Index Construction

Younghoon Kim
(nongaussian@gmail.com)

It contains some modified slides, which were originally written by Jim Martin, Donald Patterson Min-Yen Kan, and Zhang & Helmer, used for the Stanford CS276 class and from the Stuttgart IIR class
https://nlp.stanford.edu/IR-book/newslides.html

Index construction

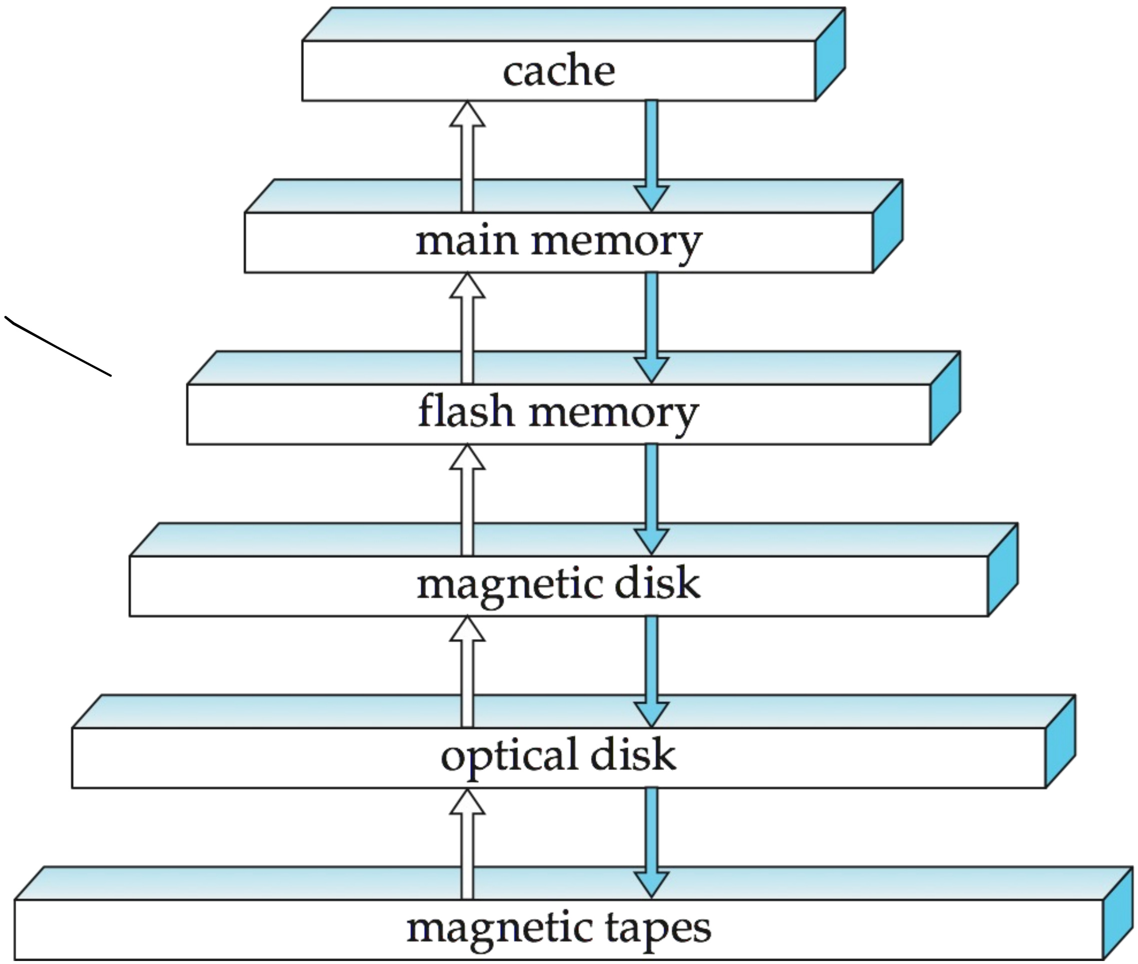
- How do we construct an index?
- What strategies can we use with limited main memory?

Hardware basics

- Many design decisions in information retrieval are based on the characteristics of hardware
- We begin by reviewing hardware basics

DISK I/O

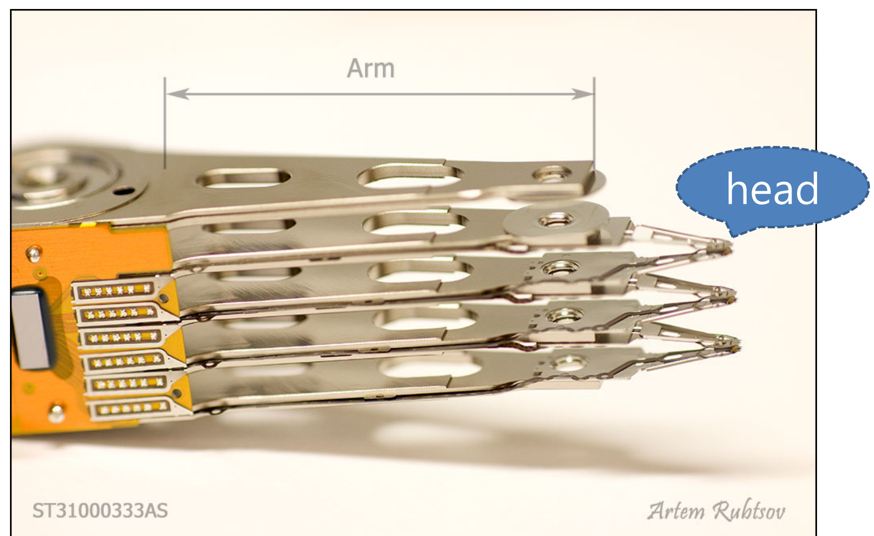
Storage Hierarchy



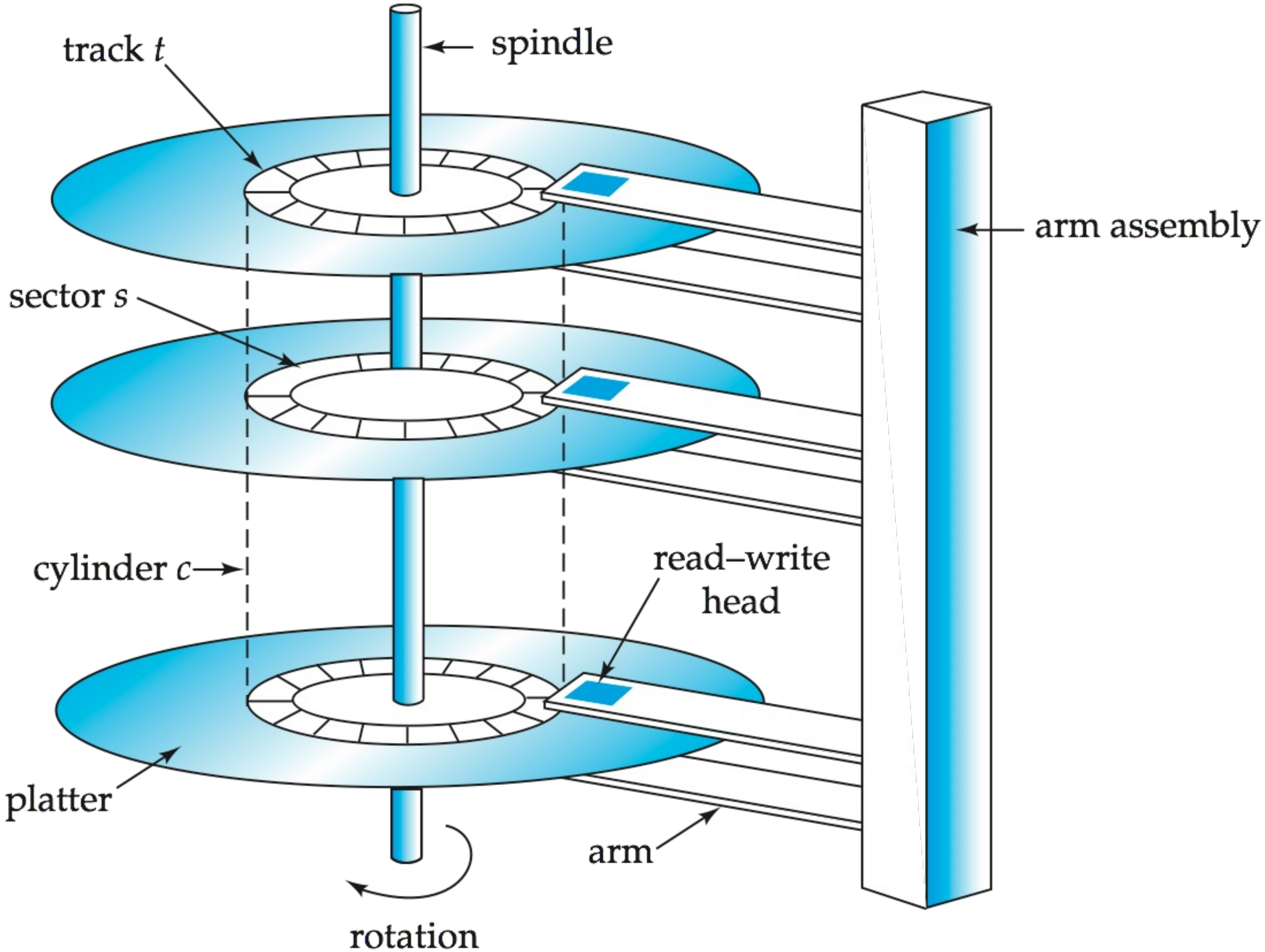
Storage Hierarchy (Cont.)

- **primary storage**: Fastest media but volatile (cache, main memory).
- **secondary storage**: next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- ~~**tertiary storage**: lowest level in hierarchy, non-volatile, slow access time~~
  - ~~also called **off-line storage**~~
  - ~~E.g. magnetic tape, optical storage~~

Magnetic Hard Disk Mechanism



Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

Magnetic Disks

- Read-write head
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.
- Surface of platter divided into circular tracks
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into sectors.
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm.

Performance Measures of Disks

- Access time (=seek time) – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - Seek time – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
    - 4 to 10 milliseconds on typical disks
  - Rotational latency – time it takes for the sector to be accessed to appear under the head.
    - Average latency is 1/2 of the worst case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- Data-transfer rate – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 100 MB per second max rate, lower for inner tracks
  - Multiple disks may share a controller, so rate that controller can handle is also important
    - E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
    - Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)

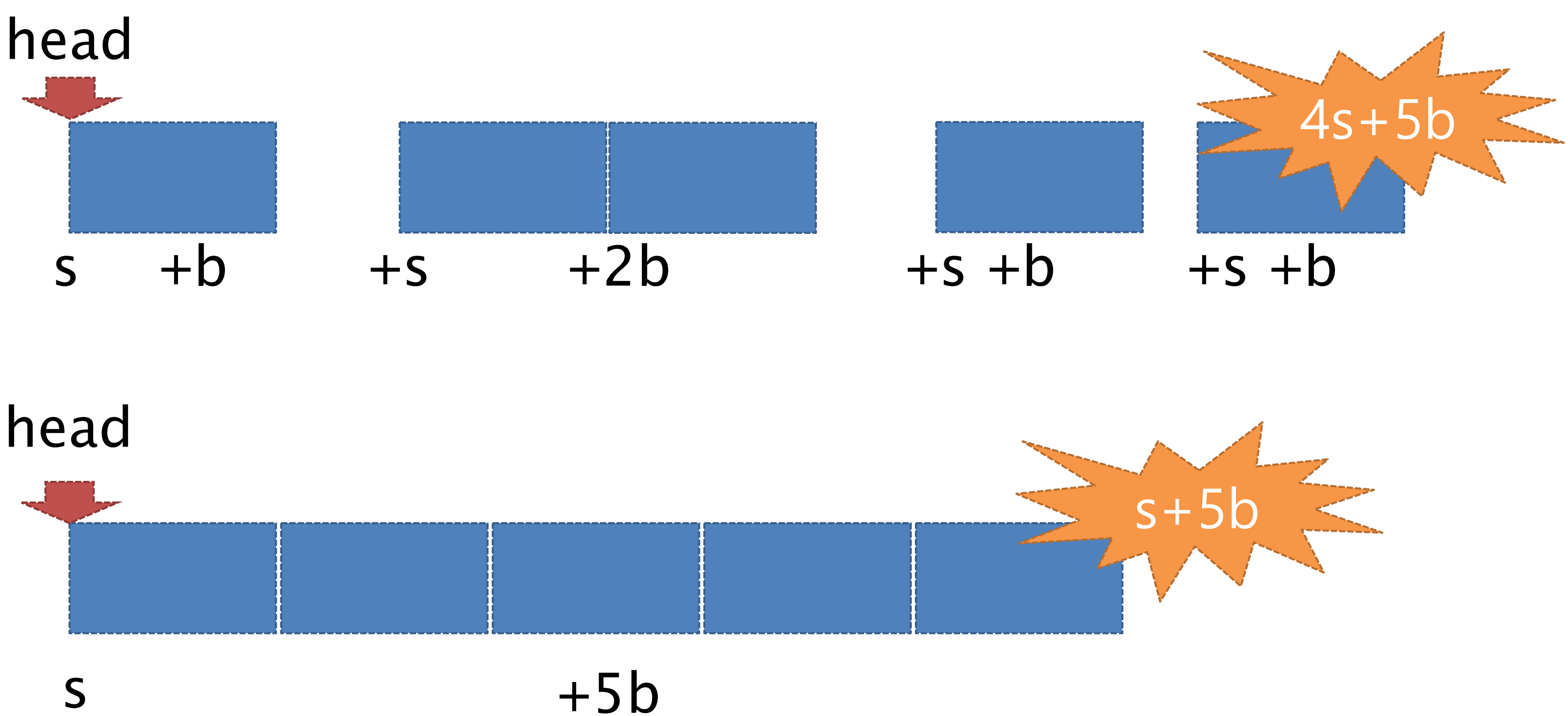
Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
  - data is transferred between disk and main memory in blocks
  - sizes range from 512 bytes to several kilobytes
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
    - Typical block sizes today range from **4** to **16** kilobytes

Optimization of Disk Block Access

- **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
  - E.g. Store related information on the same or nearby cylinders.
  - Files may get fragmented over time
    - E.g., if data is inserted to/deleted from the file
    - Or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
    - Sequential access to a fragmented file results in increased disk arm movement
  - Some systems have utilities to defragment the file system, in order to speed up file access

Optimization of Disk Block Access



SORT-BASED INDEX CONSTRUCTION

Index Construction

- Grouping postings with term
  - → sort
- Scaling sort for large data
  - → external merge-sort

Scaling Index Construction

- Grouping postings

Term	docID
I	1
did	1
enact	1
julius	1
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambit	2

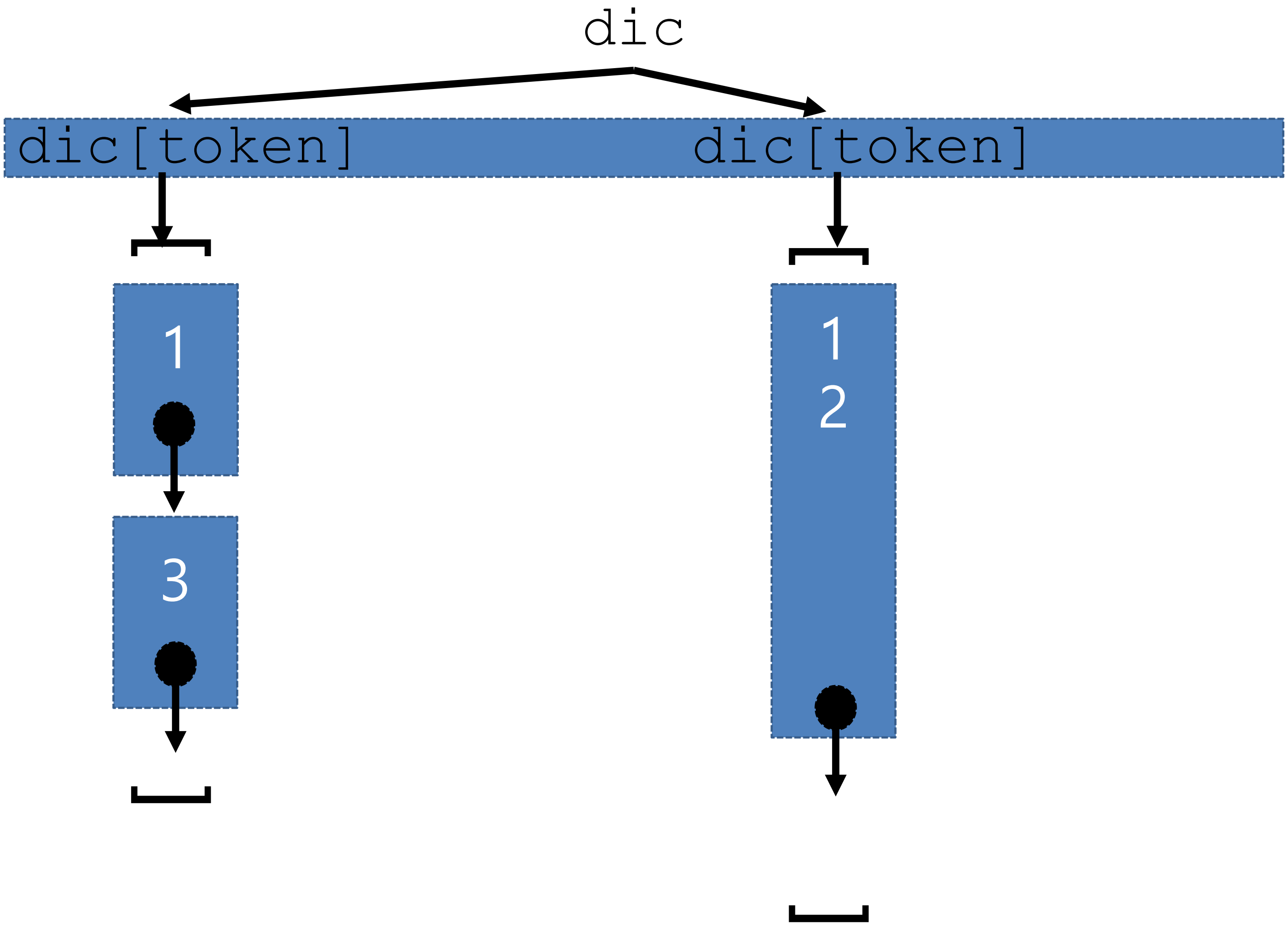
```python
dic = dict()

for token, docid in postings:
  if token not in dic:
    dic[token] = list()

  dic[token].append(docid)
```

Term	docID
ambit	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
I	1
it	2
julius	1
kill	1
kill	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Scaling Index Construction



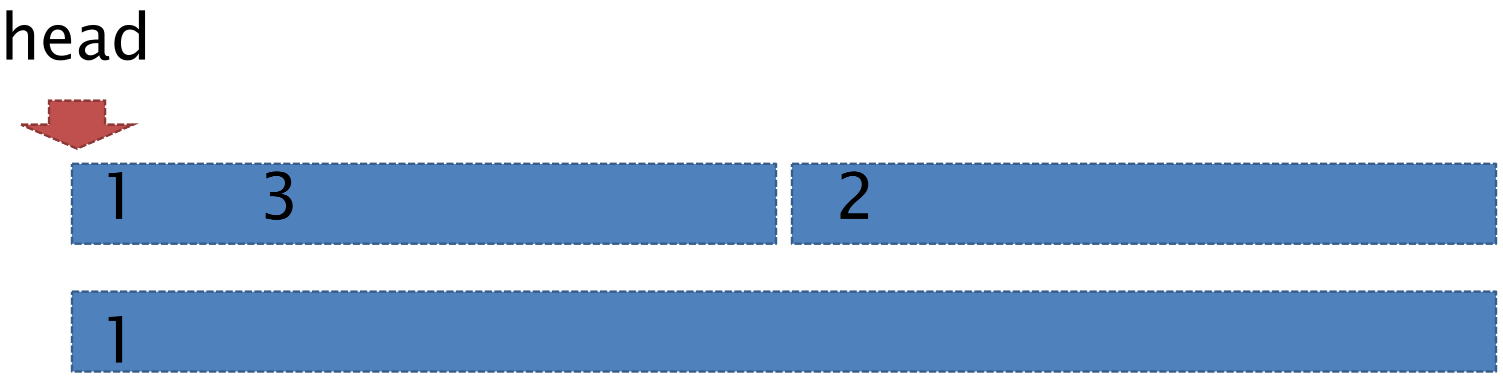
Scaling Index Construction

- In-memory index construction does not scale
  - Can’t stuff entire collection into memory, sort, then write back
- **How can we construct an index for very large collections?**
  - Considering the hardware constraints we just learned about . . .
  - Memory, disk, speed, etc.

Let’s talk about straightforward methods

Sort using disk as "memory"?

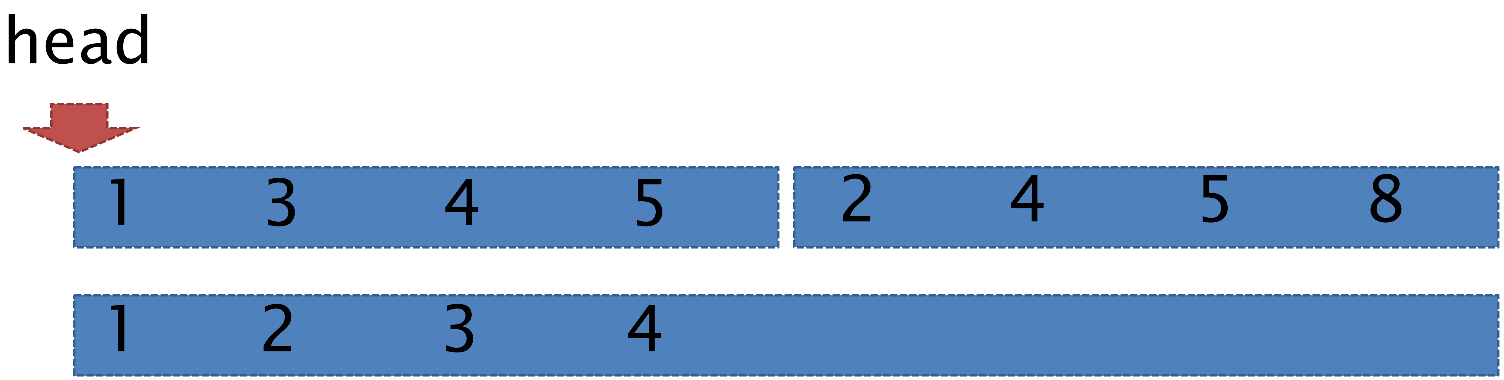
- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?



- No: Sorting on disk is too slow due to **too many disk seeks**.

Sort using disk as "memory"?

- Block-based disk I/O



BLOCK-BASED DISK I/O

Example: Join Operation

- Join
  - Two relations ‘student’ and ‘takes’ on student IDs
  - E.g., SELECT r.id FROM student as r JOIN takes as s ON r.id = s.studentID
- Examples use the following information
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400
  - That is,
    - A block can hold 50 student records and 25 takes records respectively
    - $n_r = 5{,}000$
    - $n_s = 10{,}000$
    - $b_r = 100$
    - $b_s = 400$

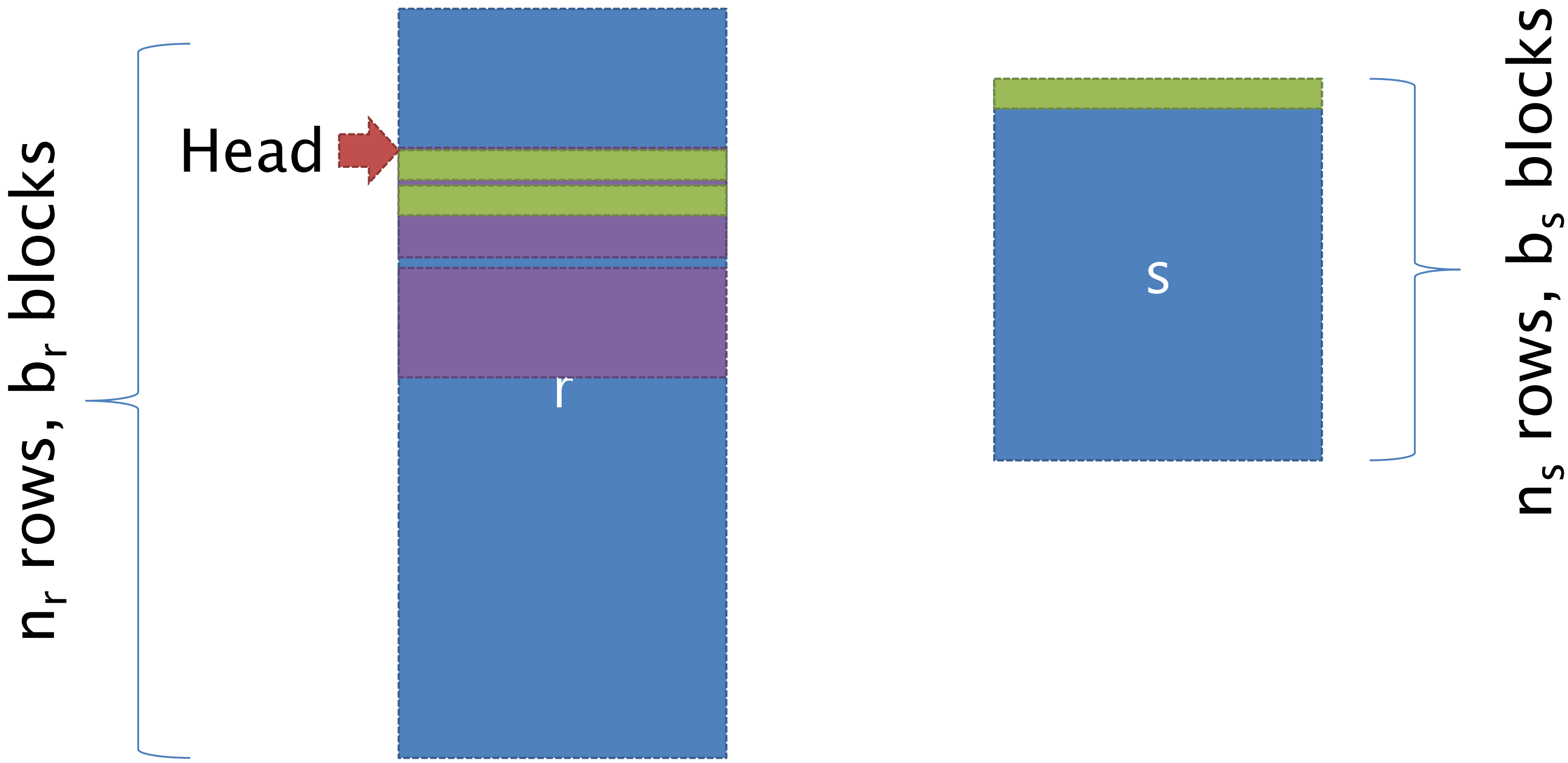
Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$ (θ: join condition)

```
for each tuple t_r in r do begin
    for each tuple t_s  in s do begin
        test pair (t_r, t_s) to see if they satisfy θ
        if they do, add (t_r, t_s) to the result
    end
end
```

- *r* is called the **outer relation** and *s* the **inner relation** of the join
- Assume that no indices can be used with any kind of join condition

Nested-Loop Join



# of block transfers: b_r + $n_r * b_s$

# of disk seeks: b_r + n_r

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

  $n_r * b_s + b_r$ block transfers, plus

  $n_r + b_r$ seeks

- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 ∗ 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 ∗ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- Block nested-loops algorithm (next slide) is preferable.

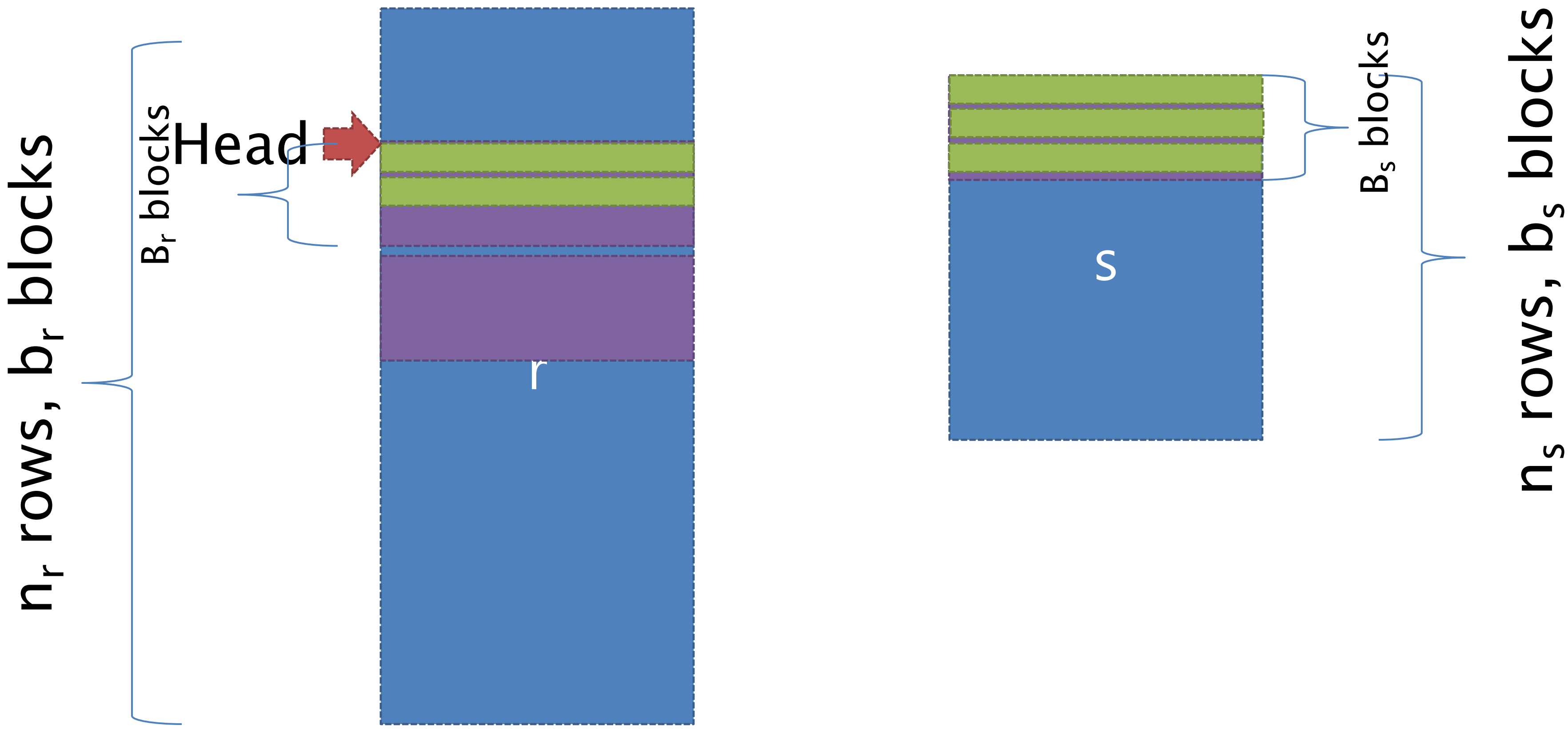
Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

for each block B_r **of** r **do begin**
    for each block B_s **of** s **do begin**
        for each tuple t_r **in** B_r **do begin**
            for each tuple t_s **in** B_s **do begin**
                Check if (t_r, t_s) satisfy the join condition
                if they do, add (t_r, t_s) to the result
            end
        end
    end
end

Block Nested-Loop Join

Available blocks: $B_r + B_s$



# of block transfers: $b_r \quad + \quad b_s \cdot \left\lceil \frac{b_r}{B_r} \right\rceil$

# of disk seeks: $\left\lceil \frac{b_r}{B_r} \right\rceil \quad + \quad \left\lceil \frac{b_r}{B_r} \right\rceil$

Block Nested-Loop Join (Cont.)

- Available memory: $B_r + B_s$ blocks

  Block transfers: $b_r + b_s \cdot \left\lceil \frac{b_r}{B_r} \right\rceil$

  Disk seeks: $2\left\lceil \frac{b_r}{B_r} \right\rceil$

- In the worst case: $B_r = 1$ and $B_s = 1$
  - $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
    - 100 * 400 + 100 = 40,100 block transfers
    - 2 * 100 = 200 seeks

Nested-loop Join:
2,000,100 block transfers +
5,100 seeks

Block Nested-Loop Join (Cont.)

- I/O costs

  Block transfers: $b_r + b_s \cdot \left\lceil \frac{b_r}{B_r} \right\rceil$

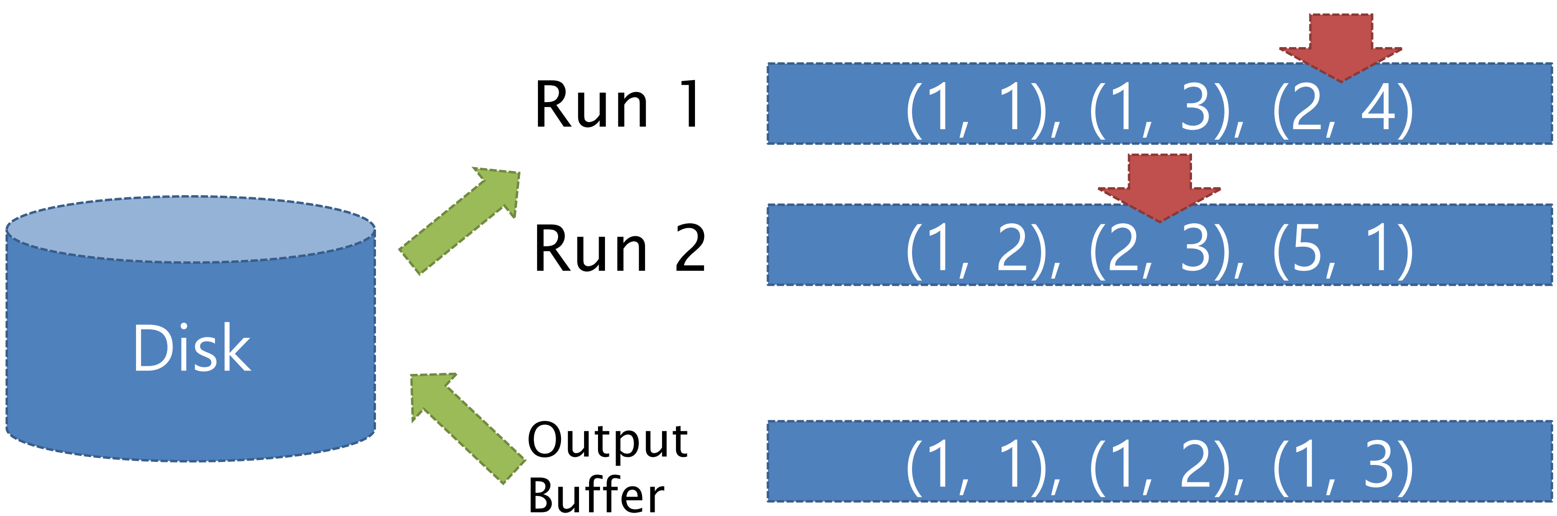
  Disk seeks: $2 \left\lceil \frac{b_r}{B_r} \right\rceil$

- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use (M – 2) memory blocks as the buffer for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - **Cost = $\boldsymbol{b_r + b_s \cdot \left\lceil \frac{b_r}{M-2} \right\rceil}$ block transfers + 2 $\boldsymbol{\left\lceil \frac{b_r}{M-2} \right\rceil}$ seeks**

EXTERNAL MERGE SORT

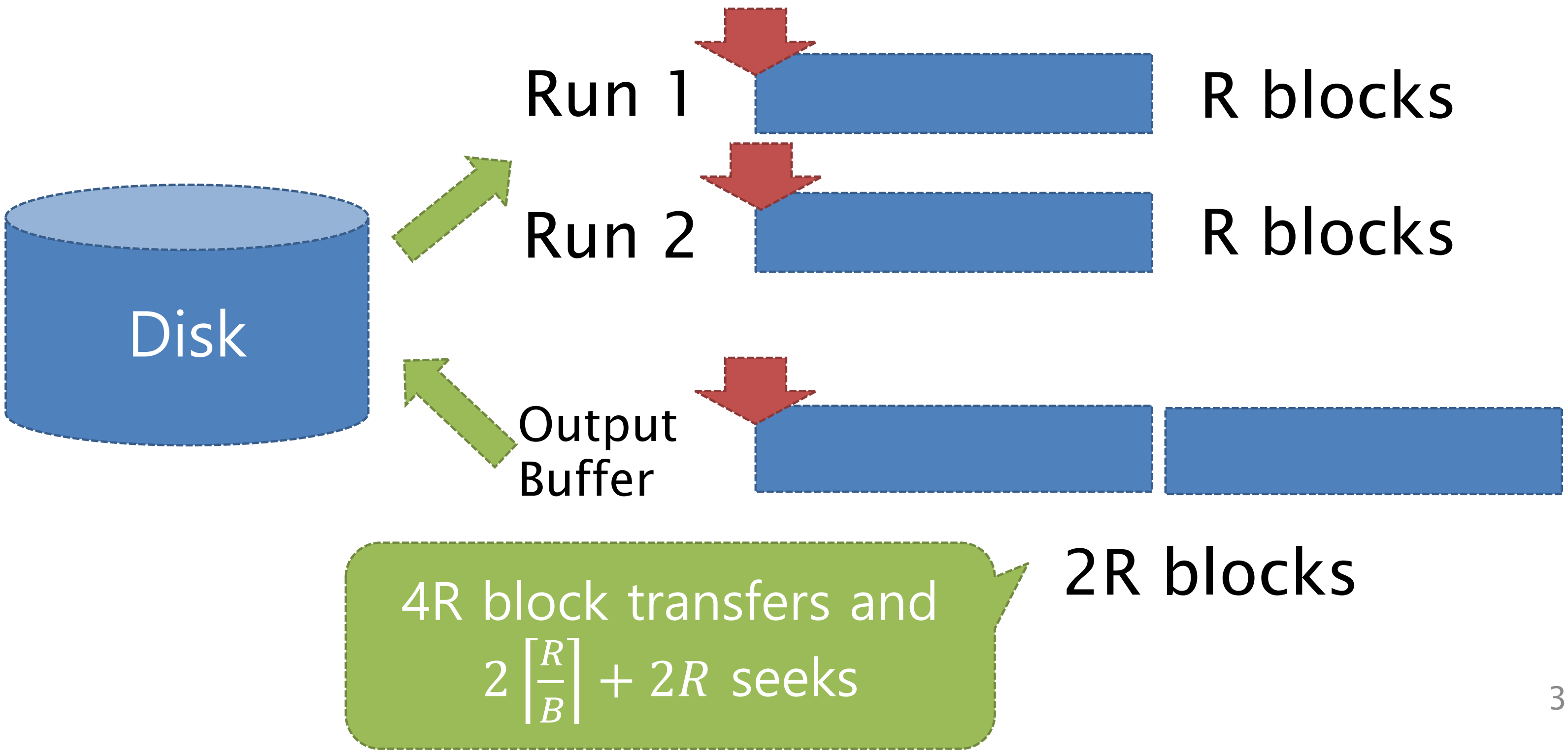
BSBI: Blocked sort-based Indexing (Sorting with fewer *disk seeks*)

- 12-byte (4+4+4) records *(termID, docID, pos).*
- Run(Block aligned): Already sorted list of records
- Basic idea of algorithm:
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.



I/O Cost of Merging Two Runs

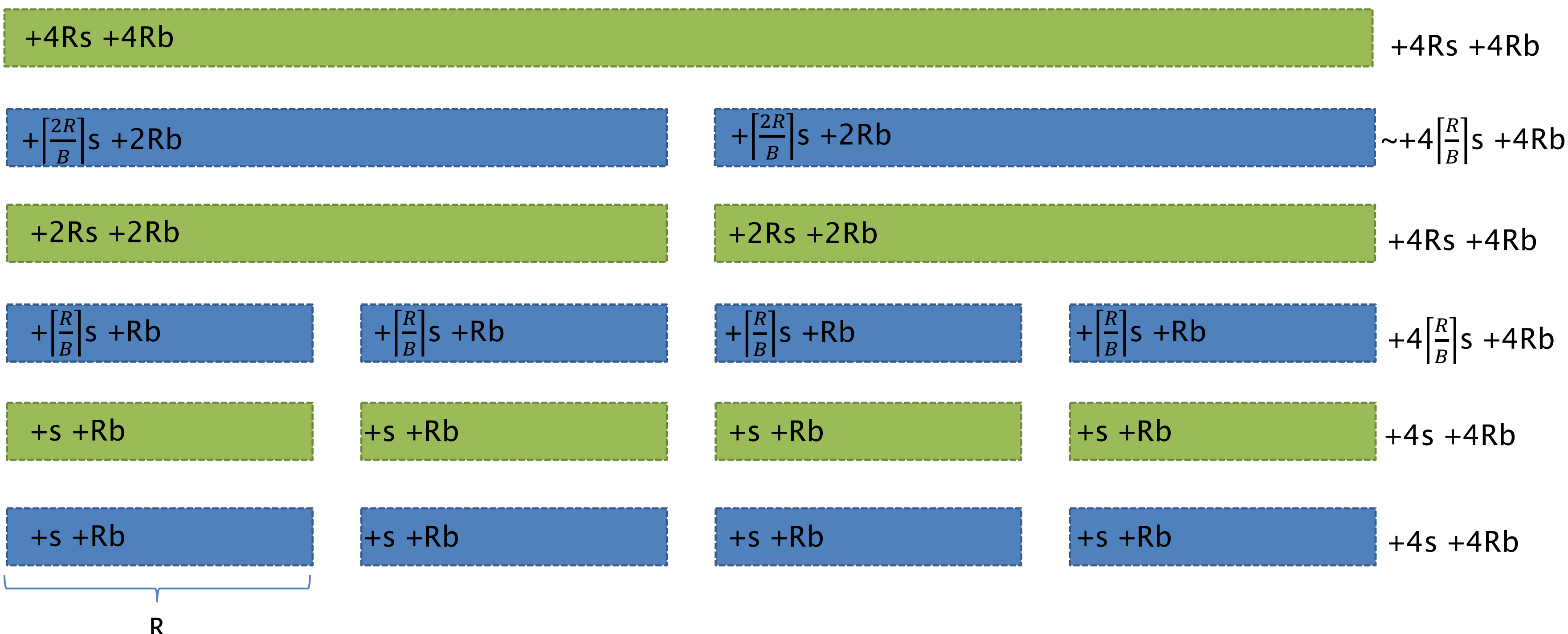
- Assume
  - We merge two runs and write the intermediated sorted run on disk
  - We are available M (=2B + 1) blocks of memory



Sorting N blocks of records

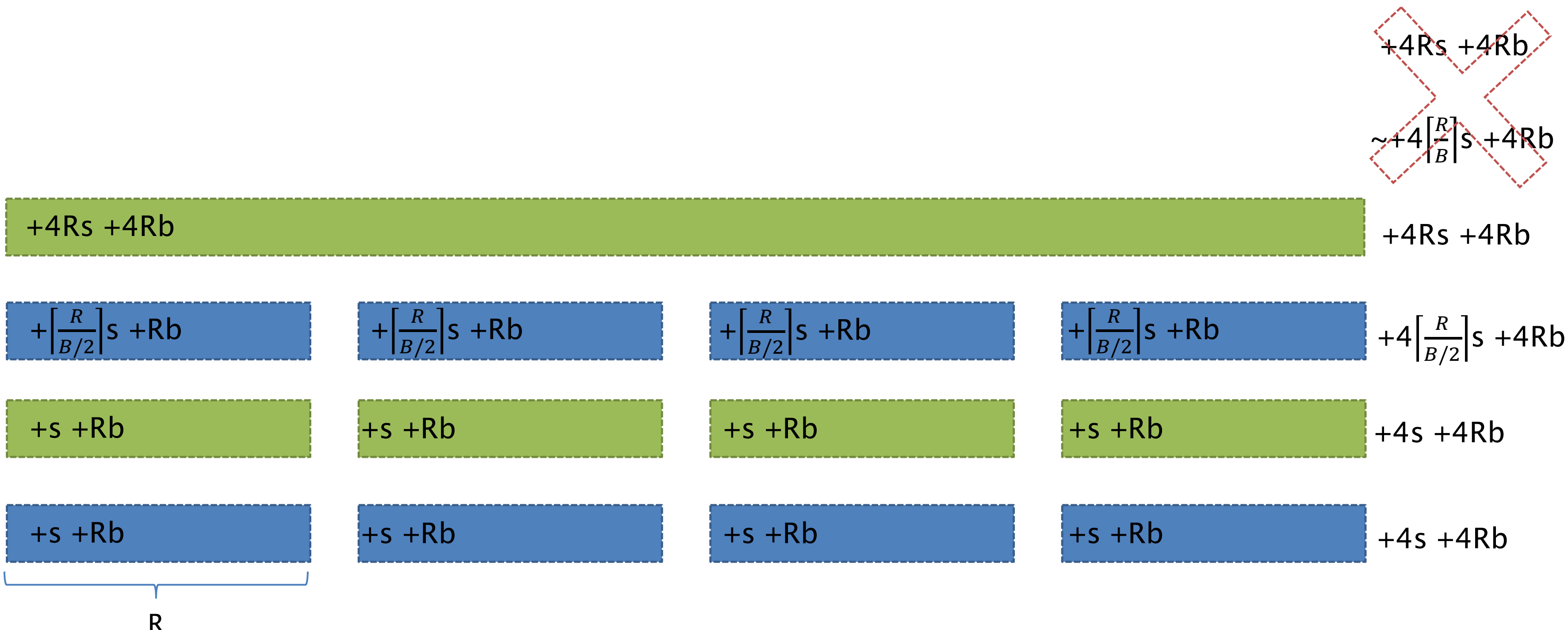
- Mergesort do binary merges, with a merge tree of $\log_2 N$ depth where N = the number of smallest runs
- During each level, read into memory runs, merge, write back to disk
- Assume M (=2B + 1) blocks of memory available and the initial run size is R blocks

Temporary setting



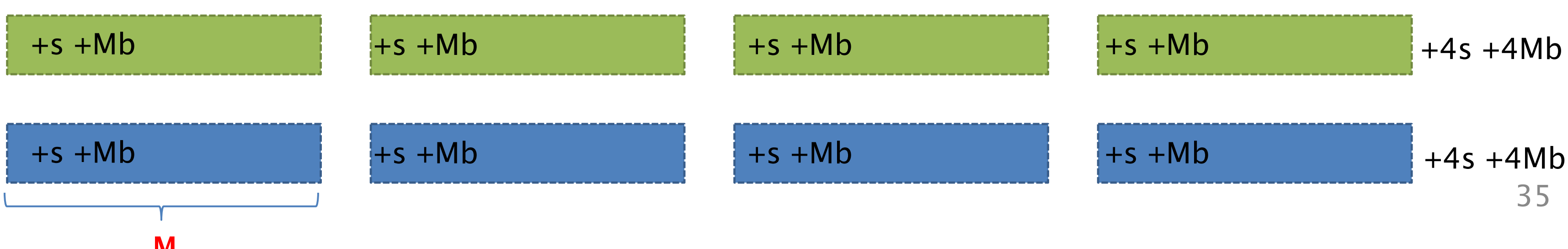
Binary Merge vs. Multi-way Merge

- But it is more efficient to do **a multi-way merge**, where you are reading from all blocks simultaneously
- Assume M (=2B + 1) blocks of memory available and the initial run size is R blocks
  - Use B/2 blocks for each run



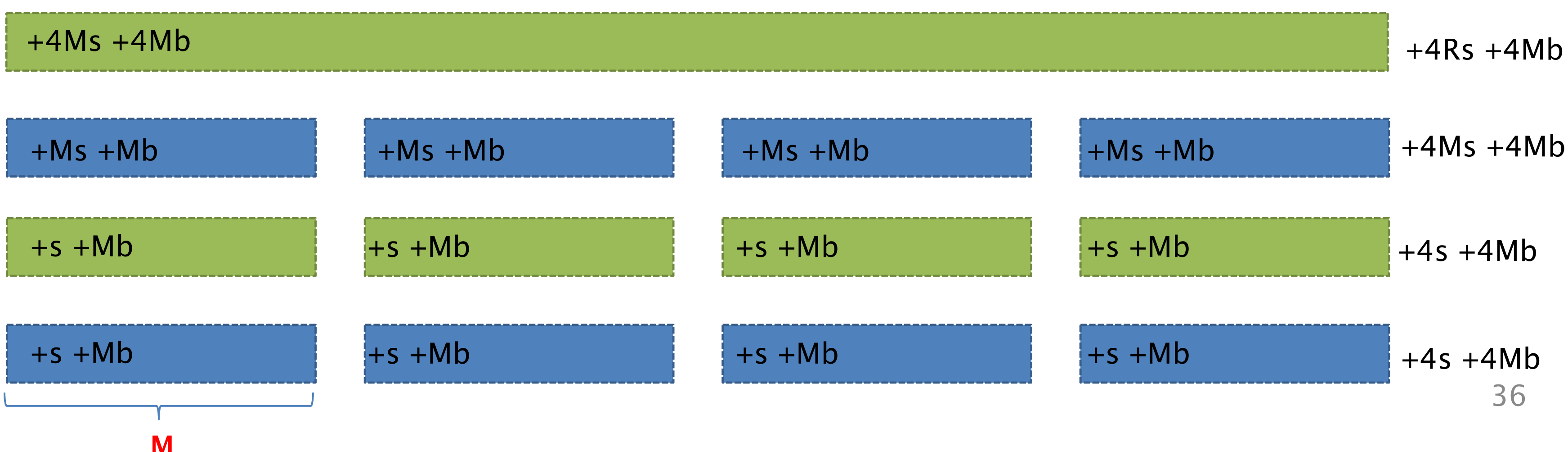
Multi-way Merge: Determine R

- Given
  - M blocks of available memory
  - m: m-way merge
- The depth of merge tree
  - $\log_m n_R$ where n_R is the number of initial runs
  - To reduce n_R, we use M blocks for sorting each initial run and generate M-blocks long initial runs



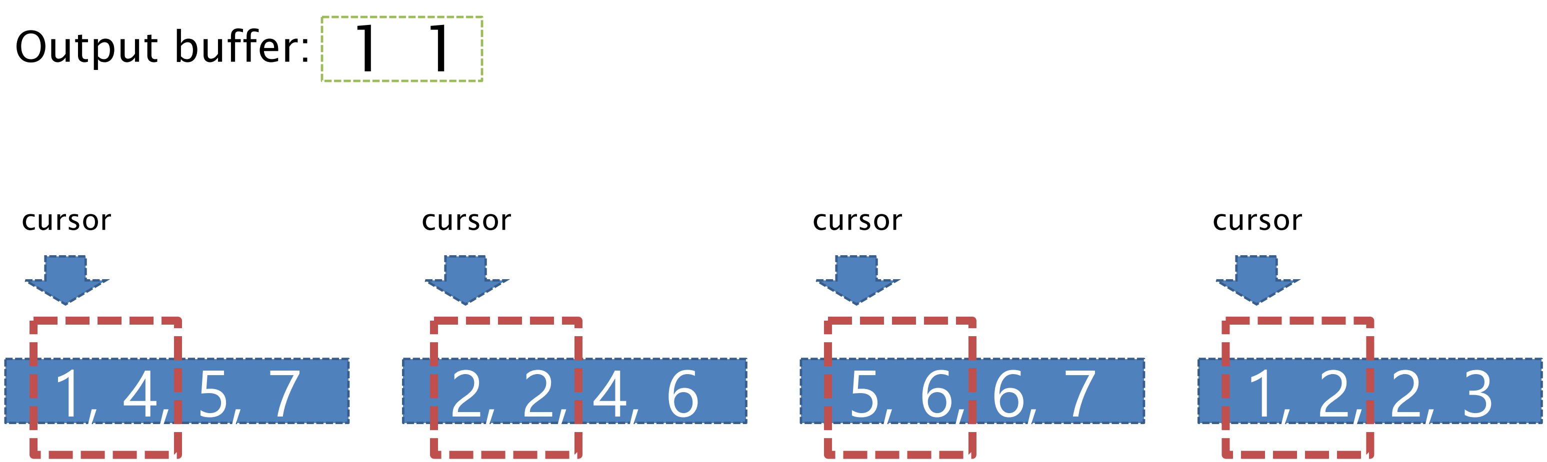
Multi-way Merge: Determine m

- Given
  - M blocks of available memory
  - m: m-way merge
- The depth of merge tree
  - $\log_m n_R$ where n_R is the number of initial runs
  - To increase m, we perform (M-1)-way merge: use 1-blocks for the read buffer for each run



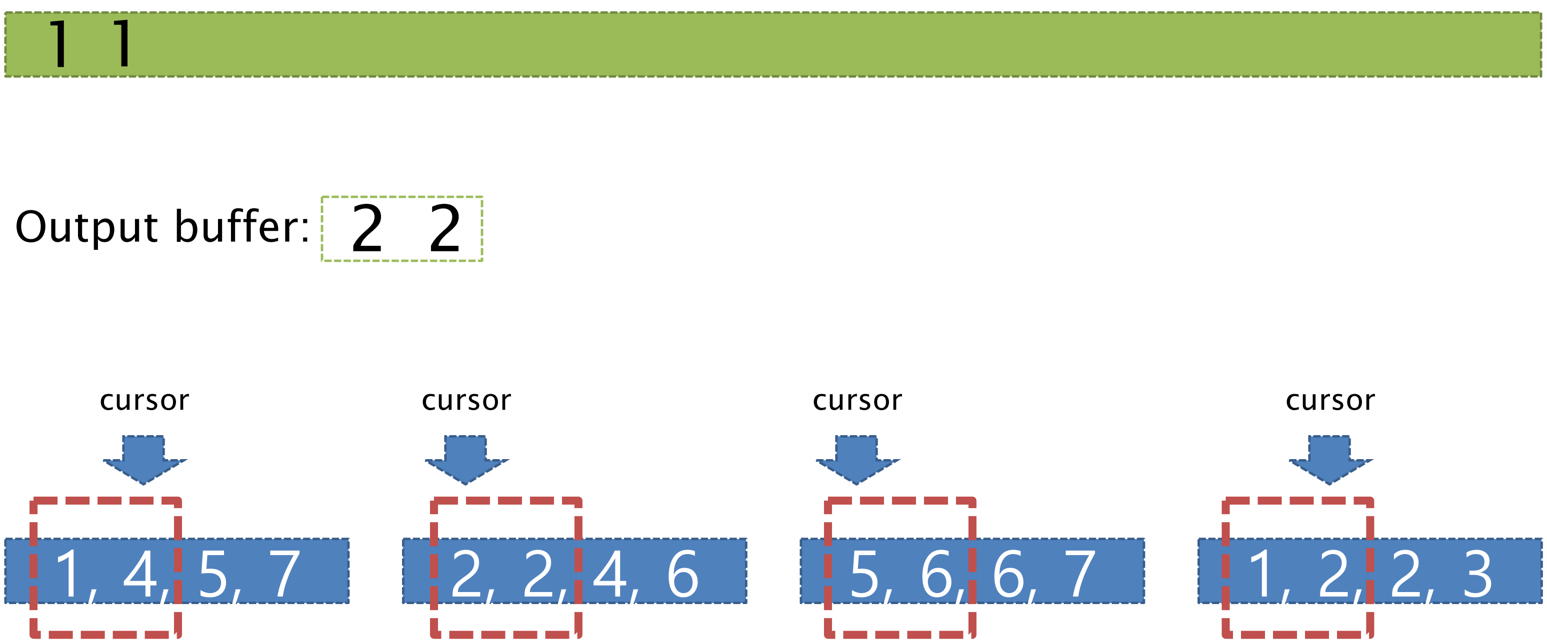
Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer



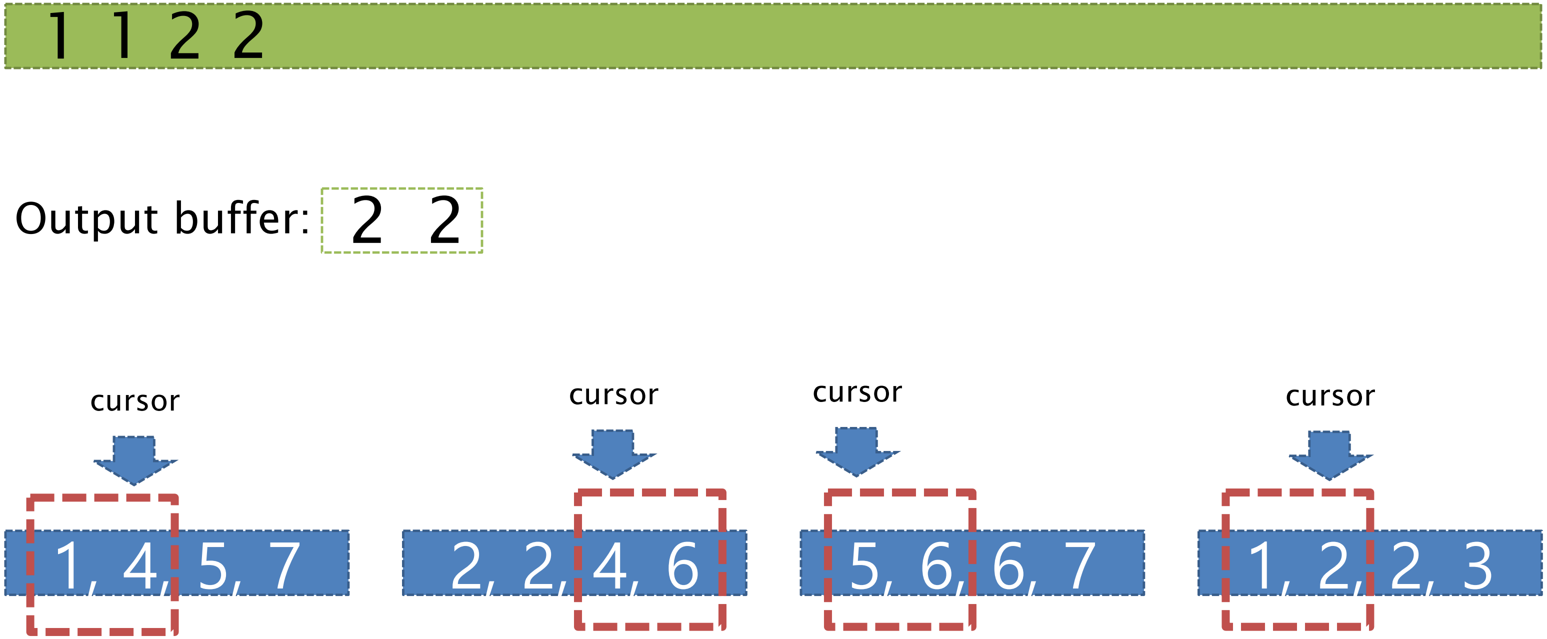
Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer



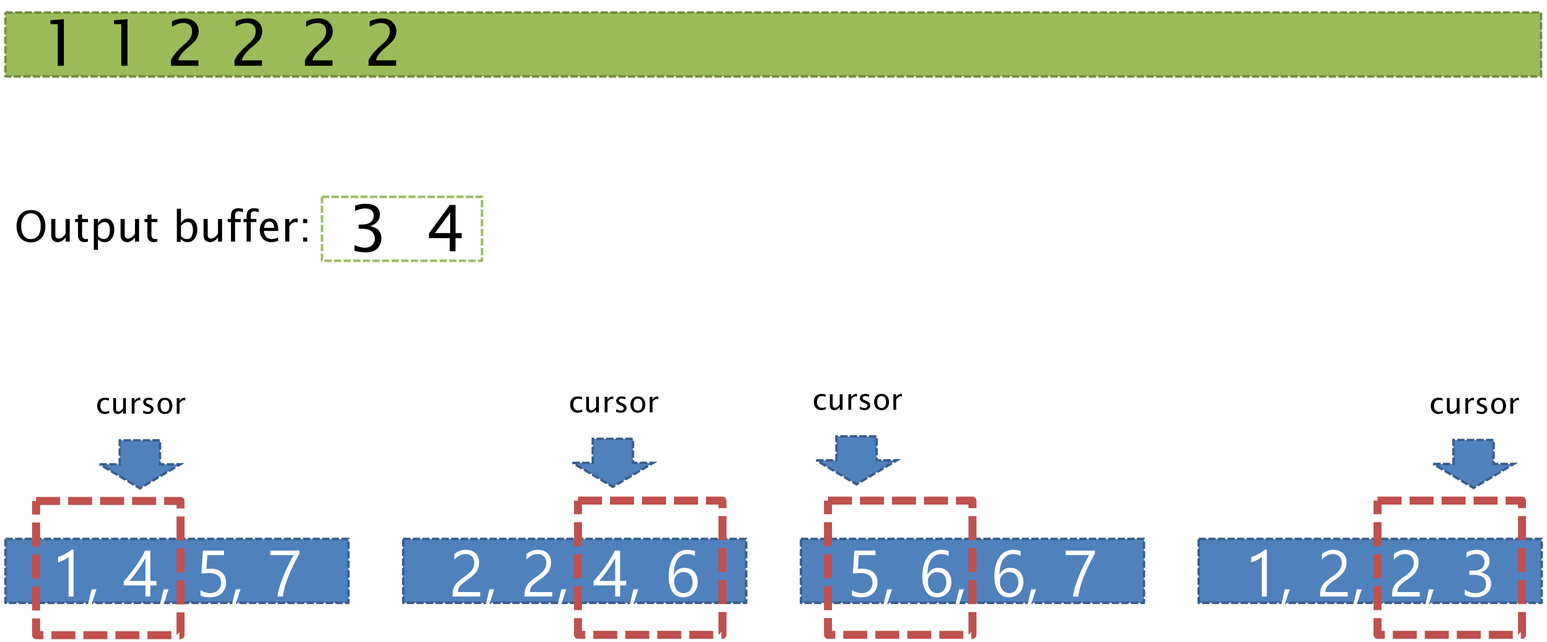
Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer



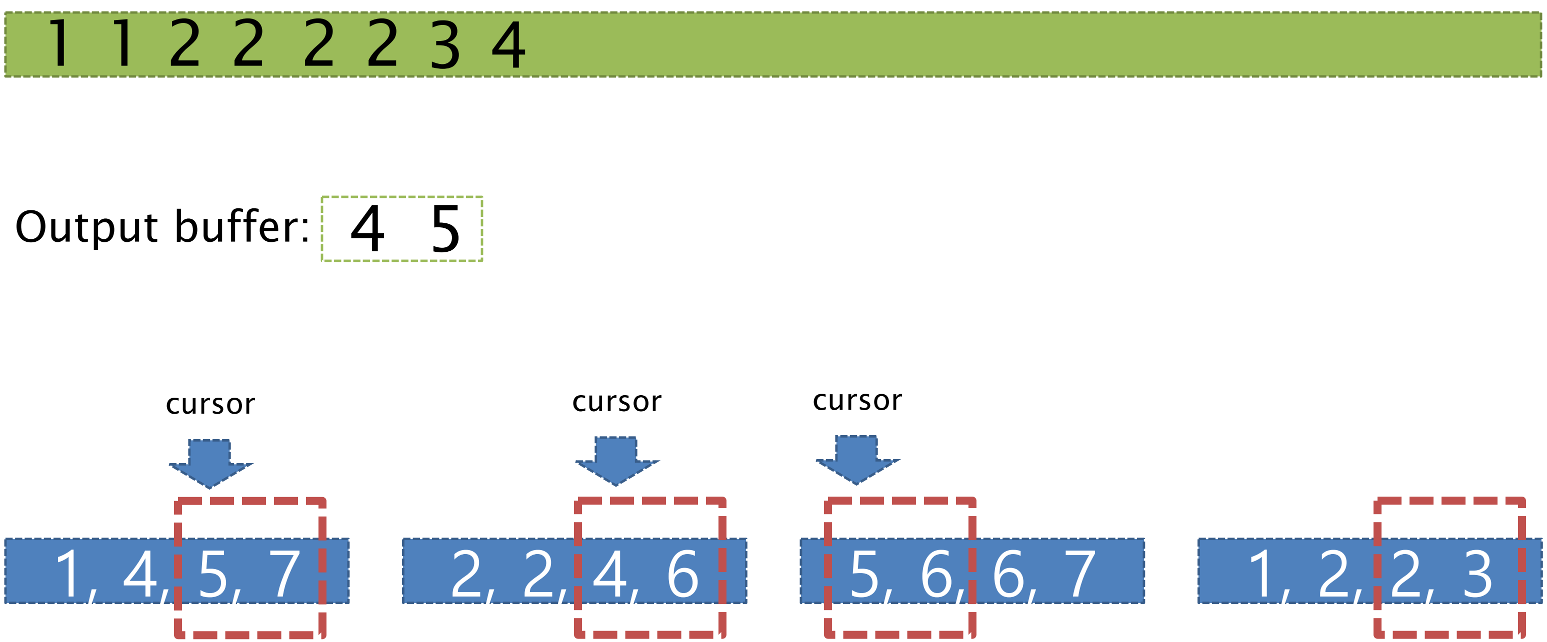
Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer



Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer



Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer

1 1 2 2 2 2 3 4 4 5

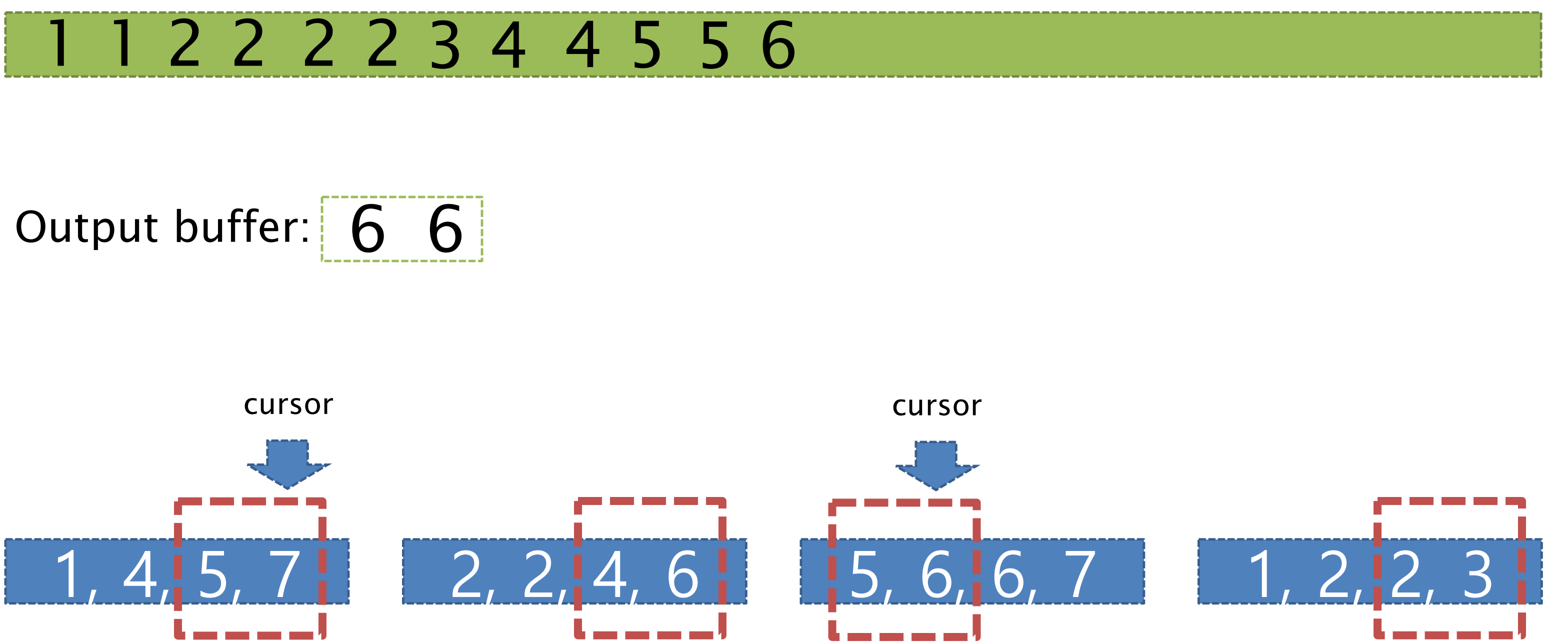
Output buffer: 5 6

cursor cursor cursor

1, 4, 5, 7   2, 2, 4, 6   5, 6, 6, 7   1, 2, 2, 3

Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer

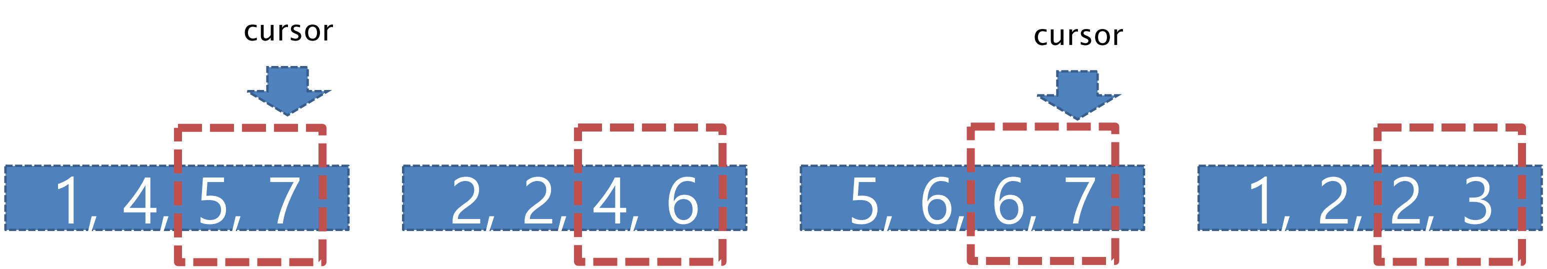


Multi-way Merge

- Access m runs
  - Use a block of memory for read buffer for each run
  - Use a block of memory for output buffer
  - Among the head of m runs, takes the smallest and put into output buffer

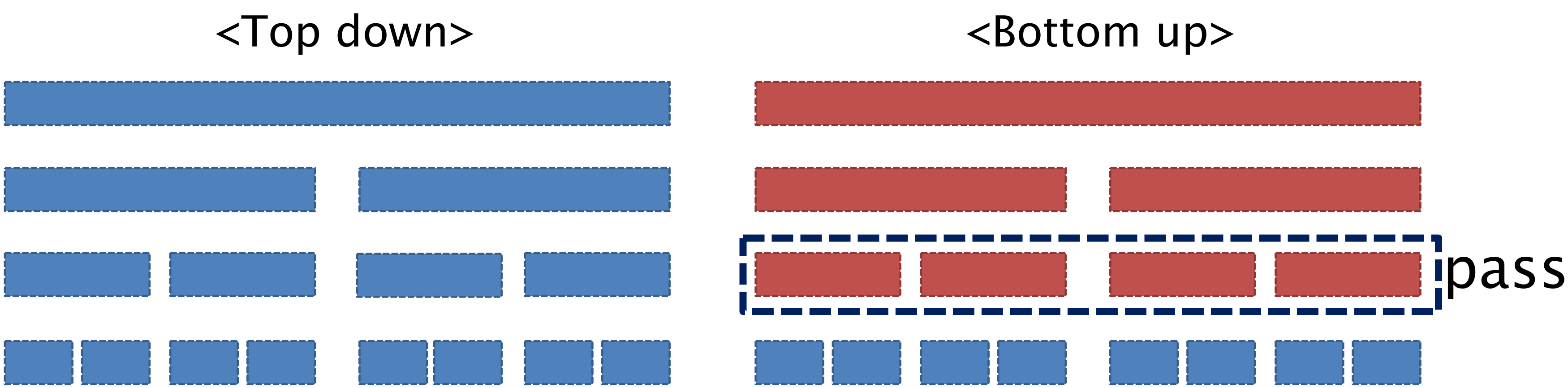
1 1 2 2 2 2 3 4 4 5 5 6 6 6

Output buffer: 7 7



Top Down vs. Bottom Up

- Traditional memory-based binary merge sort algorithm
  - Top down is easier to implement
- External m-way merge sort algorithm
  - Bottom up is better



External Merge-Sort

- Create sorted initial runs. Let i be 0 initially.
  Repeatedly do the following till the end of the data:
  (a) Read **M blocks** of postings into memory
  (b) Sort the in-memory blocks using quick sort.
  (c) Write sorted data to run R_i on disk; increment i.
  Let the final value of i be N
- Merge the runs (next slide)…..

Let *M* denote **AVAILABLE**
Blocks in main memory

External Merge-Sort (Cont.)

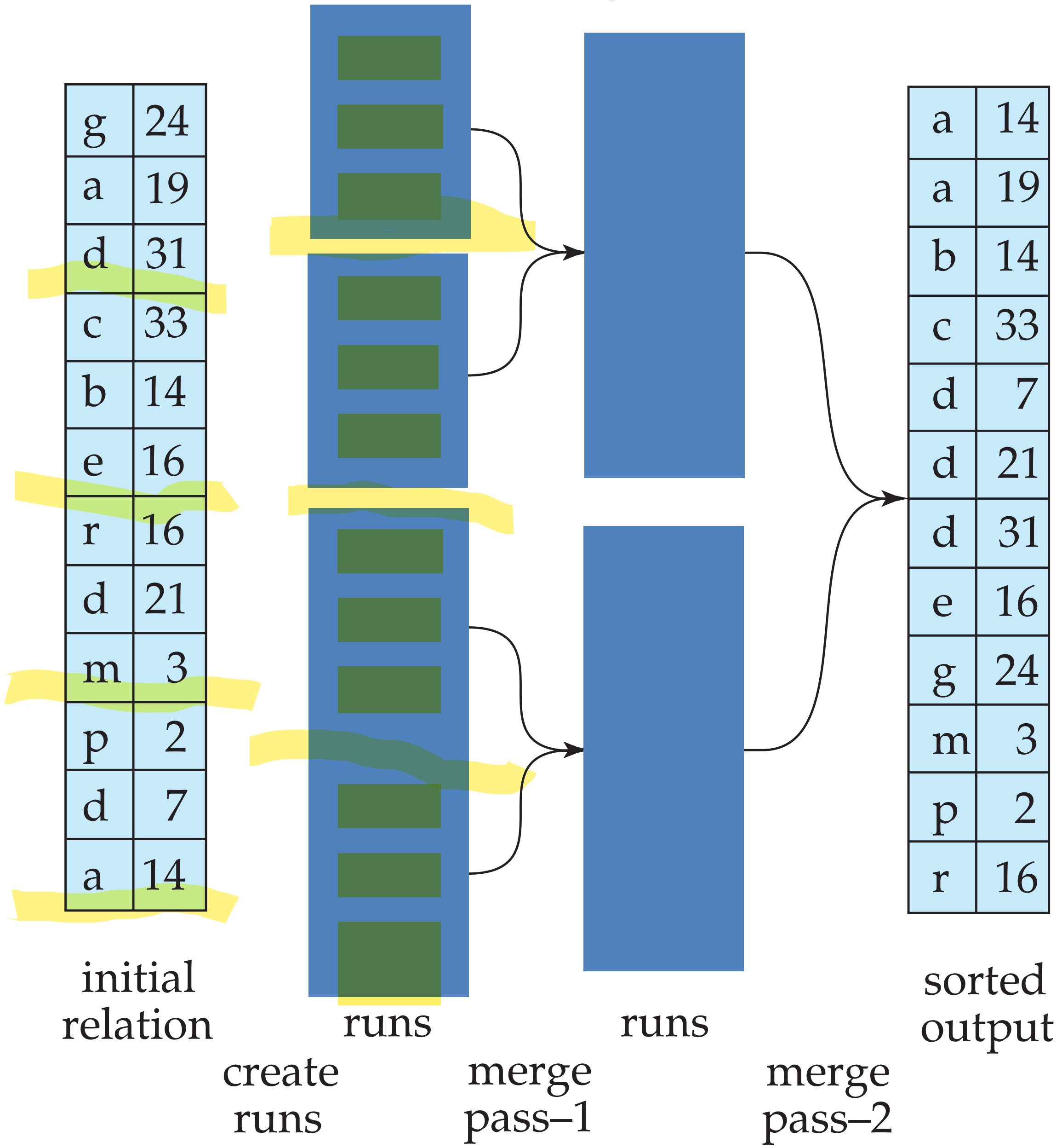
2. **Merge the runs (N-way merge)**. If $N < M$ ($N <= M-1$),
   1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer. If the output buffer is full write it to disk.
      3. **If** the buffer page becomes empty **then** read the next block (if any) of the run into the buffer.
   3. **until** all input buffer pages are empty:

External Merge-Sort (Cont.)

- If $N \geq M$, **several merge *passes*** are required.
  - In each **pass**, contiguous groups of M - 1 runs are merged using the previous procedure.
  - A pass reduces the number of runs by a factor of M -1 and creates runs longer by the same factor.
    - E.g. If there are 90 runs and M=11, one pass reduces the number of runs to 9 and each generated run is 10 times as long as the initial run
  - Repeated passes are performed till all runs have been merged into one.

Example:

M=3


g 24
a 19
d 31
c 33
b 14
e 16
r 16
d 21
m 3
p 2
d 7
a 14
initial relation
runs
runs
a 14
a 19
b 14
c 33
d 7
d 21
d 31
e 16
g 24
m 3
p 2
r 16
sorted output
create runs
merge pass–1
merge pass–2

Cost Estimation

- Cost analysis:
  - **M** blocks(=pages) are available in main memory
  - Postings on $\mathbf{b_r}$ blocks ➔ $\left\lceil \frac{b_r}{M} \right\rceil$ runs are generated!
  - Total number of **merge passes** required: $\left\lceil \log_{M-1} \left\lceil \frac{b_r}{M} \right\rceil \right\rceil$
  - COST: Block transfers (read/write) for <u>initial run creation</u> as well as in each pass is $\mathbf{2}\boldsymbol{b_r}$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - Thus, total number of block transfers for external sorting:

$$2\mathrm{b_r} + 2\mathrm{b_r}\left(\left\lceil \log_{M-1} \left\lceil \frac{b_r}{M} \right\rceil \right\rceil\right) - b_r$$

  - Seeks: next slide

Cost Estimation (Cont.)

- Cost of seeks
  - **During run generation:** one seek to read each run and one seek to write each run
    - $2\left\lceil \frac{b_r}{M} \right\rceil$ seeks
  - **During the merge phase**
    - Need ***$2b_r$* seeks** for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:

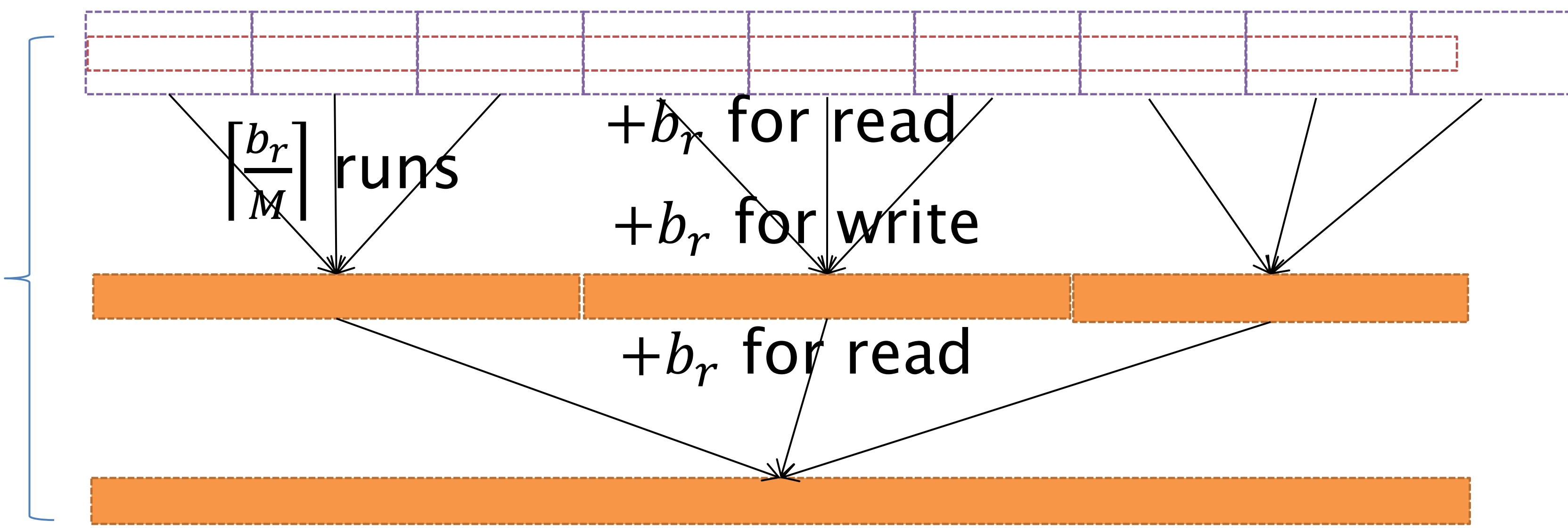
$$2\left\lceil \frac{b_r}{M} \right\rceil + 2b_r \left\lceil \log_{M-1} \left\lceil \frac{b_r}{M} \right\rceil \right\rceil - b_r$$

Cost Estimation: Block Transfers

- Cost analysis:

M blocks

$2b_r$ for initial run generation



$\left\lceil \log_{M-1} \left\lceil \frac{b_r}{M} \right\rceil \right\rceil$ passes

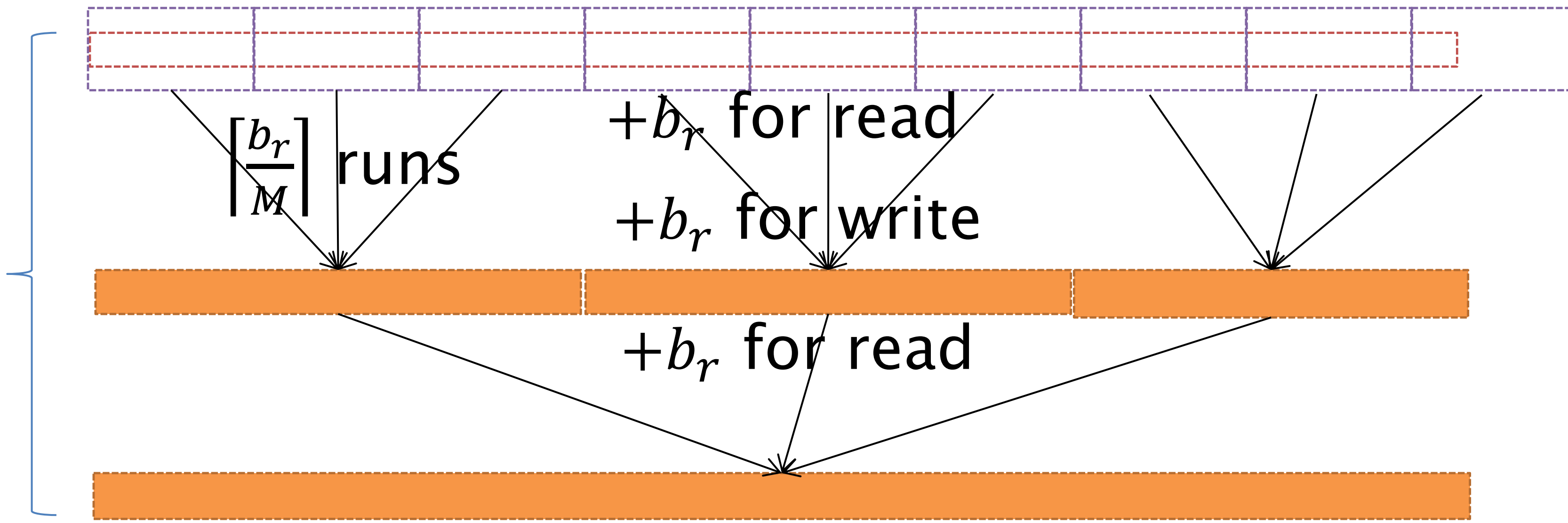
$\Rightarrow \; 2\mathrm{b_r} + 2\mathrm{b_r}\left(\left\lceil \log_{M-1} \left\lceil \frac{b_r}{M} \right\rceil \right\rceil\right) - b_r$

Cost Estimation: Disk Seeks

- Cost analysis:

M blocks

$2\left\lceil \frac{b_r}{M} \right\rceil$ for initial run generation



$\left\lceil \log_{M-1} \left\lceil \frac{b_r}{M} \right\rceil \right\rceil$ passes

$$\Rightarrow 2\left\lceil \frac{b_r}{M} \right\rceil + 2\mathrm{b_r}\left(\left\lceil \log_{M-1} \left\lceil \frac{b_r}{M} \right\rceil \right\rceil\right) - b_r$$

Exercise:

M=5

Output buffer

4

13	4	4		4
4	13	6		6
35	35	8		8
49	43	13		10
43	49	29		11
46	29	35		13
29	46	43		29
85	83	43		30
88	85	46		35
83	88	46		43
8	8	49		43
85	50	50		46
74	46	74		46
46	74	79		47
50	85	83		49
6	6	84		50
79	43	85		65
91	79	85		74
43	84	88		79
84	91	91	10	83
65	10		11	83
30	11		30	84
10	30		47	85
47	47		65	85
11	65		83	88
83	83			91

Exercise:

13	4
4	13
35	29
49	35
43	43
46	46
29	49
85	83
88	85
83	88
8	6
85	8
74	43
46	46
50	50
6	74
79	79
91	84
43	85
84	91
65	10
30	11
10	30
47	47
11	65
83	83

Output buffer

M=5 and 2 records per bloc

$\rightarrow \frac{26}{10} = 3$

4way 가능

1번만에 가능

Exercise

M=3 and 1 records per block

86
60
90
59
48
19
86
94
23
36
48
7
9
63
74
29
39
20

Quiz

- Sort the postings using external merge-sort algorithm
  - 30 postings
  - 3 blocks of memory are available
  - Each block(a page) can hold up to 3 postings
  - A block is used for the read buffer of each run
- **How many blocks to read/write?**
- **How many disk seeks occurs?**

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2
was	2

Distributed Indexing

- External merge-sort is the best choice using a single computer
- Nonetheless, web-scale indexing must use a distributed computing cluster
- How do we exploit such a pool of machines for indexing? ➔ MapReduce