BIG DATA MANAGEMENT

CZ4123

MEMORY HIERARCHY (PART III)

CACHE CONSCIOUS DATA-PROCESSING

Siqiang Luo

Assistant Professor

OVERVIEW

- In previous lectures, we show the basic concepts and analysis of memory hierarchy.

- In this lecture, we show some design principles to make use of memory hierarchy for processing big data. We will discuss following two cases.
  - Array access patterns
    - Spatial locality designs
    - Temporal locality designs

  - Big data sorting

MOTIVATING QUESTION

Suppose we have a 10000x10000 matrix of integers, which is stored in a 2-dimentional array A[10000][10000]. We want to set all of its value to 1. Cache size = 5000 integers. How would you write your code?

Solution X

```
for (int i=0;i<10000;i++)
    for(int j=0;j<10000;j++)
        A[i][j]=1;
```

Solution Y

```
for (int j=0;j<10000;j++)
    for(int i=0;i<10000;i++)
        A[i][j]=1;
```

Which one is better, X or Y?

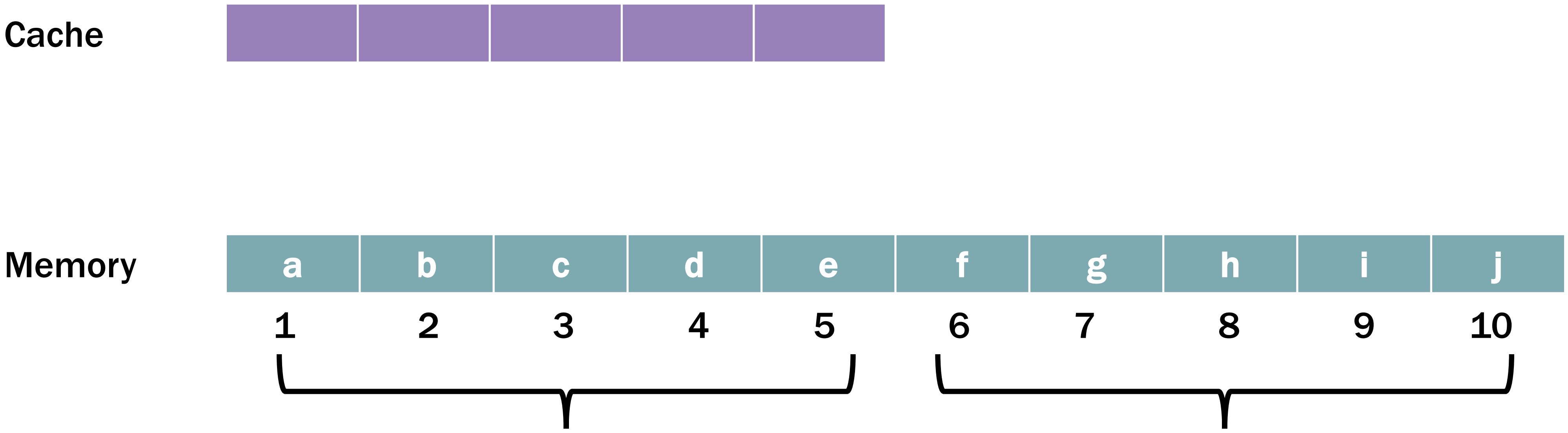
ACCESSING AN ARRAY

- We may access an array with different access patterns
  - Access pattern means the position sequence of accessing an array.
  - Different access patterns may impact the utility of cache

ACCESS PATTERN 1

We have a 10-integer array stored in main memory,
cache size = 5 integers, transfer size (cache line)= 5 integers

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache

a	b	c	d	e

Cache Miss: 1
Cache Hit: 0

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache

a	b	c	d	e

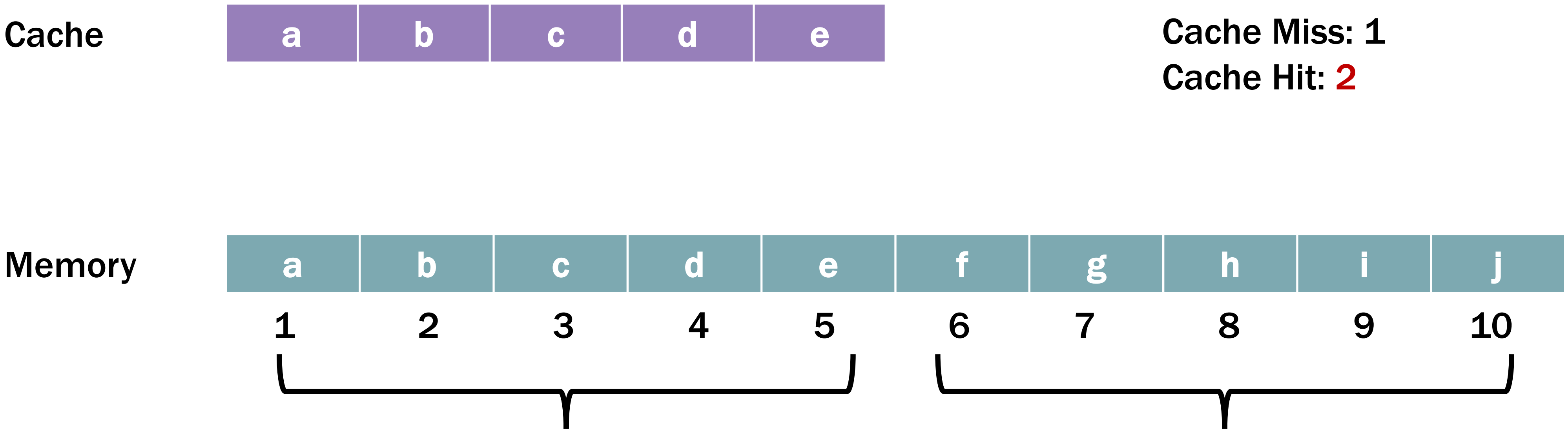
Cache Miss: 1
Cache Hit: 1

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



Cache Miss: 1
Cache Hit: 2

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache	a	b	c	d	e

Cache Miss: 1
Cache Hit: 3

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache

a	b	c	d	e

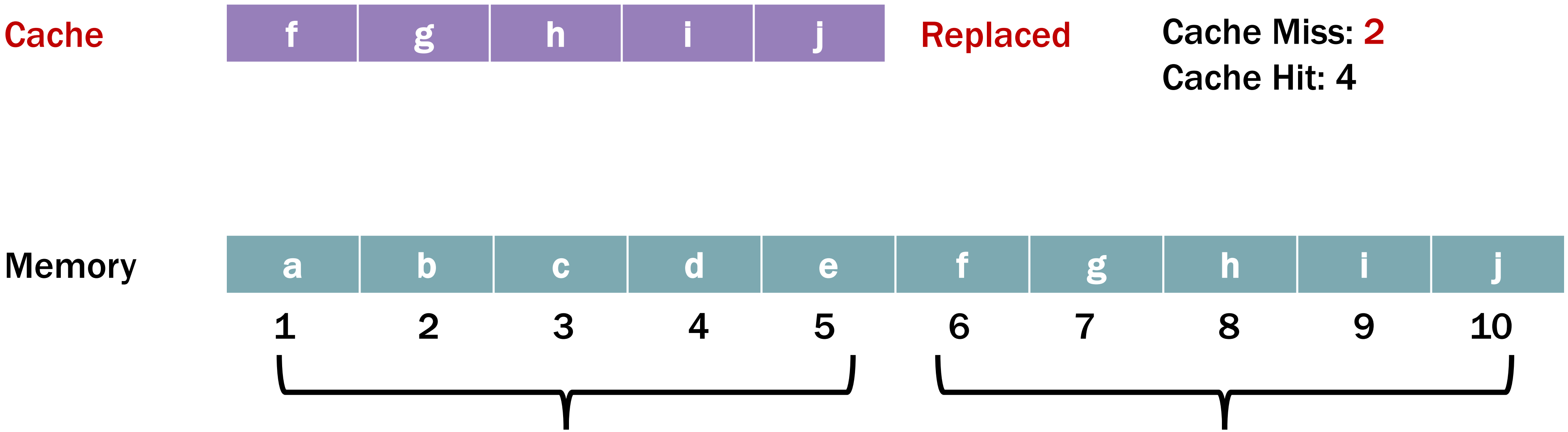
Cache Miss: 1
Cache Hit: 4

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache					
	f	g	h	i	j

Cache Miss: 2
Cache Hit: 5

Memory										
	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache	f	g	h	i	j

Cache Miss: 2
Cache Hit: 6

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache

f	g	h	i	j

Cache Miss: 2
Cache Hit: 7

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 1

Access pattern: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Cache

f	g	h	i	j

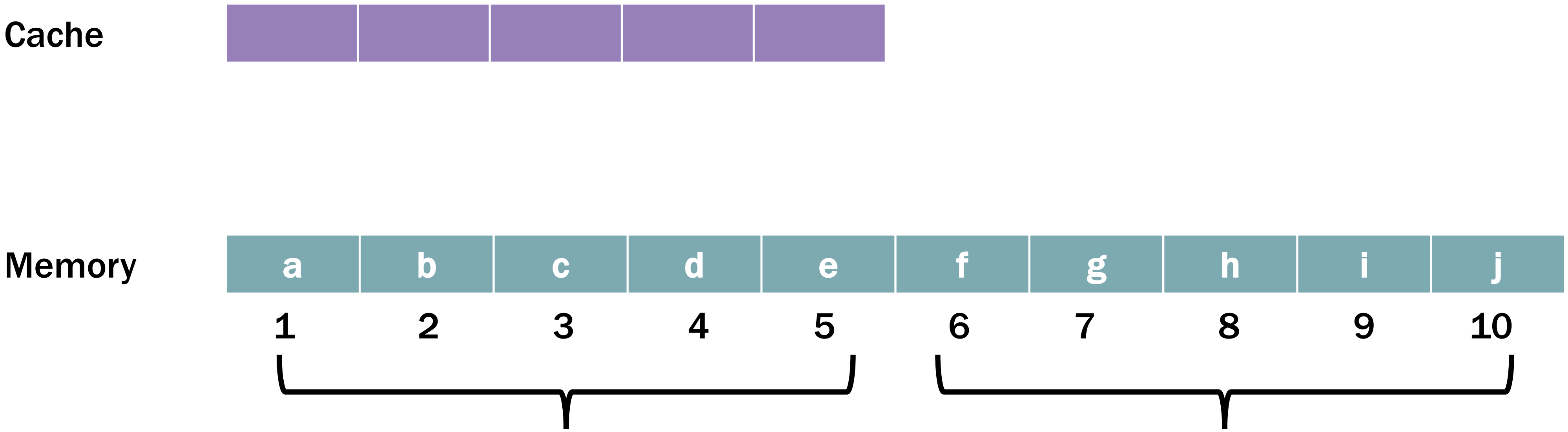
Cache Miss: 2
Cache Hit: 8

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10



ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache

a	b	c	d	e

Cache Miss: 1
Cache Hit: 0

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache

f	g	h	i	j

Replaced

Cache Miss: 2
Cache Hit: 0

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache	a	b	c	d	e	Replaced

Cache Miss: 3
Cache Hit: 0

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache

f	g	h	i	j

Replaced

Cache Miss: 4
Cache Hit: 0

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache	a	b	c	d	e	Replaced

Cache Miss: 5
Cache Hit: 0

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache

f	g	h	i	j

Replaced

Cache Miss: 6
Cache Hit: 0

Memory

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache	a	b	c	d	e	Replaced

Cache Miss: 7
Cache Hit: 0

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache	f	g	h	i	j	Replaced

Cache Miss: 8
Cache Hit: 0

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

Cache	a	b	c	d	e

Replaced

Cache Miss: 9
Cache Hit: 0

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

ACCESS PATTERN 2

Access pattern: 1, 6, 2, 7, 3, 8, 4, 9, 5, 10

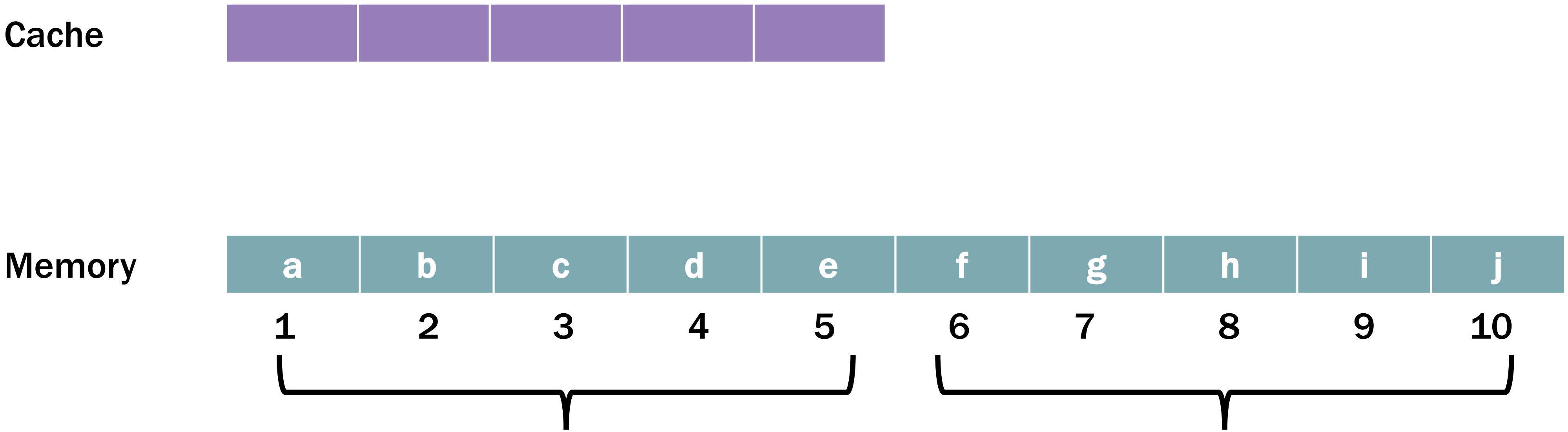
Cache	f	g	h	i	j	Replaced

Cache Miss: 10
Cache Hit: 0

Memory	a	b	c	d	e	f	g	h	i	j
	1	2	3	4	5	6	7	8	9	10

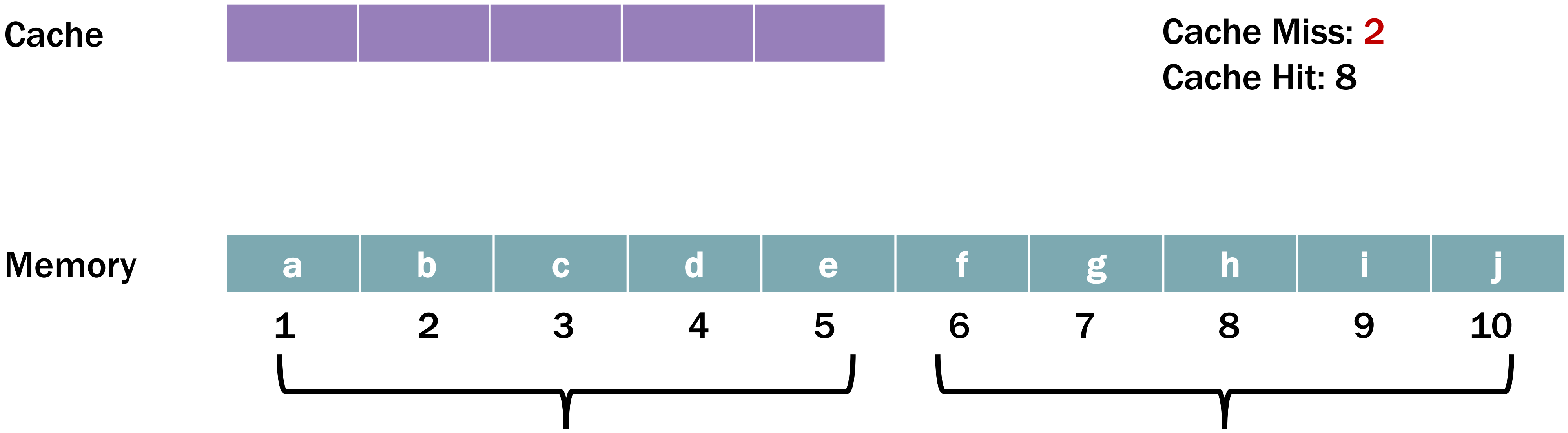
ACCESS PATTERN 3

Access pattern: 1, 6, 8, 7, 7, 8, 9, 9, 6, 10



ACCESS PATTERN 3

Access pattern: 1, 6, 8, 7, 7, 8, 9, 9, 6, 10



Cache Miss: 2
Cache Hit: 8

SUMMARY

- Access pattern with spatial locality (pattern 1) or temporal locality (pattern 3) is cache friendly.

- The adversarial access (pattern 2) may cause the worst case read performance

- When data is big, the impact is very significant

- Whenever possible, we store the data in a way that the access pattern has certain locality.

2D ARRAYS

Suppose we have a 10000x10000 matrix of integers, which is stored in a 2-dimentional array A[10000][10000]. We want to set all of its value to 1. Cache size = 5000 integers. How would you write your code?

Solution X

```
for (int i=0;i<10000;i++)
    for(int j=0;j<10000;j++)
        A[i][j]=1;
```

Solution Y

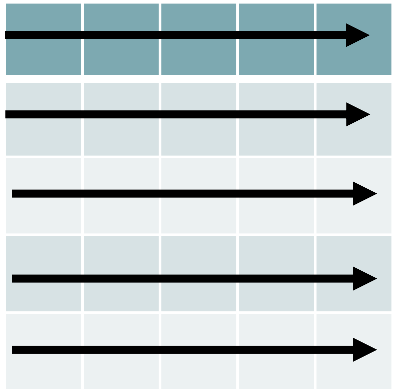
```
for (int j=0;j<10000;j++)
    for(int i=0;i<10000;i++)
        A[i][j]=1;
```

Which one is better, X or Y?

2D ARRAYS

Solution X

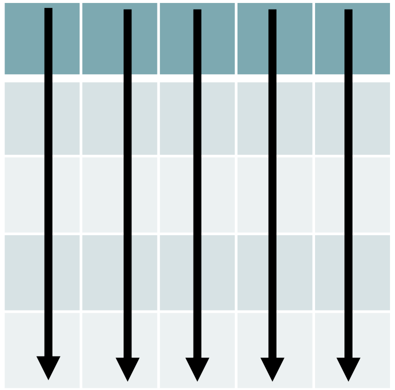
```
for (int i=0;i<10000;i++)
    for(int j=0;j<10000;j++)
        A[i][j]=1;
```



Cache friendly

Solution Y

```
for (int j=0;j<10000;j++)
    for(int i=0;i<10000;i++)
        A[i][j]=1;
```



Always evict cache

Which one is better, X or Y?

SORTING

In the big data era, it happens very often that the data volume is too big to hold in main memory.

Suppose we have 1 TB of integers stored in disk and we want to sort the array. My computer only has 16GB main memory, what should I do?

SORTING

In the big data era, it happens too often that the data is too big to be stored in main memory.

Suppose we have 1 TB of integers stored in disk and we want to sort the array. My computer only has 16GB main memory, what should I do?

We have learnt a lot of sorting algorithms, can we simply use them?

-- We can't allocate enough large array in memory when coding.

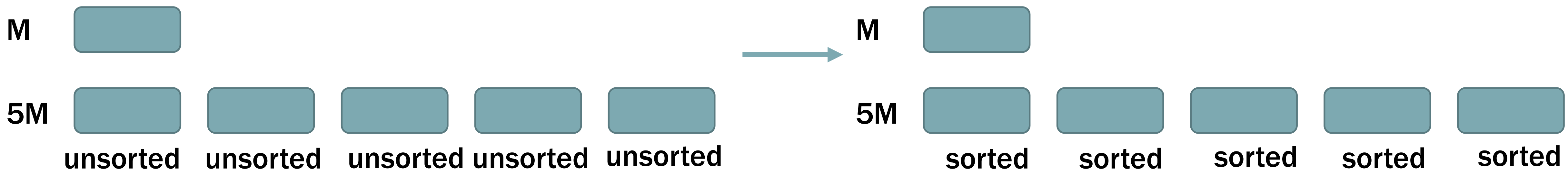
SORTING

The problem of sorting a big array larger than the main memory is called external memory sorting.

Suppose the size of main memory is M, the array size is 5M.

SORTING

Suppose the size of main memory is M, the array size is 5M.

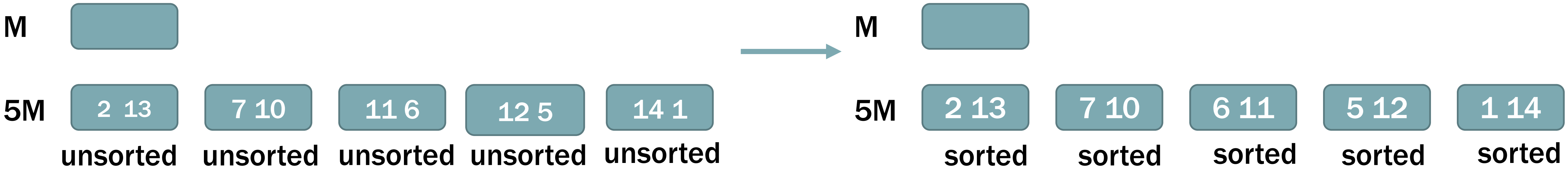


Step 1:

Cut 5M arrays into 5 parts. Each part is put in main memory to sort (using any sorting algorithm we learnt, e.g., quick sort)

SORTING

Suppose the size of main memory is M, the array size is 5M.

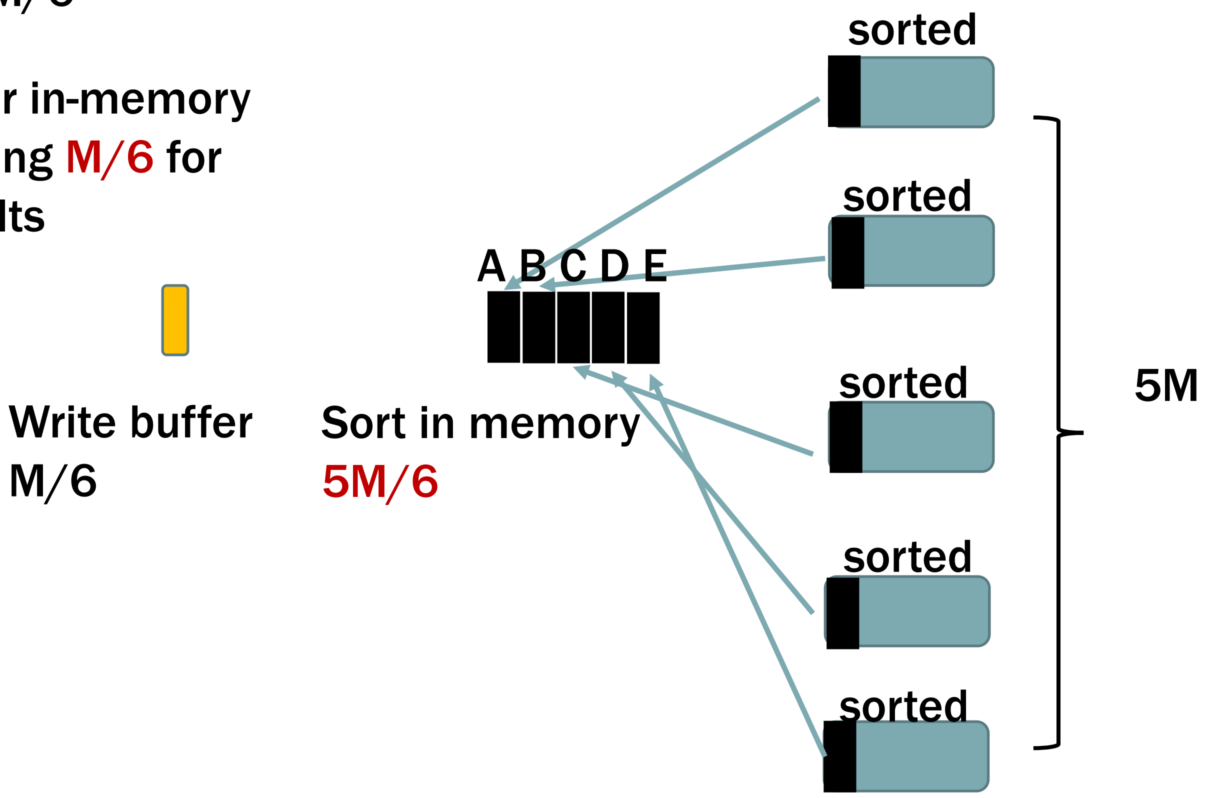


Step 1:

Cut 5M arrays into 5 parts. Each part is put in main memory to sort (using any sorting algorithm we learnt, e.g., quick sort)

SORTING

5M/6 in total for in-memory sorting, remaining M/6 for writing the results



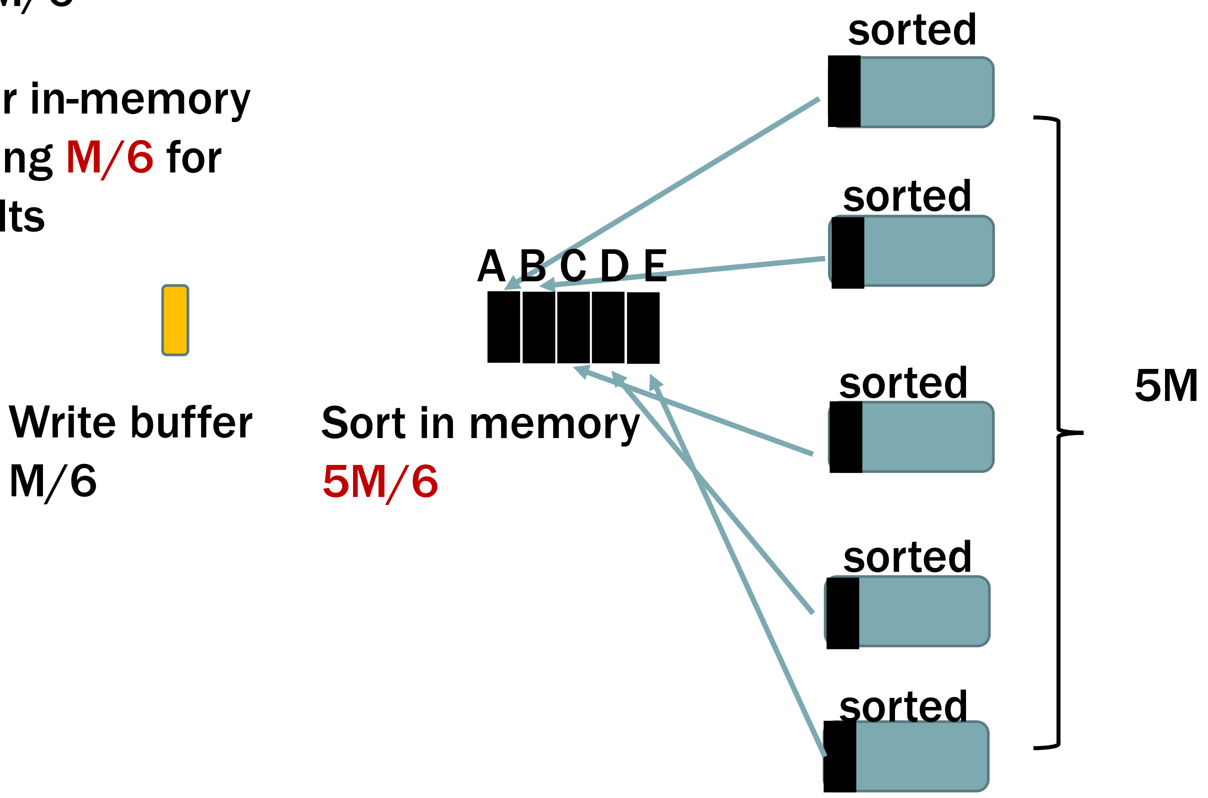
Step 2:

Apply 5-way merge sort to the first M/6 of each sorted part.

SORTING

or size M/6

5M/6 in total for in-memory sorting, remaining M/6 for writing the results



Step 2:

Apply 5-way merge sort to the first M/6 of each sorted part.

Details of merge sort

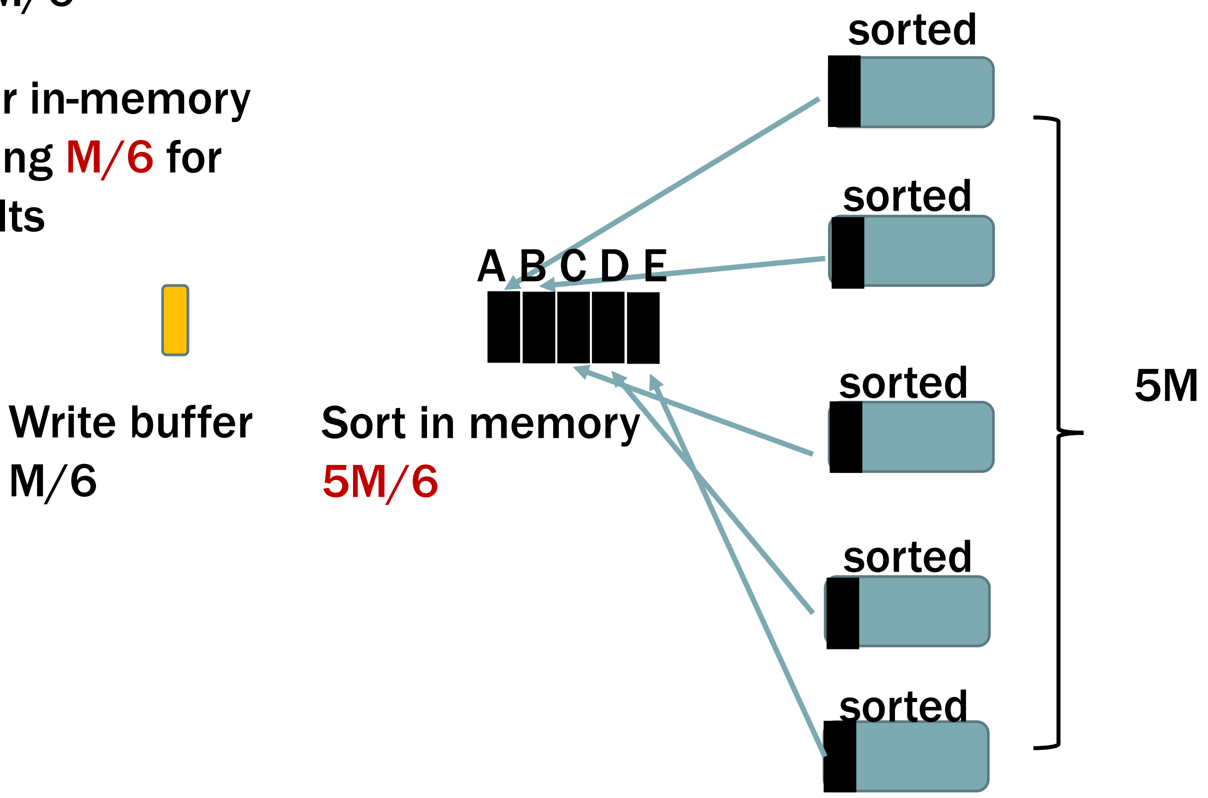
1. 5 iterators scanning each of {A,B,C,D,E} from left to right;
2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

A	2, 13
B	7, 10
C	6, 11
D	5, 12
E	1, 14

SORTING

or size M/6

5M/6 in total for in-memory sorting, remaining M/6 for writing the results



Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;
2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

A	2, 13
B	7, 10
C	6, 11
D	5, 12
E	1, 14

Step 3:

Whenever one of {A,B,C,D,E} is empty, refill it from the source part.

Whenever the write-buffer is full, write it to the disk.

DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

A: 2, 13

B: 7, 10

C: 6, 11

D: 5, 12

E: 1, 14

Write buffer (size 2)

DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;
2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

A	2, 13
B	7, 10
C	6, 11
D	5, 12
E	14

Write buffer

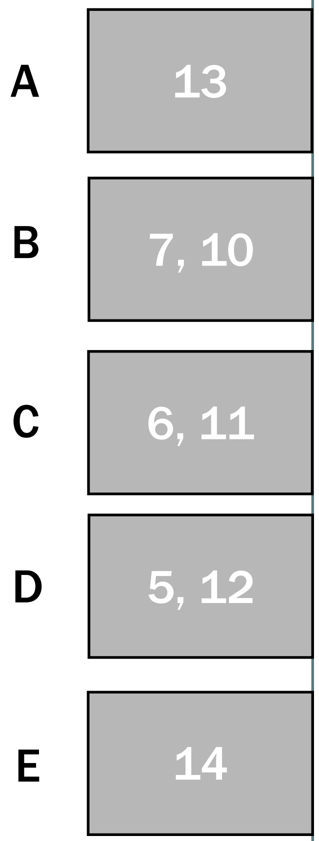
1

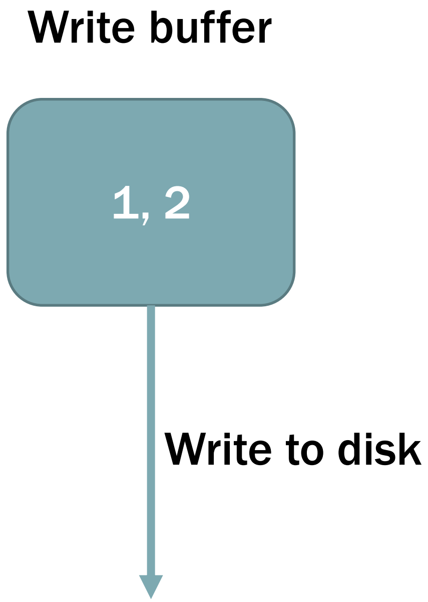
DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;





DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

A	13
B	7, 10
C	6, 11
D	5, 12
E	14

Write buffer

DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

A: 13

B: 7, 10

C: 6, 11

D: 12

E: 14

Write buffer

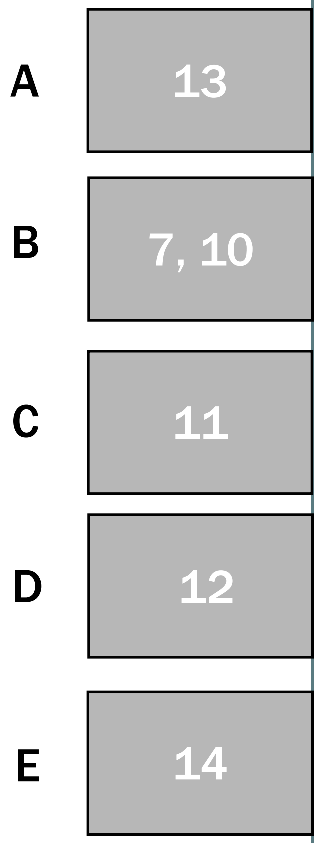
5

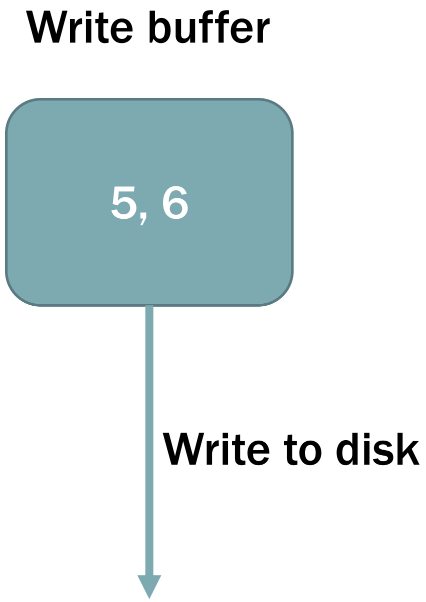
DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;



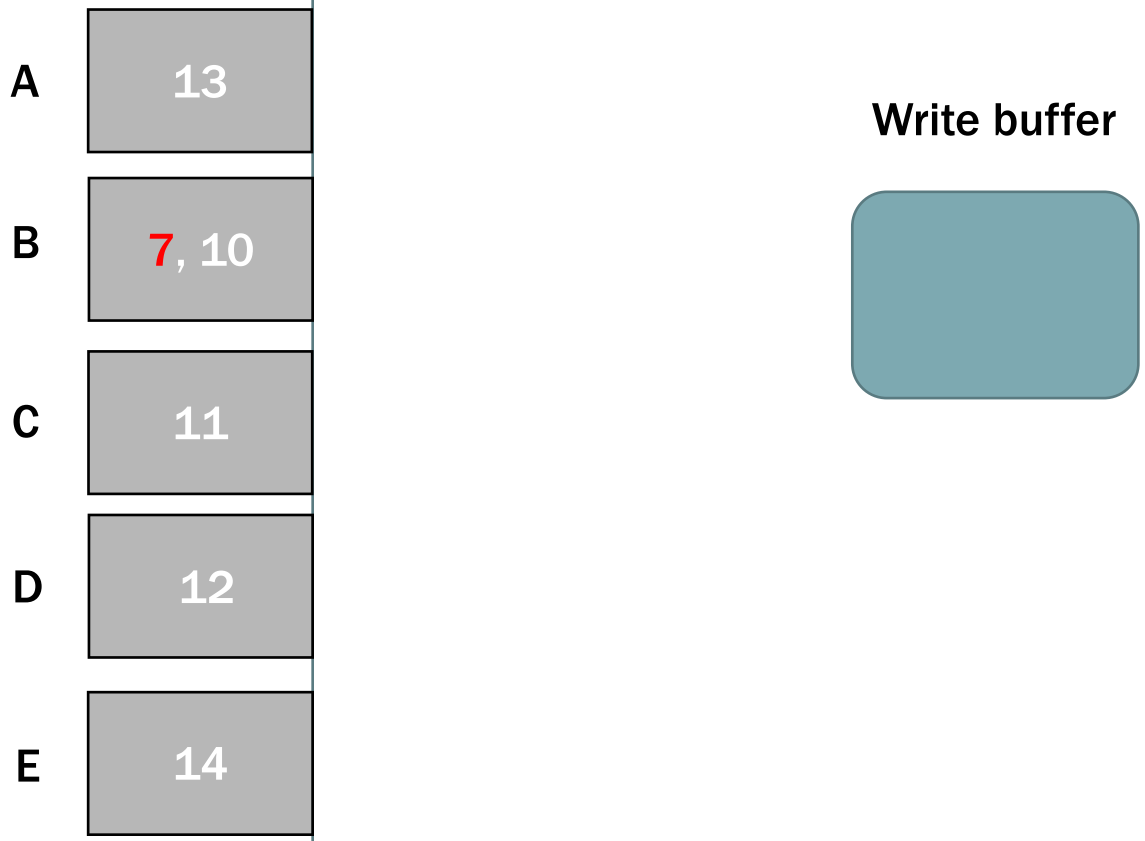


DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

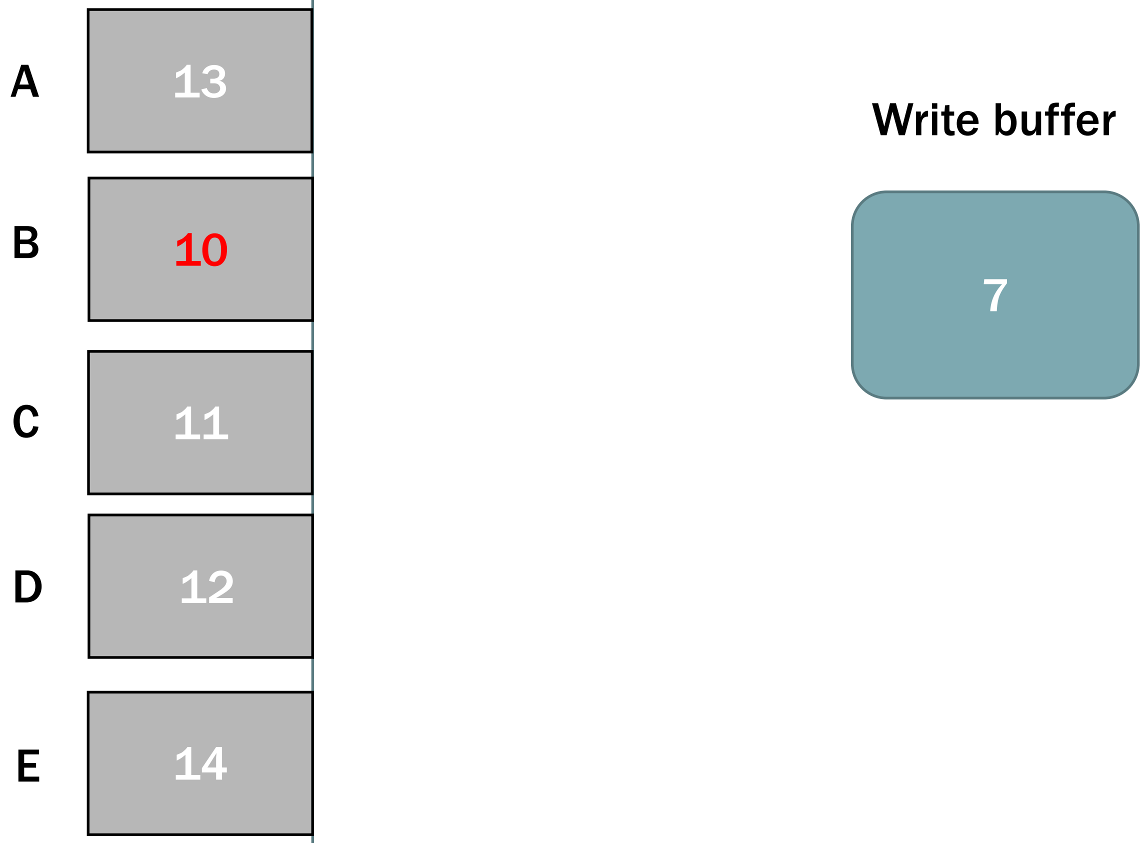


DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;

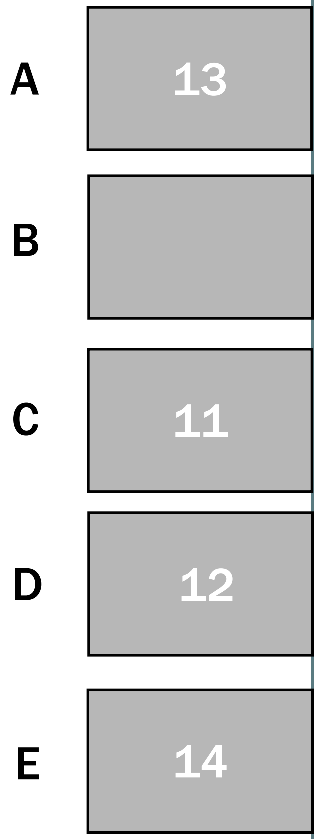


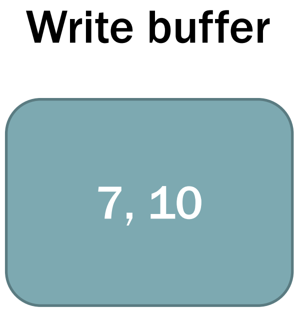
DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;



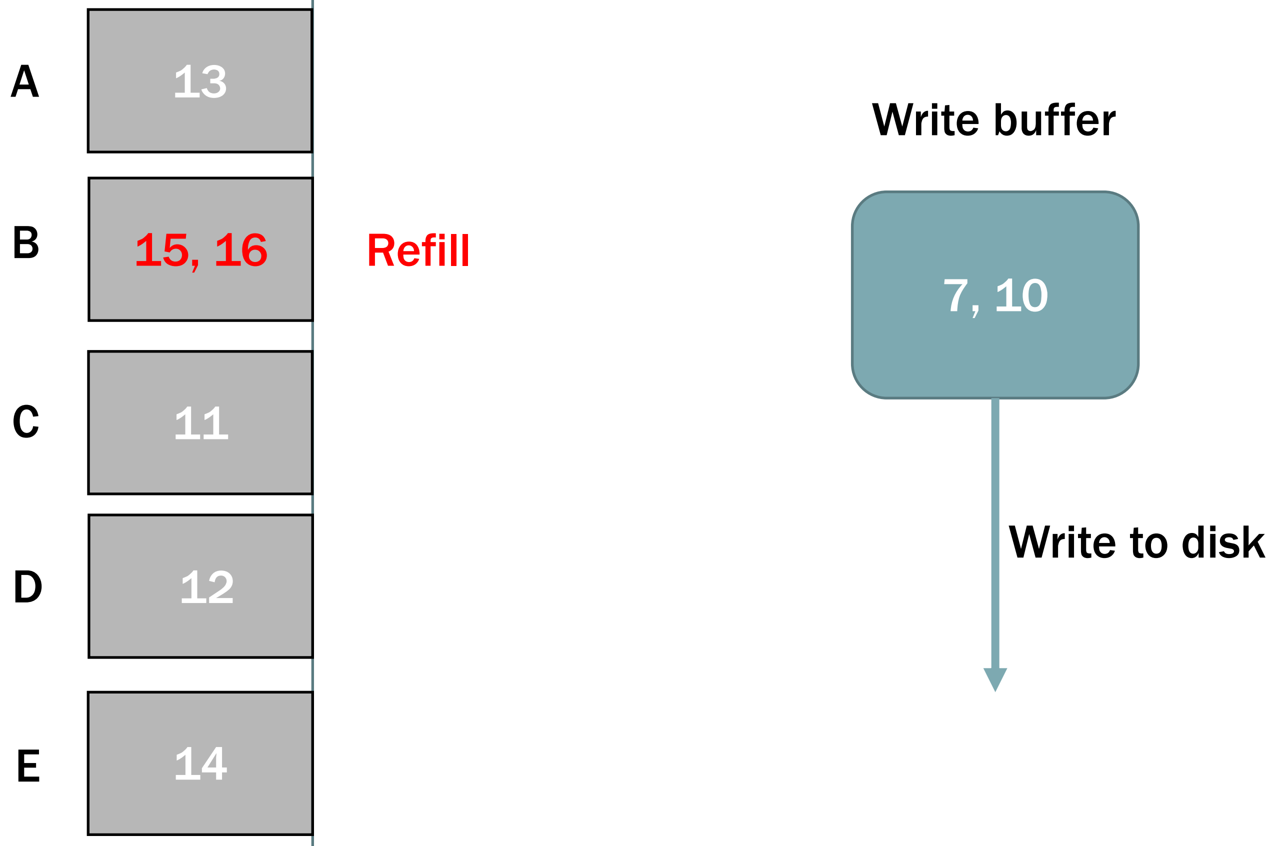


DETAILED EXAMPLE FOR (STEP 2&3)

Details of merge sort

1. 5 iterators scanning each of {A,B,C,D,E} from left to right;

2. Put the smallest-value (pointed by the 5 iterators) in the write-buffer; forward the corresponding iterator;



What's the cost?

COST ANALYSIS

Let N be the data size, M be the memory size, and the B be the page (transfer unit) size.

- Step 1: Read all the N items once and write back to the disk once, incurring O(N/B) I/Os.

COST ANALYSIS

Let N be the data size, M be the memory size, and the B be the page (transfer unit) size.

- Step 1: Read all the N items once and write back to the disk once, incurring O(N/B) I/Os.

- Step 2&3: During merge and writeback, each item is read from and written to the disk exactly once, incurring O(N/B) I/Os.

COST ANALYSIS

Let N be the data size, M be the memory size, and the B be the page (transfer unit) size.

- Step 1: Read all the N items once and write back to the disk once, incurring O(N/B) I/Os.

- Step 2&3: During merge and writeback, each item is read from and written to the disk exactly once, incurring O(N/B) I/Os.

Does this analysis apply to all situations?

COST ANALYSIS

To make the analysis hold, we should have

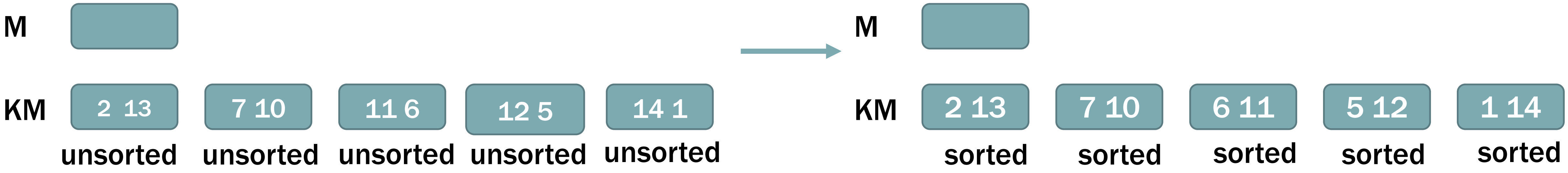
- ❑ The size of write buffer is at least one page size (size B)

- ❑ The process of in-memory merge sort should be fit in main memory.

Does this analysis apply to all situations?

SORTING

Suppose the size of main memory is M, the array size is KM(=N).



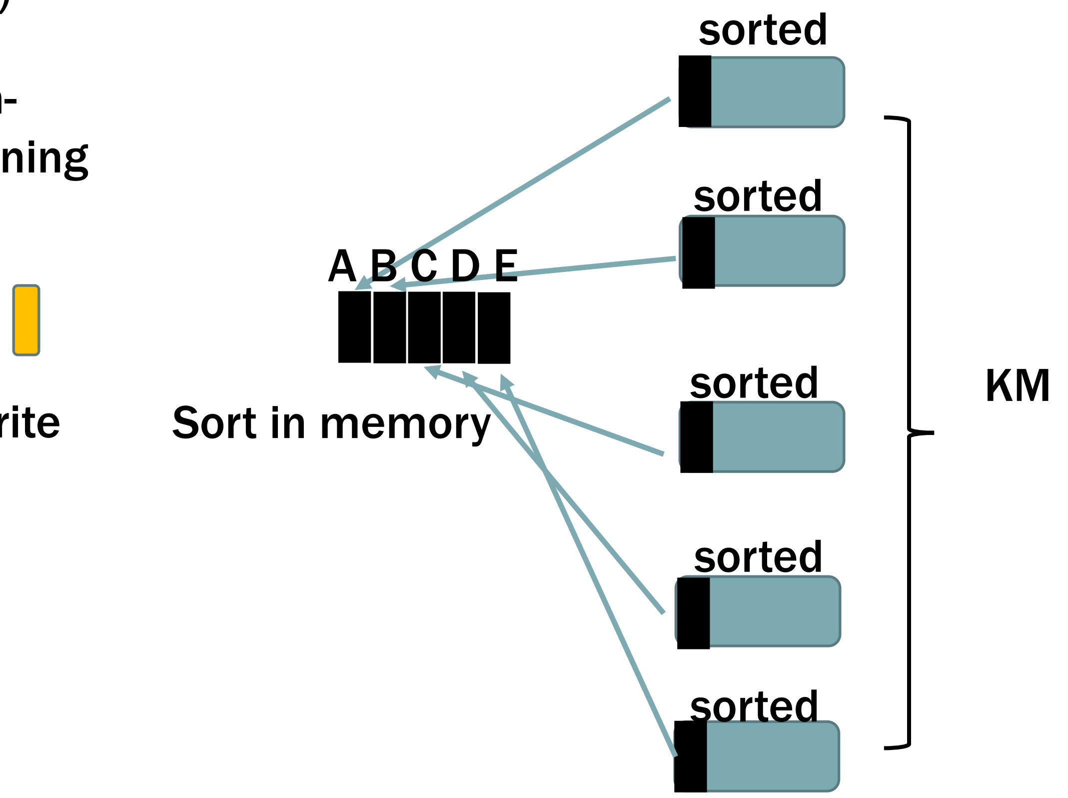
Step 1:

Cut KM arrays into K parts. Each part is put in main memory to sort (using any sorting algorithm we learnt, e.g., quick sort)

SORTING

or size M/(K+1)

KM/(K+1) in total for in-memory sorting, remaining M/(K+1) for writing the results



Step 2:

Apply K-way merge sort to the first M/(K+1) of each sorted part.

THE RANGE OF K

- Condition 1:

A write buffer has at least one page size (denoted by B).

Hence, we have $M/(K+1) >= B$, giving us **$K <= M/B-1$**

- Condition 2:

The process of K-way merge-sort should be fit in remaining memory size (excluding write buffer).

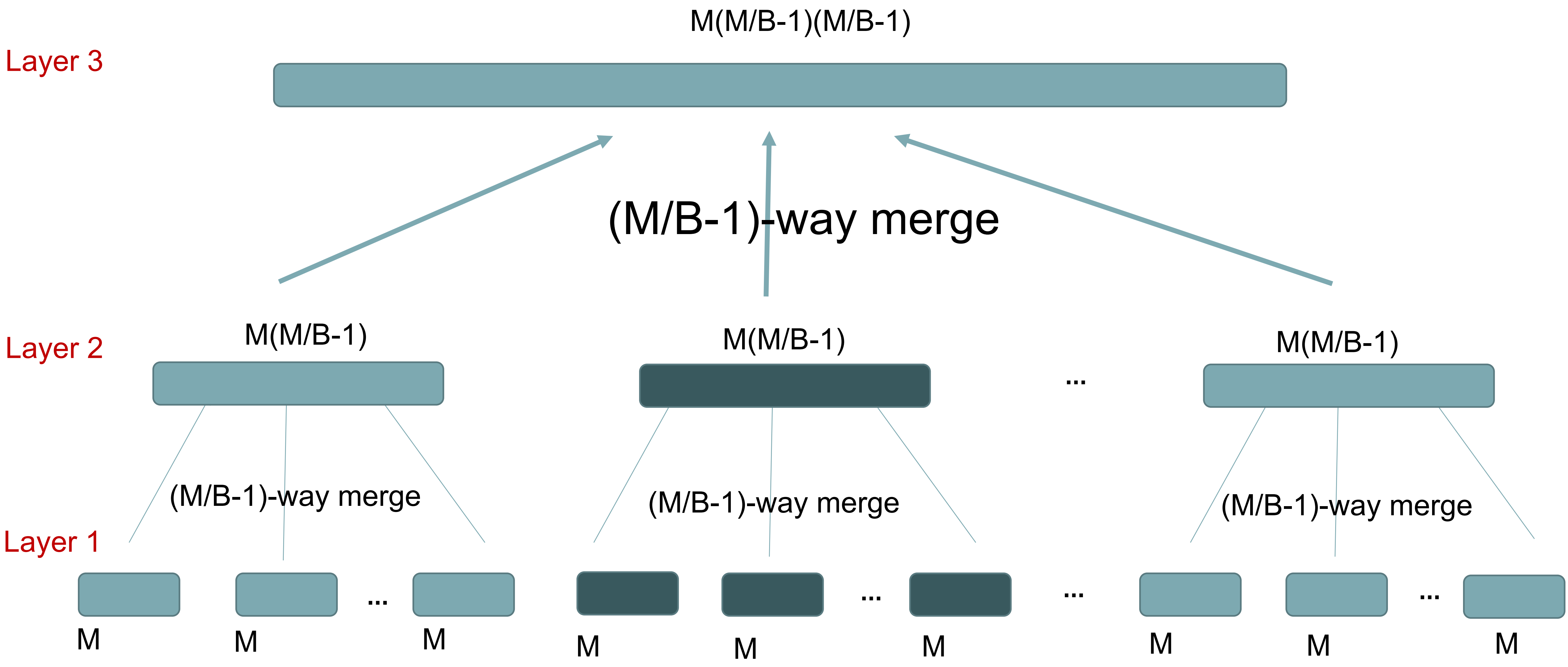
Hence, $K \times B <= KM/(K+1)$, also giving us **$K <= M/B-1$**

EXTEND TO GENERAL CASE

- The cost analysis holds when N is at most (M/B-1)M.

- We can extend the method to general N by recursively applying the (M/B-1)-way merge.

RECURSIVELY APPLY (M/B-1)-WAY MERGE



GENERAL ANALYSIS

- If there are L layers, then $M(M/B-1)^{L-1} = N$ because N items should be sorted, giving

  $L = O(\log_{M/B} N/M)$

$$
\begin{aligned}
M(M/B-1)^{L-1} &= N \\
\Leftrightarrow (M/B-1)^{L-1} &= N/M \\
\Leftrightarrow \log_{M/B-1}(M/B-1)^{L-1} &= \log_{M/B-1} N/M \\
\Leftrightarrow L-1 &= \log_{M/B-1} N/M \\
\Leftrightarrow L &= 1 + \log_{M/B-1} N/M \\
\Leftrightarrow L &= O(\log_{M/B} N/M)
\end{aligned}
$$

- The merges in each layer costs O(N/B) I/Os.
- Hence, the total cost is $O((N/B) \log_{M/B} N/M)$ I/Os.

We finish Memory Hierarchy!

Next lecture: Column Store