(M)$
3. $\log_2(M-1)$
4. 2^M

Practice Question

Q. True/False: A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.

True

Q. True/False: If Local Depth of the overflowing bucket is equal to the global depth, only then the directories are doubled and the global depth is incremented by 1.

True

Limitations

- The directory size may increase significantly if several records are hashed on the same bucket.
- Size of every bucket is fixed.
- Memory is wasted in pointers when the global depth and local depth difference becomes drastic.

External Sorting

- why sorting?
  - bulk loading in B+ tree, duplicate elimination, users want the data sorted, useful in join algorithms e.t.c
- what about the standard algorithms?
  - merge sort, quick sort, heap sort → in-memory
  - the data typically does not fit in memory
  - e.g how to sort 1TB data with 8GB of RAM?

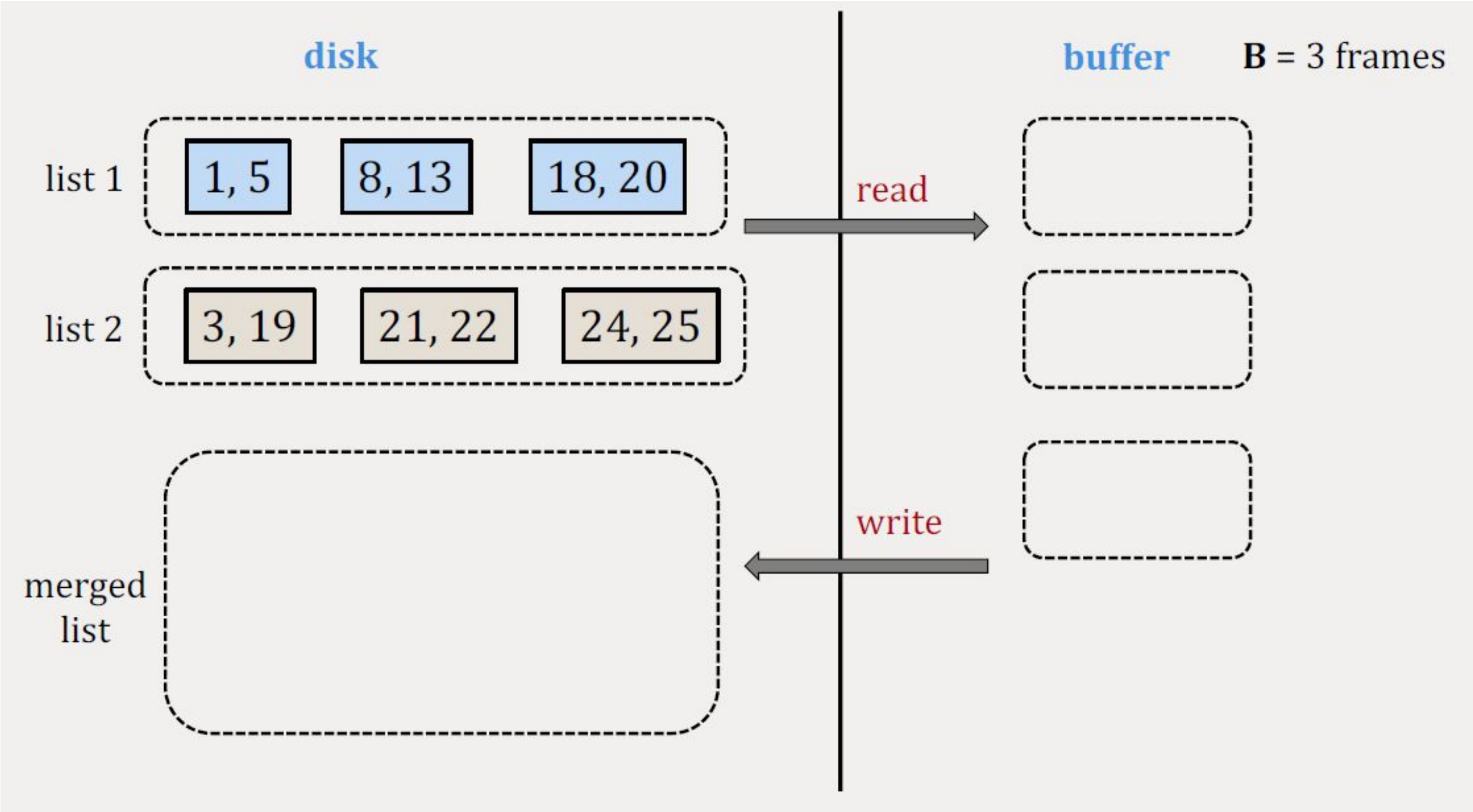
External Merge

- **Input:** 2 sorted lists (with M and N pages)
- **Output:** 1 merged sorted list (with M+N pages)

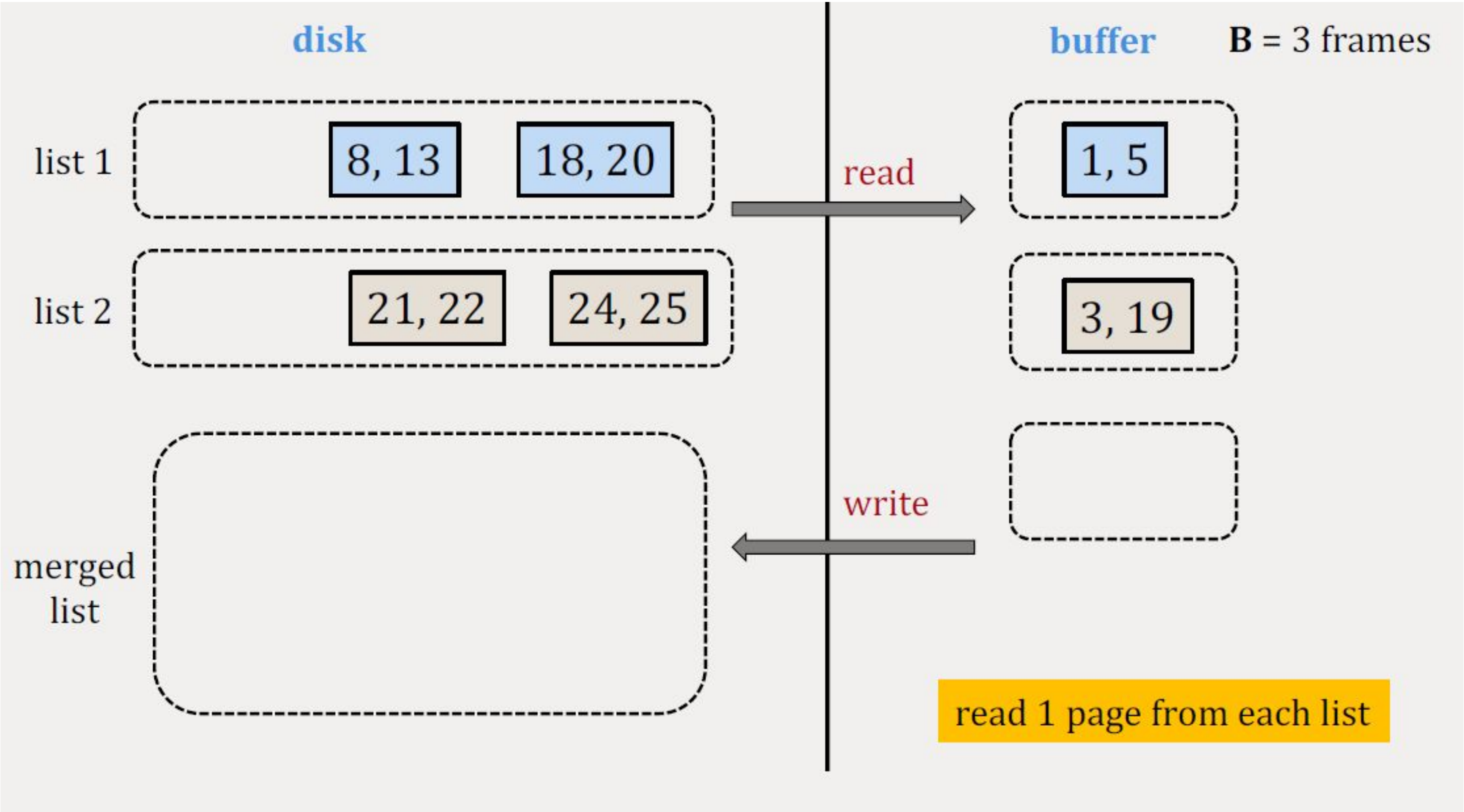
Can we do the merging efficiently in terms of I/O using a buffer pool of size at least 3?

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

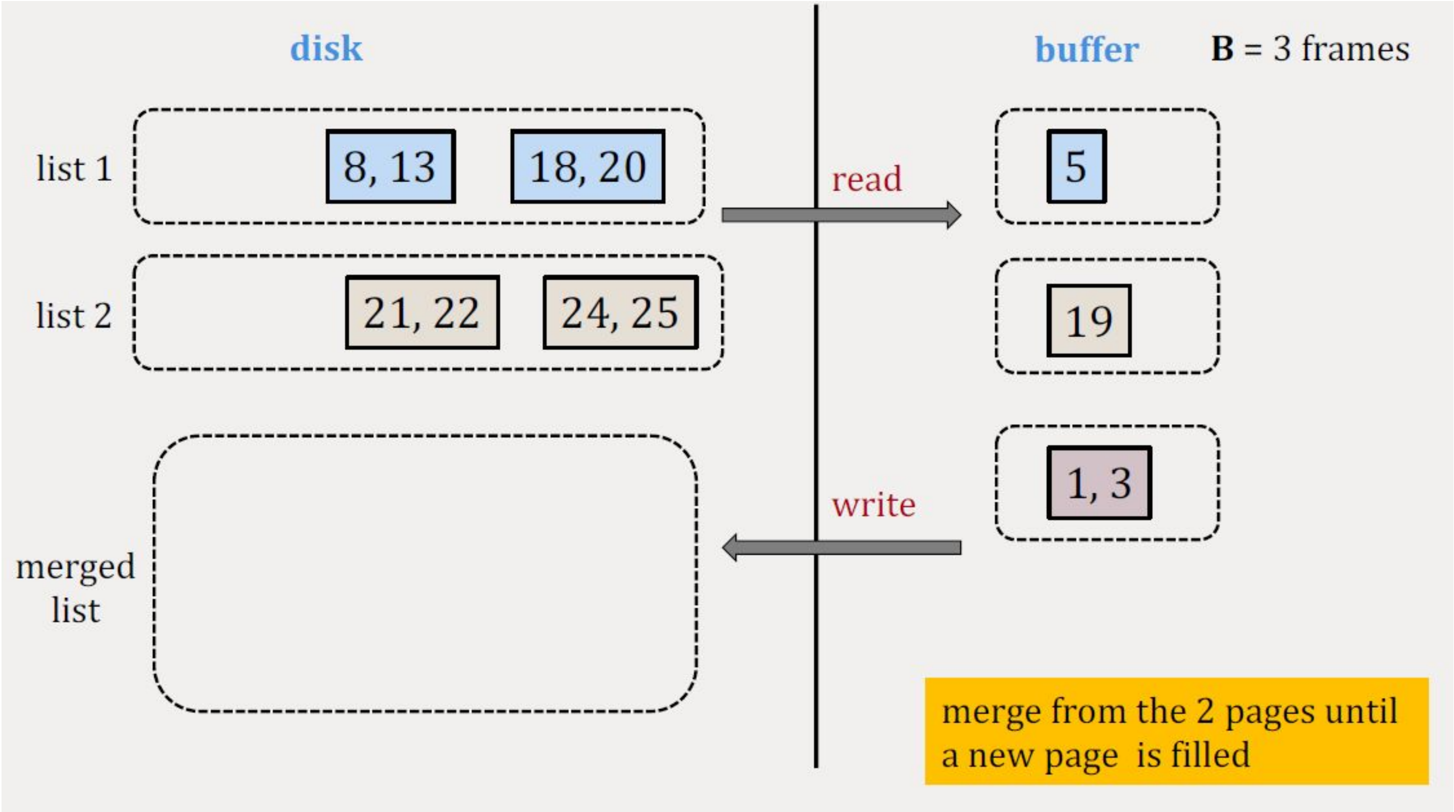
External Merge Example(1)



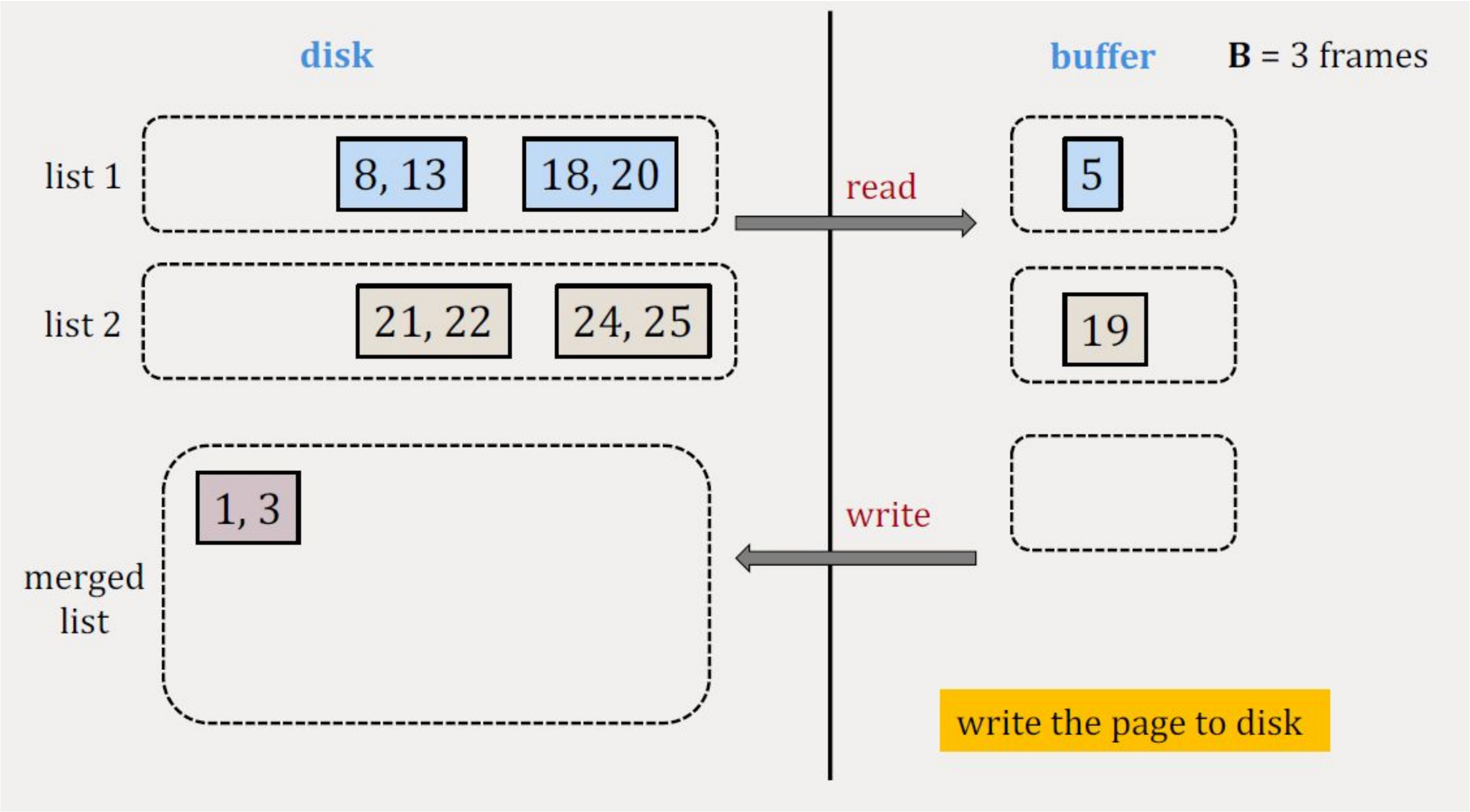
External Merge Example(2)



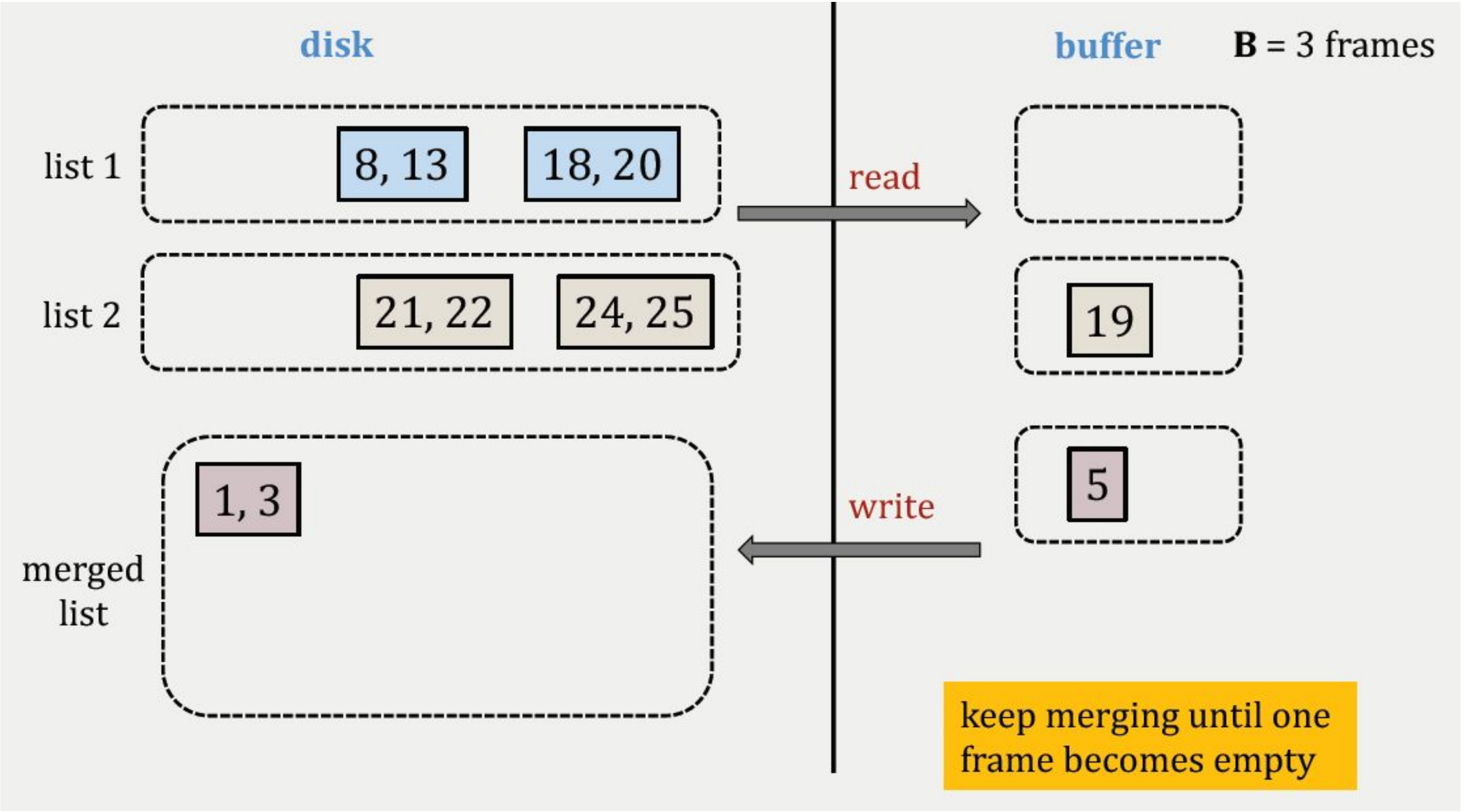
External Merge Example(3)



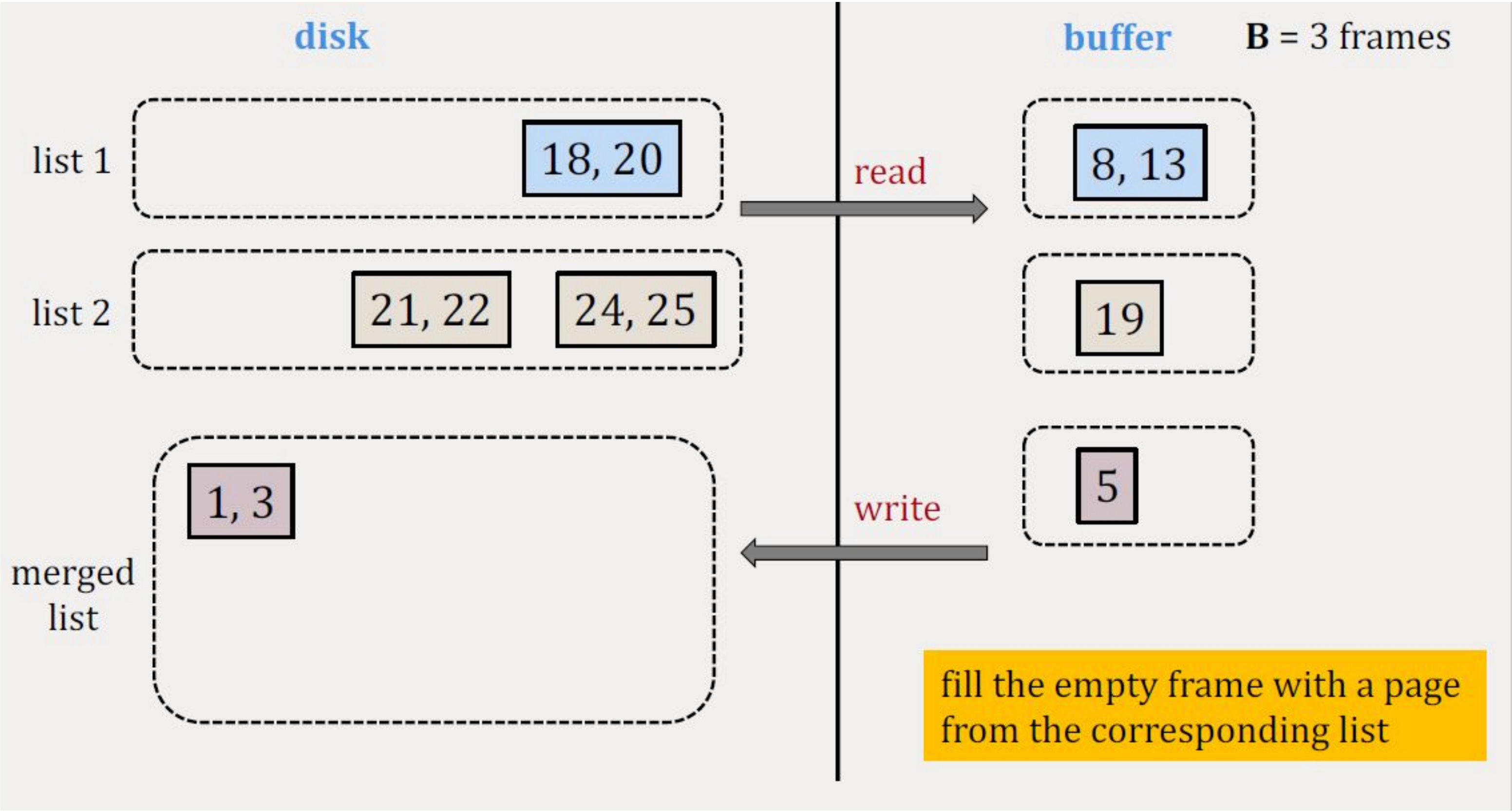
External Merge Example(4)



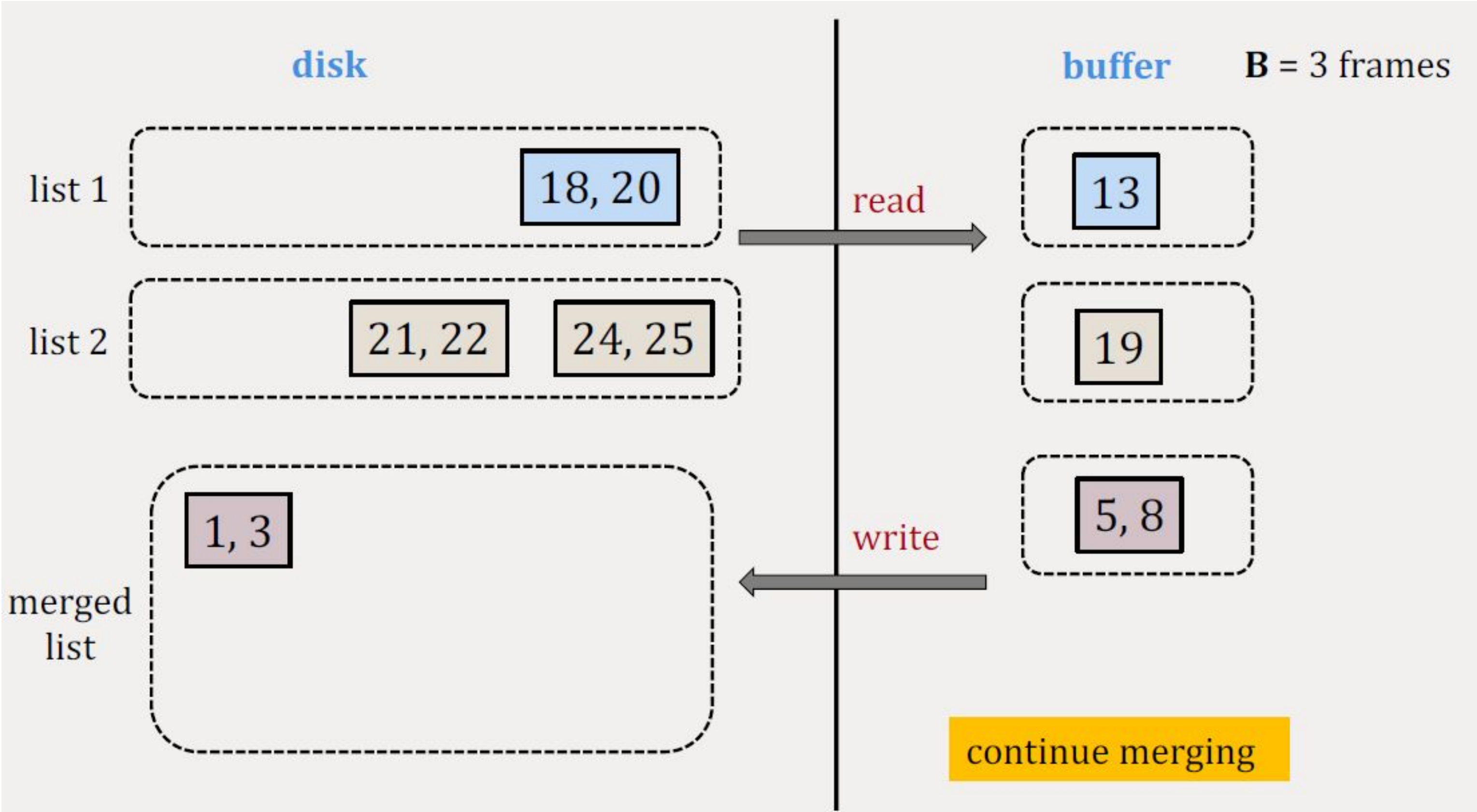
External Merge Example(5)



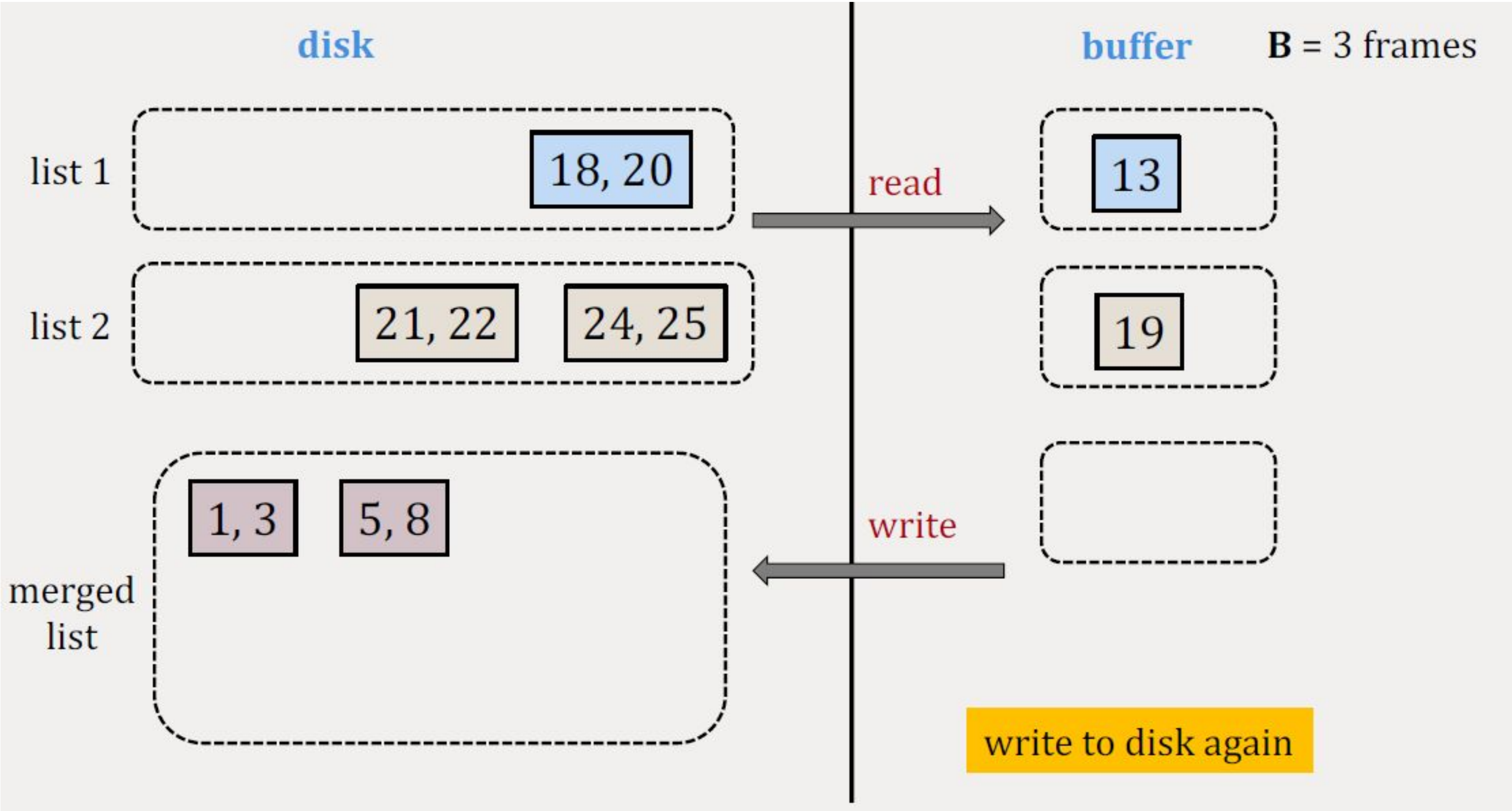
External Merge Example(6)



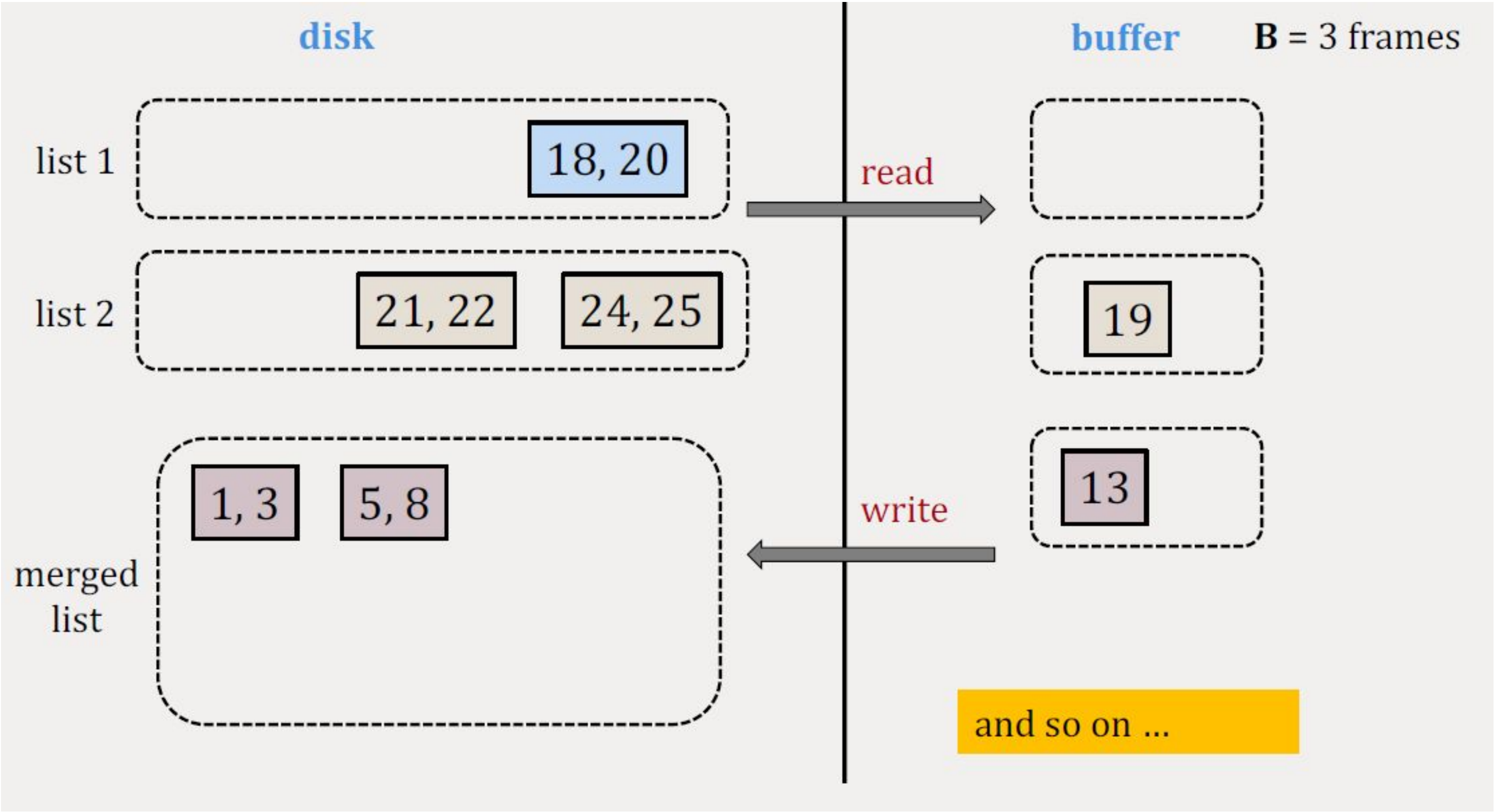
External Merge Example(7)



External Merge Example(8)



External Merge Example(9)



The Sorting Problem

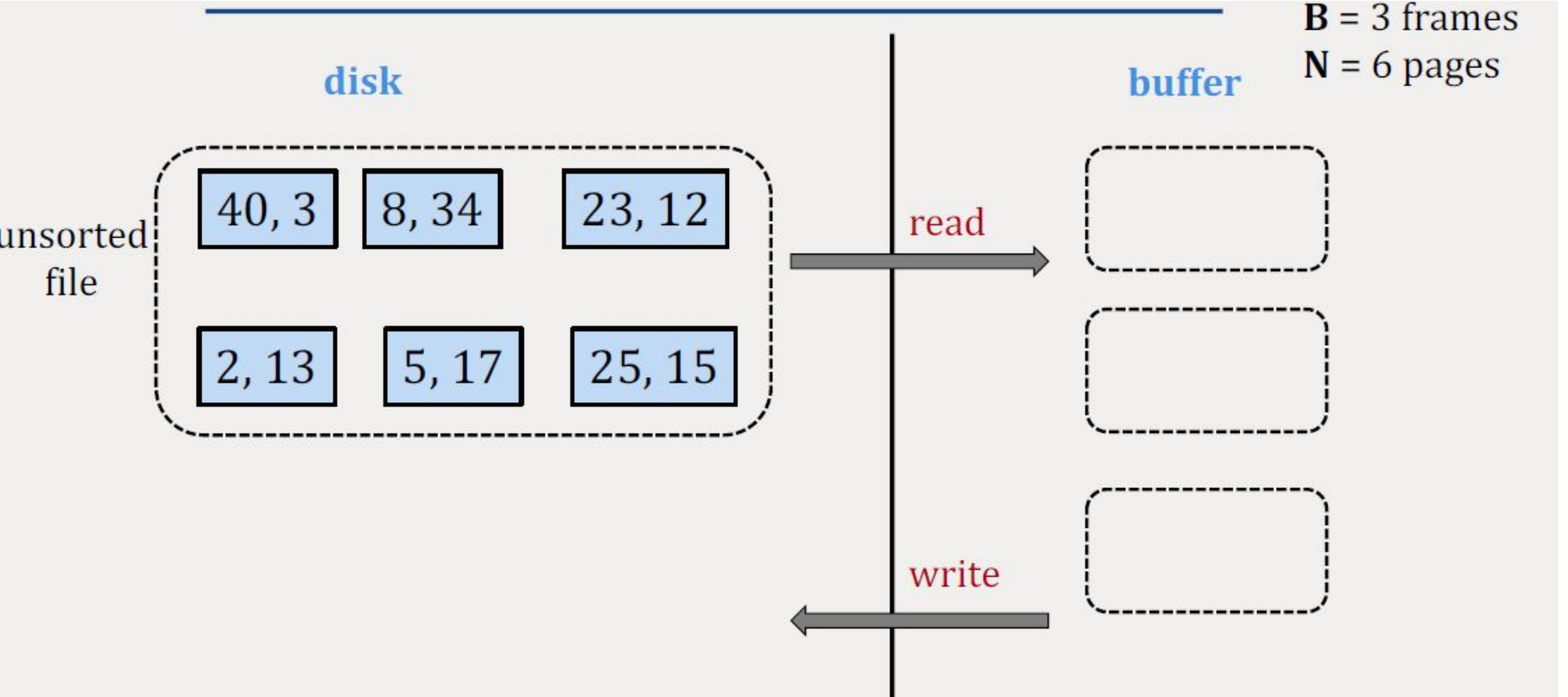
- **B** available pages in the buffer pool
- a relation R of **N** pages (where **N>B**)

SORTING: output the same relation sorted on a given attribute

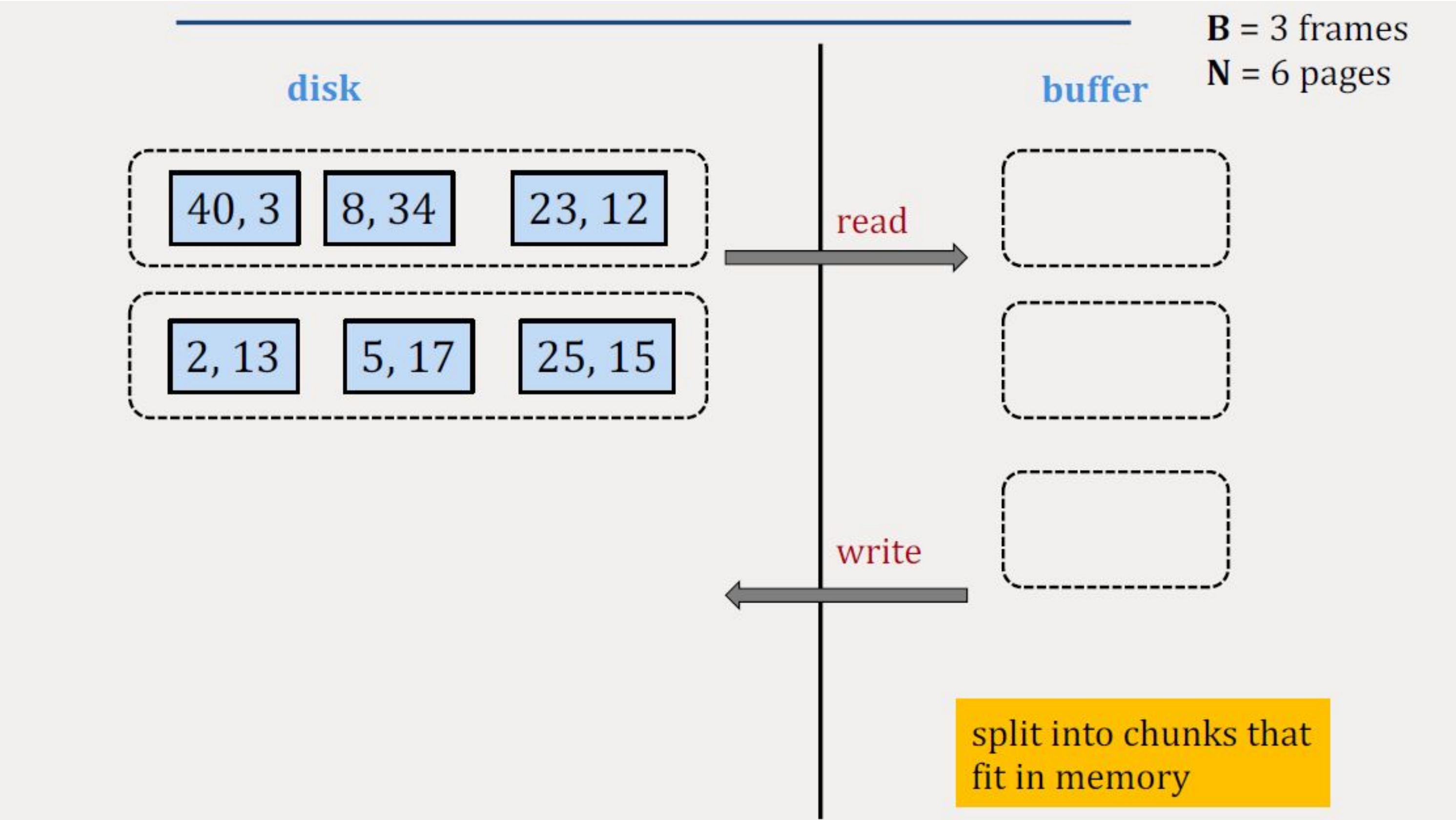
Key Idea

- 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

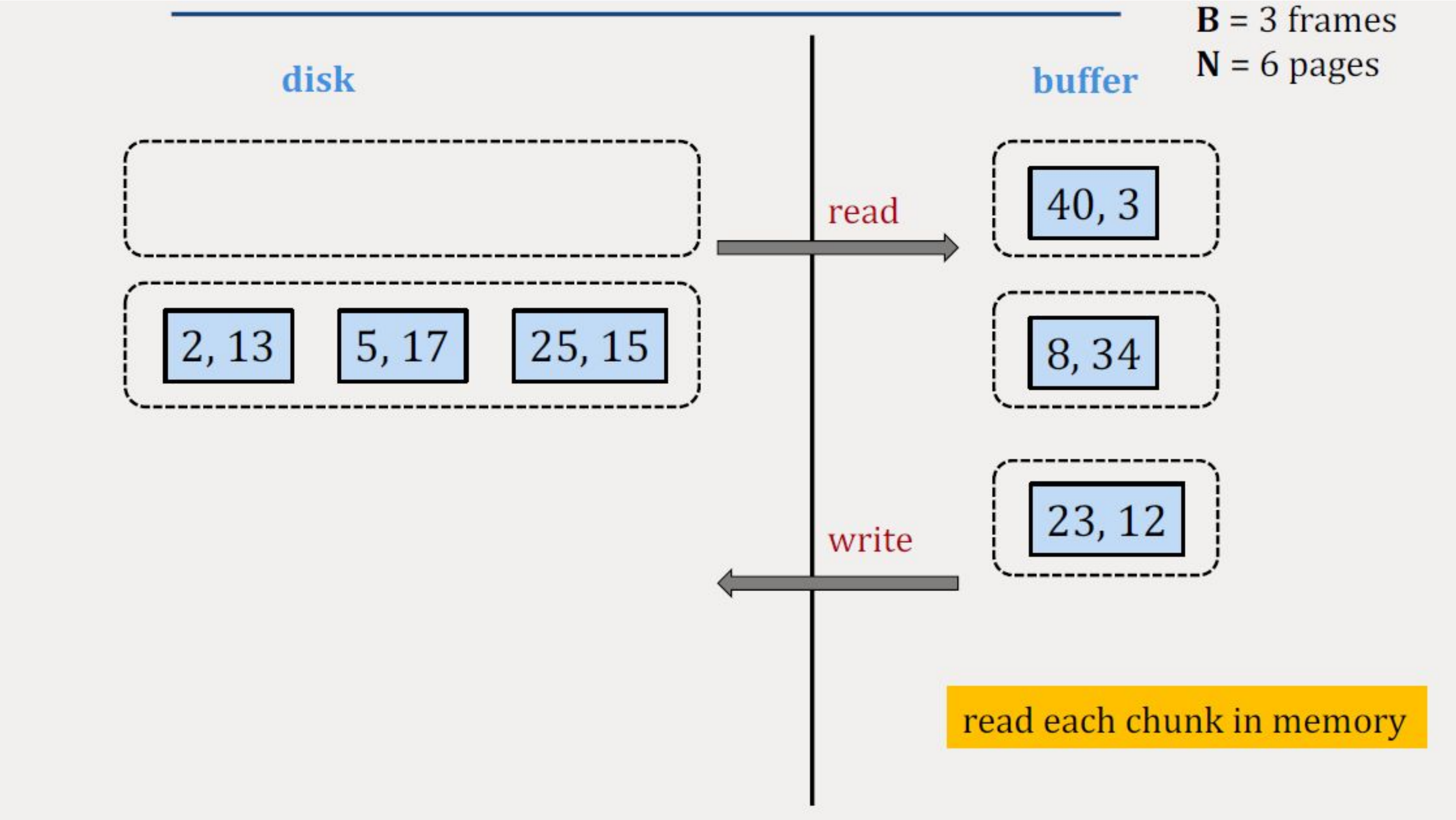
2-Way Sort Example(1)



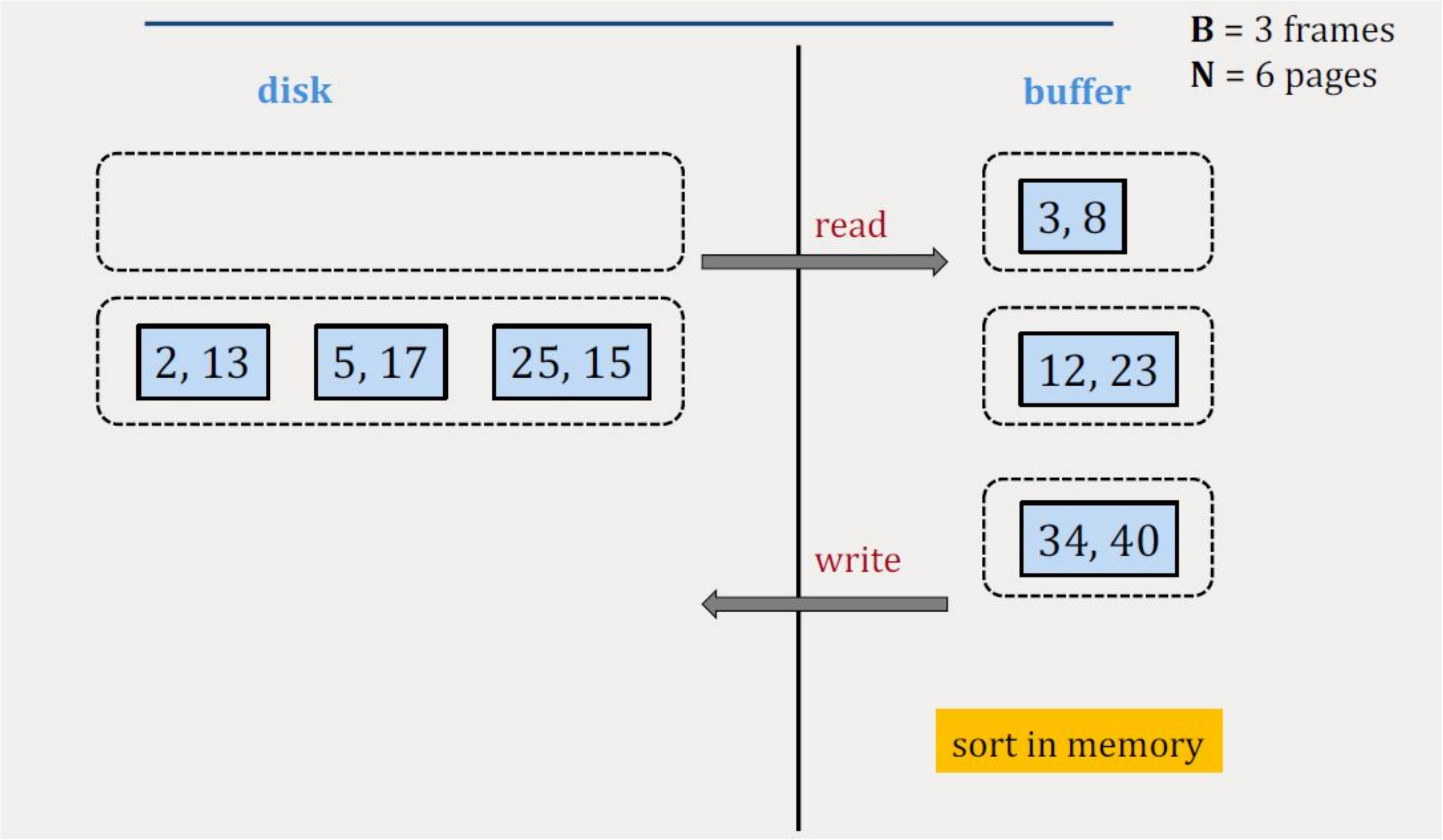
2-Way Sort Example(2)



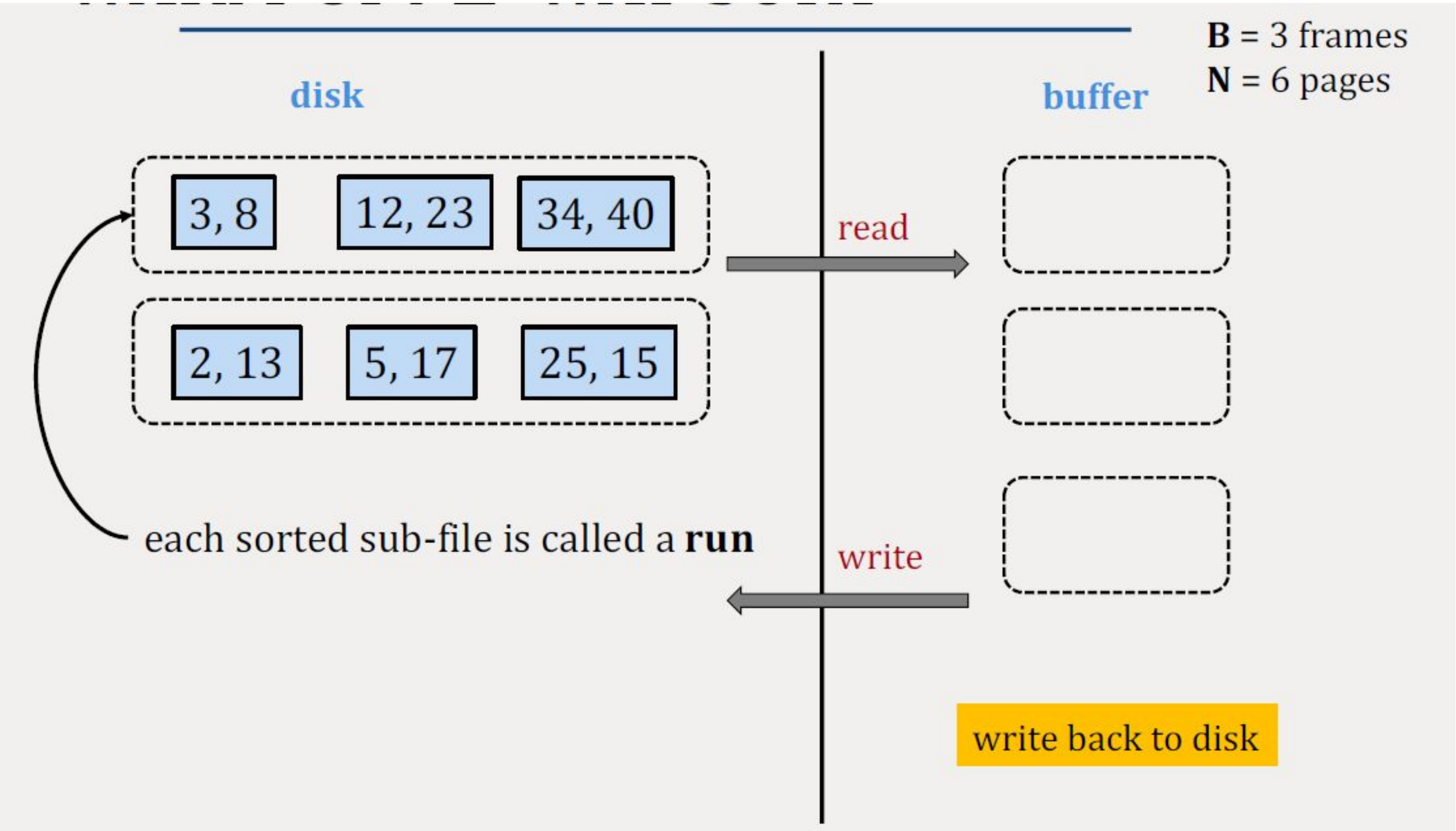
2-Way Sort Example(3)



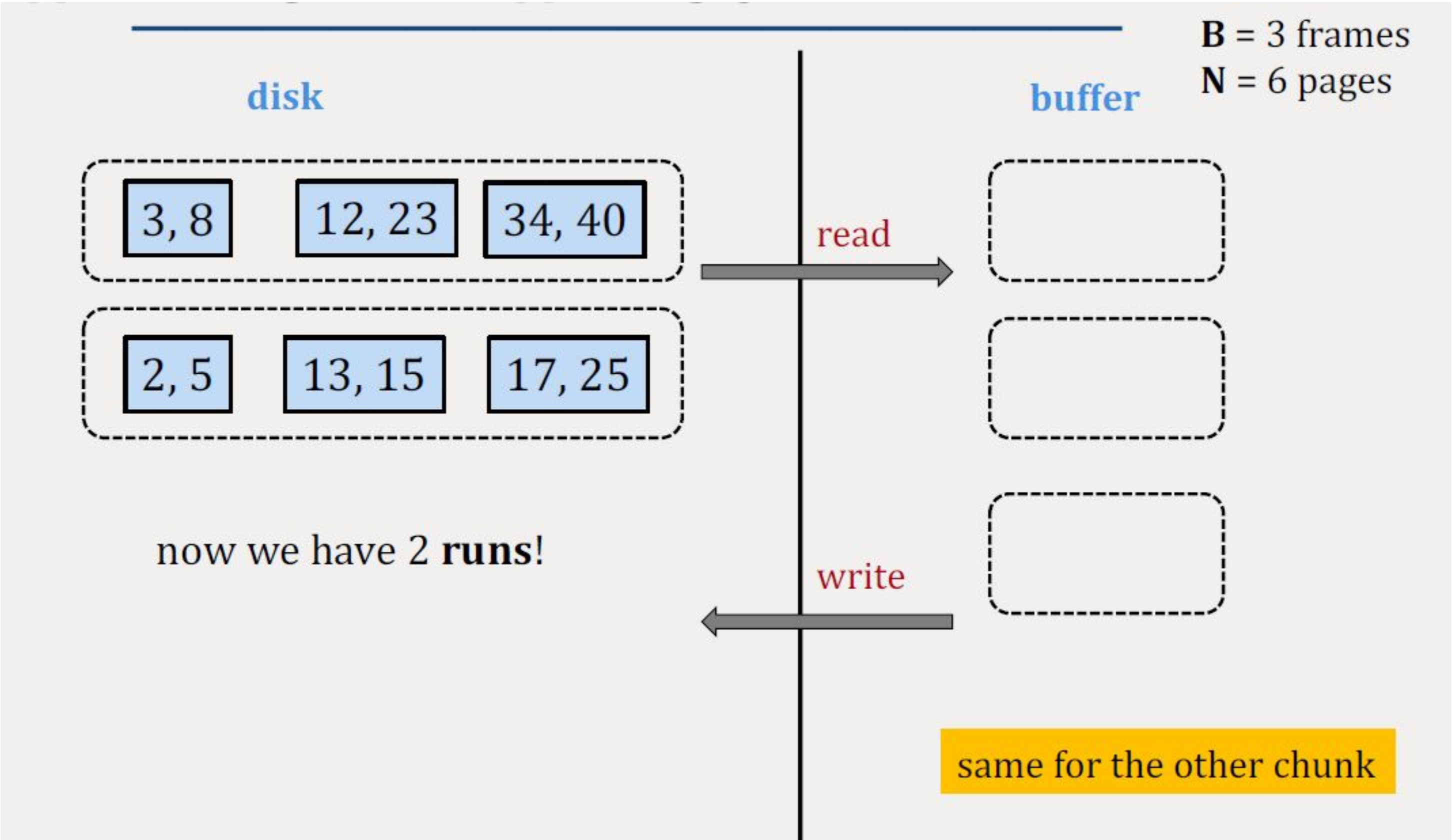
2-Way Sort Example(4)



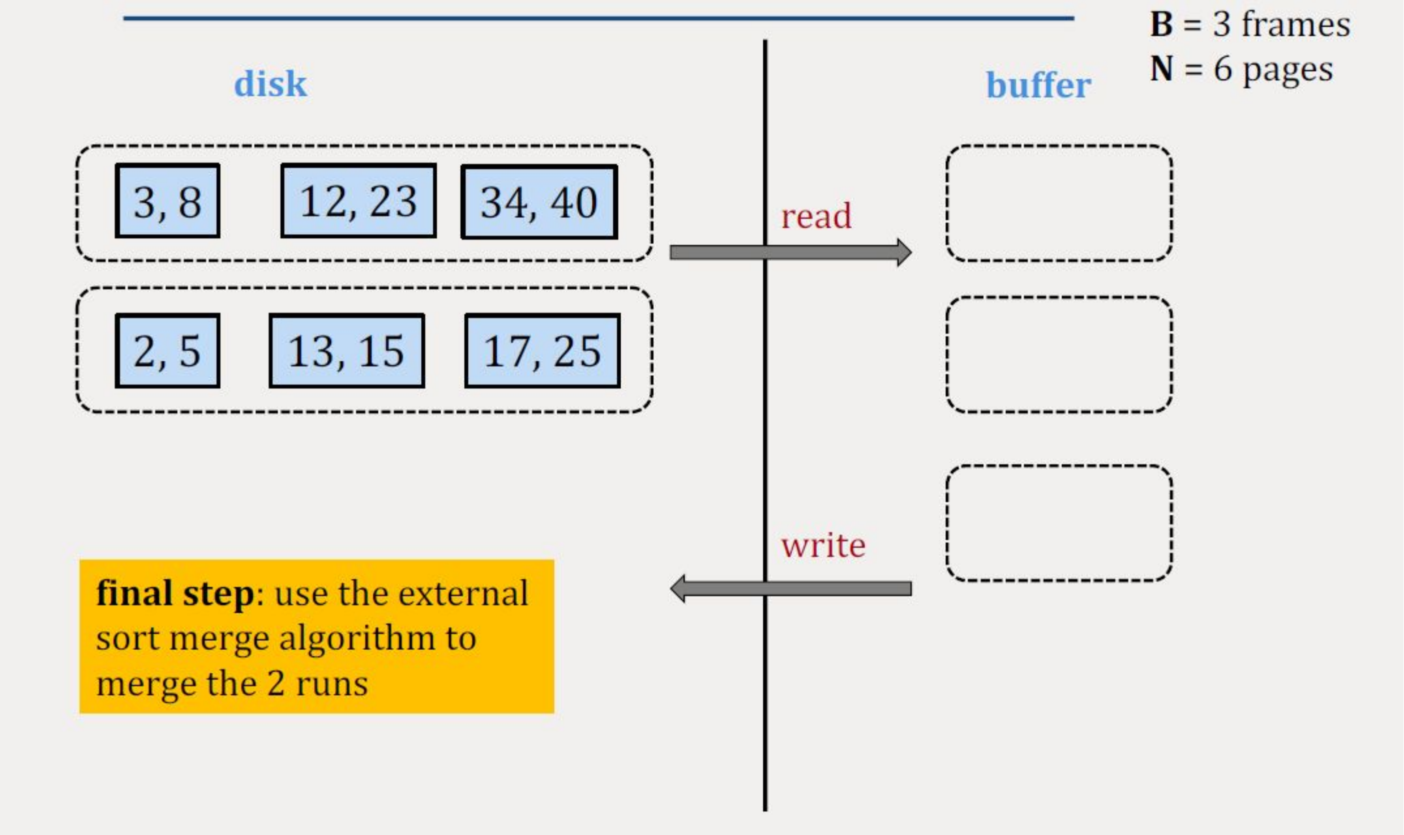
2-Way Sort Example(5)



2-Way Sort Example(6)



2-Way Sort Example(7)



Calculating the I/O cost

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

- **Pass 0:** sorting
  - 1 read + 1 write for every page
  - total cost = 6 * (1+1) = 12 I/Os
- **Pass 1:** merging
  - total cost = 2* (3+3) = 12 I/Os

Total: 24 I/Os

External Sorting Problem

Given two files A and B and each file includes 4 pages (each page contains 4 entries only) as shown as the following:

A: [[1,4,2, 4], [5,2,18,9], [28,7,16,5], [13, 14, 11, 12]]

B: [[9,5,2,1], [25,5, 31, 0], [45, 33, 12, 16], [19, 13, 16, 21]]

Assume you have 3 buffer pages in main memory, show the “external sort” procedure for the given example.

Solution (1)

- **First Run (Sort Phase)** → split into chunks of 3 (buffer pages 3), bring each chunk into memory and apply the in-memory sorting algorithm, write it back to disk. End up with 3 sorted chunks in disk.

Input:

A: [[1,4,2, 4], [5,2,18,9], [28,7,16,5], [13, 14, 11, 12]]

B: [[9,5,2,1], [25,5, 31, 0], [45, 33, 12, 16], [19, 13, 16, 21]]

Output:

[1, 2, 2, 4] [4, 5, 5, 7] [9, 16, 18, 28]

[0, 1, 2, 5] [5, 9, 11, 12] [13, 14, 25, 31]

[12, 13, 16, 16] [19, 21, 33, 45]

I/O cost: read & write of 8 pages → 16 I/O (in terms of pages)

Solution (2)

- **Second Run (Merge Phase)** → Take two pages each time from the sorted chunks into memory, merge them and write them to disk

Input:

[1, 2, 2, 4] [4, 5, 5, 7] [9, 16, 18, 28]

[0, 1, 2, 5] [5, 9, 11, 12] [13, 14, 25, 31]

[12, 13, 16, 16] [19, 21, 33, 45]

Output:

[0, 1, 1, 2] [2, 2, 4, 4] [5, 5, 5, 5] [7, 9, 9, 11] [12, 13, 14, 16] [18, 25, 28, 31]

[12, 13, 16, 16] [19, 21, 33, 45]

I/O cost: read & write of 6 pages => 12 I/O

Solution (3)

- **Third Run (Merge Phase)** → Take two pages each time from the sorted chunks into memory, merge them and write them to disk

Input:

[0, 1, 1, 2] [2, 2, 4, 4] [5, 5, 5, 5] [7, 9, 9, 11] [12, 13, 14, 16] [18, 25, 28, 31]

[12, 13, 16, 16] [19, 21, 33, 45]

Output:

[0, 1, 1, 2] [2, 2, 4, 4] [5, 5, 5, 5] [7, 9, 9, 11] [12, 12, 13, 13] [14, 16, 16, 16] [18, 19, 21, 25] [28, 31, 33, 45]

I/O cost: read & write of 6 pages => 16 I/O

Total I/O cost: 44 I/O

Thanks!