

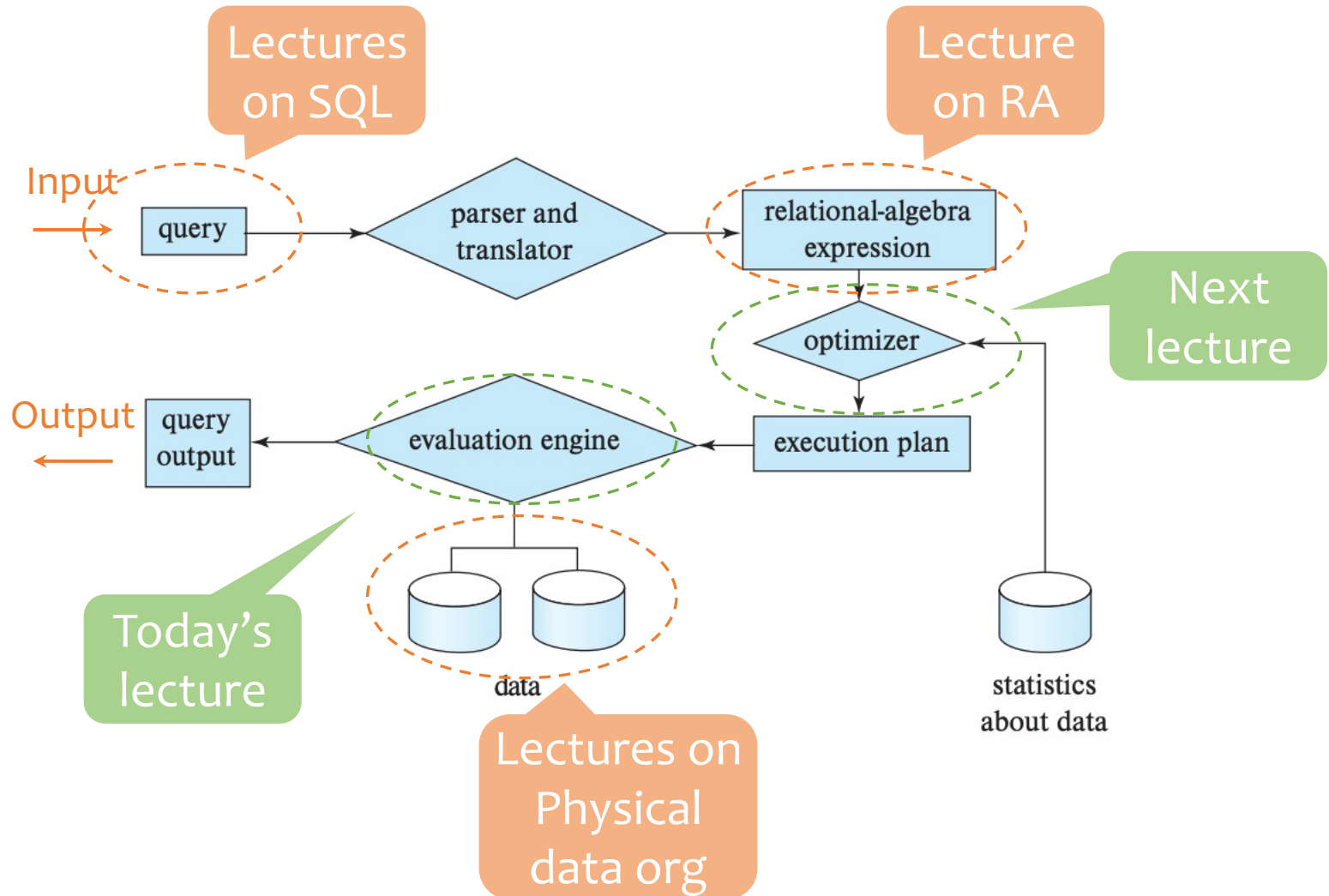
Query Processing

CS348 Spring 2023

Instructor: Sujaya Maiyya

Sections: **002 & 004 only**

Overview



Overview (cont.)

- Many different ways of processing the same query
 - Scan? Sort? Hash? Use an index?
 - All have different performance characteristics and/or make different assumptions about data
- Best choice depends on the situation
 - Implement all alternatives
 - Let the **query optimizer** choose at run-time (next lecture)

Outline

- Scan

```
select * from User where pop = 0.8
```

- Index

```
select * from User, Member where  
User.uid = Member.uid;
```

- Sort

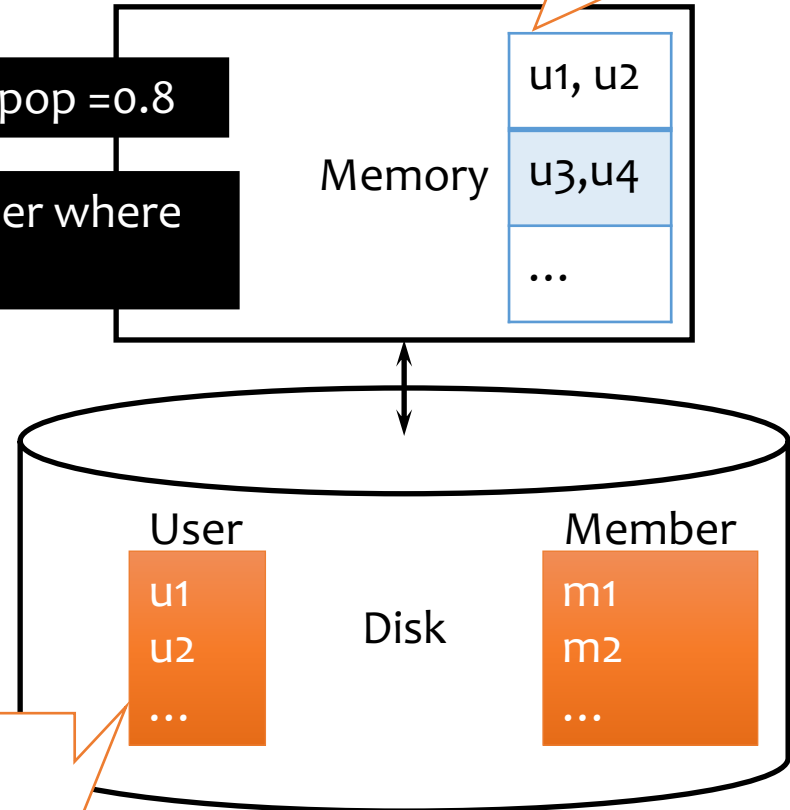
- Hash (Optional)

Number of rows for a table $|Users|$

Number of disk blocks for a table

$$B(Users) = \frac{|Users|}{\# \text{ of rows per block}}$$

Number of memory blocks available: M



Notation

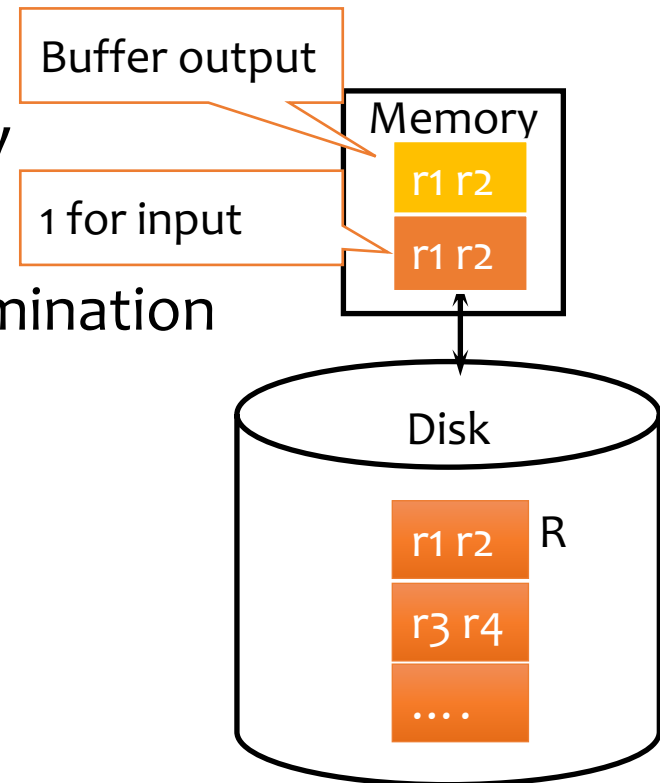
- Relations: R, S
- Tuples: r, s
- Number of tuples: $|R|, |S|$
- Number of disk blocks: $B(R), B(S)$
- Number of memory blocks available: M
- Cost metric
 - Number of I/O's
 - Memory requirement

Scanning-based algorithms



Table scan

- Scan table R and process the query
 - Selection over R
 - Projection of R without duplicate elimination
- I/O's: $B(R)$
 - Trick for selection:
 - stop early if it is a lookup by key
- Memory requirement: 2 (blocks)
 - 1 for input, 1 for buffer output
 - Increase memory does not improve I/O
- Not counting the cost of writing the result out
 - Same for any algorithm!



Basic nested-loop join

$$R \bowtie_p S$$

- For each r in a block B_R of R :
 - For each s in a block B_S of S :
 - Output rs if p is true over r and s

- R is called the **outer** table; S is called the **inner** table
- I/O's: $B(R) + |R| \cdot B(S)$

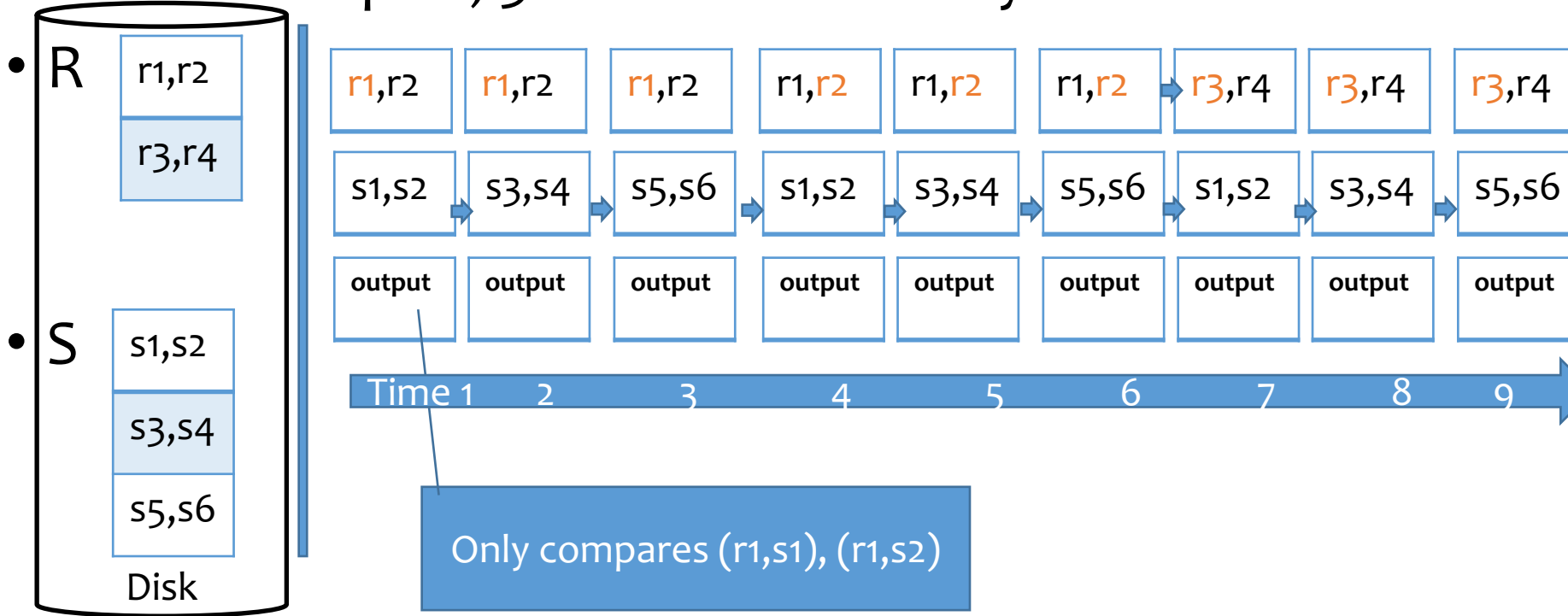
Blocks of R are moved into memory only once

Blocks of S are moved into memory $|R|$ number of times

- Memory requirement: **3**

Example for basic nested loop join

- 1block = 2 tuples, 3 blocks of memory



- Number of I/O:

$$B(R) + |R| * S(R) = 2 \text{ blocks} + 4 * 3 \text{ blocks} = 14$$

Improvement: block nested-loop join

$$R \bowtie_p S$$

- For each block B_R of R :
 - For each block B_S of S :
 - For each r in B_R :
 - For each s in B_S :
 - Output rs if p is true over r and s

- I/O's: $B(R) + B(R) \cdot B(S)$

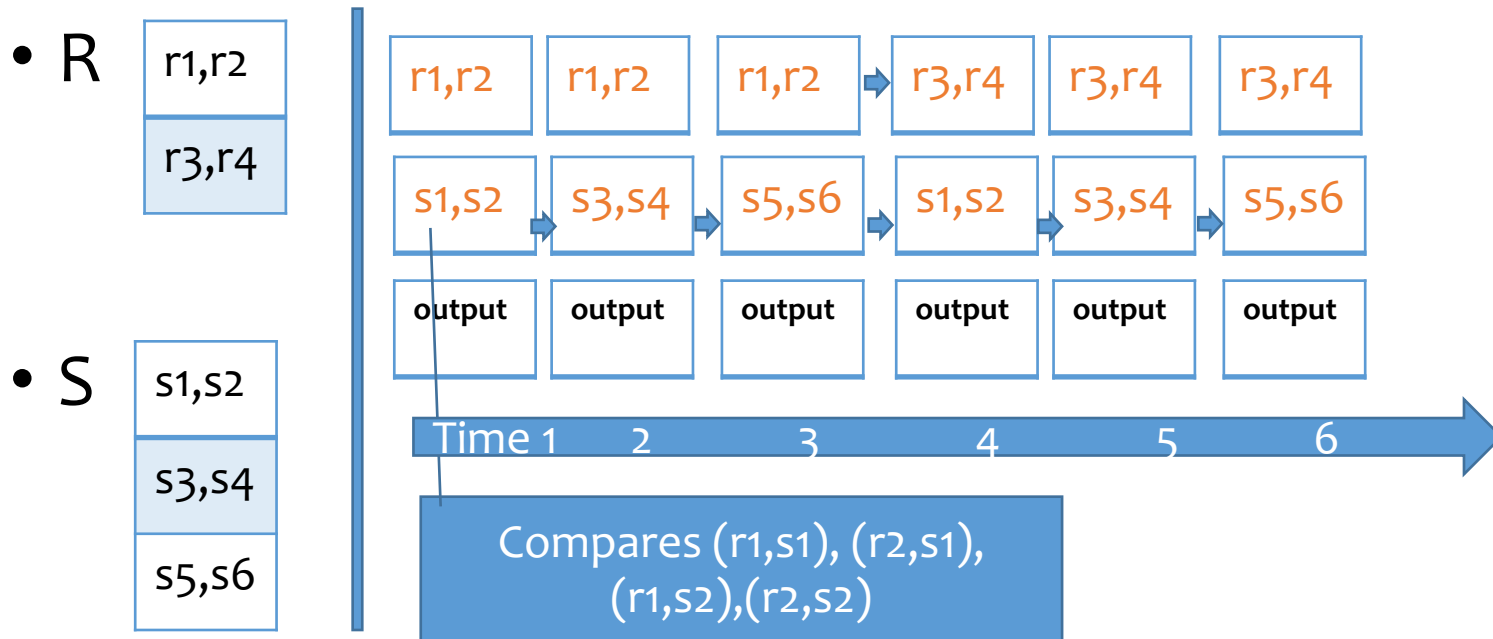
Blocks of R are moved into memory only once

Blocks of S are moved into memory $B(R)$ number of times

- Memory requirement: 3

Example for block-based nested loop join

- 1block = 2 tuples, 3 blocks of memory



- Number of I/O:

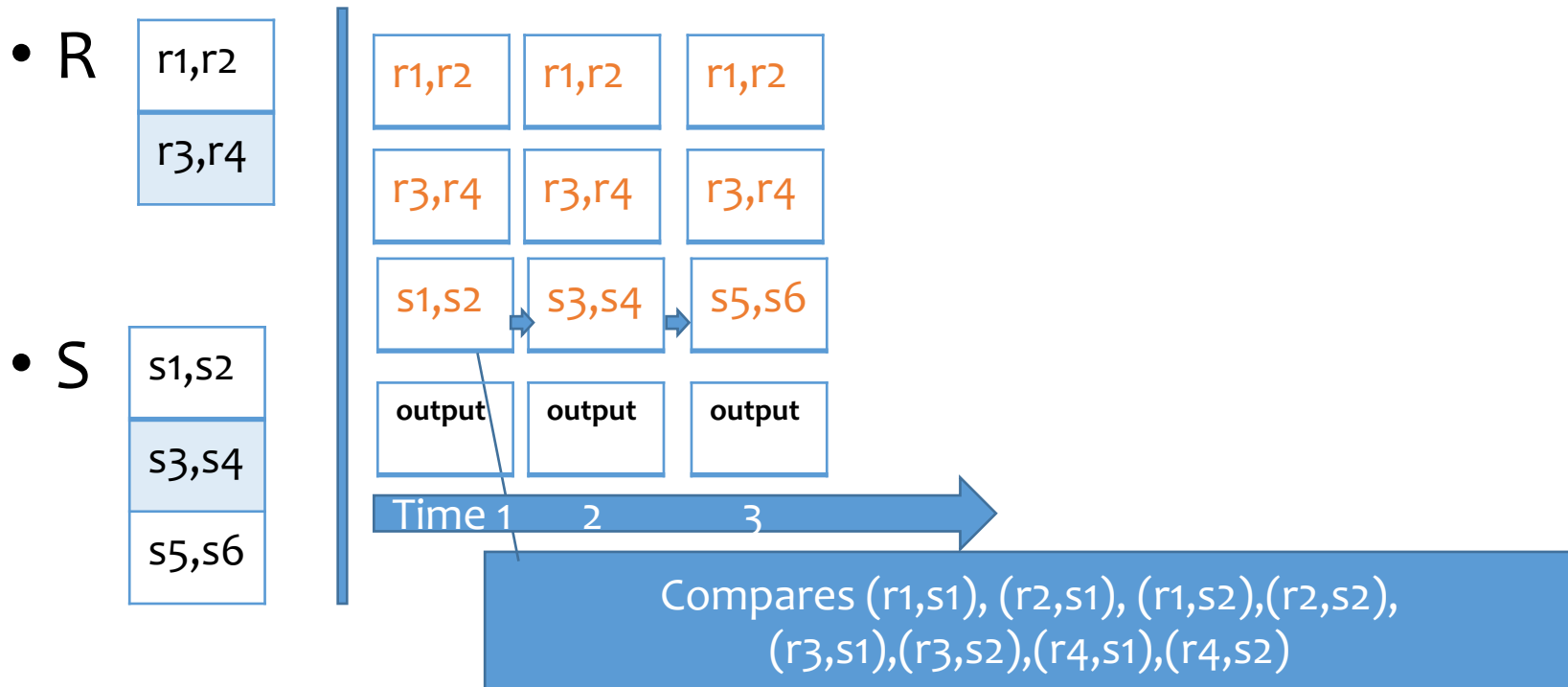
$$B(R) + B(R) * B(S) = 2 \text{ blocks} + 2 * 3 \text{ blocks} = 8$$

More improvements

- Stop early if the key of the inner table is being matched
- Make use of available memory
 - Stuff memory with as much of R as possible, stream S by, and join every S tuple with all R tuples in memory
 - I/O's: $B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S)$
 - Or, roughly: $B(R) \cdot B(S) / M$
 - Memory requirement: M (as much as possible)
- Which table would you pick as the outer? (exercise)

Example for block-based nested loop join

- 1block = 2 tuples, 4 blocks of memory



- Number of I/O:

$$B(R) + B(R)/(M-2) * S(R) = 2 \text{ blocks} + 1 * 3 \text{ blocks} = 5$$

Case study:

- System requirements:
 - Each disk/memory block can hold up to 10 rows (from any table);
 - All tables are stored compactly on disk (10 rows per block);
 - 8 memory blocks are available for query processing: $M=8$
- Database:
 - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
 - |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
 - #of blocks: $B(\text{User})=1000/10=100$; $B(\text{Group})=100/10=10$; $B(\text{Member})=50000/10=5k$
- Q1: select * from User where pop = 0.8
 - I/O cost using table scan? $B(\text{User}) = 100$
- Q2: select * from User, Member where User.uid = Member.uid;
 - I/O cost using blocked-based nested loop join

$$B(\text{User}) + \left\lceil \frac{B(\text{User})}{M-2} \right\rceil \cdot B(\text{Member}) = 100 + \left\lceil \frac{100}{8-2} \right\rceil \cdot 5000 = 85,100$$

Outline

- Scan
 - Selection, duplicate-preserving projection, nested-loop join
- Index
- Sort
- Hash (Optional)

Index-based algorithms



Selection using index

- Equality predicate: $\sigma_{A=v}(R)$
 - Use an ISAM, B⁺-tree, or hash index on $R(A)$
- Range predicate: $\sigma_{A>v}(R)$
 - Use an **ordered** index (e.g., ISAM or B⁺-tree) on $R(A)$
 - Hash index is not applicable
- Indexes other than those on $R(A)$ may be useful
 - Example: B⁺-tree index on $R(A, B)$
 - How about B⁺-tree index on $R(B, A)$?

Index versus table scan

Situations where index clearly wins:

- **Index-only queries** which do not require retrieving actual tuples
 - Example: $\pi_A(\sigma_{A>v}(R))$
- Primary index clustered according to search key
 - One lookup leads to all result tuples in their entirety

Index versus table scan (cont'd)

BUT(!):

- Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$
 - Need to follow pointers to get the actual result tuples
 - Say that 20% of R satisfies $A > v$
 - Could happen even for equality predicates
 - I/O's for scan-based selection: $B(R)$
 - I/O's for index-based selection: $\text{lookup} + 20\% |R|$
 - Table scan wins if a block contains more than 5 tuples!
 - $B(R) = |R|/5 < 20\%|R| + \text{lookup}$

Index nested-loop join

$$R \bowtie_{R.A=S.B} S$$

- Idea: use a value of $R.A$ to probe the index on $S(B)$
- For each block of R , and for each r in the block:
 - Use the index on $S(B)$ to retrieve s with $s.B = r.A$
 - Output rs
- I/O's: $B(R) + |R| \cdot (\text{index lookup} + \text{record fetch})$
 - Typically, the cost of an index lookup is 2-4 I/O's (depending on the index tree height if B+ tree)
 - Beats other join methods if $|R|$ is not too big
 - Better pick R to be the smaller relation
- Memory requirement: **3** (extra memory can be used to cache index, e.g. root of B+ tree)

Outline

- Scan
 - Selection, duplicate-preserving projection, nested-loop join
- Index
 - Selection, index nested-loop join
- Sort
 - External merge sort, sort-merge-join
- Hash (Optional)

Sorting-based algorithms

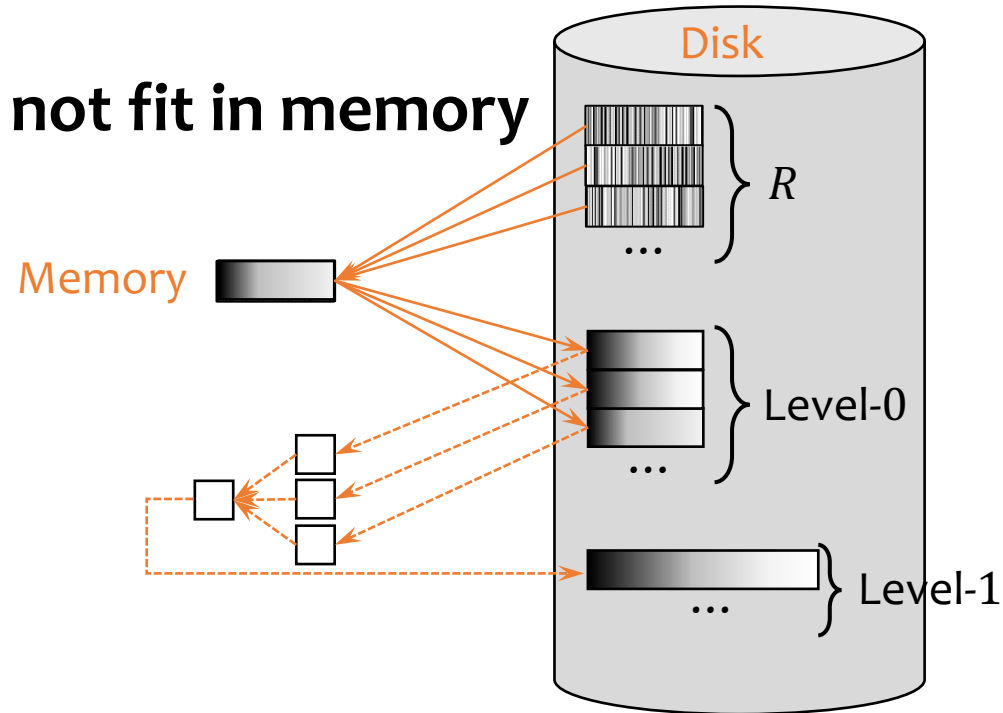


External merge sort

Recall in-memory merge sort: Sort progressively larger runs, 2, 4, 8, ..., $|R|$, by merging consecutive “runs”

Problem: sort R , but R does not fit in memory

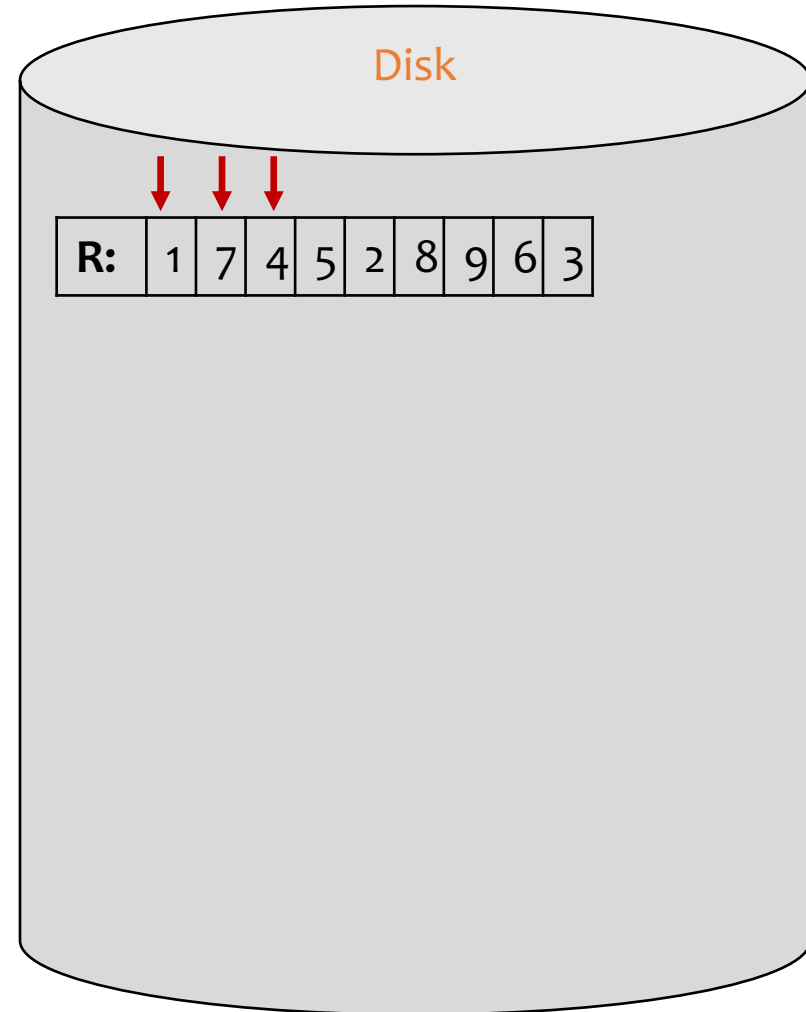
- **Phase 0:** read M blocks of R at a time, **sort** them, and write out a **level-0 run**
- **Phase 1:** **merge** $(M - 1)$ level-0 runs at a time, and write out a **level-1 run**
- **Phase 2:** **merge** $(M - 1)$ level-1 runs at a time, and write out a **level-2 run**
- ...
- **Final phase** produces one sorted run



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

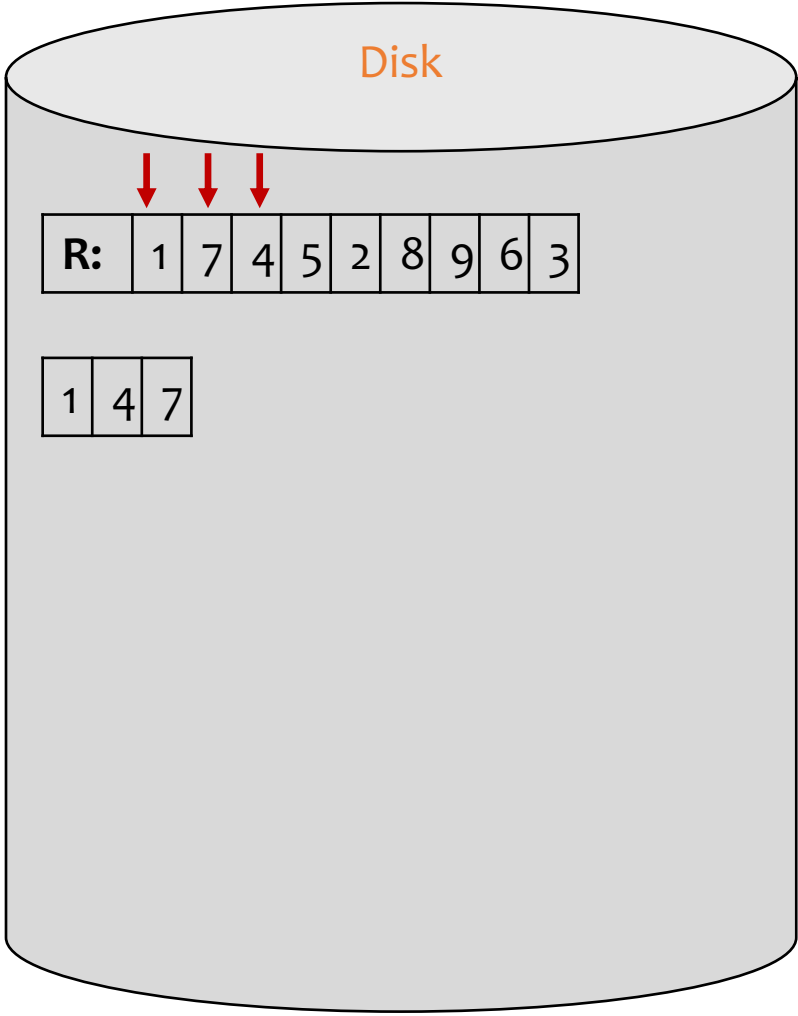
Arrows indicate the
blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

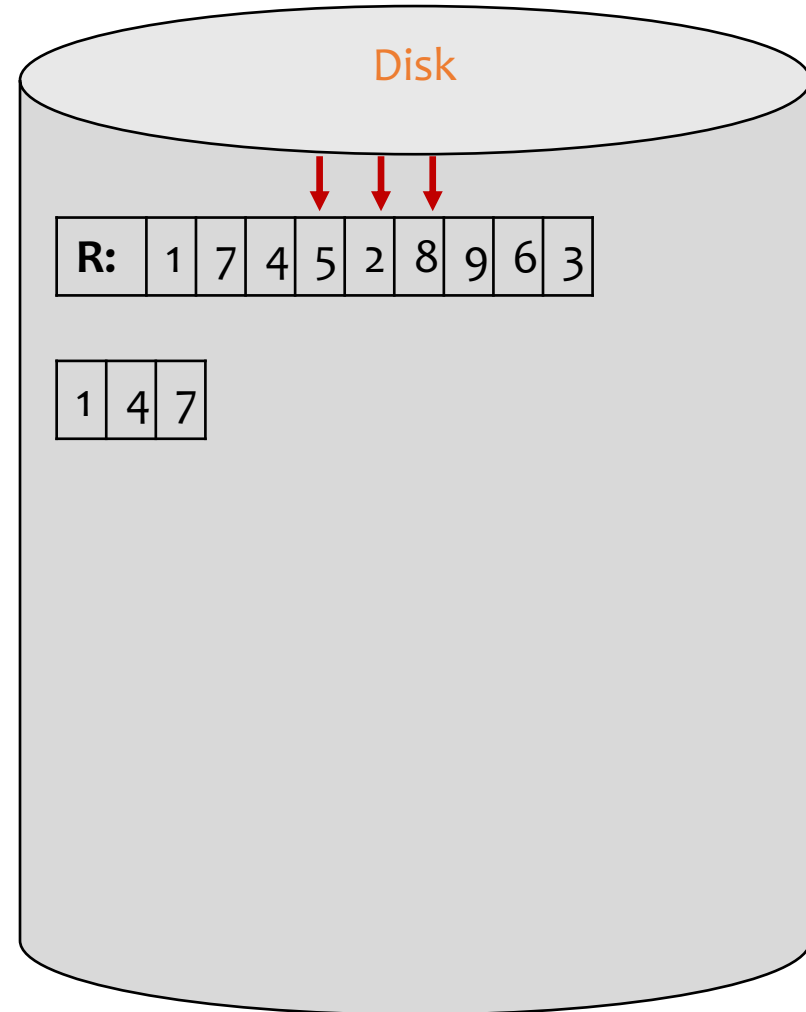
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

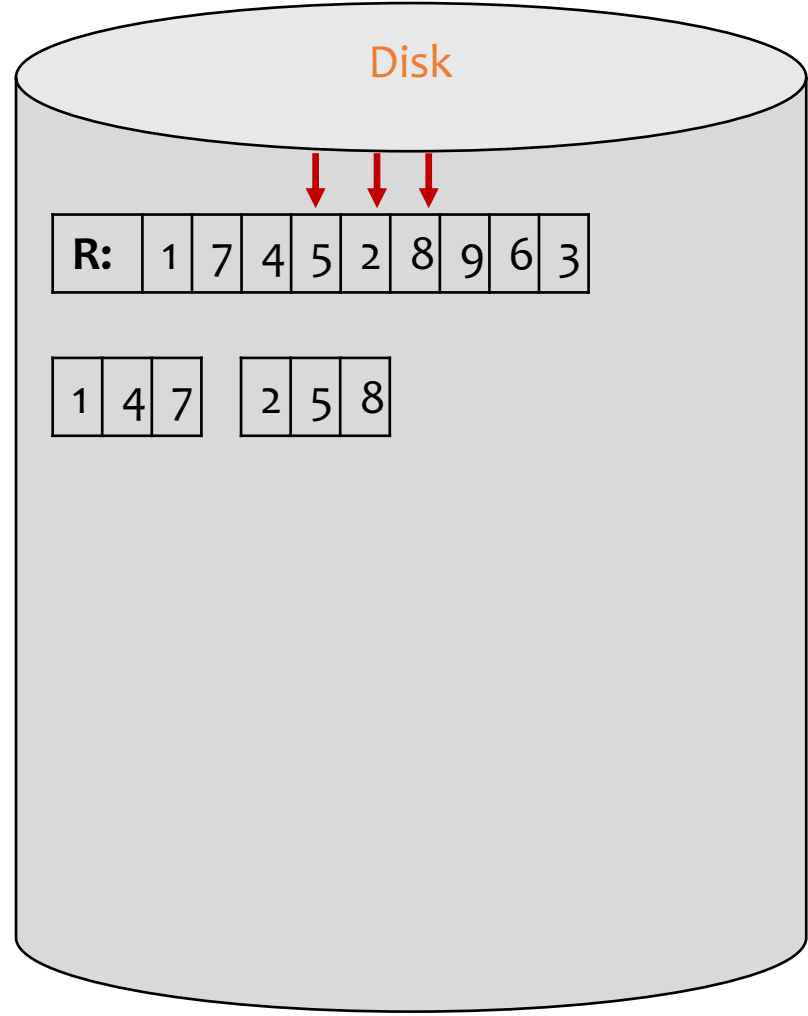
Arrows indicate the
blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

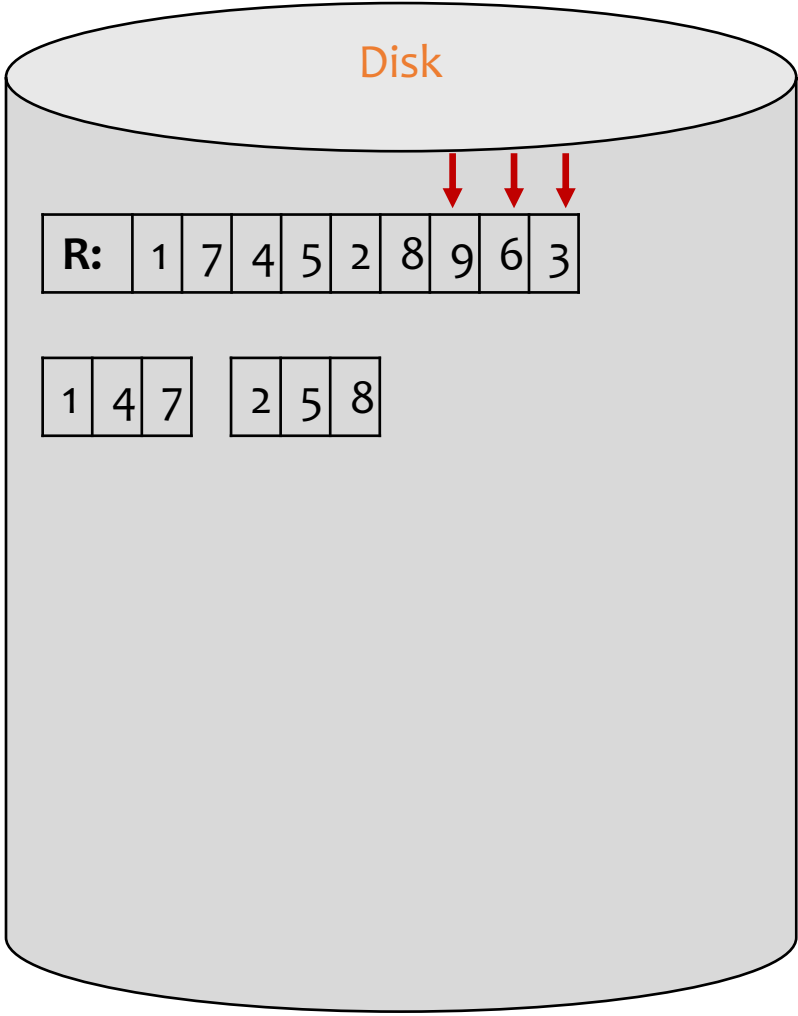
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

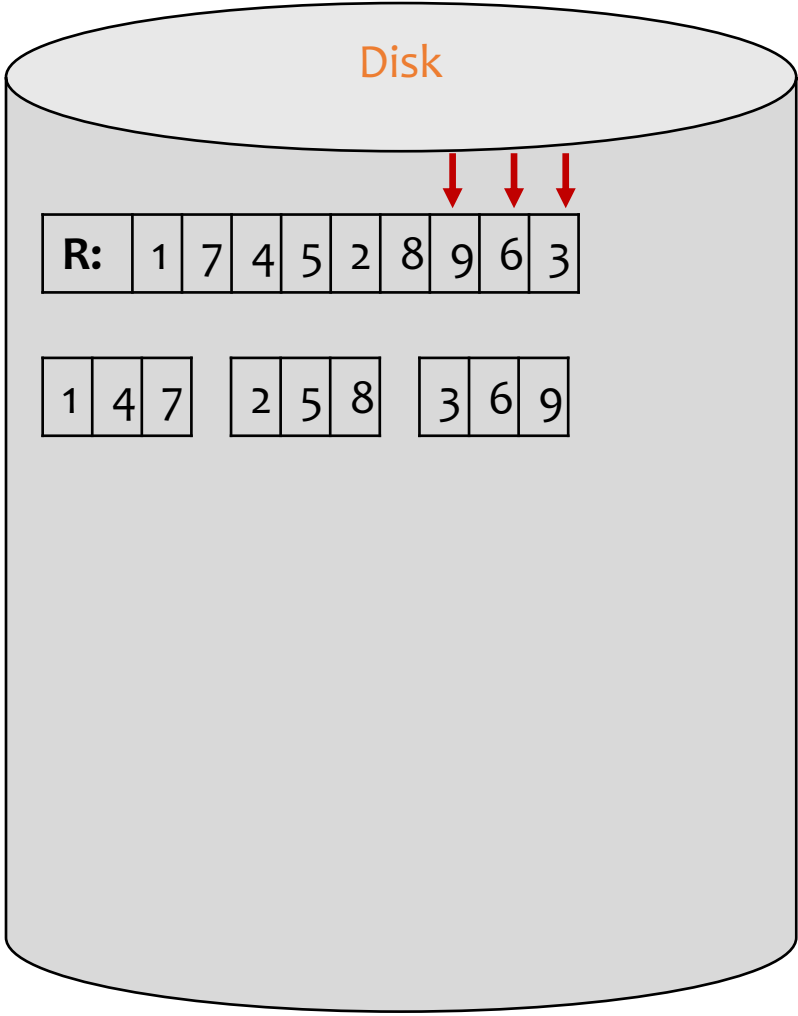
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

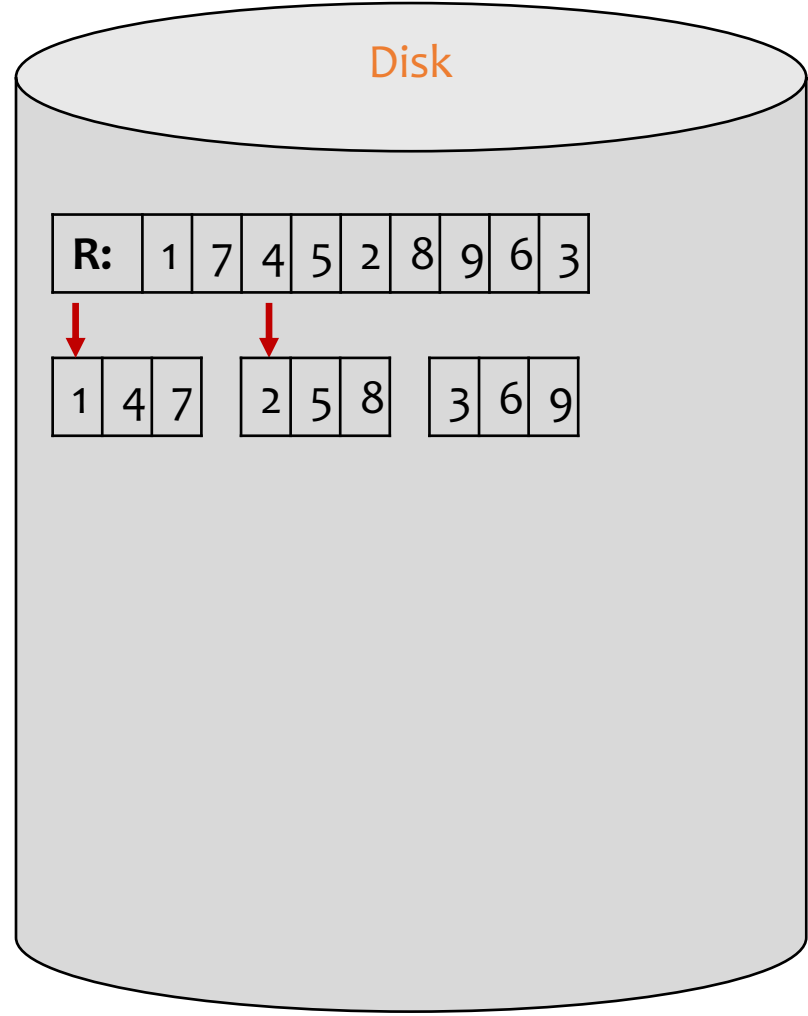
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

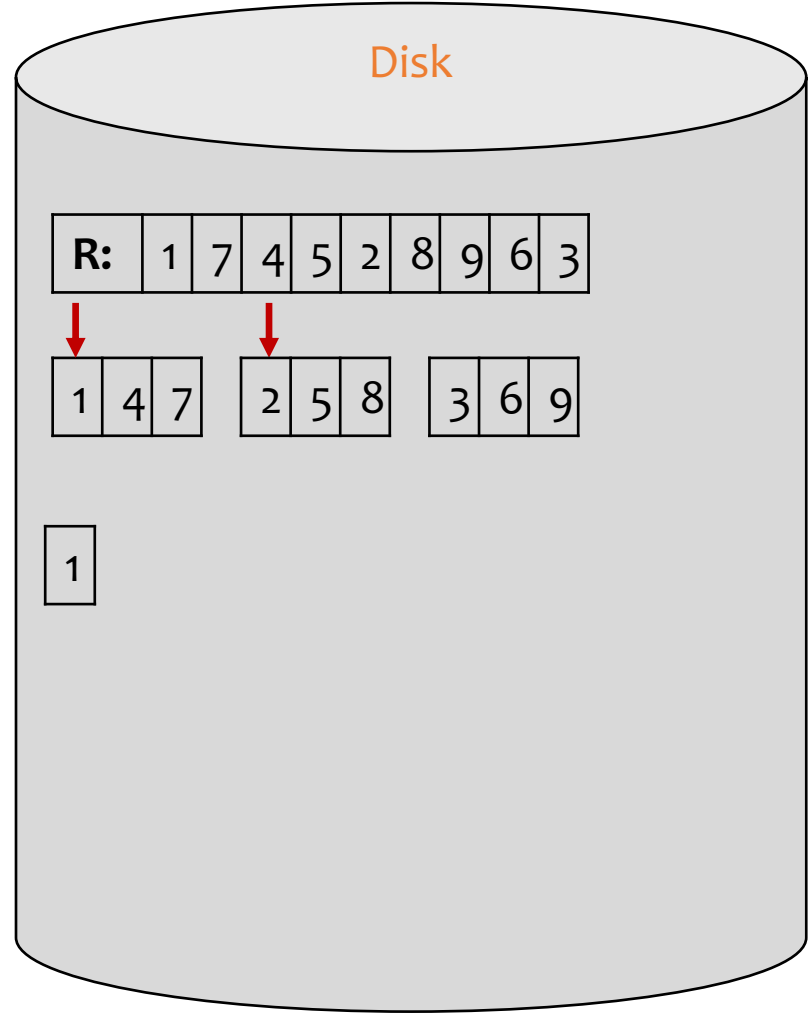
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

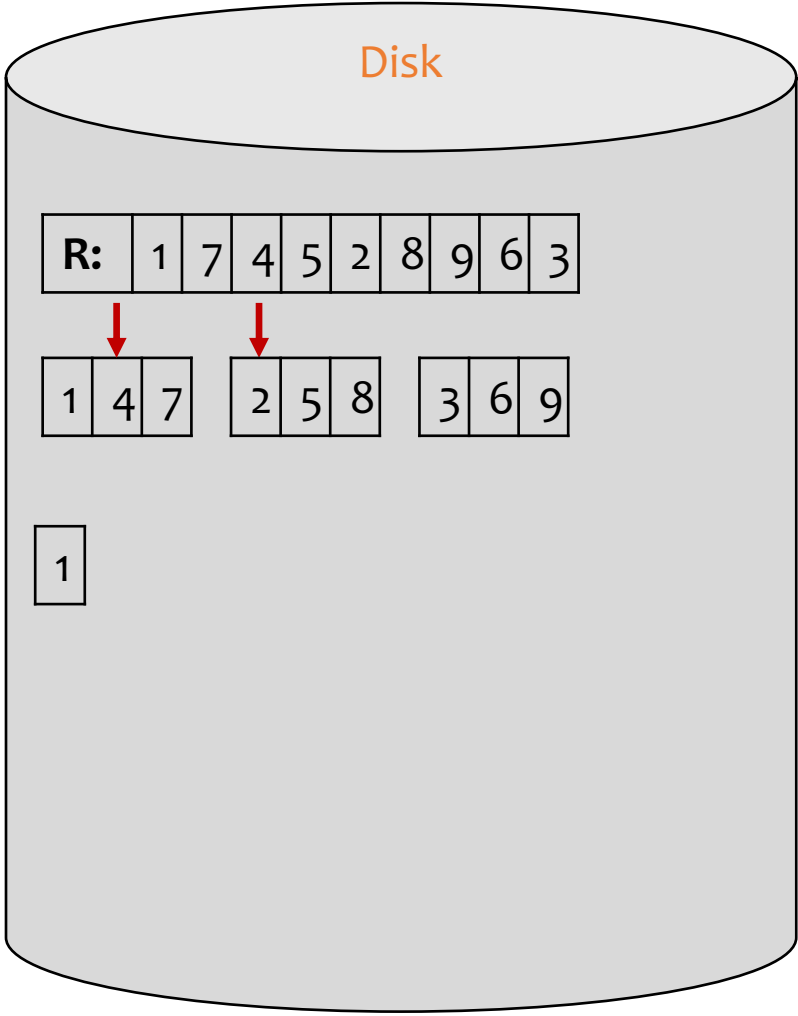
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

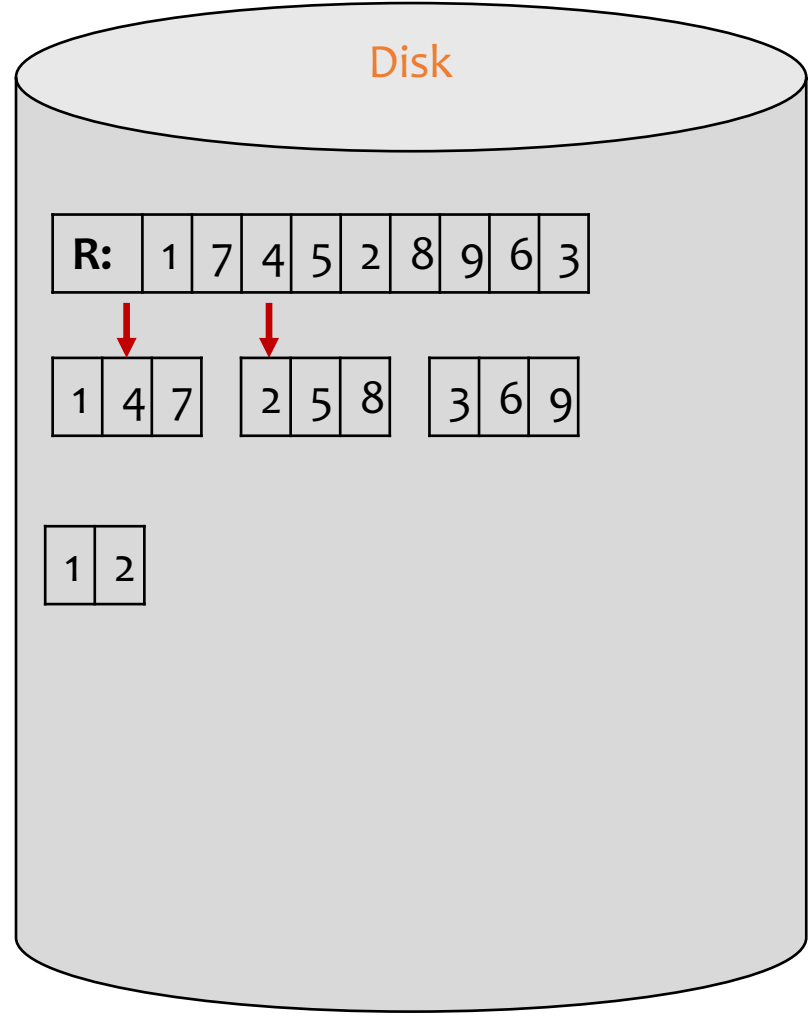
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

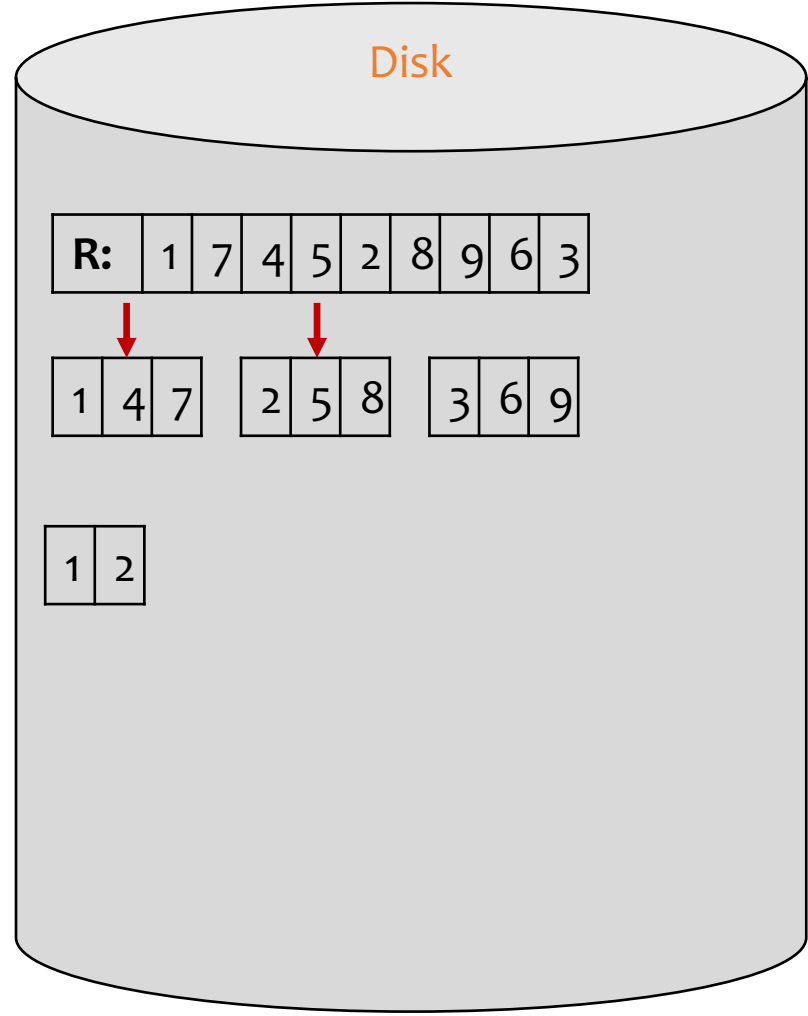
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

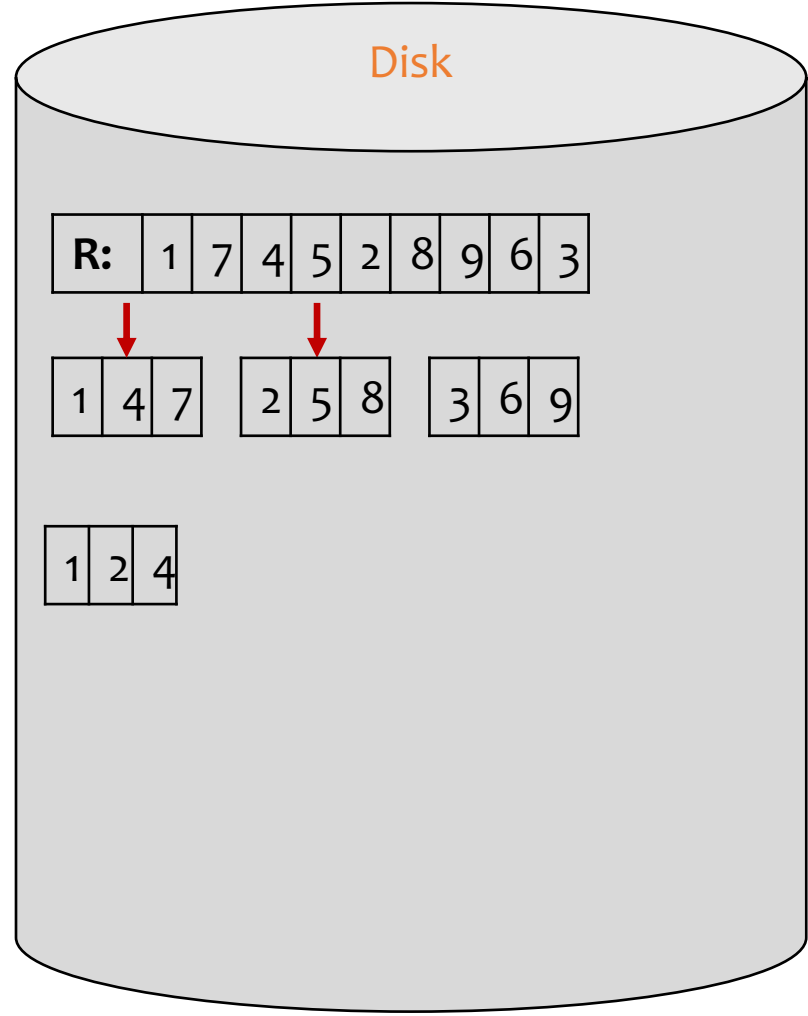
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

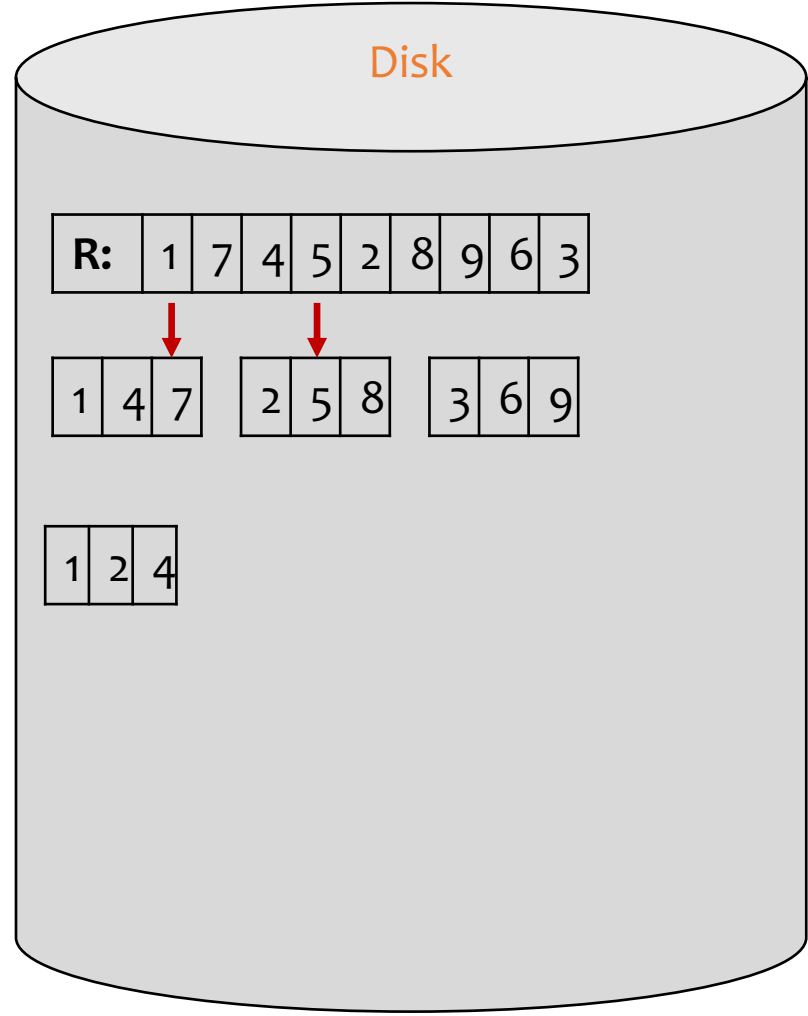
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

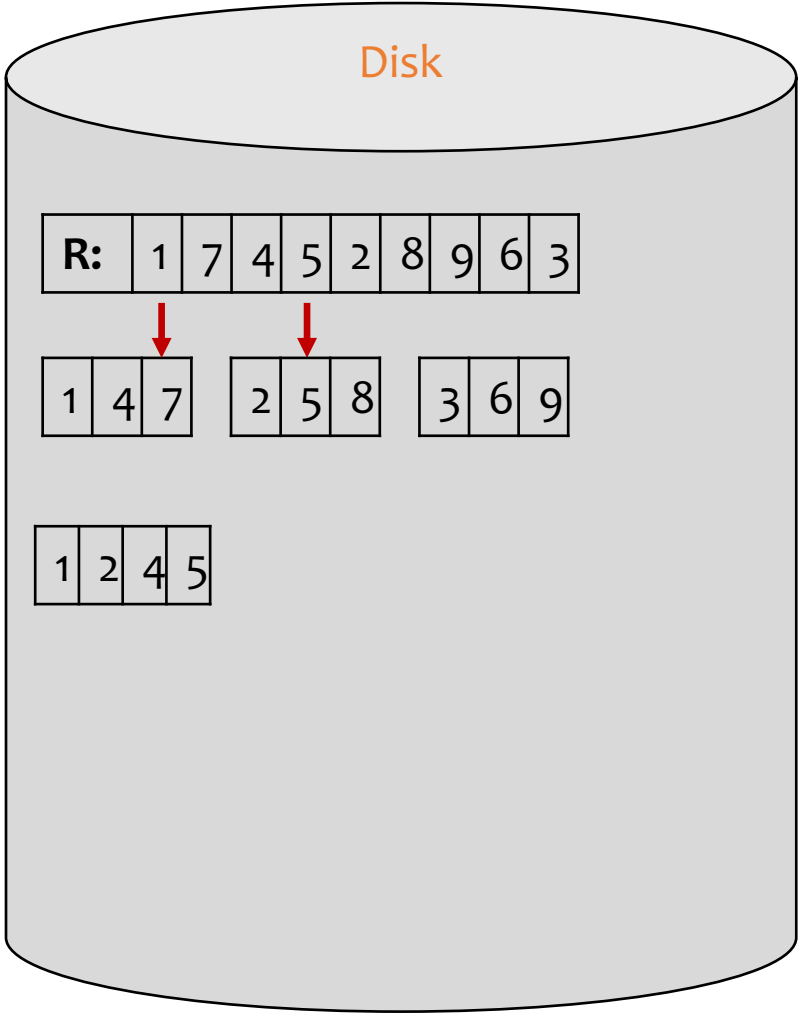
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

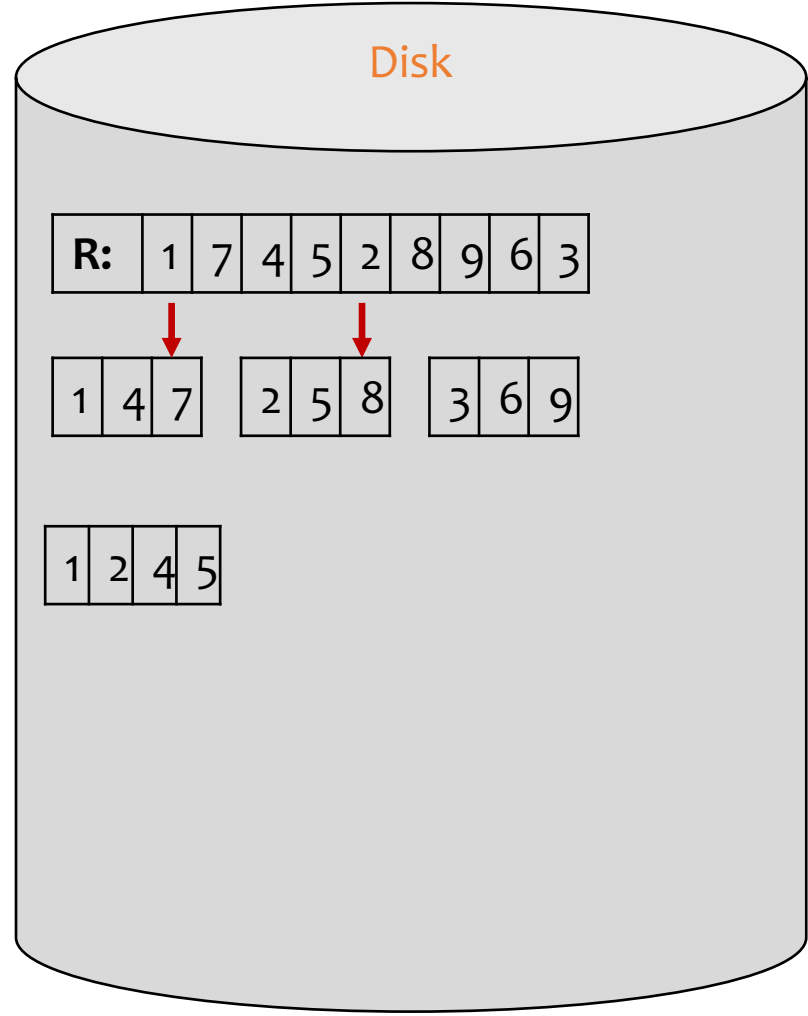
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

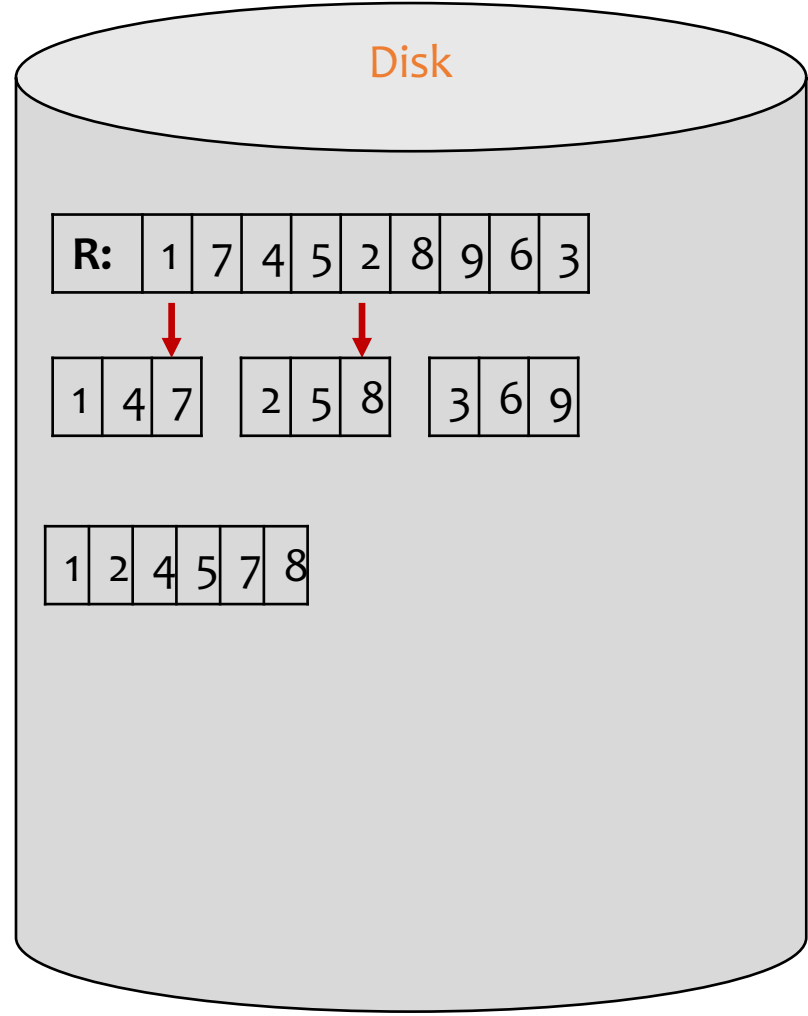
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

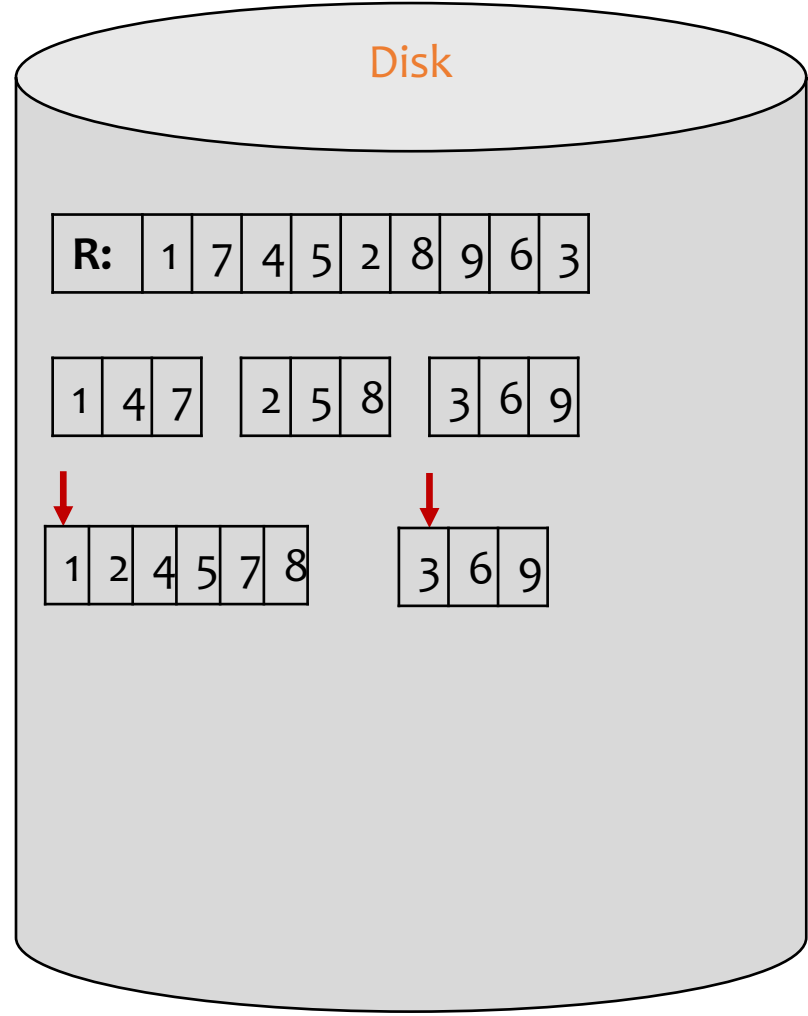
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
- Phase 2 (final)

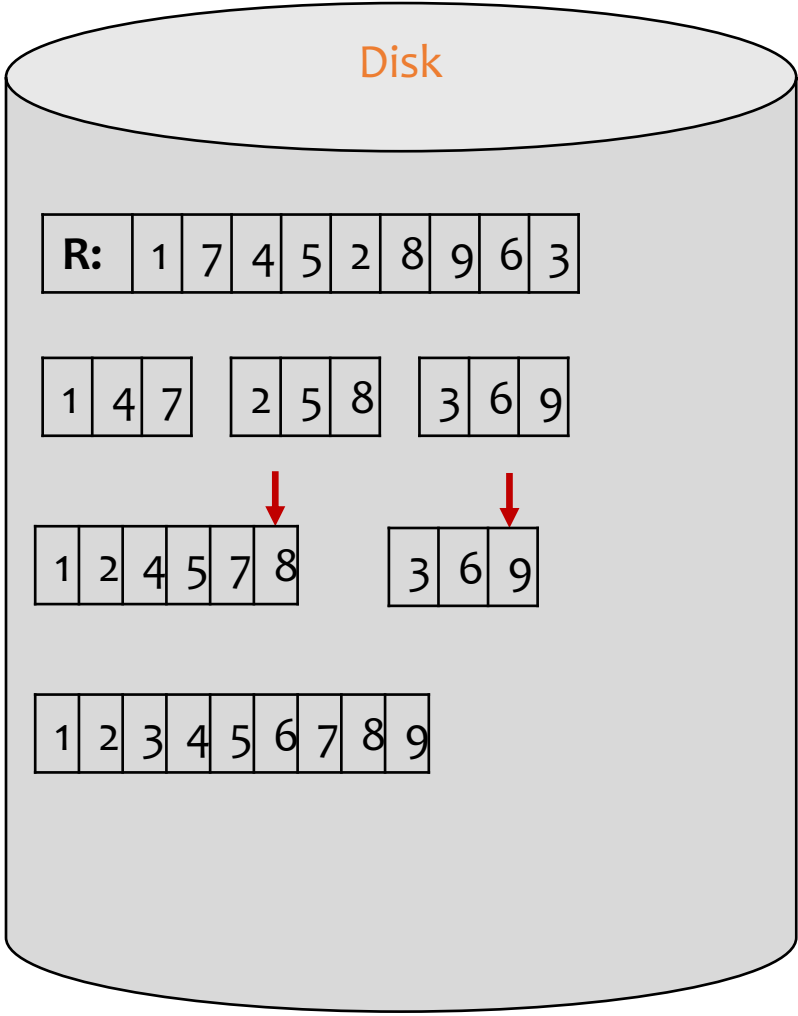
Arrows indicate the blocks in memory



Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
- Phase 2 (final)

Arrows indicate the blocks in memory



Analysis

- **Phase 0**: read M blocks of R at a time, sort them, and write out a level-0 run
 - There are $\left\lceil \frac{B(R)}{M} \right\rceil$ level-0 sorted runs
- **Phase i** : merge $(M - 1)$ level- $(i - 1)$ runs at a time, and write out a level- i run
 - $(M - 1)$ memory blocks for input, 1 to buffer output
 - The number of level- i runs = $\left\lceil \frac{\text{number of level-}(i-1) \text{ runs}}{M-1} \right\rceil$
 - $\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil$ number of such phases
 - **Final pass** produces one sorted run

I/O cost is $2 \cdot B(R)$

I/O cost is $2 \cdot B(R)$
times # of phases

Subtract $B(R)$ for the final pass

Performance of external merge sort

- I/O's

- $2B(R) \cdot \left(1 + \left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil\right) - B(R)$

- Roughly, this is $O(B(R) \times \log_M B(R))$

- Memory requirement: M (as much as possible)

Case study (optional):

- System requirements:
 - Each disk/memory block can hold up to 10 rows (from any table);
 - All tables are stored compactly on disk (10 rows per block);
 - 8 memory blocks are available for query processing: $M=8$
- Database:
 - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
 - |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
 - #of blocks: $B(\text{User})=1000/10=100$; $B(\text{Group})=100/10=10$;
 $B(\text{Member})=50000/10=5k$
- Q3: select * from User order by age asc;
 - I/O cost using external merge sort?

Case study (optional):

- System requirements:
 - Each disk/memory block can hold up to 10 rows (from any table);
 - All tables are stored compactly on disk (10 rows per block);
 - 8 memory blocks are available for query processing: $M=8$
- Database:
 - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
 - |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
 - #of blocks: $B(\text{User})=1000/10=100$; $B(\text{Group})=100/10=10$;
 $B(\text{Member})=50000/10=5k$
- Q3: select * from User order by age asc;
 - I/O cost using external merge sort?
 - Phase 0: read 8 blocks into memory at a time and sort it => $\text{ceil}(100/8)=13$ runs
 - Phase 1: merge 7 runs at a time => $\text{ceil}(13/7)=2$ runs
 - Phase 2: merge last 2 runs into a single run

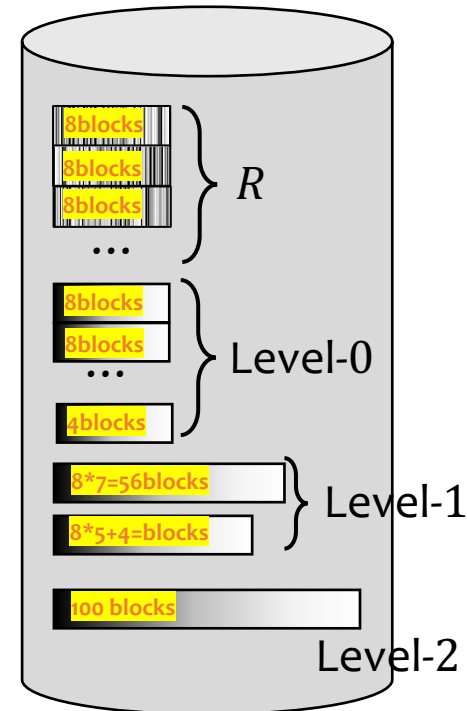
$$\text{Number of phases: } \left\lceil \log_{M-1} \left\lceil \frac{B(\text{User})}{M} \right\rceil \right\rceil + 1 = \left\lceil \log_{(8-1)} \left\lceil \frac{100}{8} \right\rceil \right\rceil + 1 = 3$$

Phase 0: read $B(\text{user})=100$ blocks, write $B(\text{User})=100$ blocks (temporary result)

Phase 1: read $B(\text{user})=100$ blocks, write $B(\text{User})=100$ blocks (temporary result)

Phase 2: read $B(\text{user})=100$ blocks, write $B(\text{User})=100$ blocks (final result, don't count)

$$\text{Total: } 2B(\text{User}) * 3 - B(\text{User}) = 5B(\text{user}) = 500$$



Operators That Use Sorting

- Pure Sort: e.g., ORDER BY
- Set Union, Difference, Intersection, or Join of R and S (next slide): When the join condition is an equality condition e.g., $R.A = S.B$,
 - All can be implemented by walking relations “in tandem” as in the merge step of merge sort.
- DISTINCT
- Group-By-and-Aggregate: Exercise: Think about how you can implement group-by-and-aggregate with sorting?

Sort-merge join

$$R \bowtie_{R.A=S.B} S$$

- Sort R and S by their join attributes; then merge
 - r, s = the first tuples in sorted R and S
 - Repeat until one of R and S is exhausted:
 - If $r.A > s.B$
 - then s = next tuple in S
 - else if $r.A < s.B$
 - then r = next tuple in R
 - else output all matching tuples, and
 r, s = next in R and S
- I/O's: **sorting** + $O(B(R) + B(S))$
 - In most cases (e.g., join of key and foreign key)
 - Worst case is $B(R) \cdot B(S)$: everything joins

Example of merge join

R:

- $r_1.A = 1$
- $r_2.A = 3$
- $r_3.A = 3$
- $r_4.A = 5$
- $r_5.A = 7$
- $r_6.A = 7$
- $r_7.A = 8$

S:

- $s_1.B = 1$
- $s_2.B = 2$
- $s_3.B = 3$
- $s_4.B = 3$
- $s_5.B = 8$

$R \bowtie_{R.A=S.B} S$:

- r_1s_1
- r_2s_3
- r_2s_4
- r_3s_3
- r_3s_4
- r_7s_5

Summary

- Scan
 - Selection, duplicate-preserving projection, nested-loop join
- Index
 - Selection, index nested-loop join
- Sort
 - External merge sort, sort-merge-join
- Hash (Optional)



Optional (won't
be tested)