# Database Management System

Lecture 8

Join

* Some materials adapted from R. Ramakrishnan, J. Gehrke and Shawn Bowers

# Today's Agenda

- Join Algorithm

# Join Algorithms

# Join Algorithms

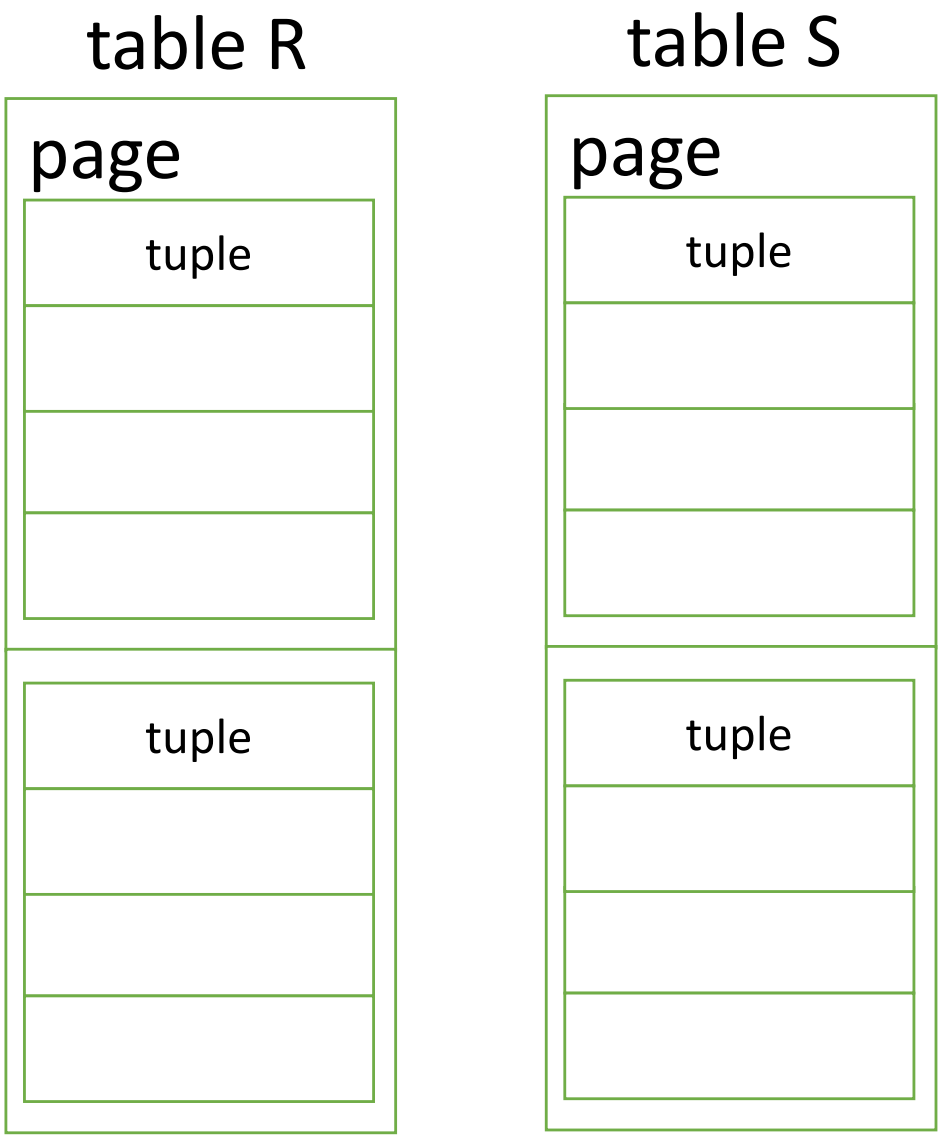
```
SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
```

Sailors(sid, snam, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- R ⋈ S is very common
  - R X S followed by a selection is inefficient ... why?
  - So we process joins (rather than cross product) when possible
  - Much effort in query processing invested in join algorithms

# Notations

- $M$ -- pages in R
- $P_R$ -- tuples per page
- $N$ -- pages in S
- $P_S$ -- tuples per page



# Join Algorithms

- Simple nested loops Join

  Join on i-th column of R and j-th column of S

  1. foreach tuple r in R do
  2. foreach tuple s in S do
  3. if $r_i$ == $s_j$ then add <r, s> to result

For R ⋈ S ...

- We call R the "*outer*" relation
- We call S the "*inner*" relation

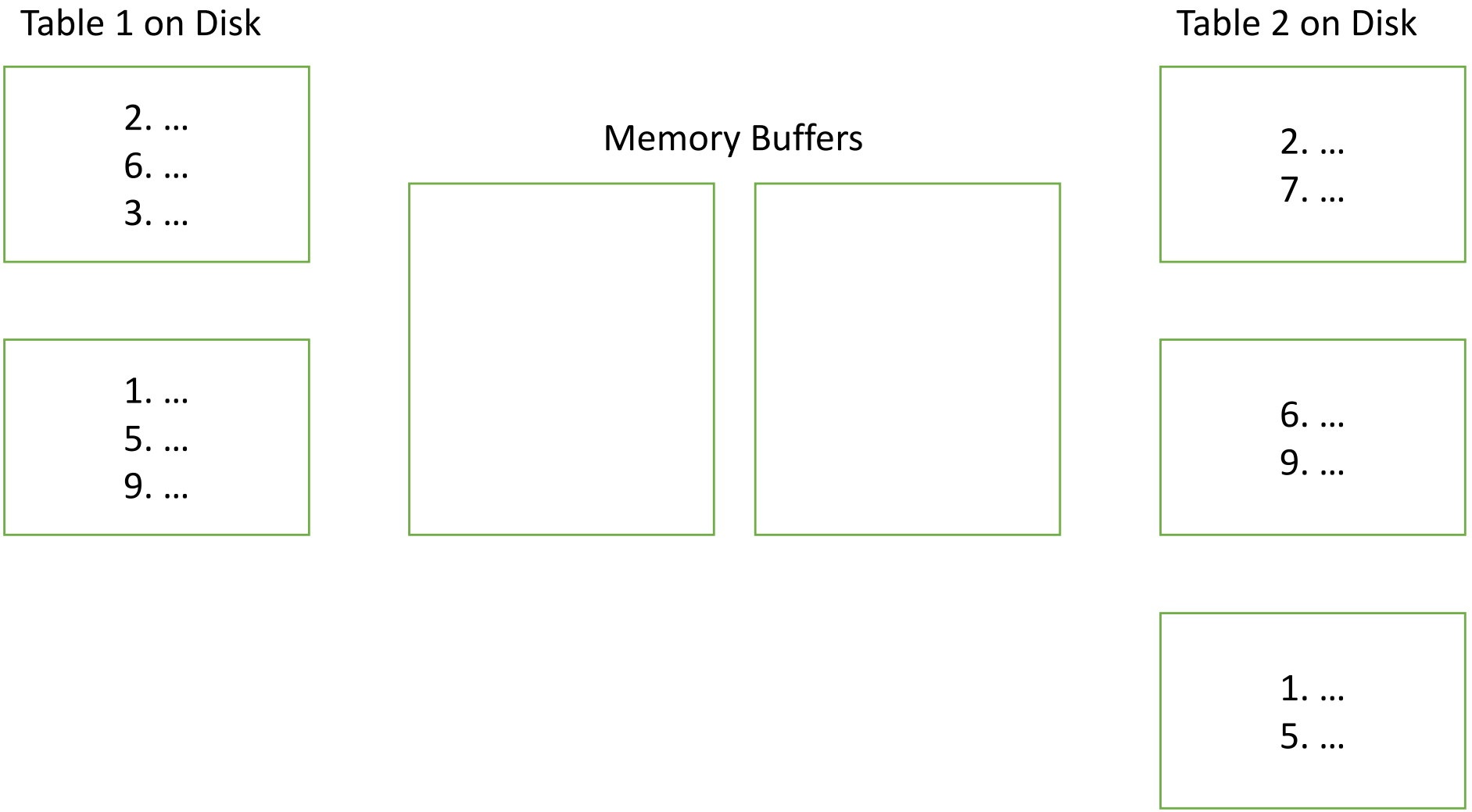
# Join Algorithms

- For each tuple in the outer relation R, we *scan* the entire inner relation S tuple-by-tuple ...
  - If $M$ = 1000 pages in R, $P_R$ = 100 tuples/page
  - If $N$ = 500 Pages in S, $P_S$ = 80 tuples/page
  - If 100 I/Os per second
  - Cost R ⋈ S = $M + (P_R * M) * N$ = 1000 + 100*1000*500 I/Os
  - 50,001,000 I/Os ≈ 500,010 seconds ≈ **6 days**!

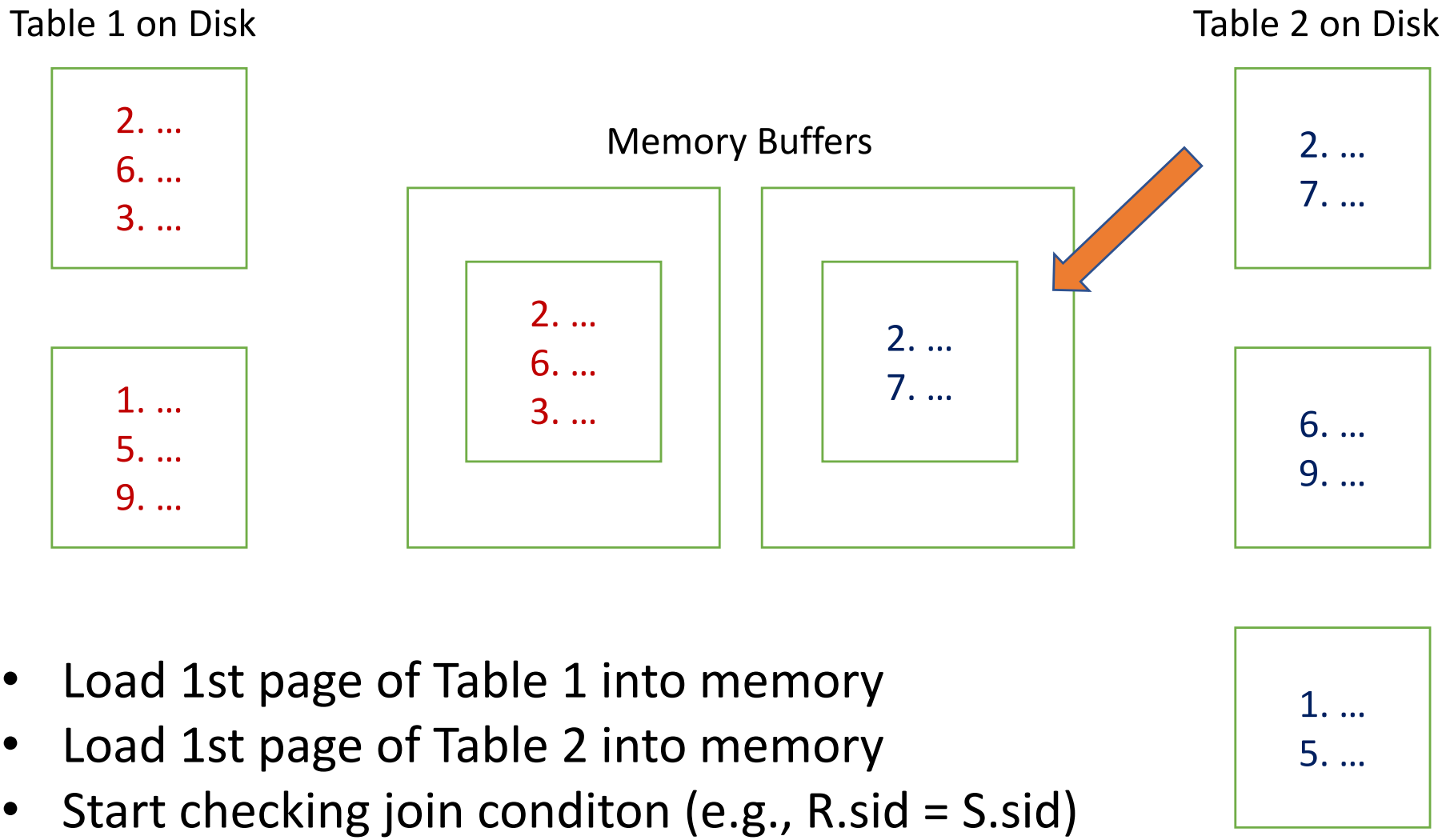
# Simple Nested Loops Join

- This example highlights
  - Simple nested loop join isn't very practical
  - We need algorithms that optimize joins
- There are also the other operations to consider ...

# Simple Nested Loops Join

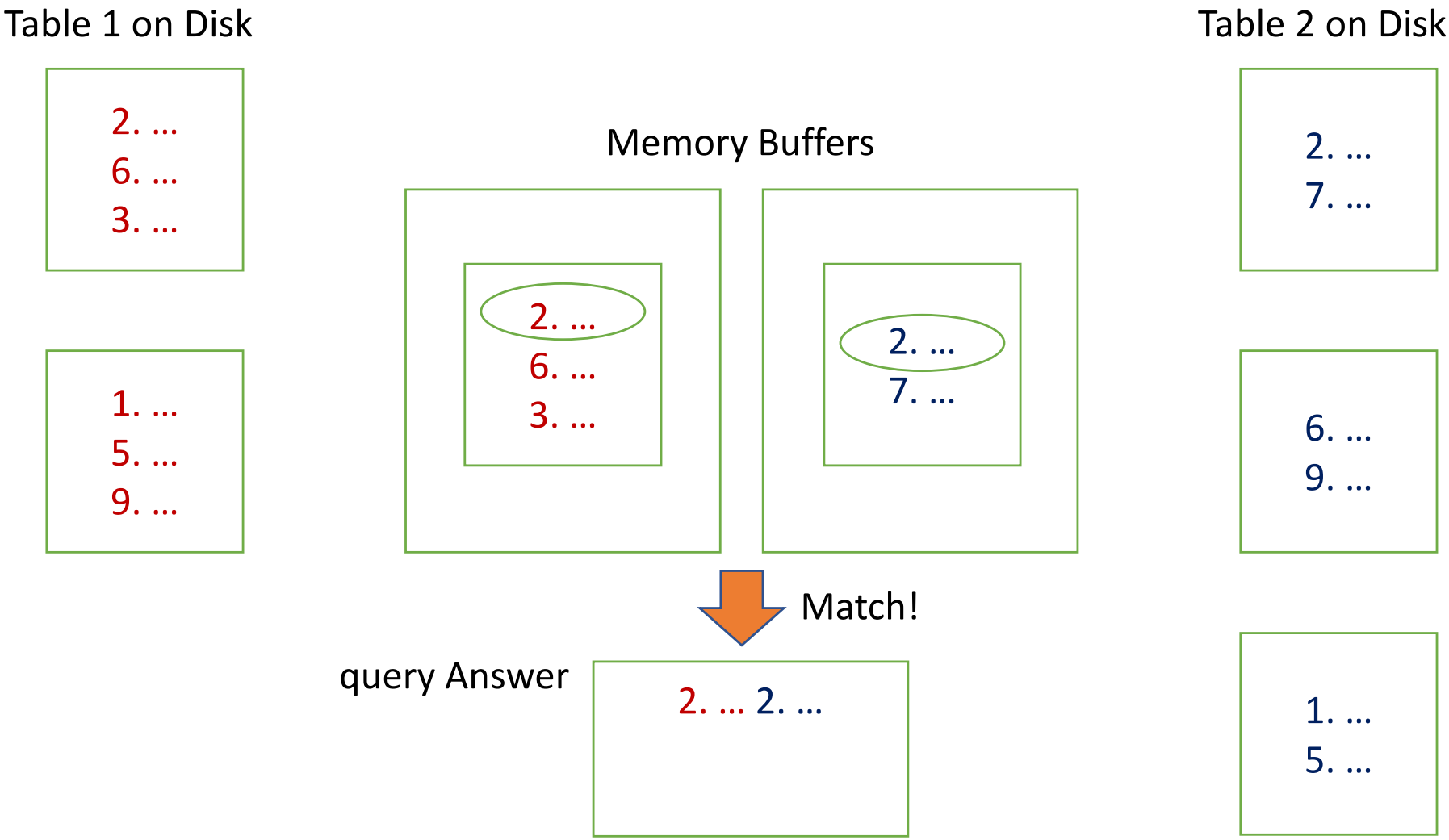


# Simple Nested Loops Join

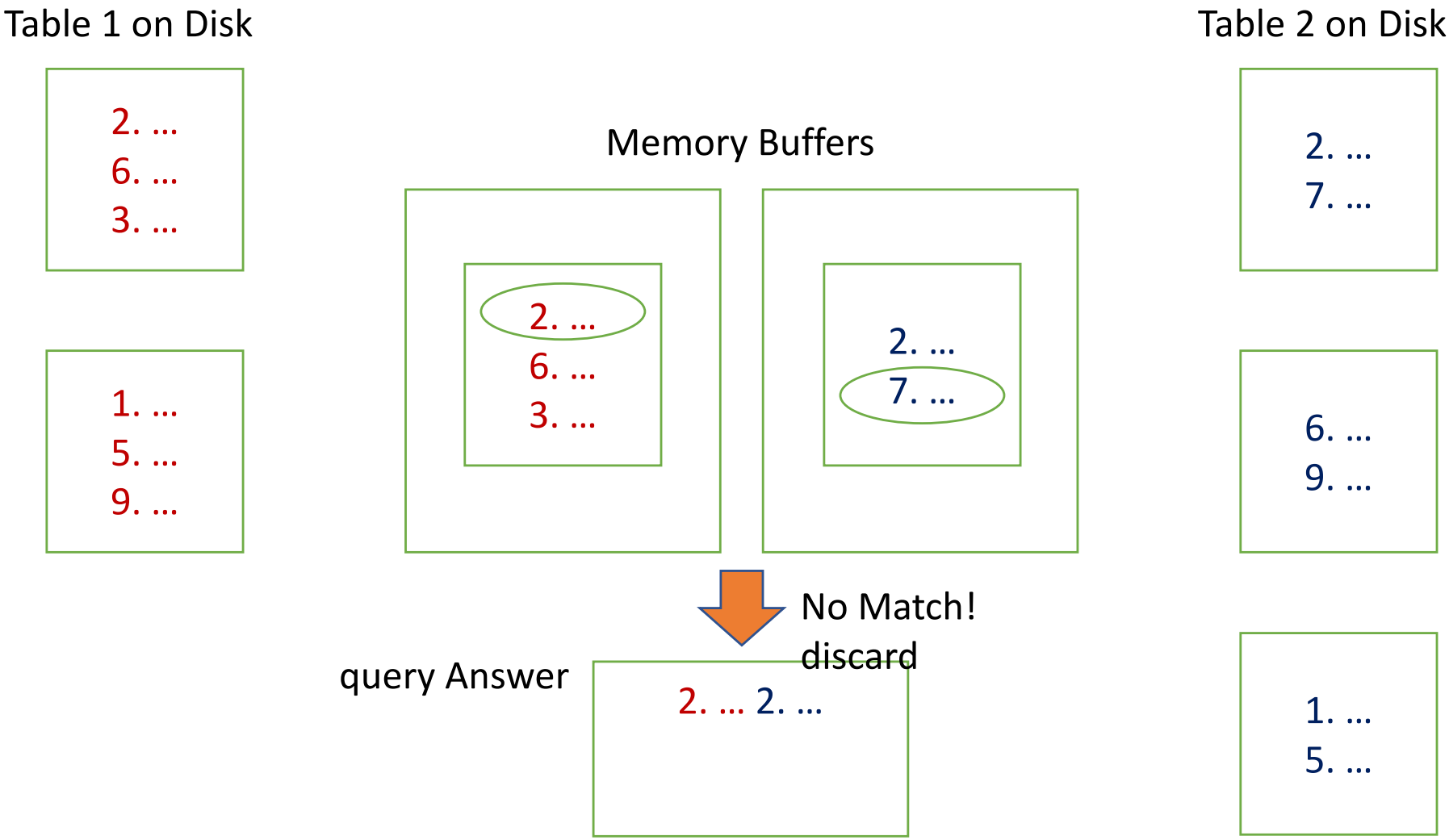



- Load 1st page of Table 1 into memory
- Load 1st page of Table 2 into memory
- Start checking join conditon (e.g., R.sid = S.sid)

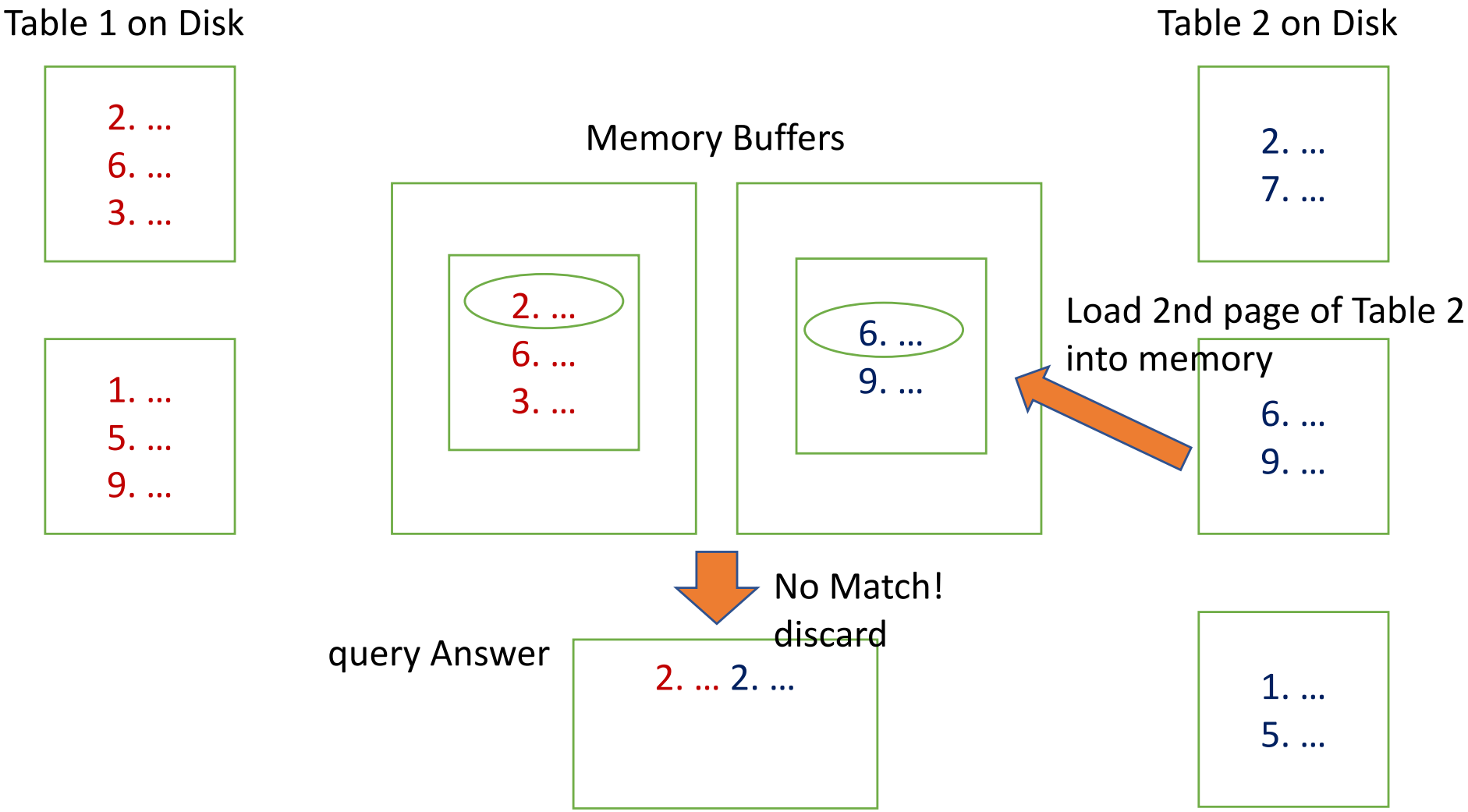
# Simple Nested Loops Join



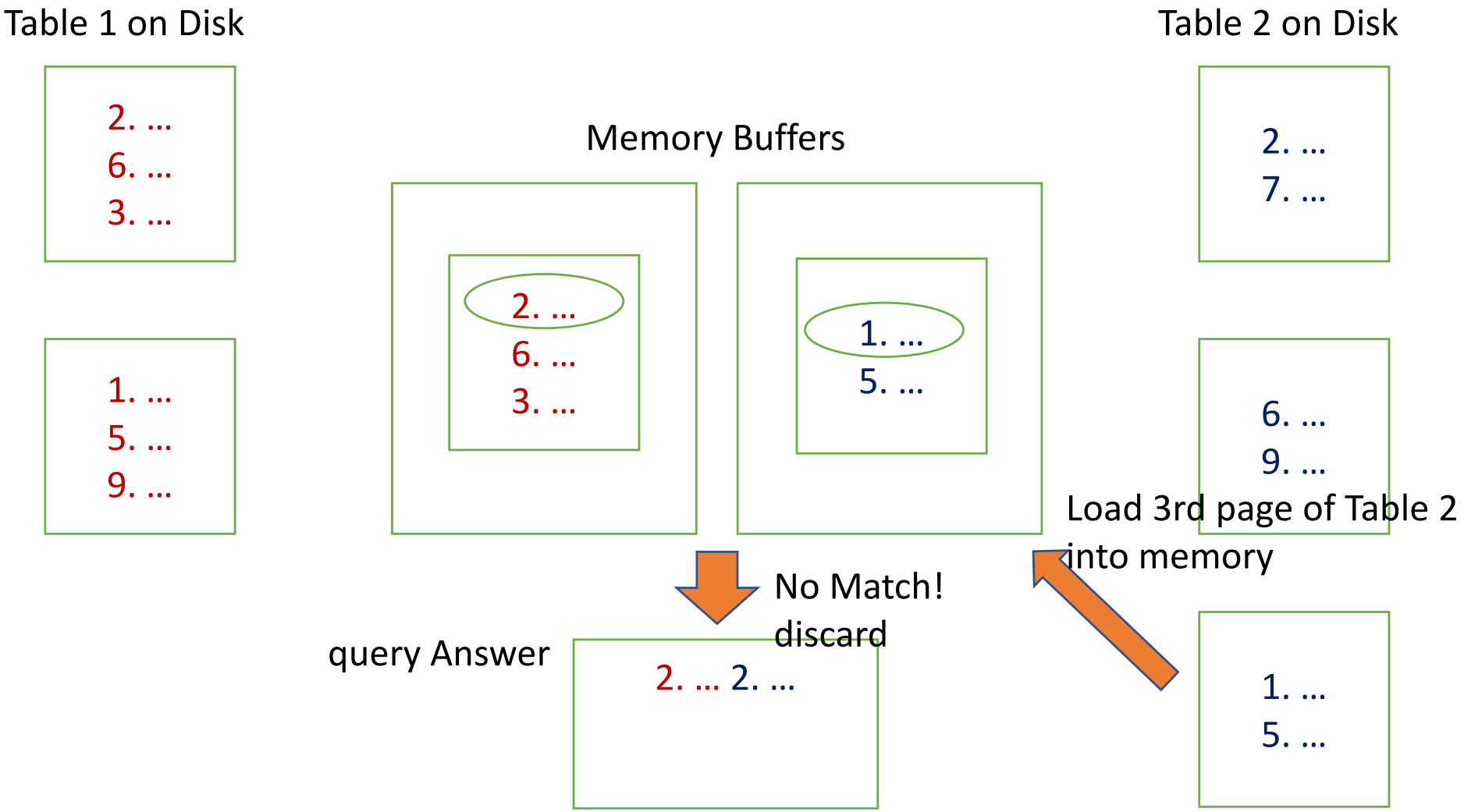
# Simple Nested Loops Join



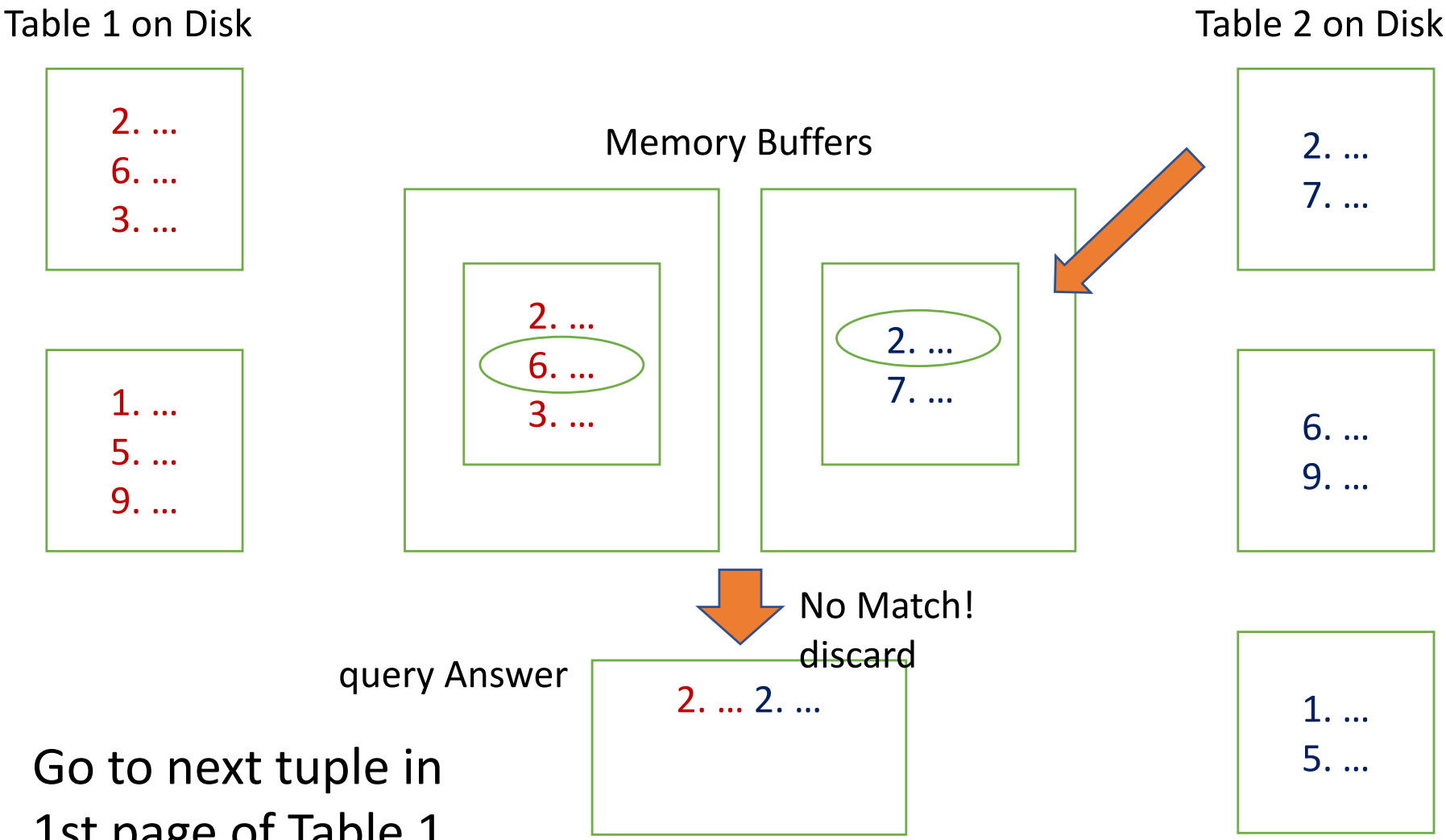
# Simple Nested Loops Join



# Simple Nested Loops Join

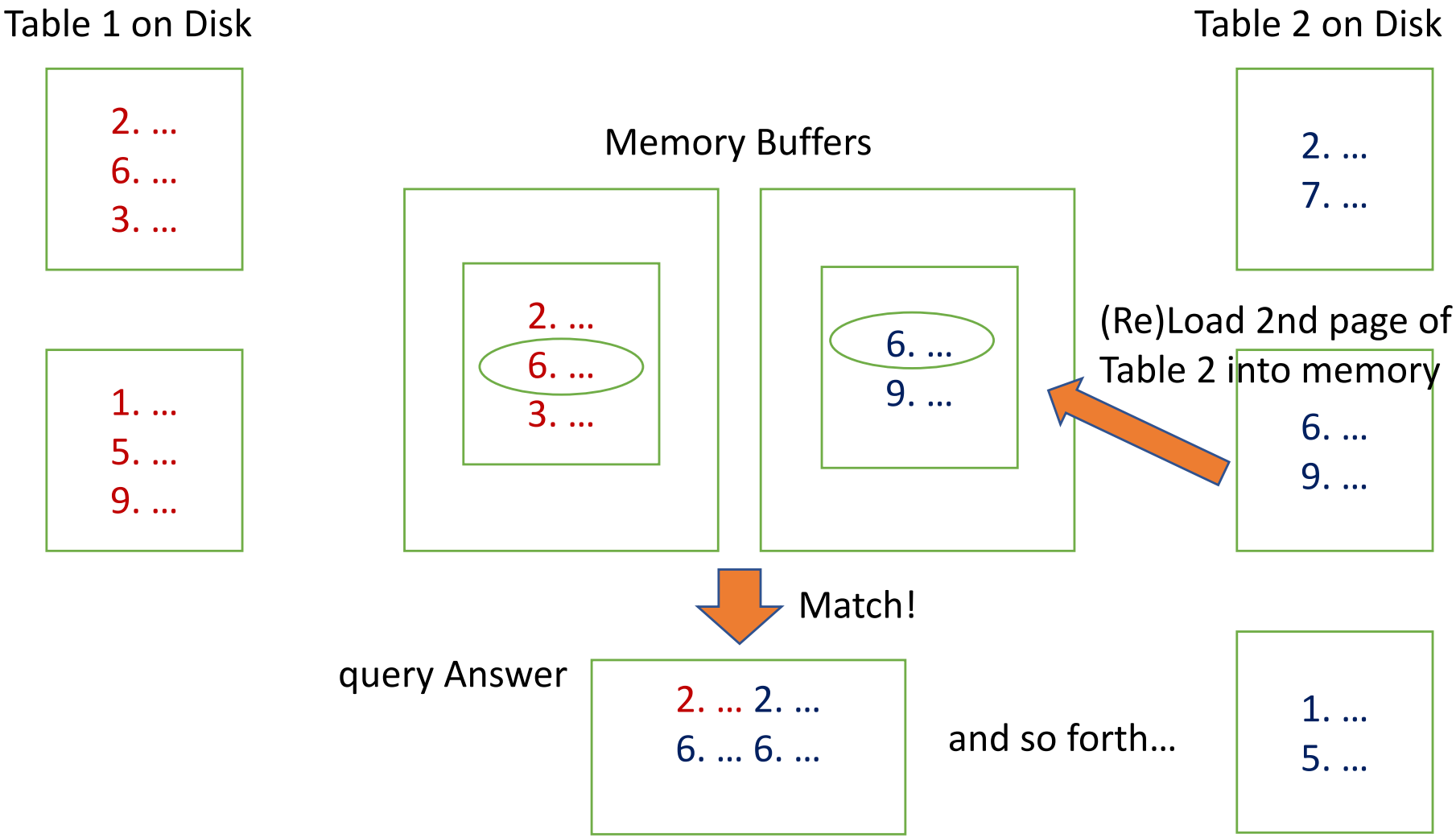


# Simple Nested Loops Join

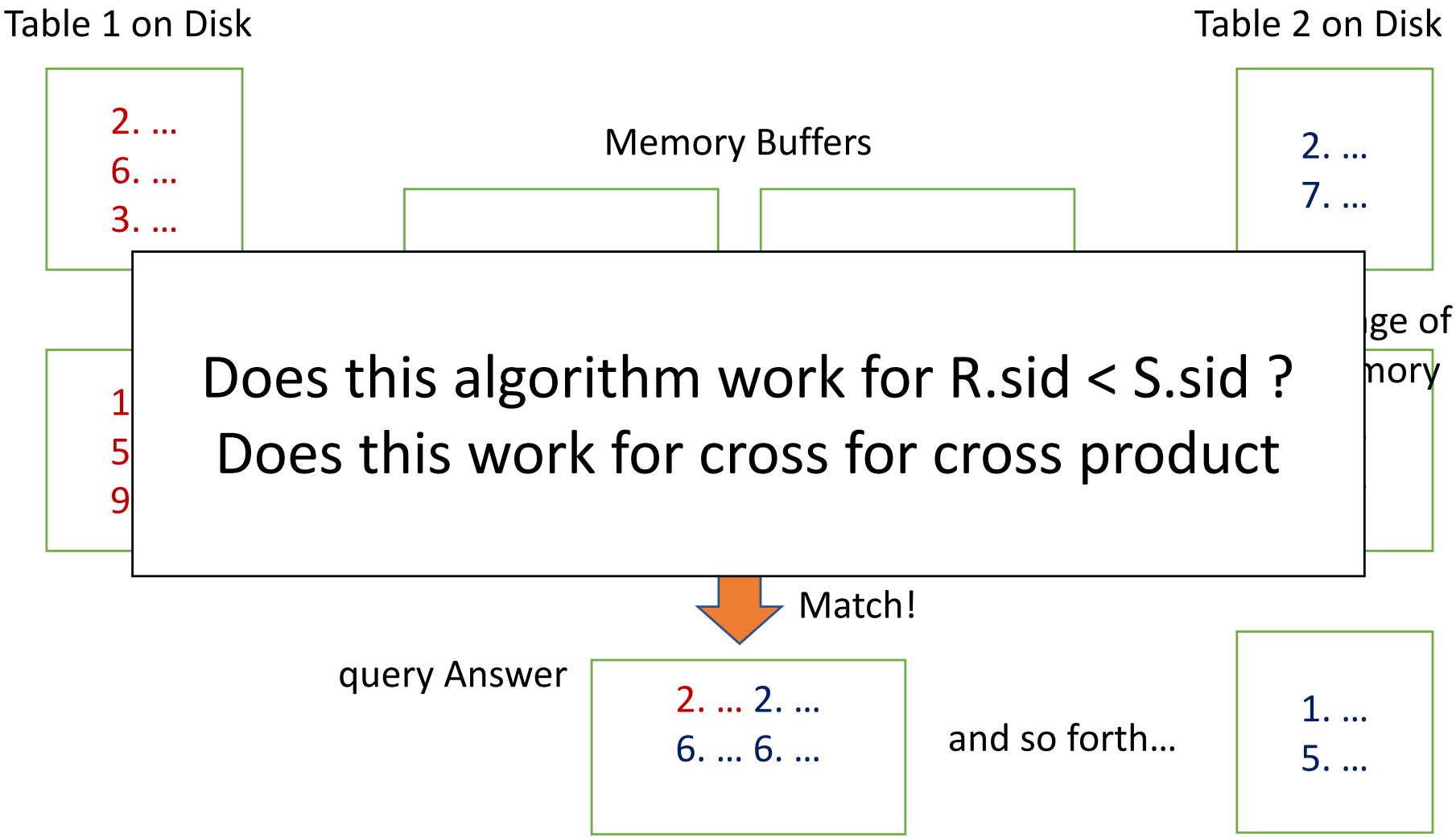



- Go to next tuple in 1st page of Table 1
- (Re)Load 1st page of Table2 into memory

# Simple Nested Loops Join



# Simple Nested Loops Join



# Join Algorithms (Revisited)

- simple nested loops join

  Join on i-th column of R and j-th column of S

  1. foreach tuple r in R do
  2. foreach tuple s in S do
  3. if $r_i == s_j$ then add <r, s> to result

- For each tuple in the outer relation R, we *scan* the entire inner relation S tuple-by-tuple ...
  - If $M$ = 1000 pages in R, $P_R$ = 100 tuples/page
  - If $N$ = 500 Pages in S, $P_S$ = 80 tuples/page
  - If 100 I/Os per second
  - Cost R ⋈ S = $M + (P_R * M) * N$ = 1000 + 100*1000*500 I/Os
  - 50,001,000 I/Os ≈ 500,010 seconds ≈ **6 days**!

# Join Algorithms (Revisited)

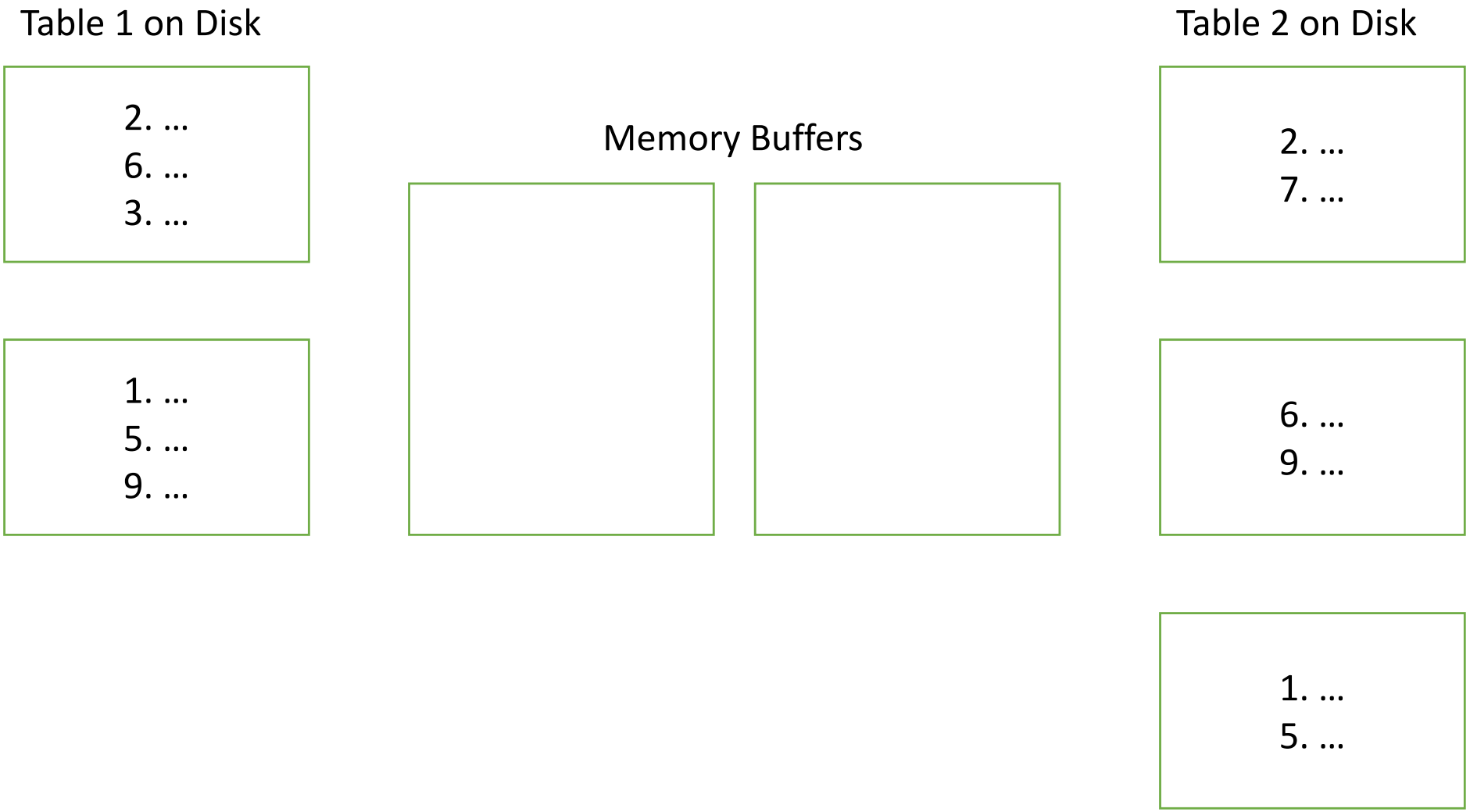
- "page-oriented" nested loops join

  Join on i-th column of R and j-th column of S

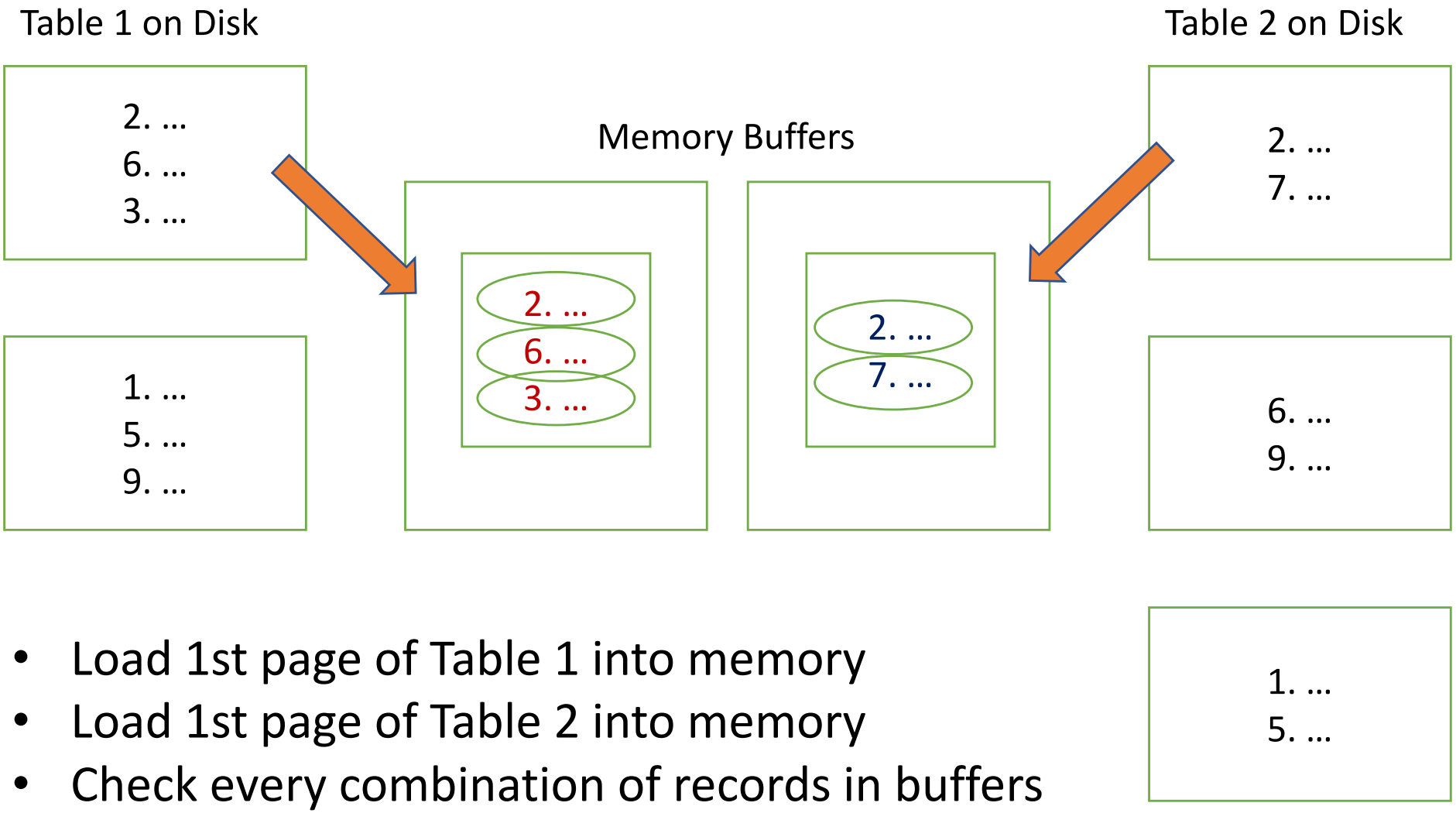
  1. foreach page of tuples in R do
  2. foreach page of tuples in S do
  3. foreach record r and s in memory
  4. if $r_i == s_j$ then add <r, s> to result

- For each *page* in R, get each *page* in S ...
  - If *M* = 1000 pages in R,*N* = 500 Pages in S,and 100 I/Os per sec.
  - Cost R ⋈ S = *M* + *M* * *N* = 1000 + 1000*500 = 501,000 I/Os
  - Cost S ⋈ R = *N* + *N* * *M* = 500 + 500*1000 = 500,500 I/Os
  - Thus, we typically use smaller relation as outer relation
  - 500,500 I/Os ≈ **1.4 hours**

# Page-Oriented Nested Loops Join

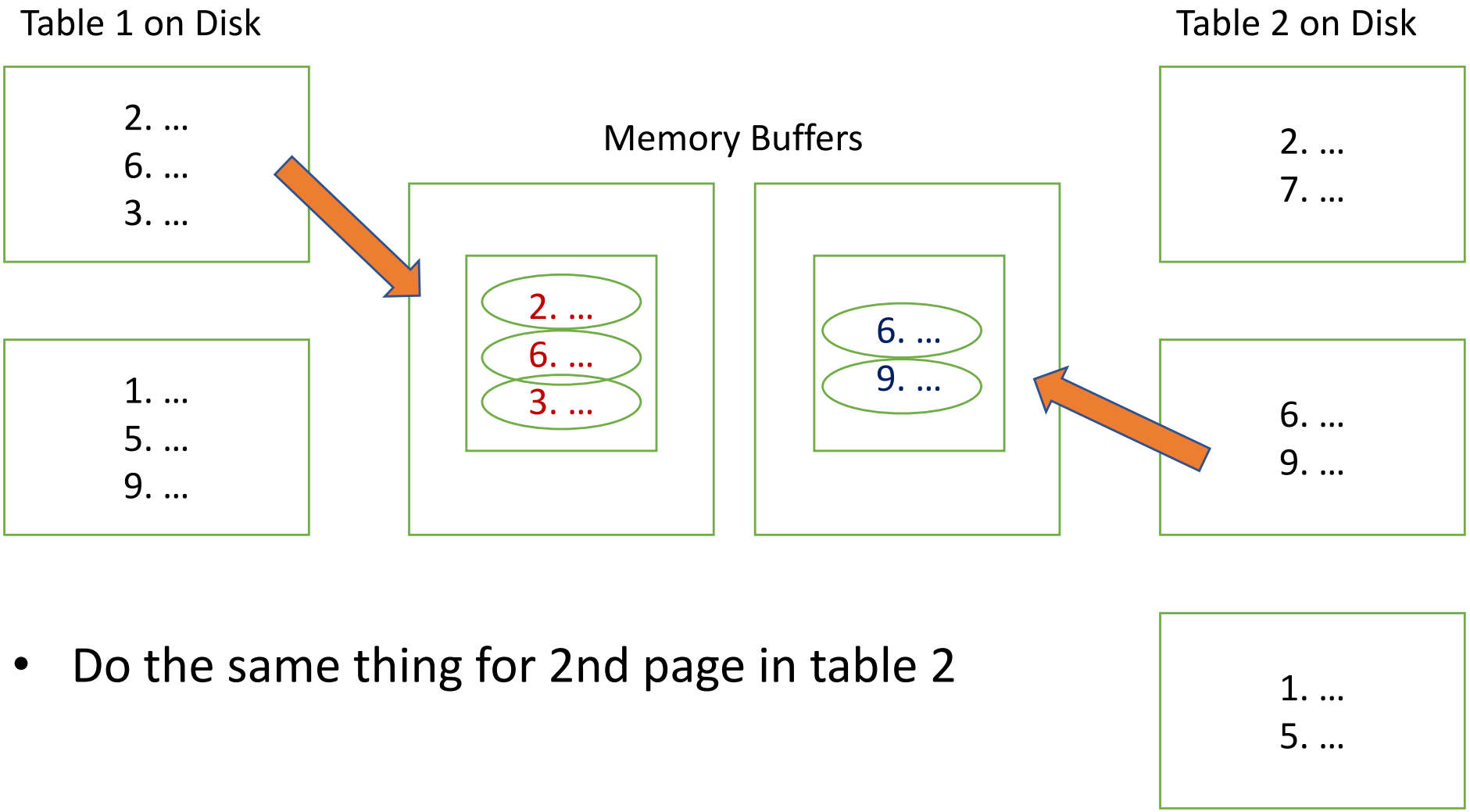


# Page-Oriented Nested Loops Join



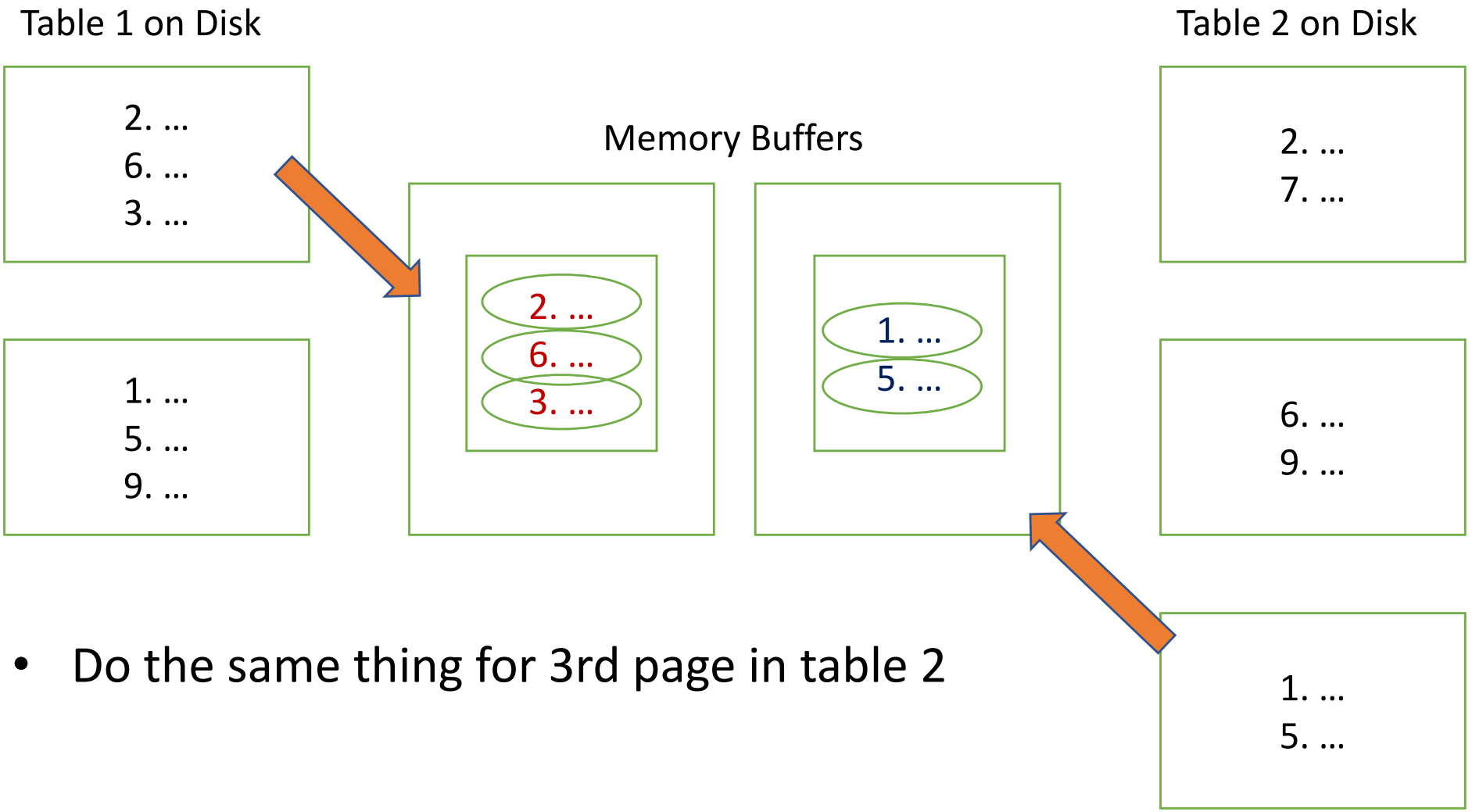

- Load 1st page of Table 1 into memory
- Load 1st page of Table 2 into memory
- Check every combination of records in buffers

# Page-Oriented Nested Loops Join



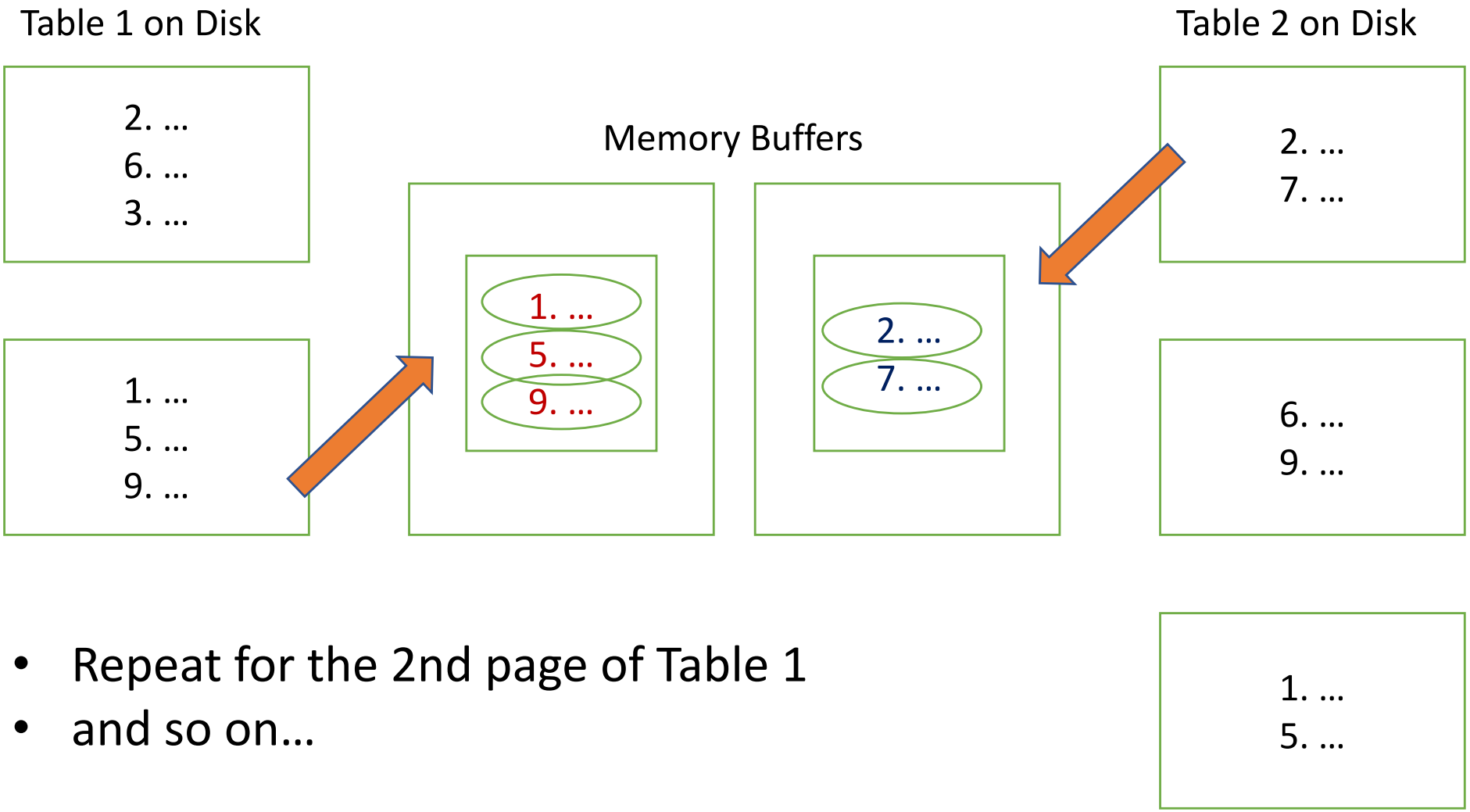

- Do the same thing for 2nd page in table 2

# Page-Oriented Nested Loops Join




- Do the same thing for 3rd page in table 2

# Page-Oriented Nested Loops Join




- Repeat for the 2nd page of Table 1
- and so on…

# Another Alternative Algorithm: Use Buffer

- ”Block” nested loops join

  Join on i-th column of R and j-th column of S

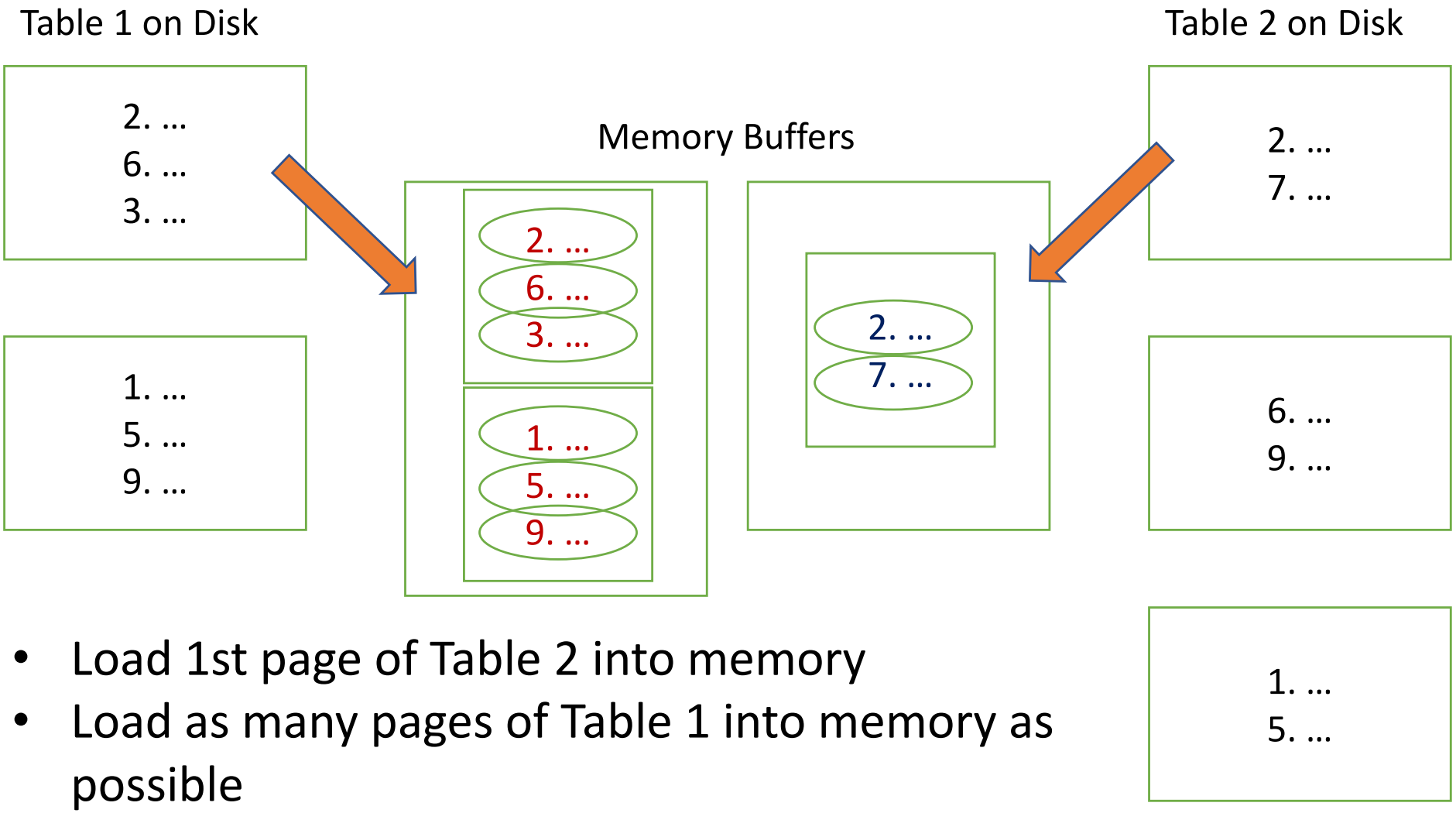
  1. Assume B pages of memory in buffer
  2. Assign one page of memory in buffer to output
  3. Load B-2 pages of tuples from R
  4. Load 1 page of tuples from S
  5. foreach record r and s in memory
  8. if ri == sj then add <r, s> to result

- For multiple *pages* in R, get each *page* in S ... check all pairs and output –

  If $M$ = 1000 pages in R,$N$ = 500 Pages in S,$B$ = 35,and 100 I/Os per sec.

  - Cost R ⋈ S = $M + (M / (B - 2)) * N$ = 1000 + (1000/33)*500 ≈ 16,000 I/Os
  - Cost S ⋈ R = $N + N * M$ = 500 + (500/33)*1000 ≈ 15,500 I/Os
  - 15,500 I/Os ≈ **3 minutes**

# Block Nested Loops Join




- Load 1st page of Table 2 into memory
- Load as many pages of Table 1 into memory as possible
- Check every combination of records in buffers

# Yet Another Alternative Algorithm: Use Index

- Index nested loops join

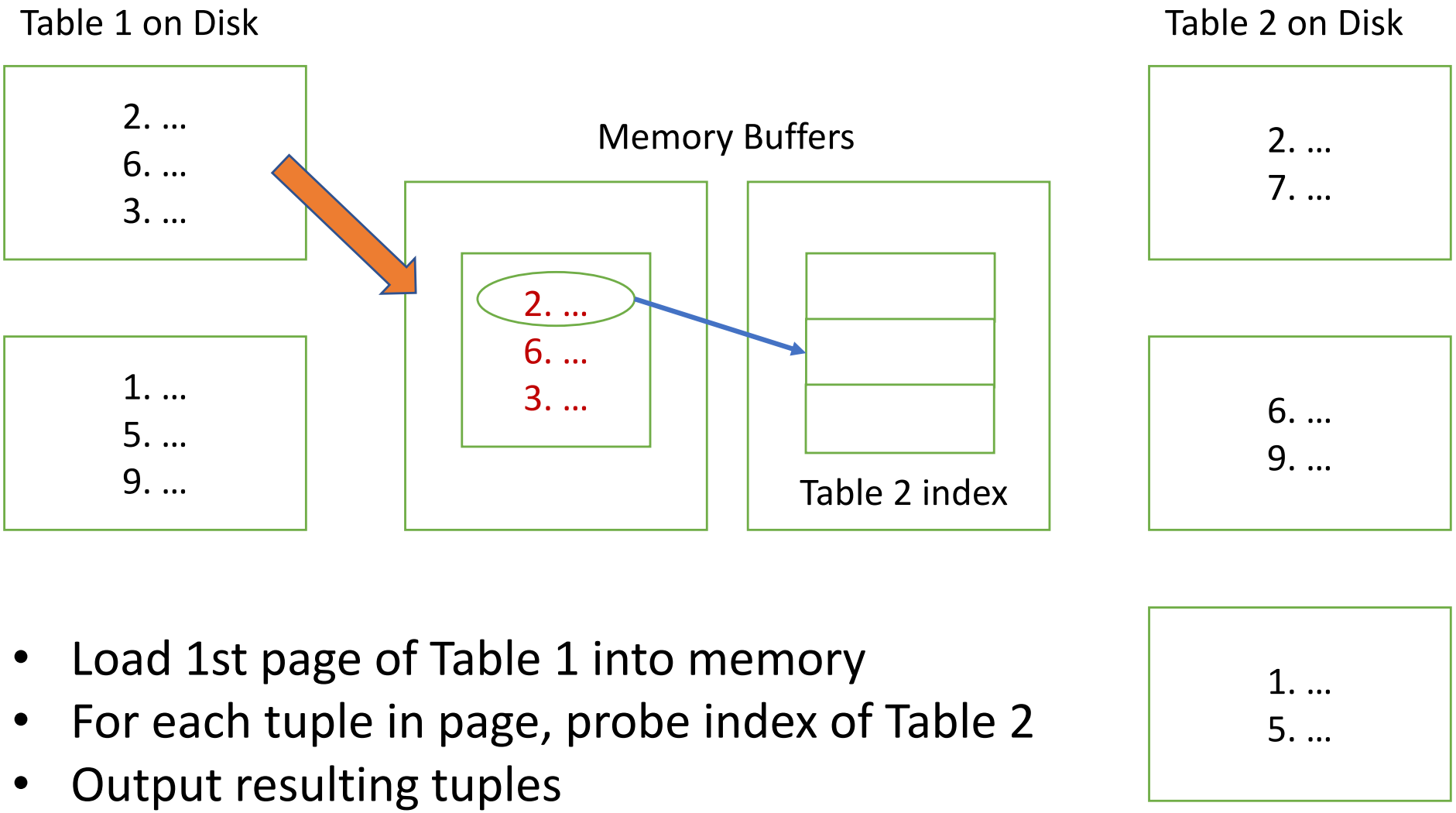
  Join on i-th column of R and j-th column of S

  1. Assuming there is an index on the j-th column of S
  2. foreach tuple r in R do
  3. find tuples s in S with matching search key $r_i$
  4. for each such s, add <r, s> to result

- For records in R, use search key to obtain matching S records
  - If *M* = 1000 pages in R, *PR* = 100 tuples/page, and 100 I/Os per sec.
  - Cost R ⋈ S = *M* + (*M*PR*) * cost of finding matching S tuples

    = 1000 + (1000*100) * 3 ≈ 300,100 I/Os ≈ **1 hour**
  - Cost S ⋈ R = 500 + (500*80)*4 ≈ 160,500 I/Os ≈ **30 minutes**
  - If probing R is 2 I/Os, then ≈ 15 minutes

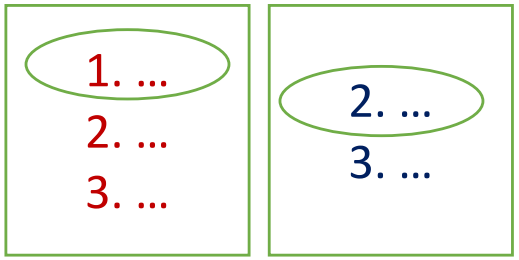
# Page-Oriented Nested Loops Join




- Load 1st page of Table 1 into memory
- For each tuple in page, probe index of Table 2
- Output resulting tuples

# And Another Alternative Algorithm: Sort

- If each relation is sorted on the join attributes ...
- Cost of joining R and S can be reduced to *M + N*




- Compare 1st in R and 1st in S
- If match output <r, s>
- Otherwise discard smallest and repeat

- But what if R and S are not sorted?
  - We need to sort them
  - The *Challenge*: The tables do not fit into memory!
  - The *Solution*: External Sorting
  - Note that other relational operator algorithms also require sorting

# N-Way External Sorting

- Employ the “merge” step in the mergesort algorithm
- On the first pass:
  - Read pages of file until memory (buffers) full
  - Sort data in buffer pages on (search/sort) key
  - Write result back out to disk
- Result is a “*sorted run*” ...

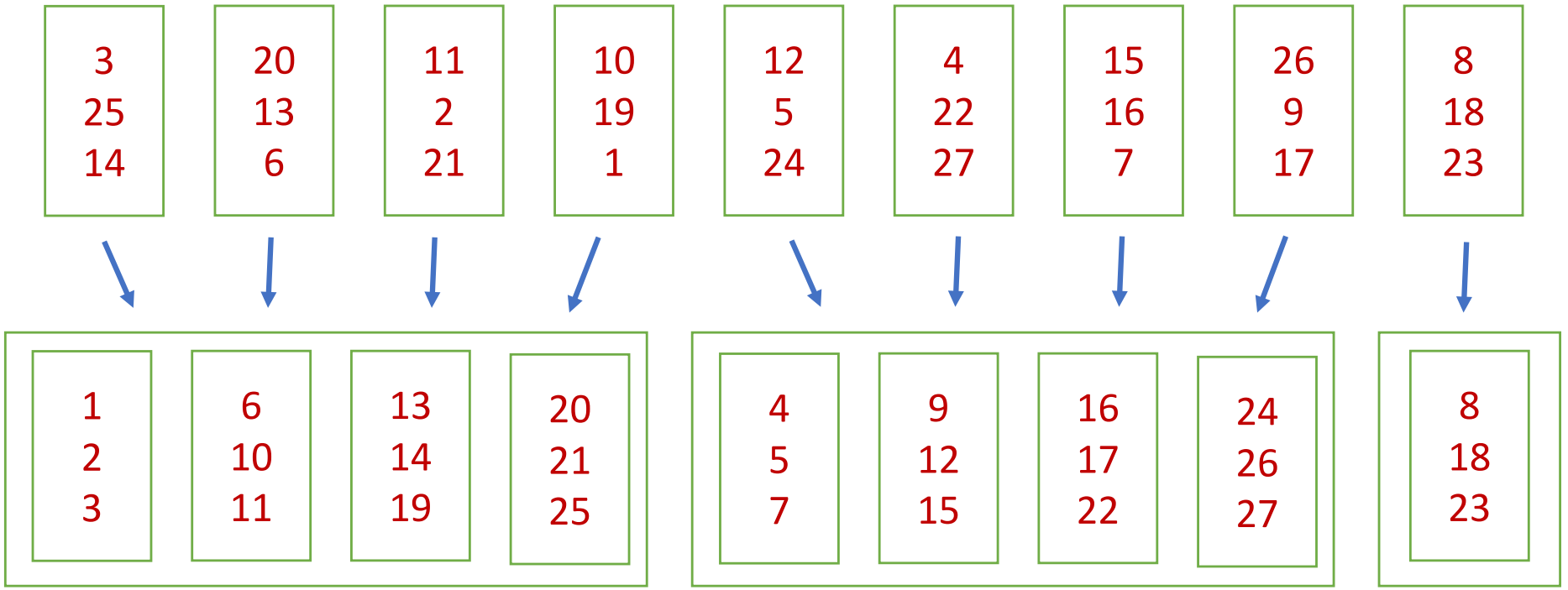
  A sorted run consists of a (sub-) set of small *sorted files*

# N-Way External Sorting

- Employ the "*merge*" step in the mergesort algorithm
- Once we have a "sorted run"
  - Do an "N-way" merge

    ... rather than a 2-way merge as in mergesort
  - $N = B - 1$ is the number of available buffers
  - One buffer reserved for output
- Results in a set of additional passes
- In each pass we create larger sorted sub-files

# Ex: 4 Buffer Pages

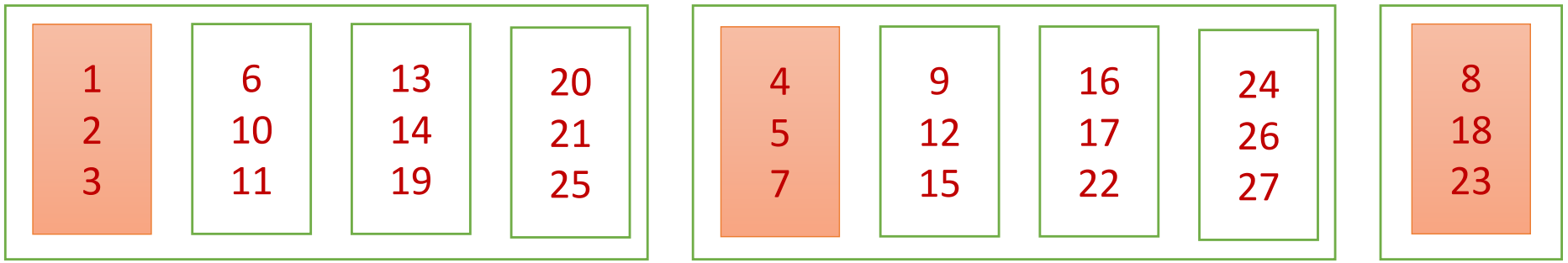



*New File of 3 sorted sub-files*

## First Pass:

load B = 4 pages, sort, and store as sorted sub-files

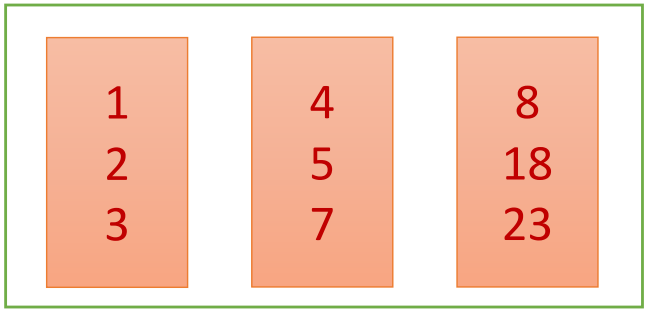
# Ex: 4 Buffer Pages




*In this case 3-way merge*

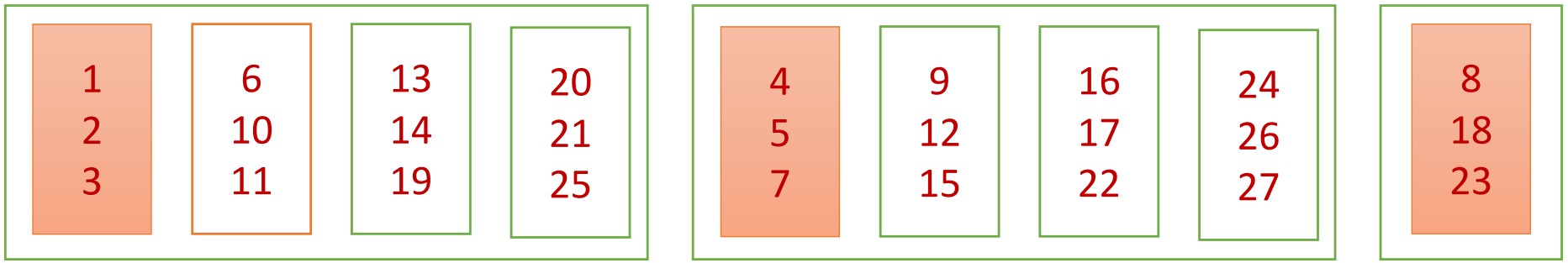
Buffer

Load Buffer




Merge B – 1 of the sorted sub files (sorted runs)

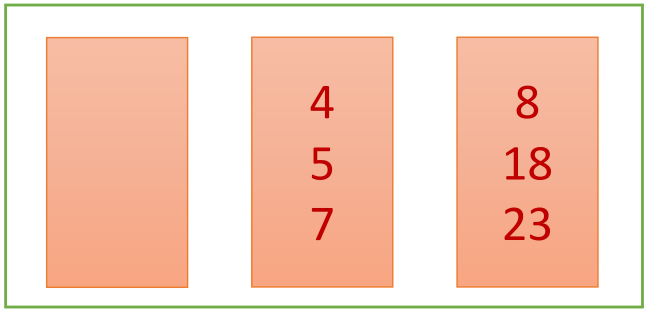
# Ex: 4 Buffer Pages




*In this case 3-way merge*

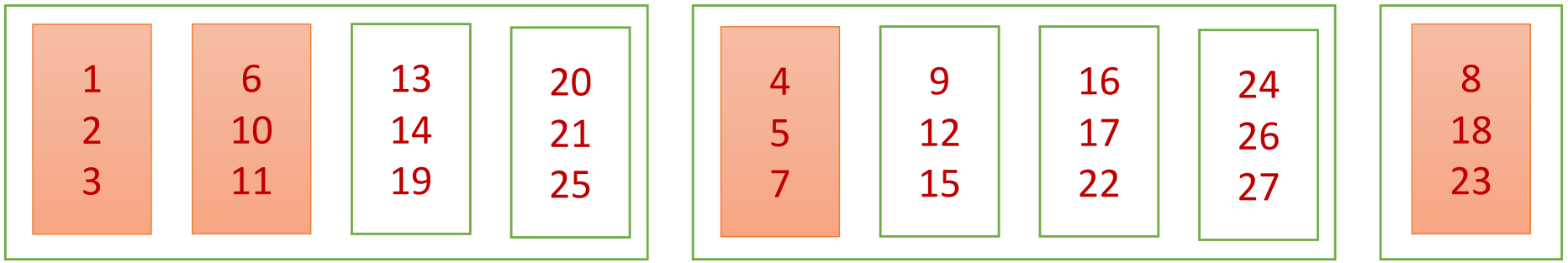
Buffer

Output
1st Page




Merge B – 1 of the sorted sub files (sorted runs)

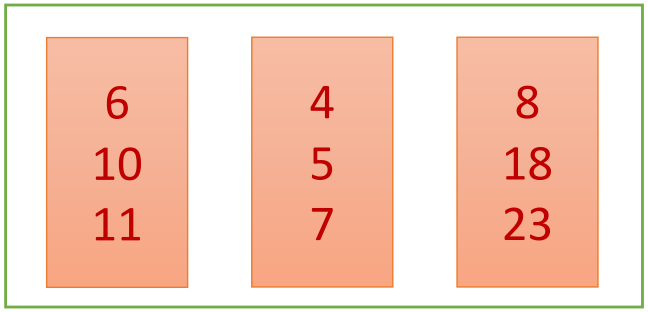
# Ex: 4 Buffer Pages




*In this case 3-way merge*

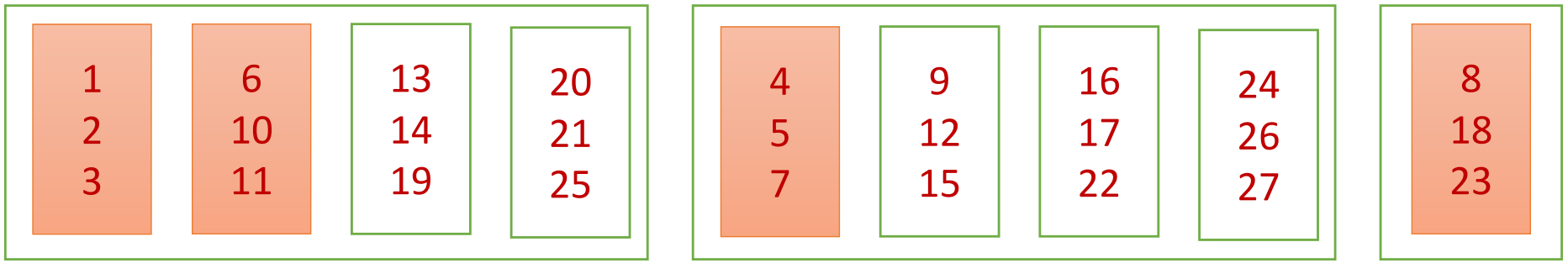
Buffer

Load Buffer again




Merge B – 1 of the sorted sub files (sorted runs)

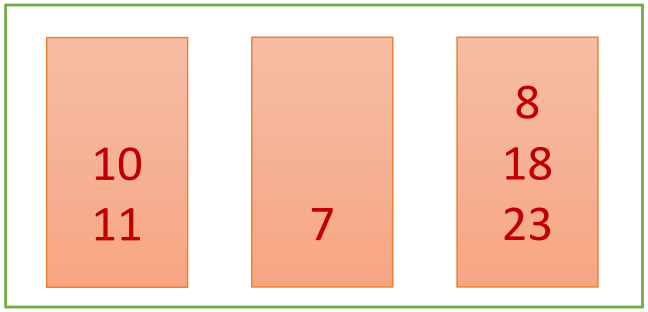
# Ex: 4 Buffer Pages




*In this case 3-way merge*

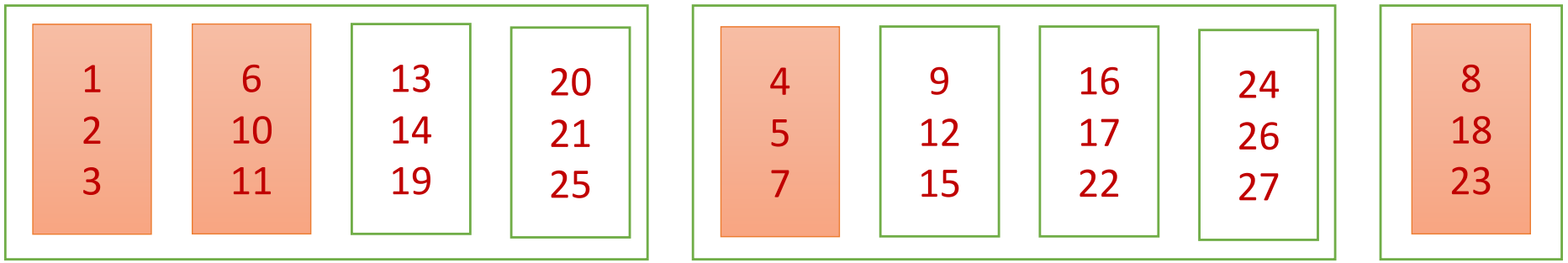
Buffer

Output
2nd Page




Merge B – 1 of the sorted sub files (sorted runs)

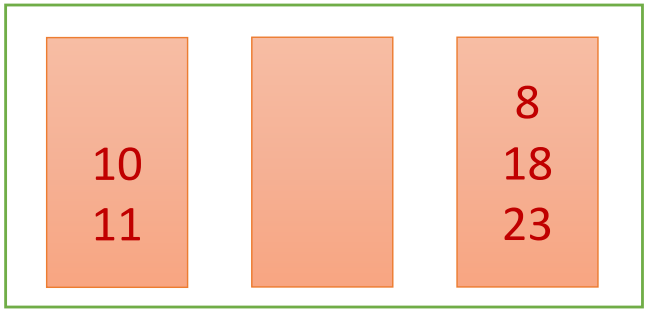
# Ex: 4 Buffer Pages




*In this case 3-way merge*

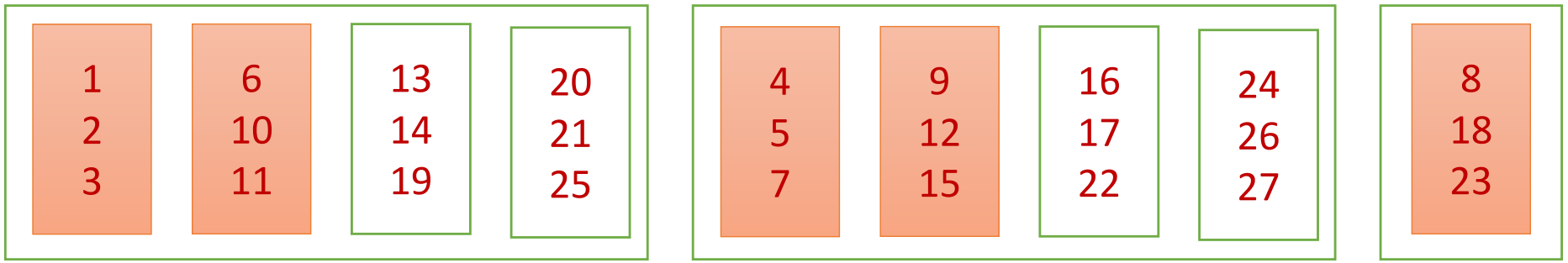
Buffer

Output
3rd Page




Merge B – 1 of the sorted sub files (sorted runs)

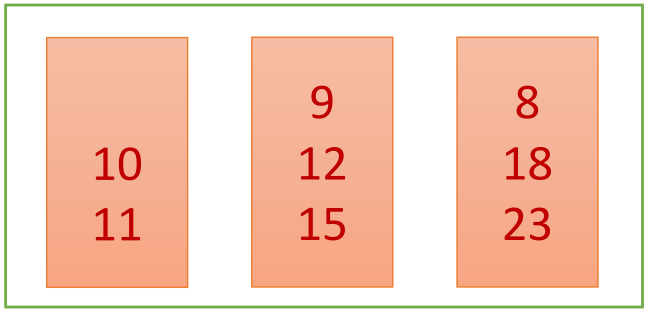
# Ex: 4 Buffer Pages




*In this case 3-way merge*

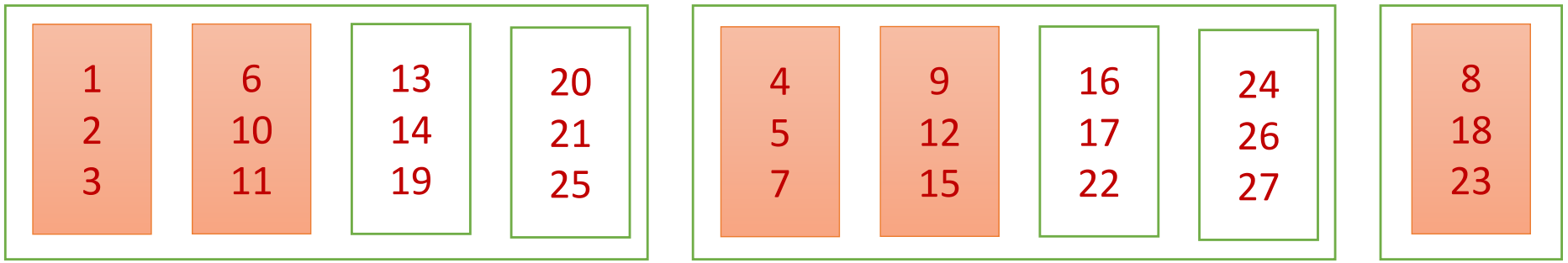
Buffer

Load Buffer again




Merge B – 1 of the sorted sub files (sorted runs)

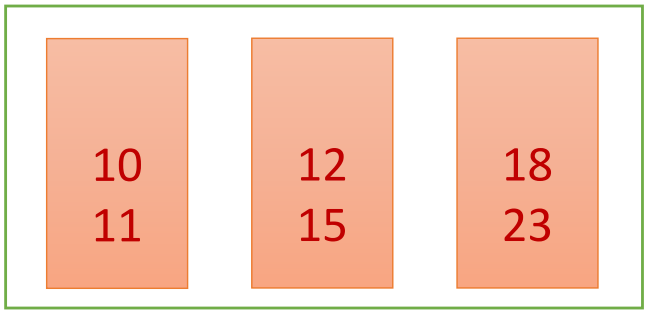
# Ex: 4 Buffer Pages




*In this case 3-way merge*

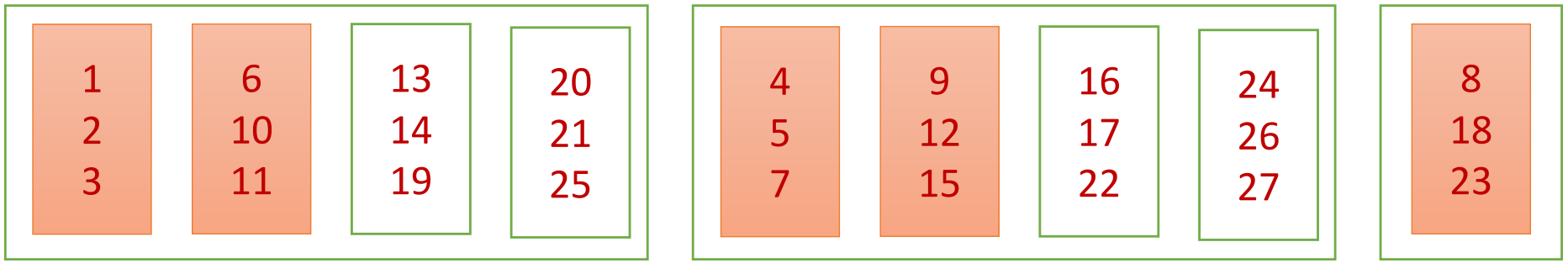
Buffer

Output
3rd Page




Merge B – 1 of the sorted sub files (sorted runs)

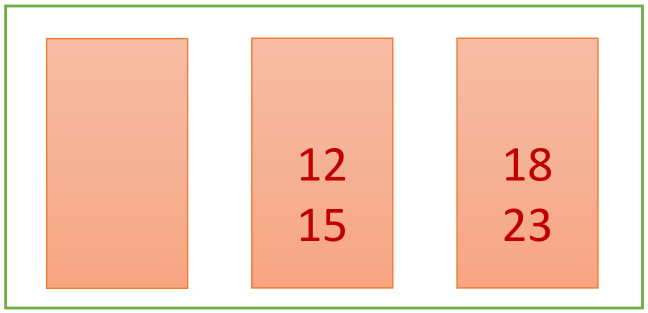
# Ex: 4 Buffer Pages




*In this case 3-way merge*

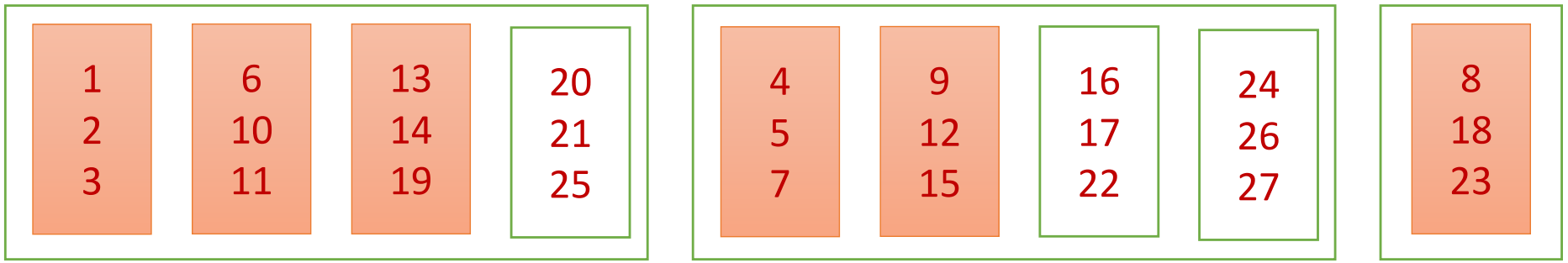
Buffer

Output
4th Page




Merge B – 1 of the sorted sub files (sorted runs)

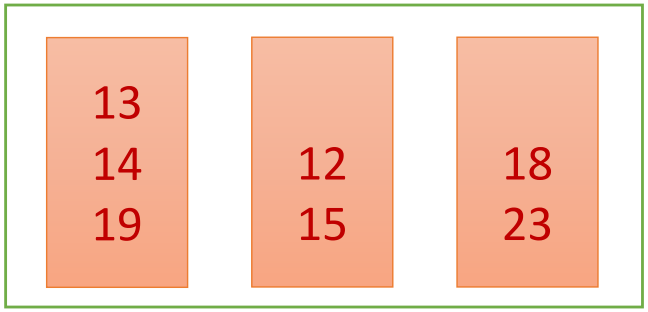
# Ex: 4 Buffer Pages




*In this case 3-way merge*

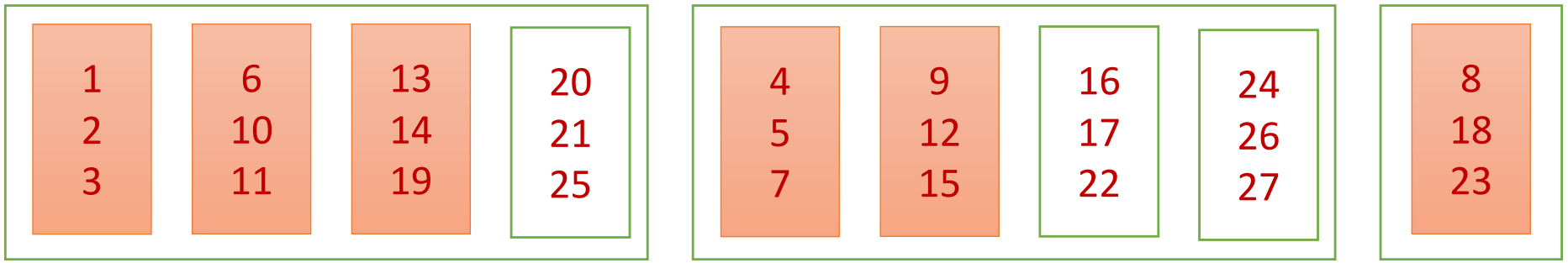
Buffer

Load Buffer again




Merge B – 1 of the sorted sub files (sorted runs)

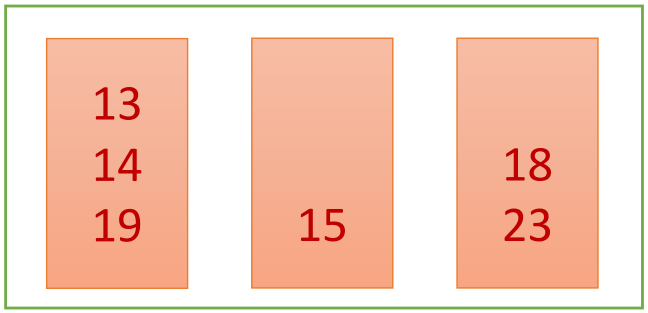
# Ex: 4 Buffer Pages




*In this case 3-way merge*

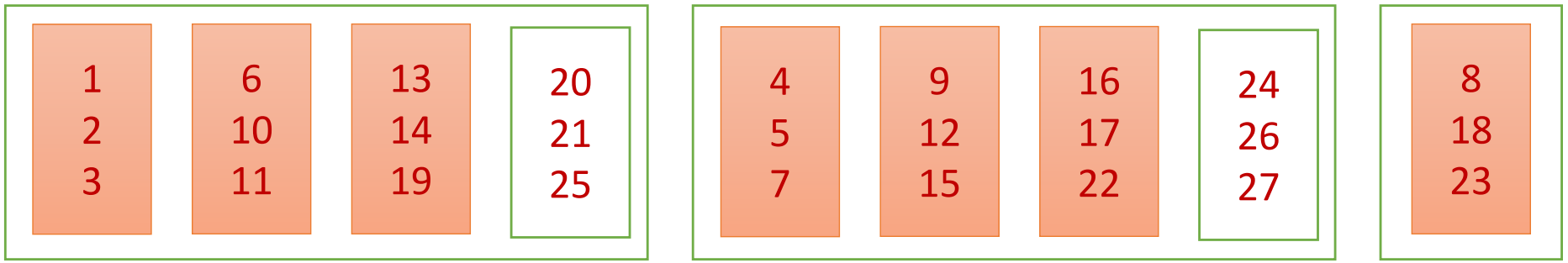
Buffer

Output
4th Page




Merge B – 1 of the sorted sub files (sorted runs)

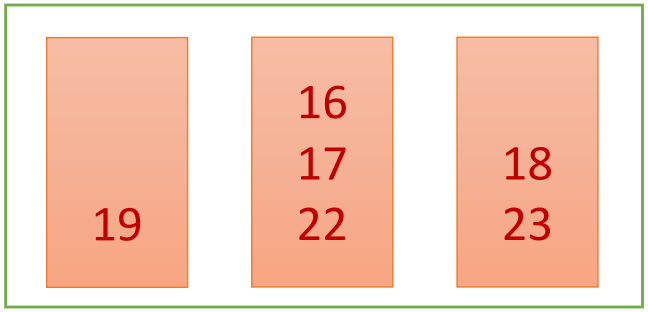
# Ex: 4 Buffer Pages



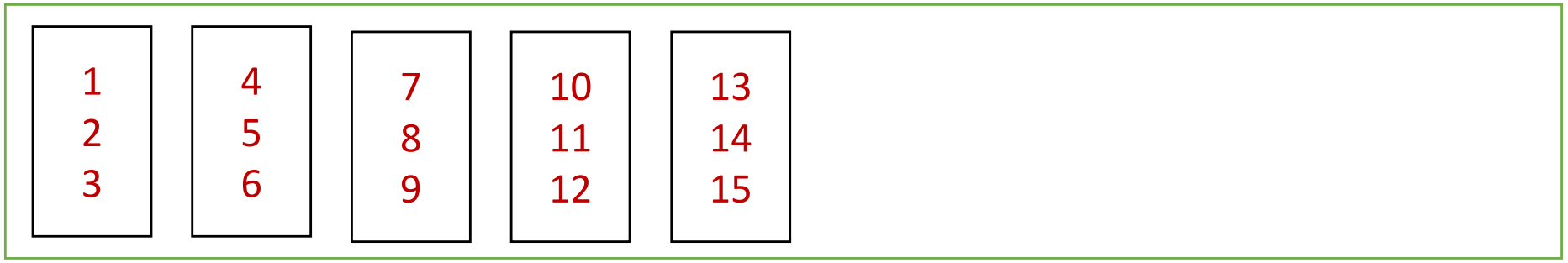

*In this case 3-way merge*

Buffer

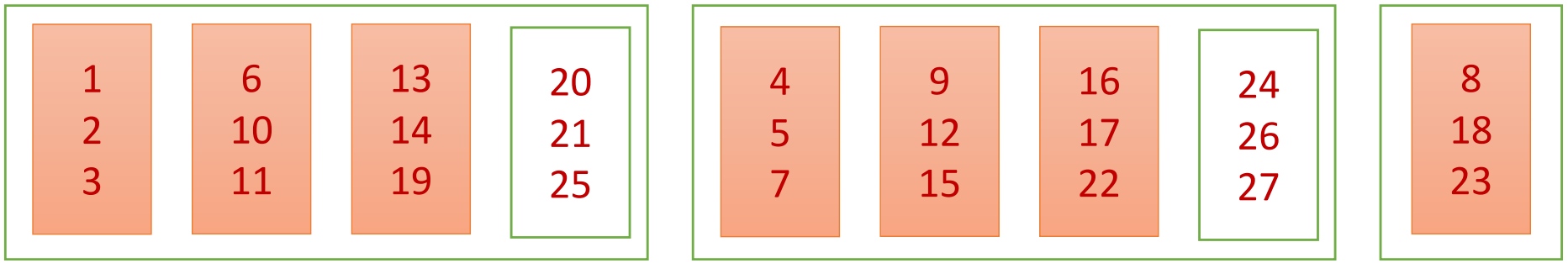
Load Buffer again




Merge B – 1 of the sorted sub files (sorted runs)



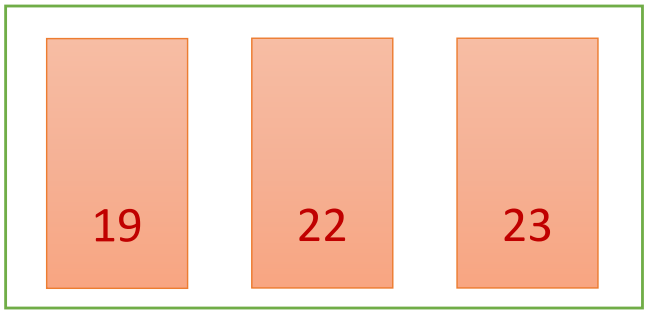
# Ex: 4 Buffer Pages



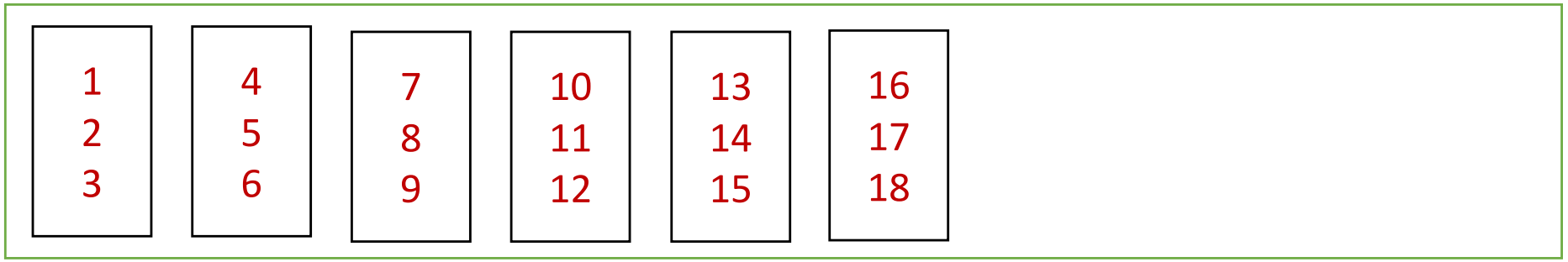

*In this case 3-way merge*

Buffer

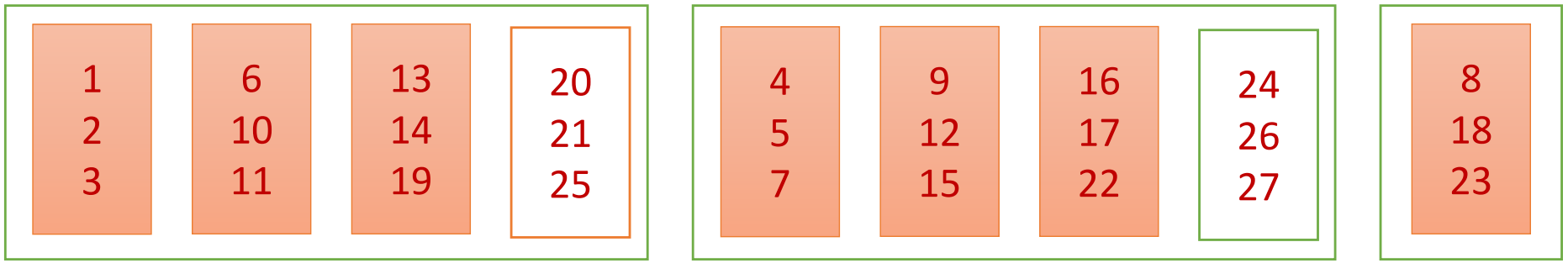
Output
6th Page




Merge B – 1 of the sorted sub files (sorted runs)

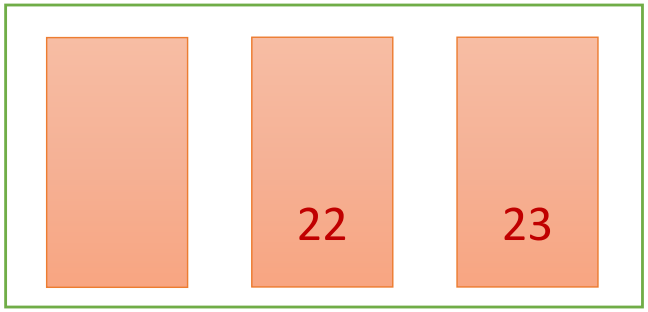


# Ex: 4 Buffer Pages



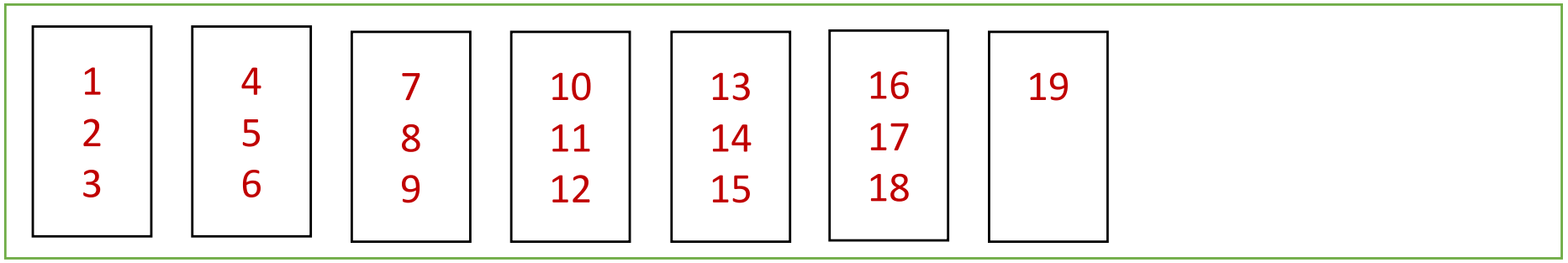

*In this case 3-way merge*

Buffer

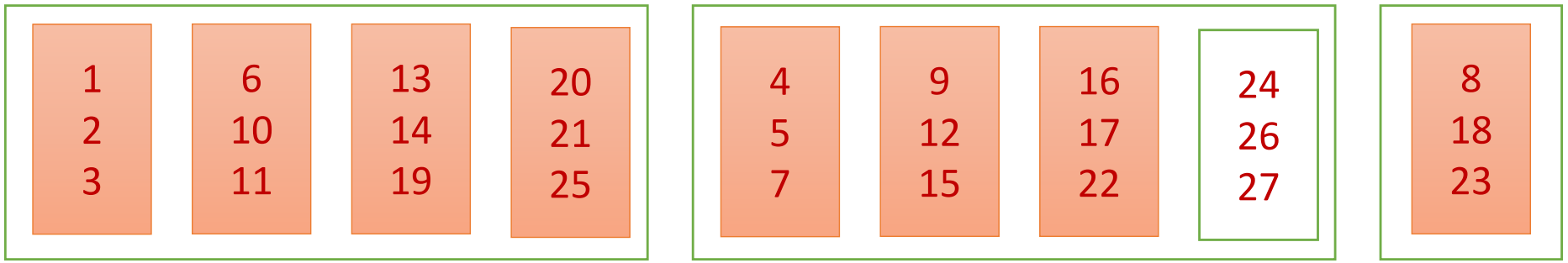



Output
7th Page

Merge B – 1 of the sorted sub files (sorted runs)

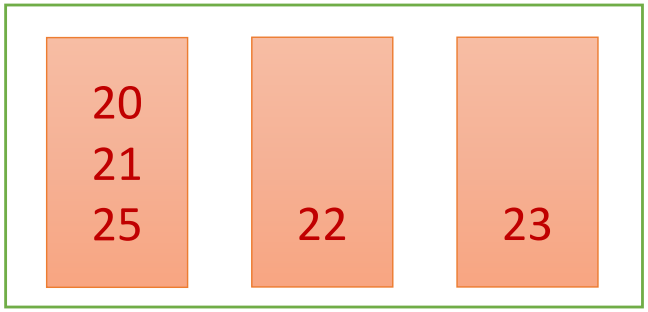


# Ex: 4 Buffer Pages



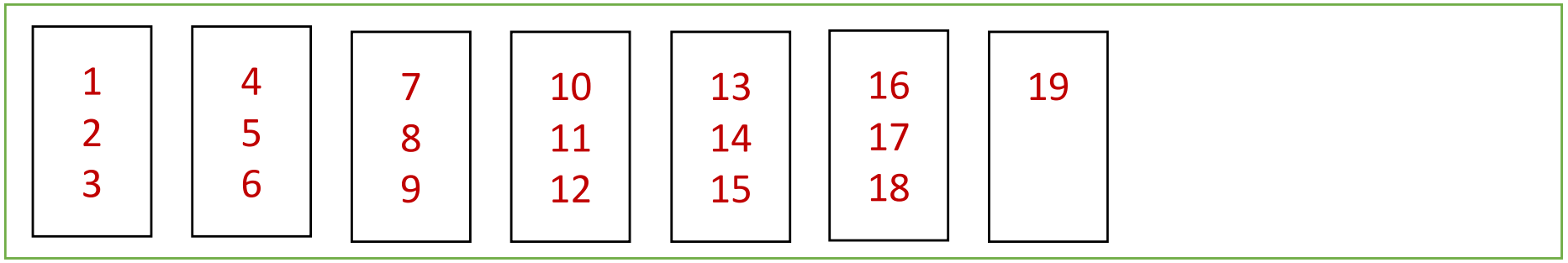

*In this case 3-way merge*

Buffer

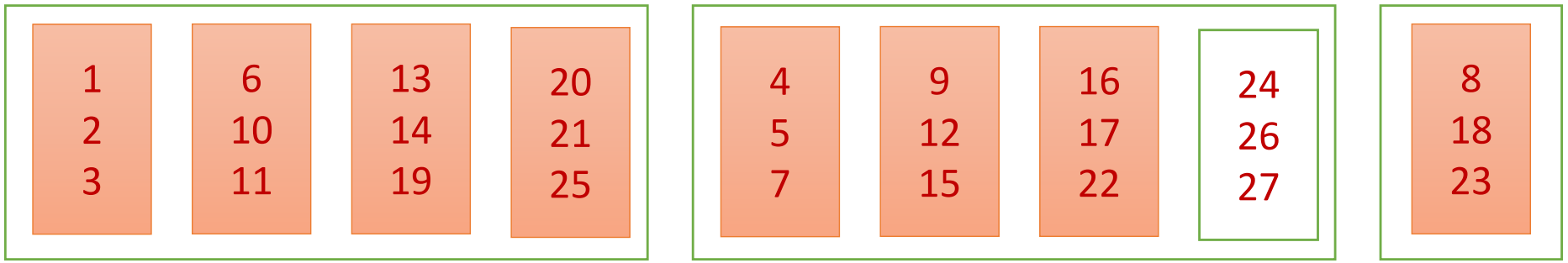



Load Buffer again

Merge B – 1 of the sorted sub files (sorted runs)



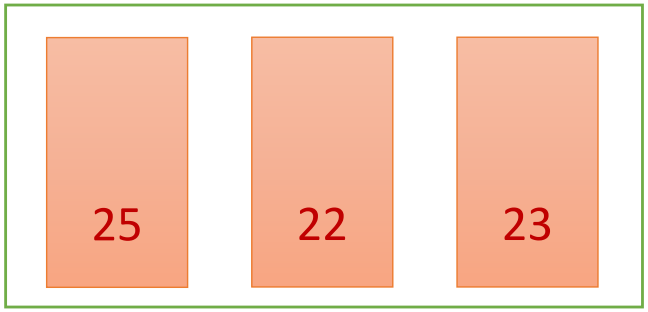
# Ex: 4 Buffer Pages



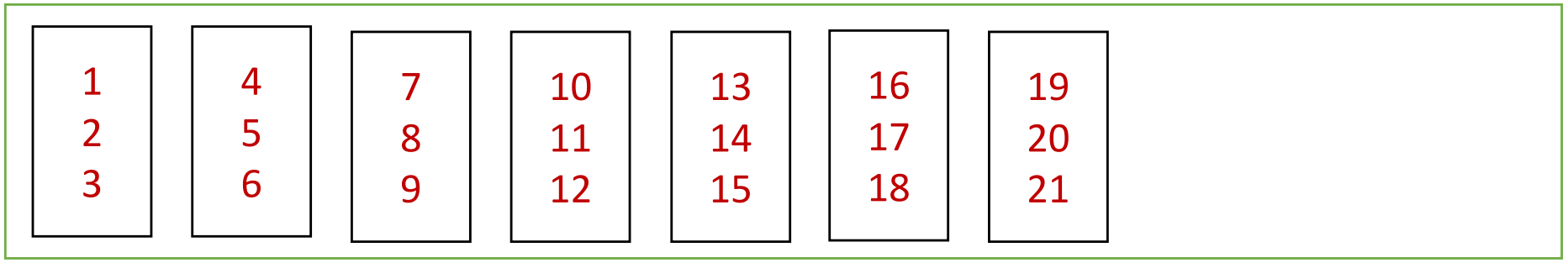

*In this case 3-way merge*

Buffer

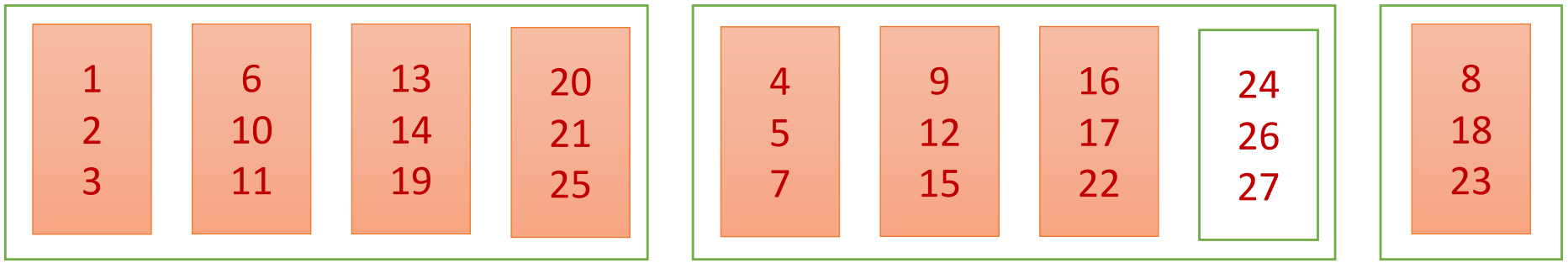
Output
7th Buffer




Merge B – 1 of the sorted sub files (sorted runs)



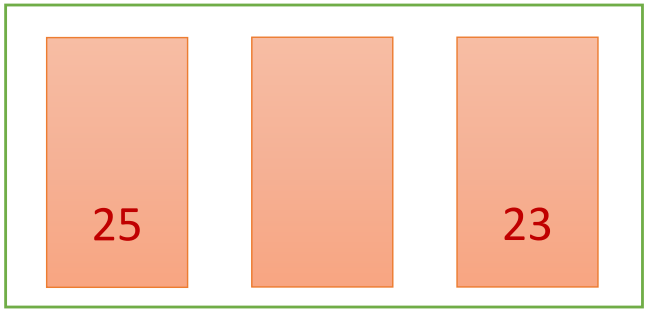
# Ex: 4 Buffer Pages



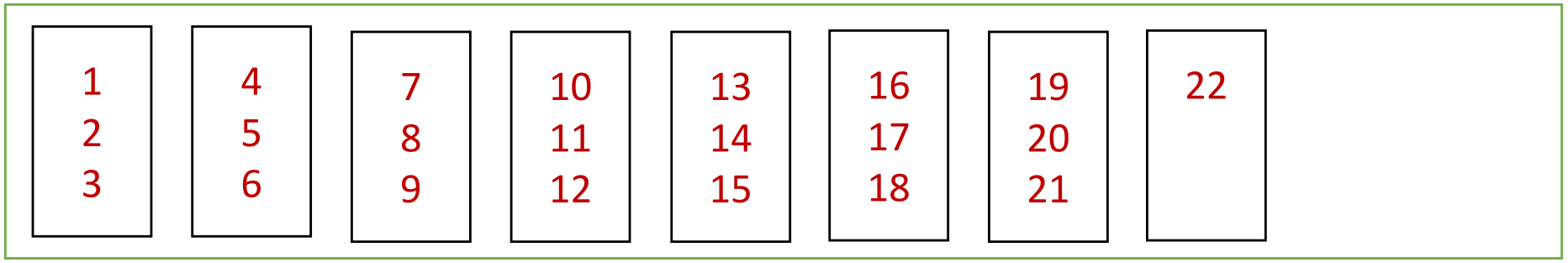

*In this case 3-way merge*

Buffer

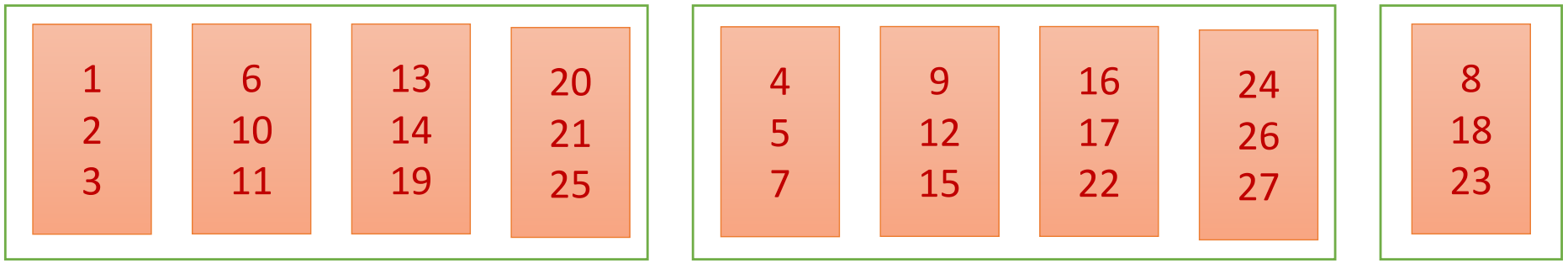
Output
8th Buffer




Merge B – 1 of the sorted sub files (sorted runs)

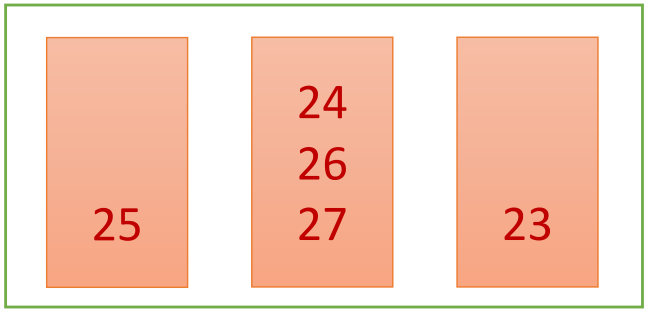


# Ex: 4 Buffer Pages



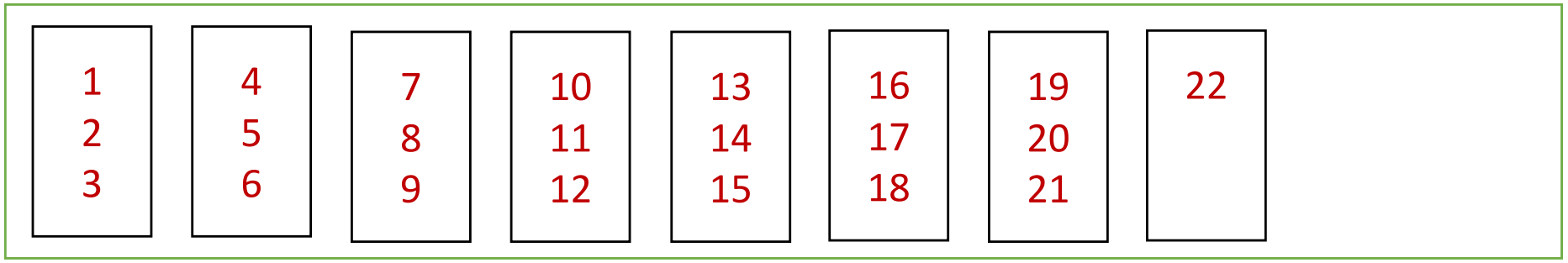

*In this case 3-way merge*

Buffer

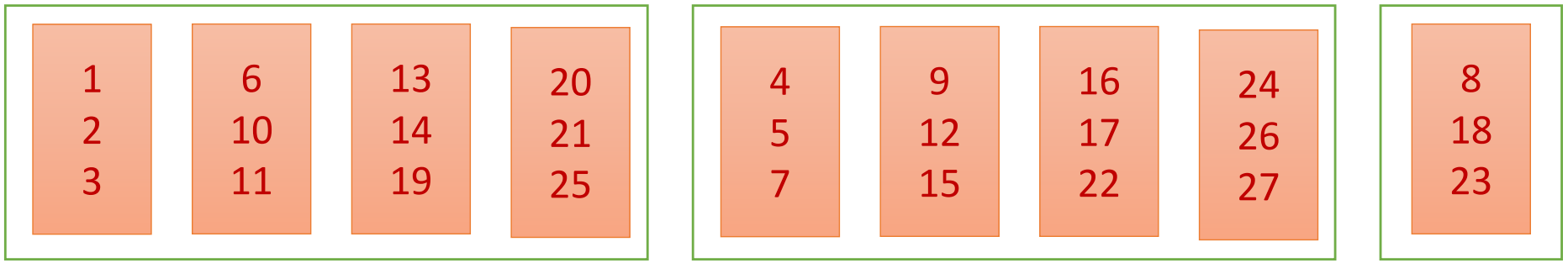



Load Buffer again

Merge B – 1 of the sorted sub files (sorted runs)



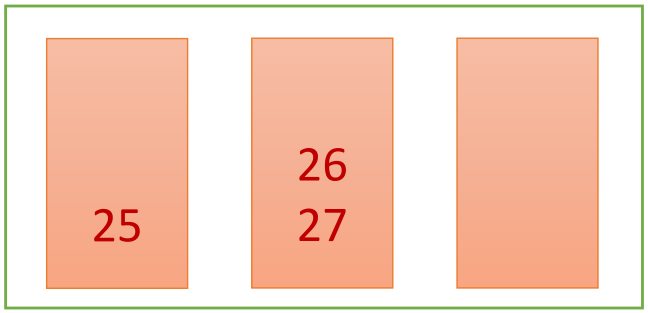
# Ex: 4 Buffer Pages



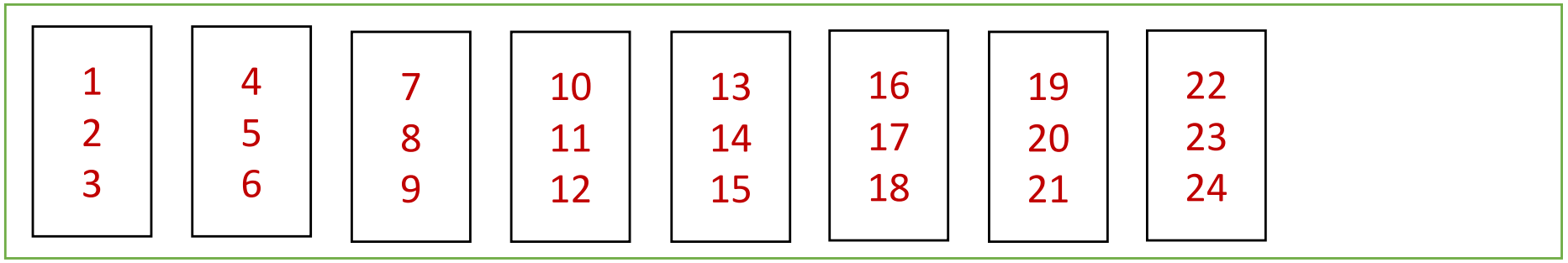

*In this case 3-way merge*

Buffer

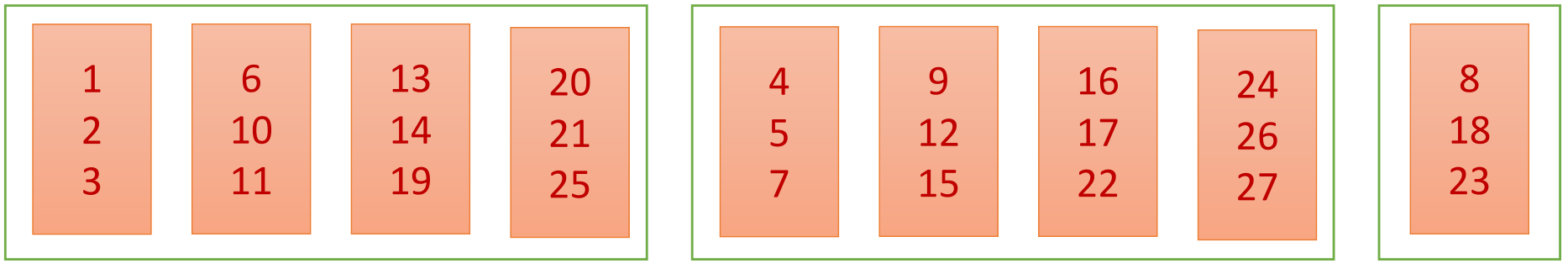
Output
8th Buffer




Merge B – 1 of the sorted sub files (sorted runs)

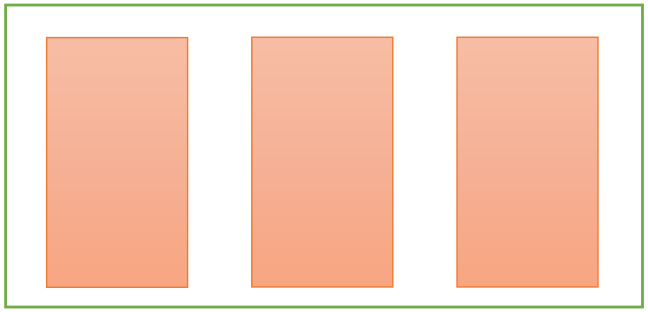


# Ex: 4 Buffer Pages



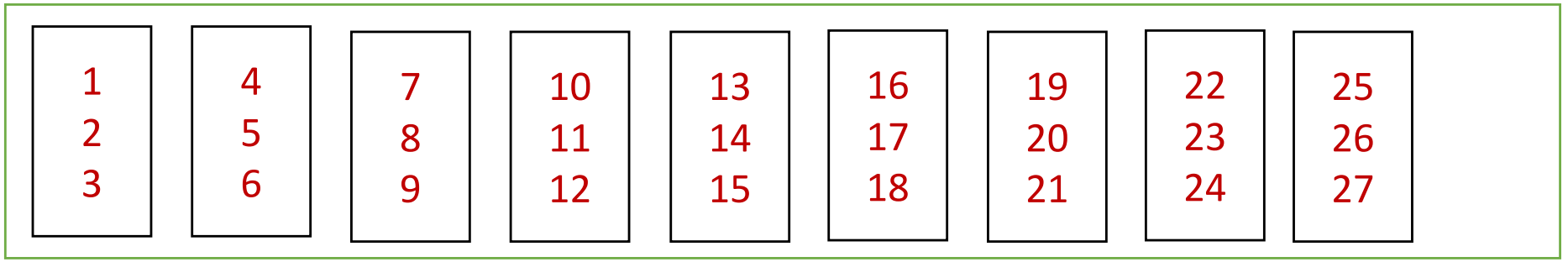

*In this case 3-way merge*

Buffer



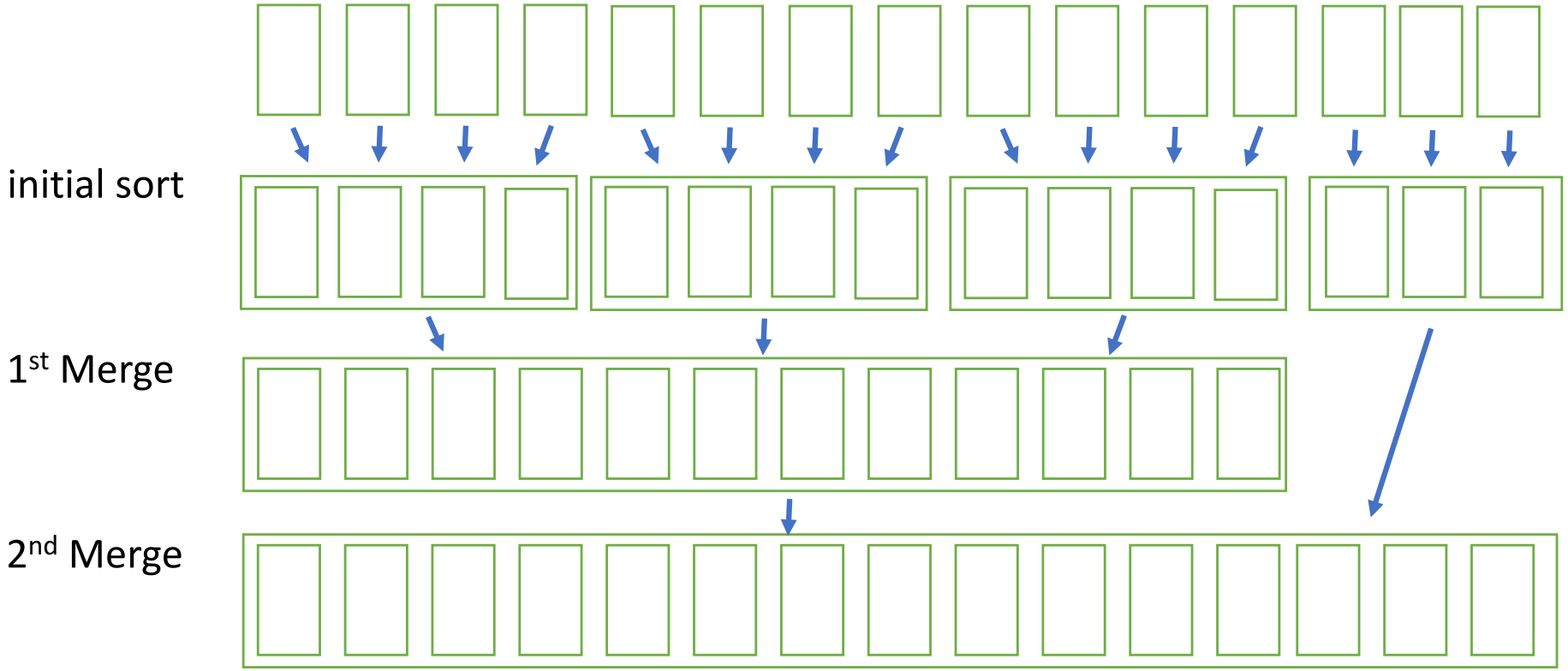
Output
9th Buffer
DONE!!

Merge B – 1 of the sorted sub files (sorted runs)



# N-Way External Sorting

- Merge may require multiple passes
- At each merge pass the number of sub-files is reduced by B – 1



# N-Way External Sorting

- The cost:
  - Each pass does 2**M* I/Os (for *M* pages in table)
  - We read and write the entire file (all pages) in each pass
  - So ... how many passes?
- Number of passes depends on buffer space available
  - Passes = $\lceil \text{Log}_{B-1}(M/B) \rceil$... Why *M*/*B*?
  - Can sort 100 million pages in 4 passes w/ 129 pages of memory
  - Can sort *M* pages using *B* memory pages in 2 passes if $\sqrt{M} < B$ (often true)

# Sort Merge Join

- Sort R on join attribute (if not already sorted)
- Sort S on join attribute (if not already sorted)
- Merge R and S
  - Scan of R until R-tuple ≥ current S-tuple
  - Then scan S until S-tuple ≥ R-tuple
  - Repeat until R-tuple = S-tuple
  - At this point, we have a match, and output
  - Then resume scanning R and S

# Sort Merge Join

- Outer relation R is scanned once
  - Each time an R-tuple r matches first S-tuple
  - We form a “group” of S-tuples that match r
  - Each such group is scanned once per matching R tuple
  - Either:
    - This group fits into memory (and the scan is “free”)
    - Or we have extra page I/Os (to reread the group)

# Sort Merge Join

- Best case cost (all matches in memory):
  - Cost to sort R + Cost to sort S + (M+N)
- Worst case cost (all R and S have same value)
  - Matching group is the entire S relation
  - Cost to Sort R + Cost to sort S + M + M*N
  - ... note this is *worse* than page-oriented nested loops! (since you also have to sort R and S)

# Sort Merge Join

- For Reserves and Sailors:
  - Reserves has 1000 pages
  - Sailors has 500 pages
  - With 35 pages in the buffer, each sorted in 2 passes
- Best case cost is:
  - 4*1000 + 4*500 + 1000 + 500 = 7500 I/Os ≈ 1 minute
  - ... multiply by 4 since it takes 2 passes and each pass reads and writes each page of file

# Sorting using B+ Trees

- Lets say the table we want to sort has a B+ Tree defined on the sorting attributes
- Can the B+ Tree help to retrieve records in order?
- It can help if the B+ Tree is clustered
  - We can retrieve records in order by traversing leaf pages
  - Records either stored in leaf pages or can be obtained from leaf pages
- It can be a very bad idea if B+ Tree is not clustered
  - Why?

# Sorting via Clustered B+ Trees

- Cost for clustered case
  - Root to the left-most leaf, then retrieve all pages
- What if it is unclustered?
  - Additional cost of retrieving data records
  - Each page fetched just once
- Always better than external sorting!

# Sorting via Clustered B+ Trees

- Similar to the case of doing a range query
- unclustered case - data entries
  - Each entry on one leaf page can point to a different page
  - In general, one I/O per data record!

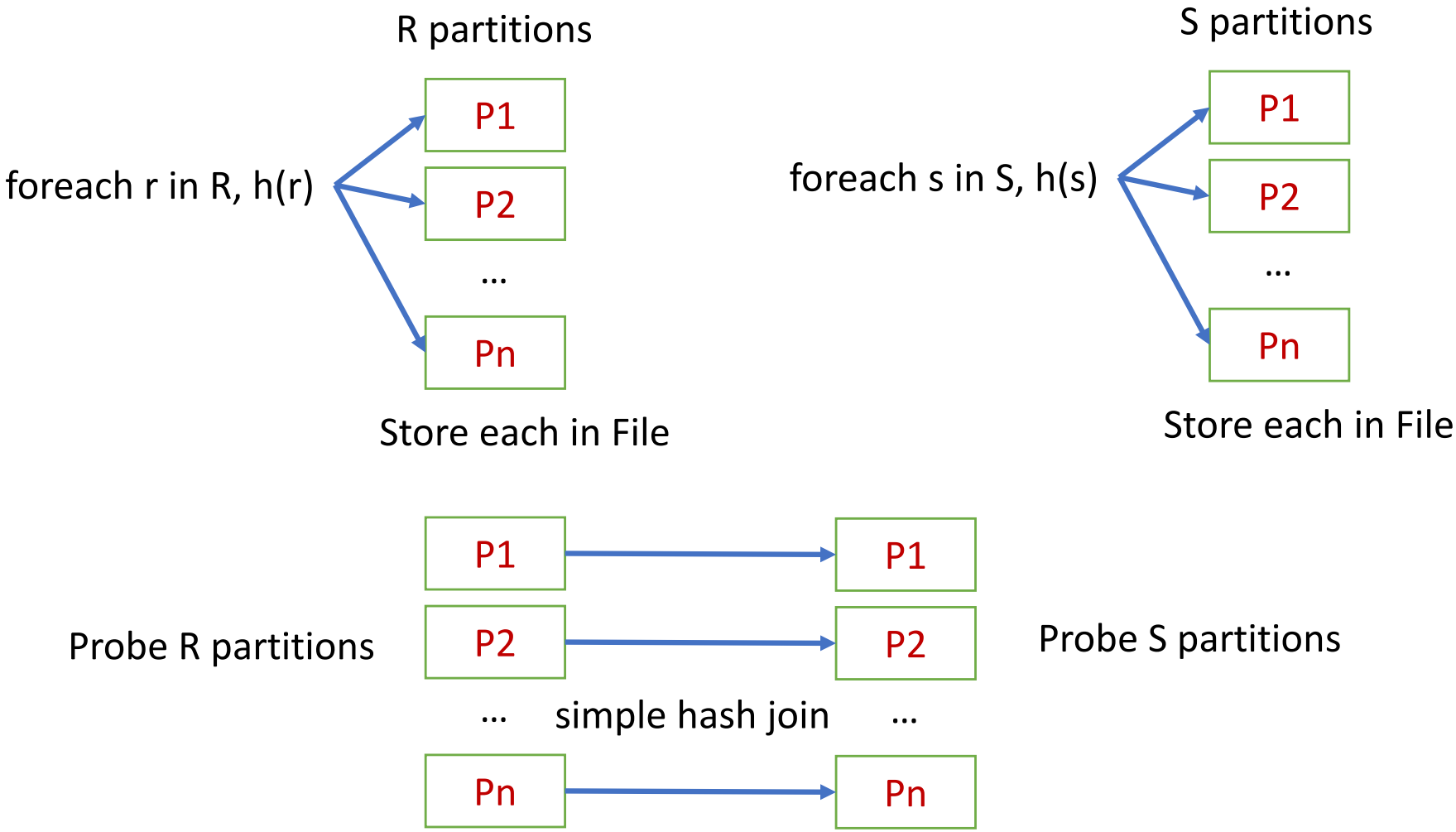
# Hash Join

- Simple case: entire S table fits into main memory
  - Build an in-memory hash index for S ("*build*" phase)
- recall a hash index maps keys to buckets of records
  - Scan R and find matching S-records ("*probe*" phase)
- this is identical to the index nested loops join
- Cost is:
  - Cost to read R (the outer relation)
  - Cost to read S (the inner relation + build index)
  - Each time we read a page in R we find all matches with S
  - So total cost is M + N!

# Hash Join

- What do we do if S does not fit into memory?
  - Define a hash function *h* that can be used to partition R and S
  - Each S partition should be small enough to fit into main memory
  - Apply *h* to R and S and store each resulting partition in a file
  - Do the simple case (index nested loop join) on *each pair* of matching partitions (files)

# Hash Join


R partitions
foreach r in R, h(r)
P1
P2
...
Pn
Store each in File
S partitions
foreach s in S, h(s)
P1
P2
...
Pn
Store each in File
Probe R partitions
P1
P2
...
Pn
simple hash join
P1
P2
...
Pn
Probe S partitions

# Hash Join

- Partitions:
  - We assume that the number of partitions k < B
  - Each partition may have many pages
- Cost of Hash Join:
  - 2*M to partition R (read and write)
  - 2*N to partition S (read and write)
  - Cost to join partitions: M + N
  - Total cost is: 3*(M+N)
  - For reserves and sailors:
  - 3*(1000 + 500) = 4500 I/Os ≈ 45 seconds

# Sort Merge Join vs. Hash Join

- Sort-Merge Join
  - Less sensitive to data “skew” (e.g., clusters of similar values)
  - Result is sorted (... more on this later)
- Hash Join
  - Highly parallelizable (join partitions concurrently)
- For inequality conditions (e.g., R.name < S.name)
  - Hash and Sort-Merge Join not applicable
  - Block nested loops likely to be the best approach

# Comparison of (approximate) costs

| Join Algorithm | I/Os | Time |
|---|---|---|
| Simple Nested Loops Join | 50,000,000 | 6 days |
| Page Nested Loops Join | 500,000 | 1.4 hours |
| Block Nested Loops Join | 16,000 | 3 minutes |
| Index Nested Loops Join | 160,500 | 30 minutes |
| Sorted-merge Join | 7500 (at best) | 1 minute (at best) |
| Hash Join | 4500 | 45 seconds |

Assuming:

- R has 1000 pages, 100 tuples/page
- S has 500 pages, 80 tuples/page
- 35 buffer pages
- 100 I/Os per second

# For Next Week

- Read
  - Ch. 16