Query Processing

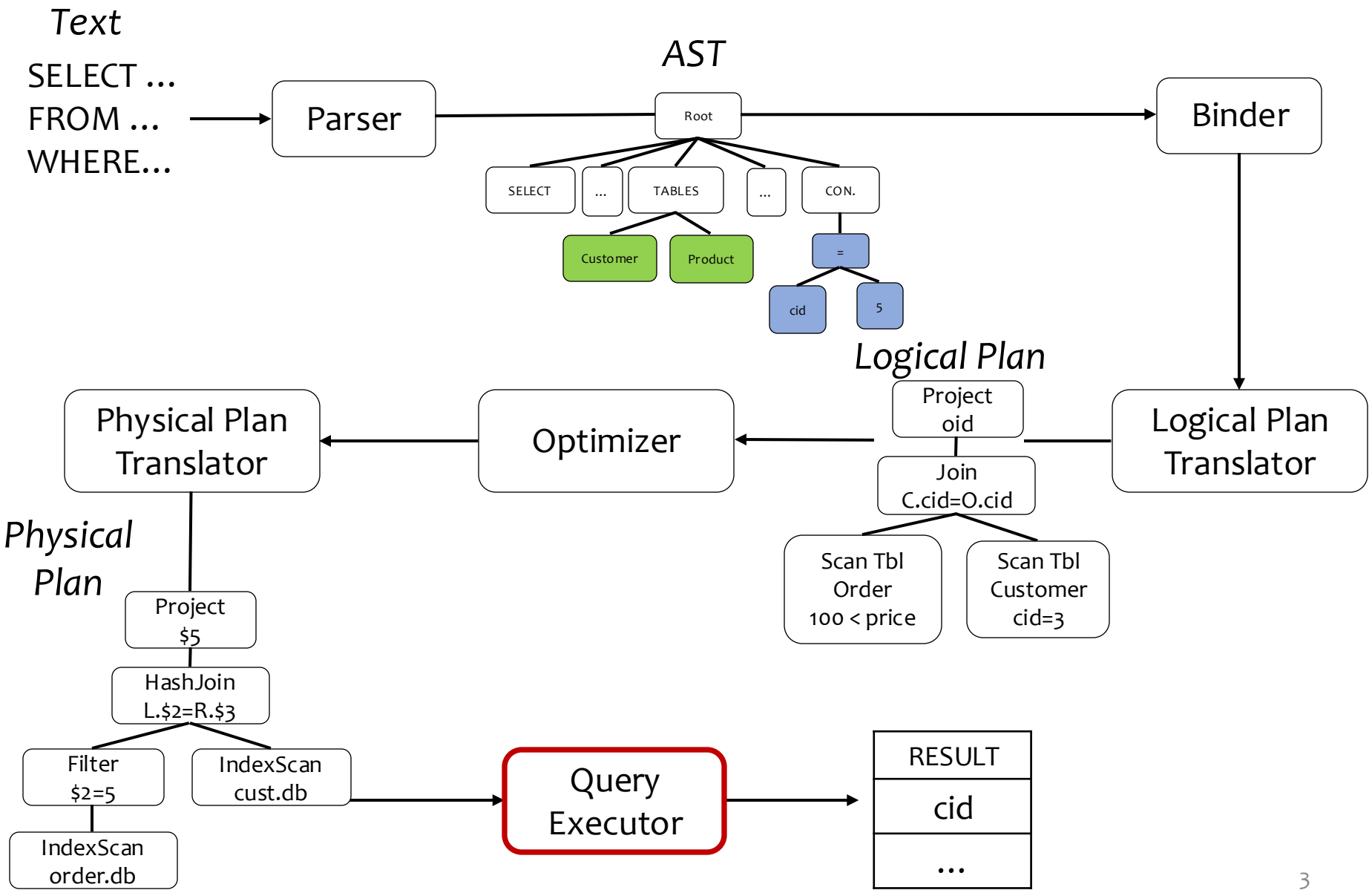
CS348

Instructor: Sujaya Maiyya

Announcements

- Midterm: On Nov 7th
  - Everything until lecture 12
    - RA, SQL, DB Design (ER + design theory)

Overview


Text
SELECT …
FROM …
WHERE…
Parser
AST
Root
SELECT
…
TABLES
…
CON.
Customer
Product
=
cid
5
Binder
Logical Plan
Project
oid
Join
C.cid=O.cid
Scan Tbl
Order
100 < price
Scan Tbl
Customer
cid=3
Logical Plan
Translator
Optimizer
Physical Plan
Translator
Physical
Plan
Project
$5
HashJoin
L.$2=R.$3
Filter
$2=5
IndexScan
order.db
IndexScan
cust.db
Query
Executor
RESULT
cid
…

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

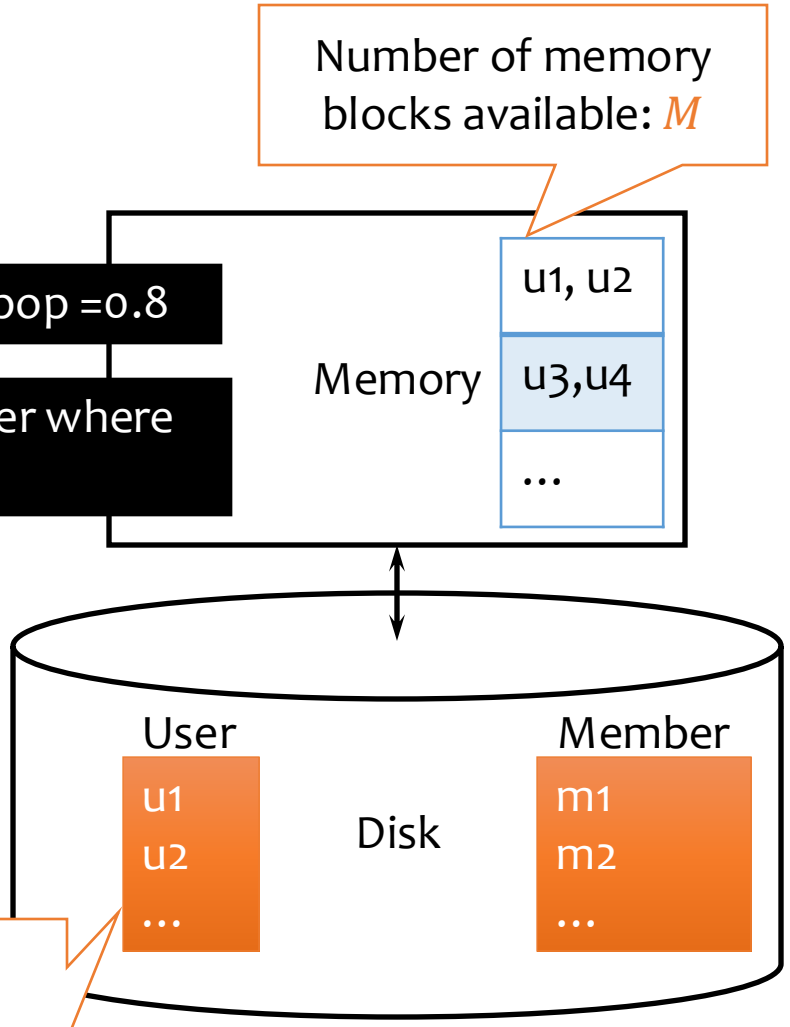
- Best choice depends on the situation
  - Implement all alternatives?
  - Let the query optimizer choose at run-time (next lecture)

Outline

- Scan
- Index
- Sort
- Hash

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

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



Number of rows for a table $|Users|$

Number of disk blocks for a table

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

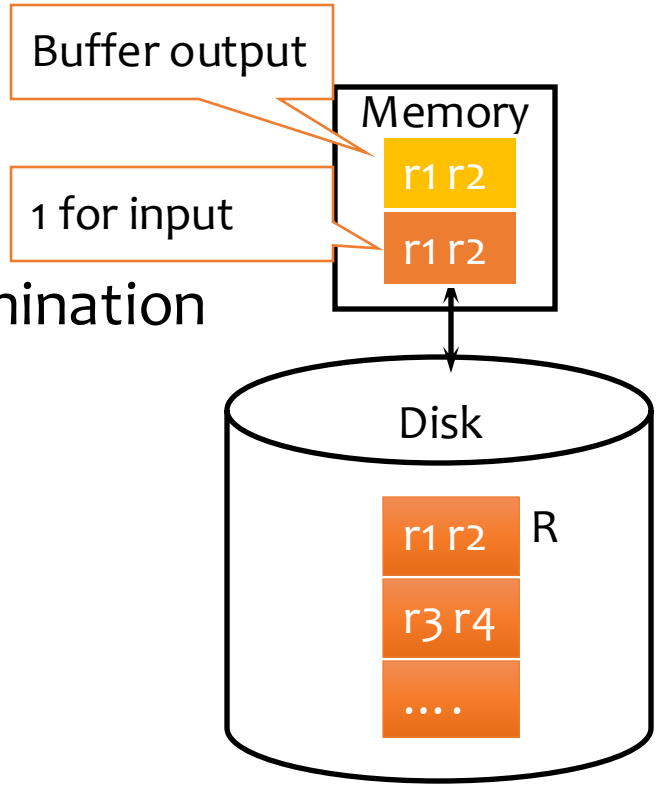
Notation

- 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: $B(R) + |R| \cdot B(S)$

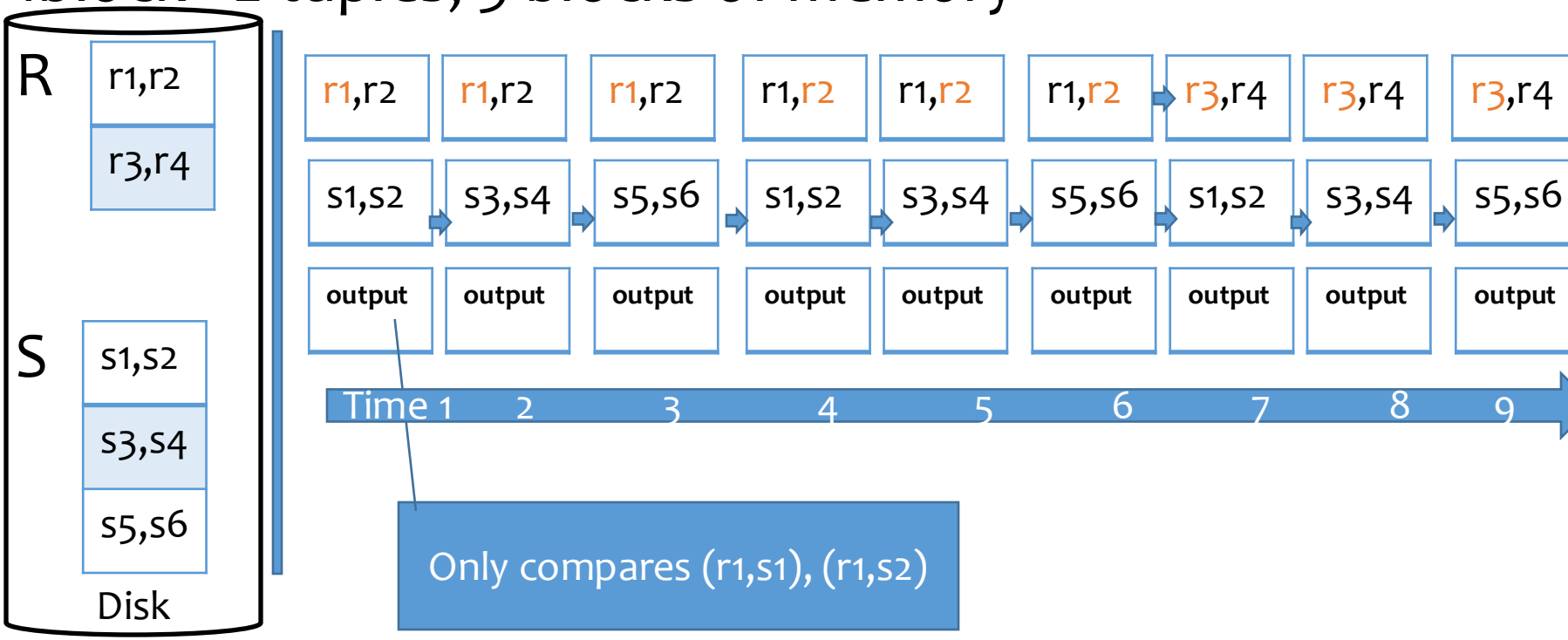
Blocks of R are moved into memory only once

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:

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

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: $B(R) + B(R) \cdot B(S)$

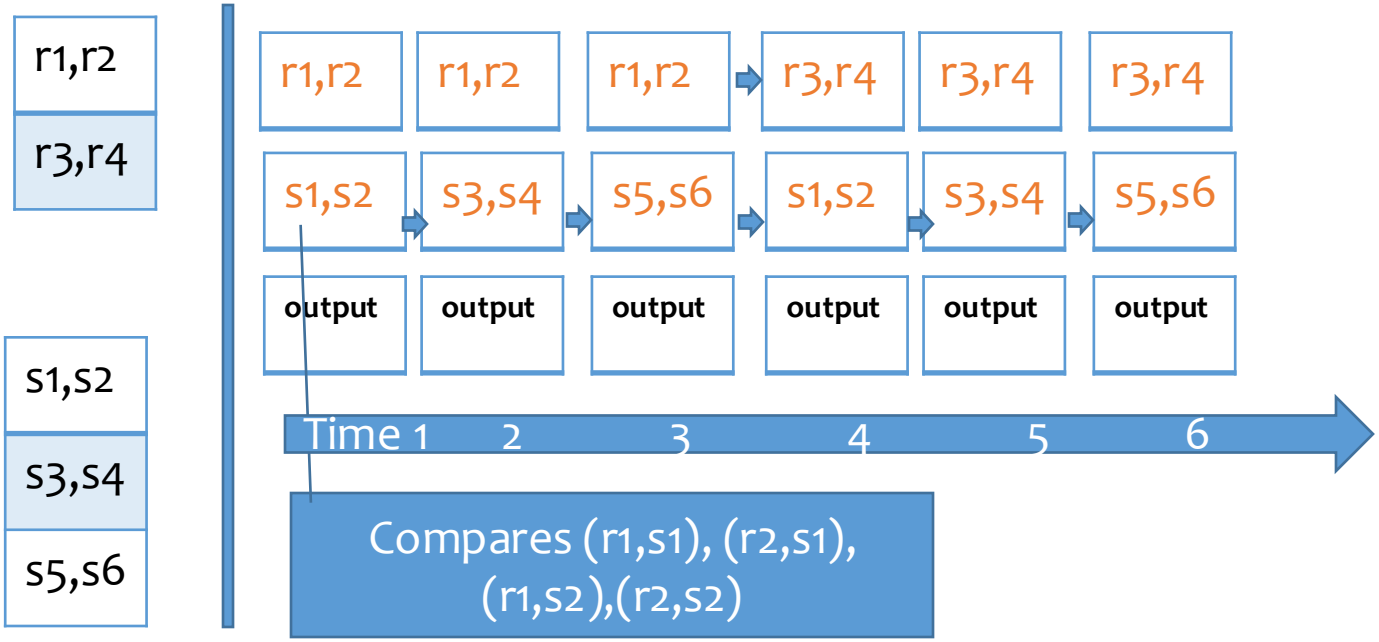
Blocks of R are moved into memory only once

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
- R
- S



- Number of I/O:

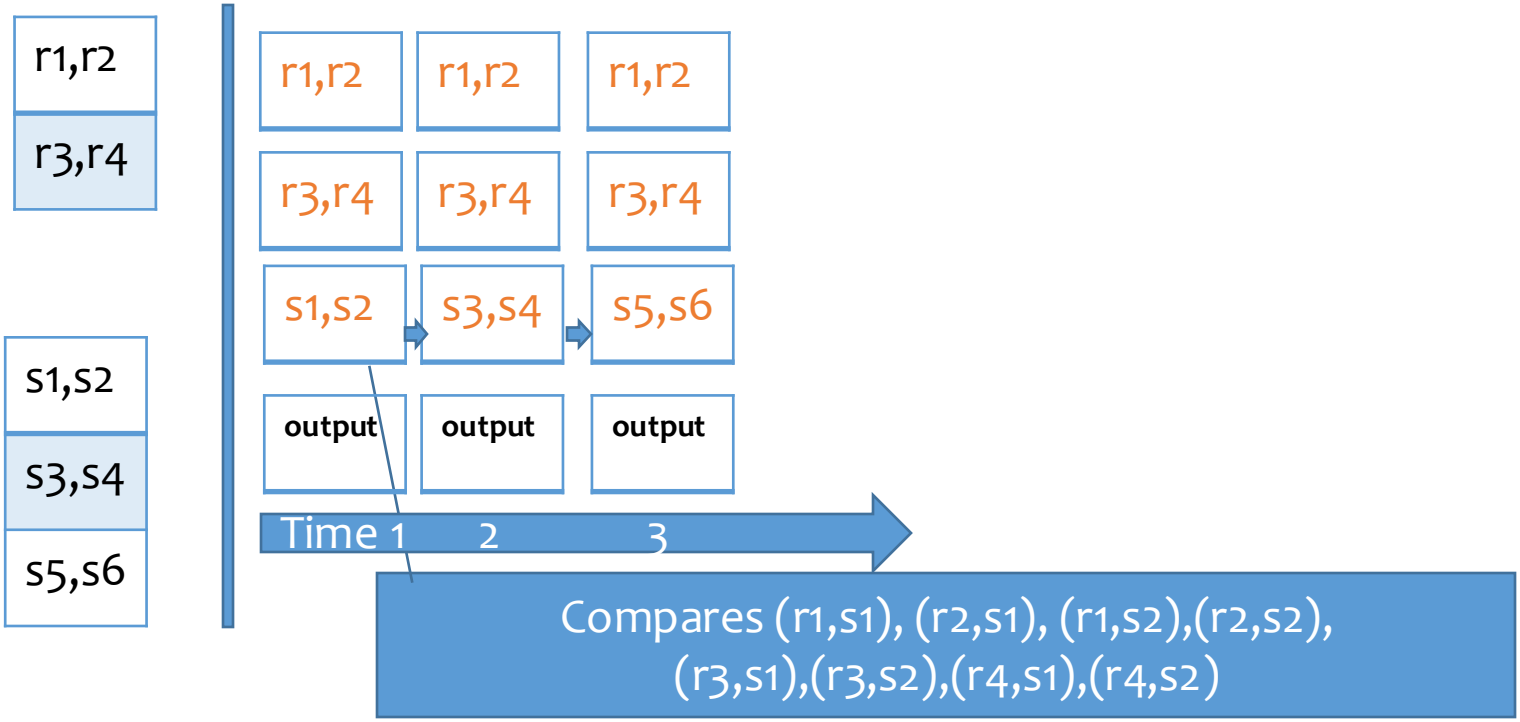
B(R) + B(R)* B(S) = 2 blocks + 2 * 3blocks = 8

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
- R
- S



- Number of I/O:
  B(R) + B(R)/(M-2)* B(S) = 2 blocks + 1 * 3blocks = 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(User)=1000/10=100; B(Group)=100/10=10; B(Member)=50000/10=5k
- Q1: select * from User where pop =0.8
  - I/O cost using table scan? $B(User) = 100$
- Q2: select * from User, Member where User.uid = Member.uid;
  - I/O cost using blocked-based nested loop join

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

Outline

- Scan
  - Selection, duplicate-preserving projection, nested-loop join
- Sort
  - External merge sort, sort-merge-join
- Hash
  - Hash join, point selection, group by and aggregations
- Index
  - Selection, index nested-loop join

Sorting-based algorithms

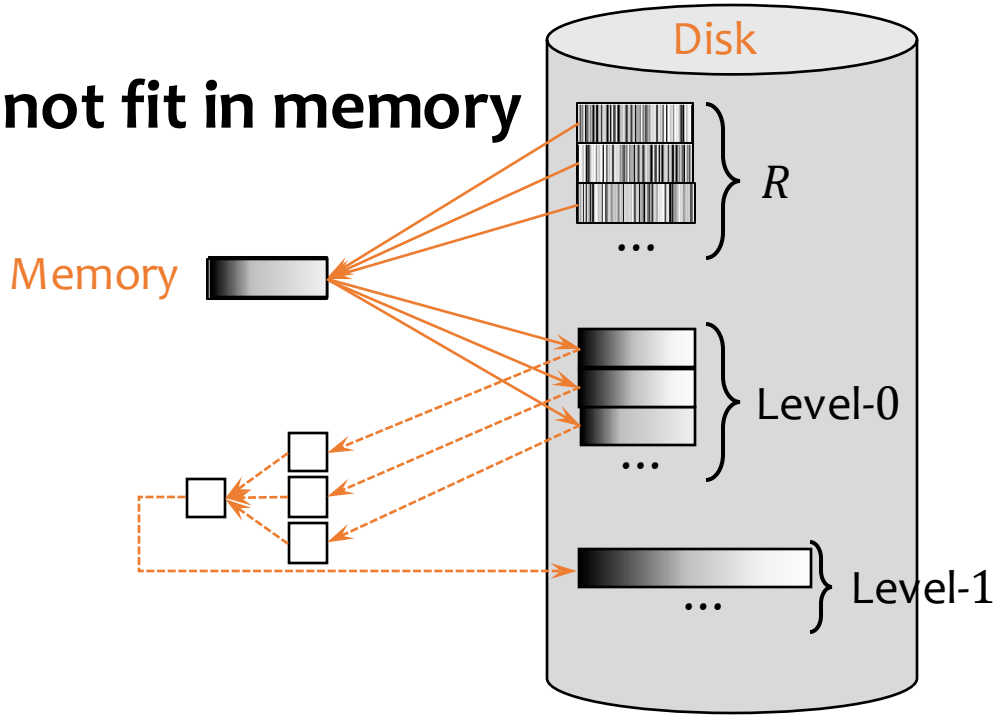
http://en.wikipedia.org/wiki/Mail_sorter#mediaviewer/File:Mail_sorting,1951.jpg

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

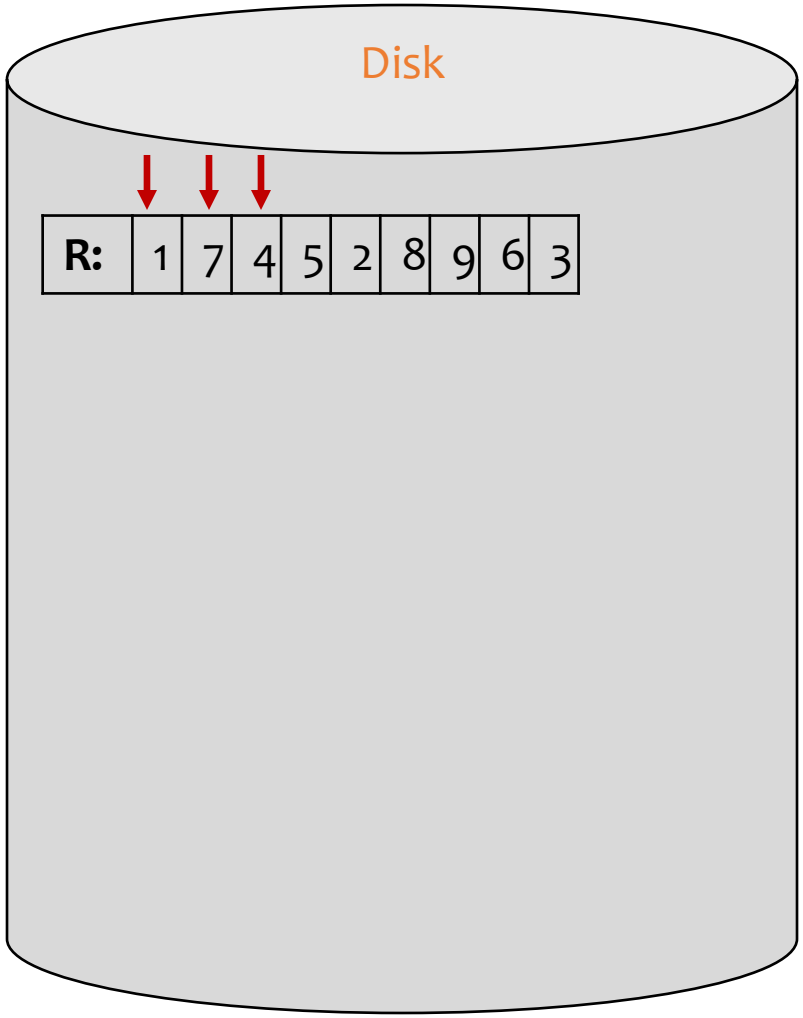
…

- 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

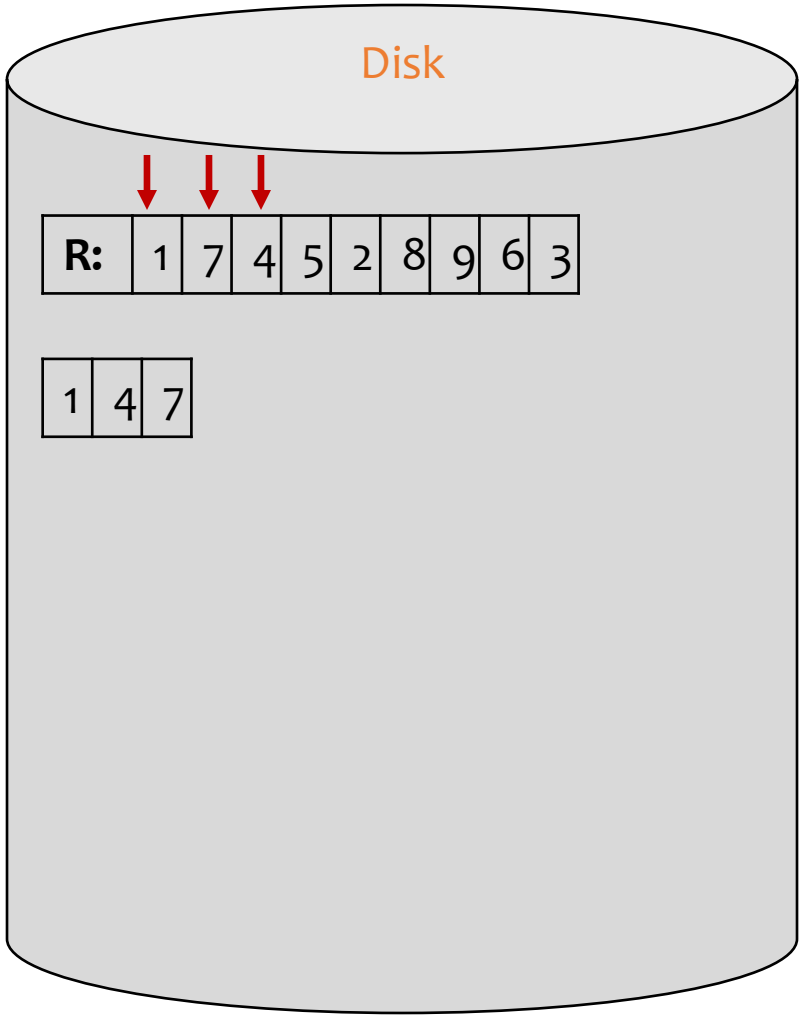
Arrows indicate the blocks in memory



Example

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

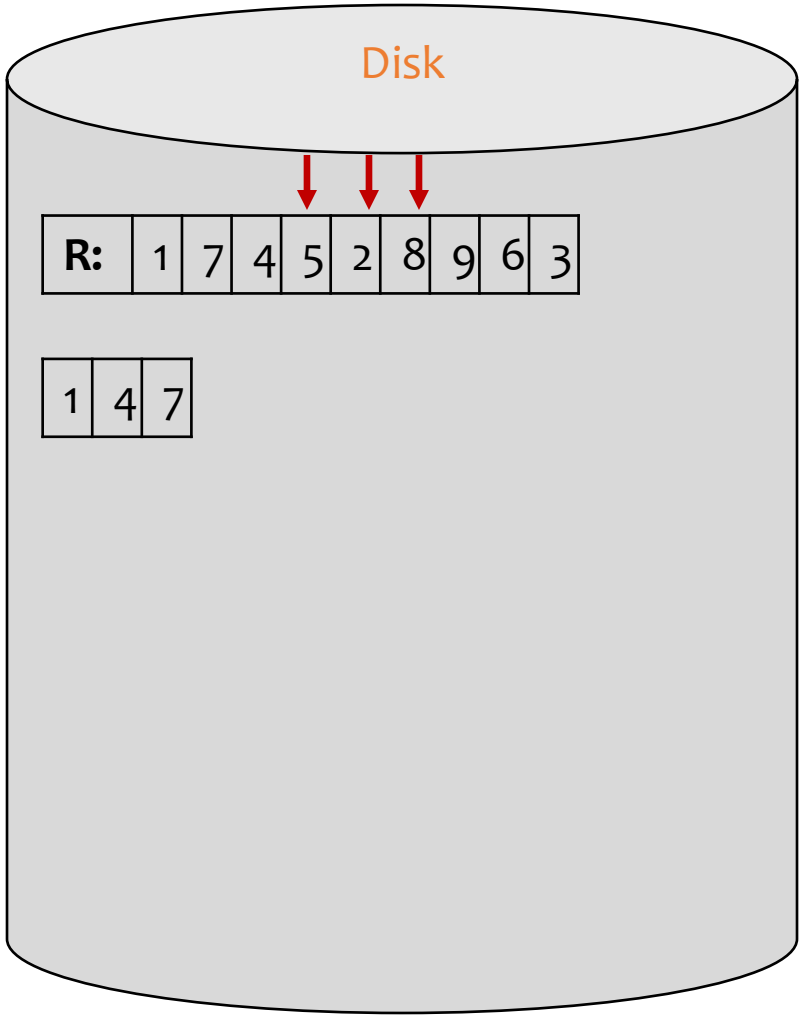
Arrows indicate the blocks in memory



Example

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

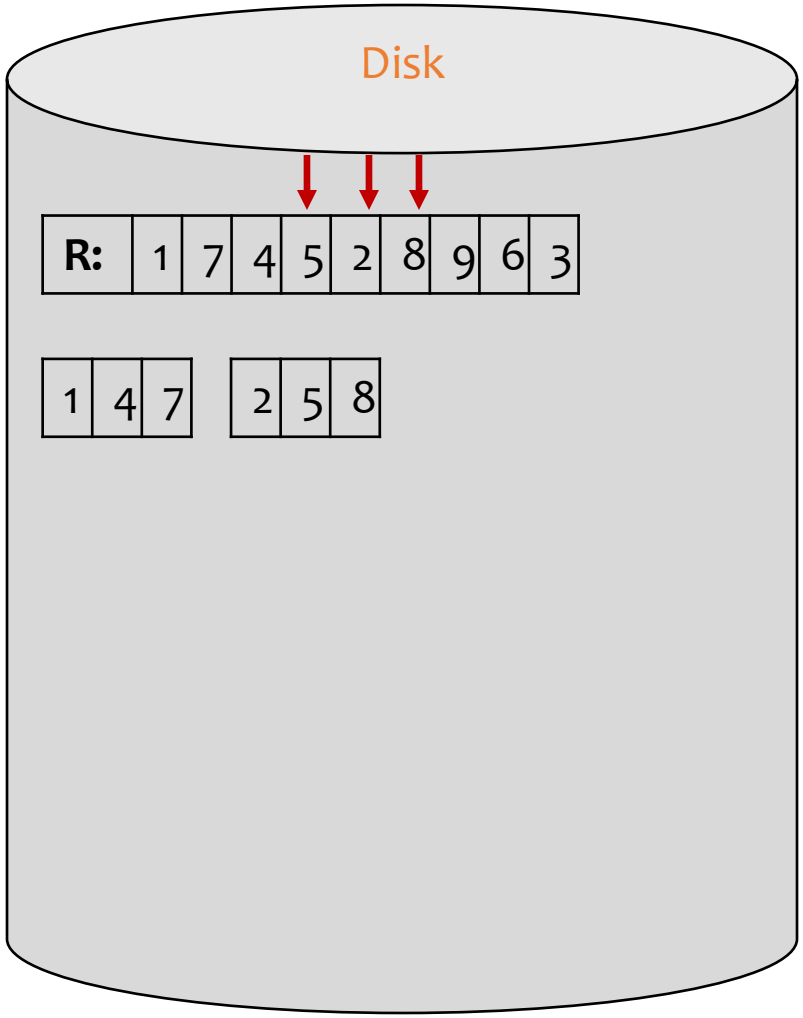
Arrows indicate the blocks in memory



Example

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

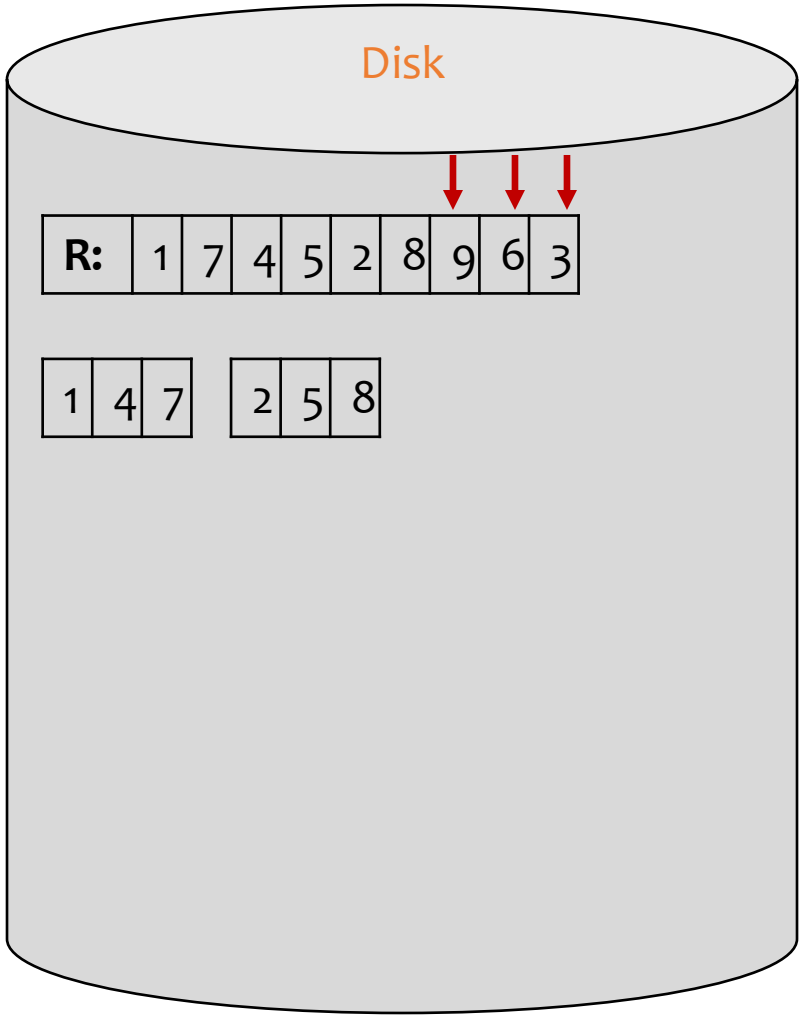
Arrows indicate the blocks in memory



Example

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

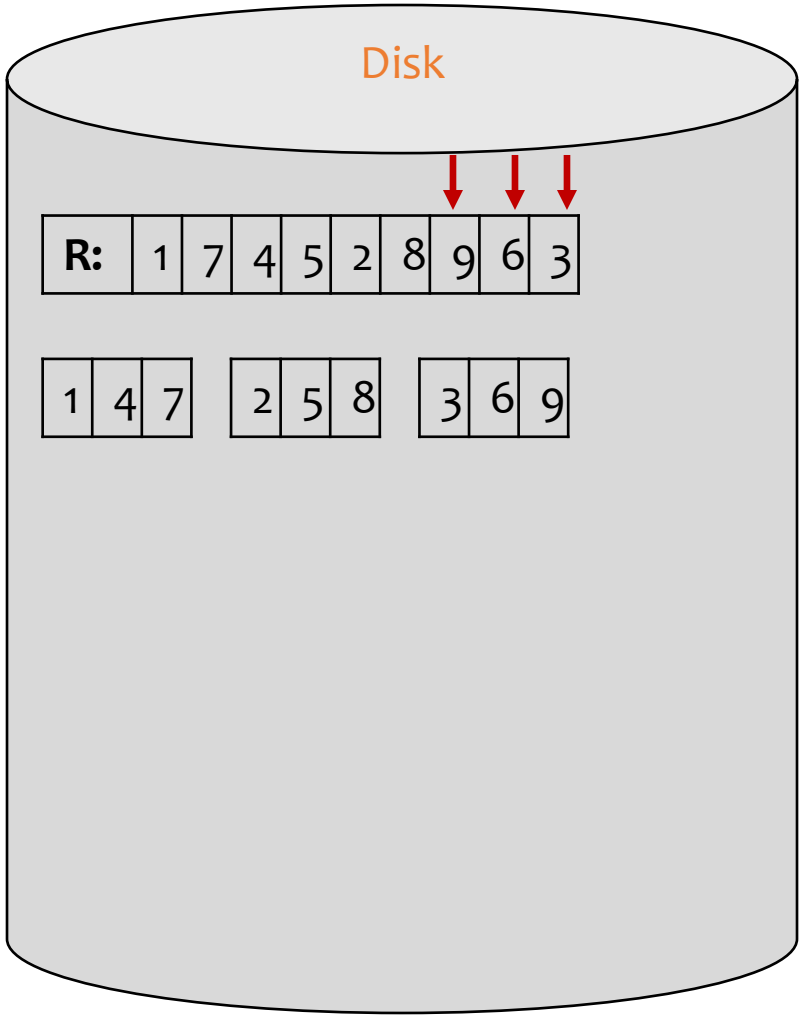
Arrows indicate the blocks in memory



Example

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

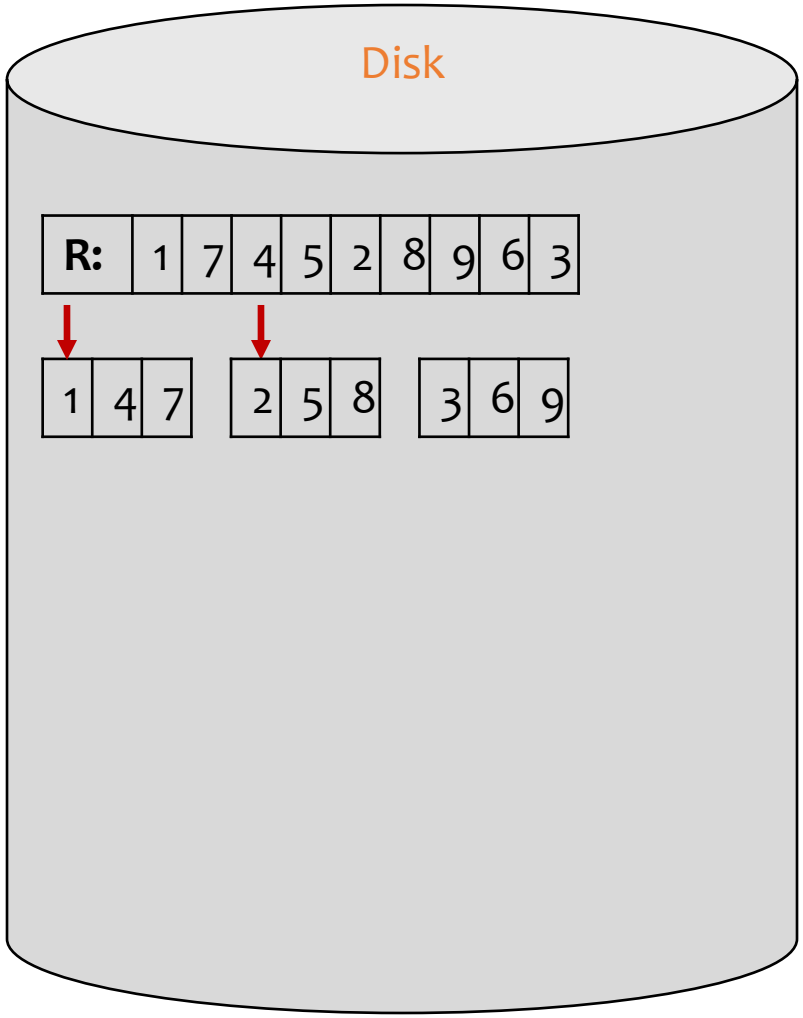
Arrows indicate the blocks in memory



Example

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

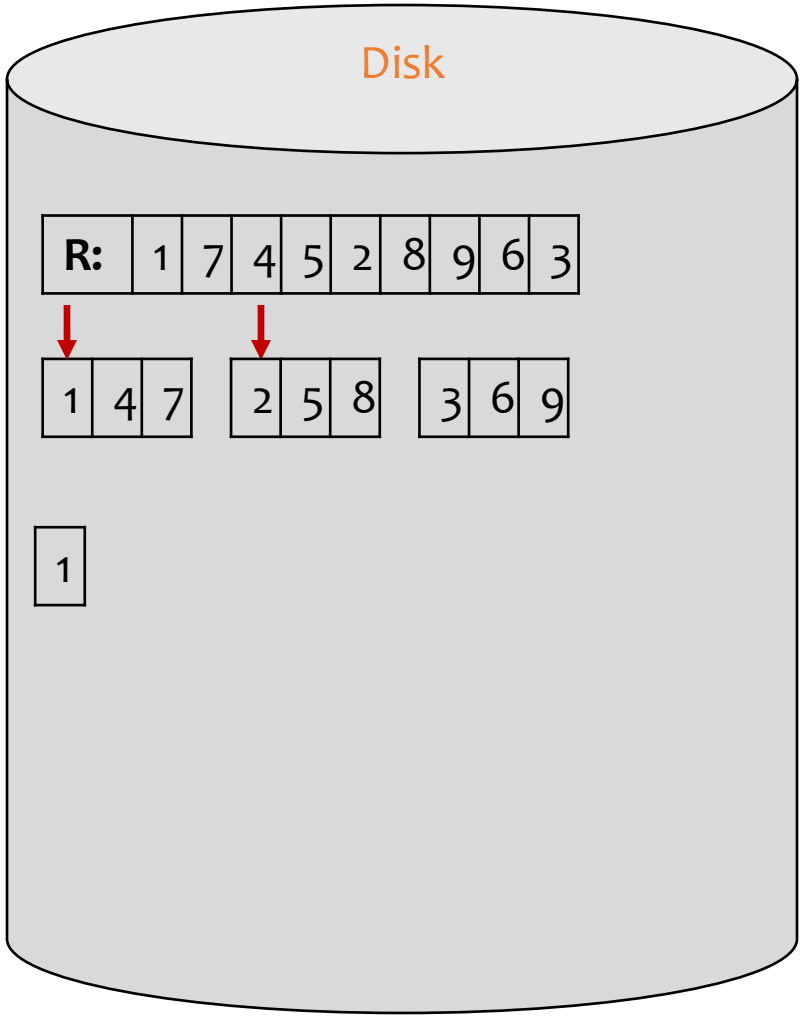
Arrows indicate the blocks in memory



Example

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

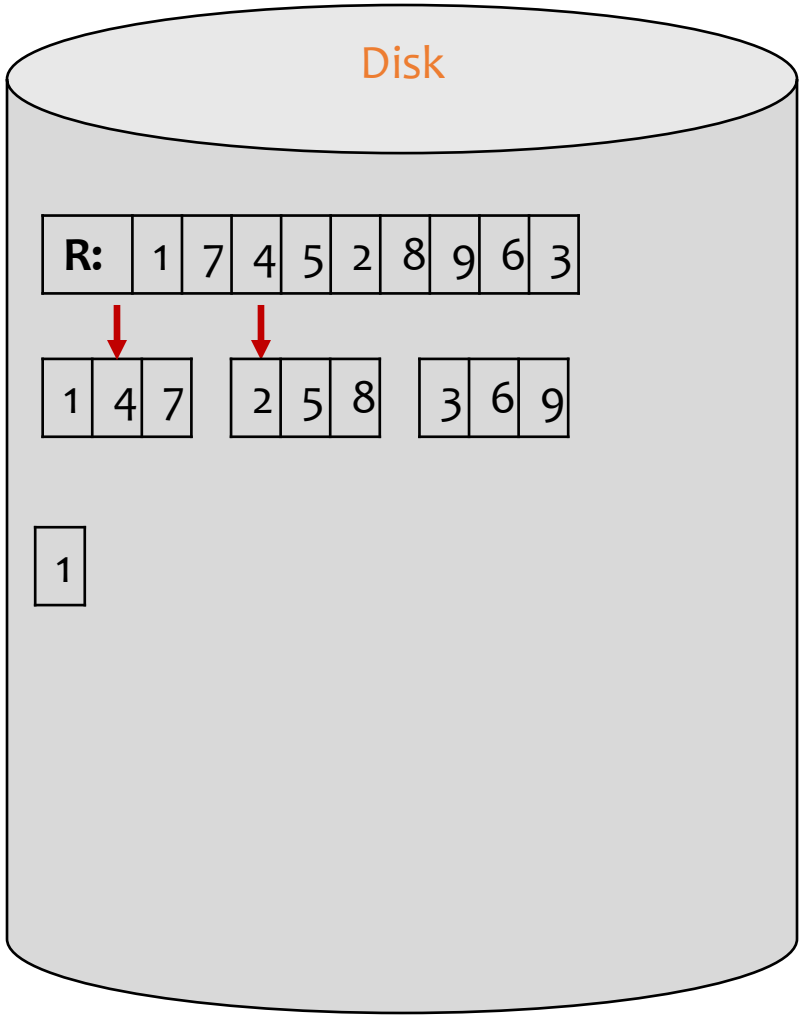
Arrows indicate the blocks in memory



Example

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

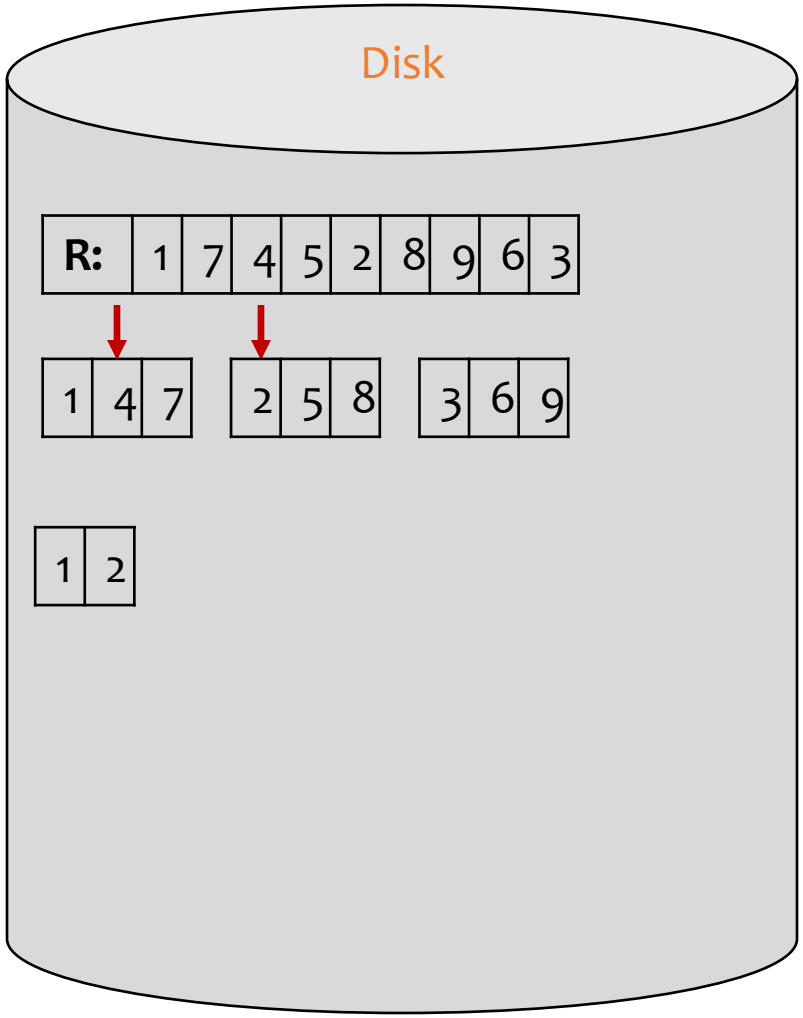
Arrows indicate the blocks in memory



Example

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

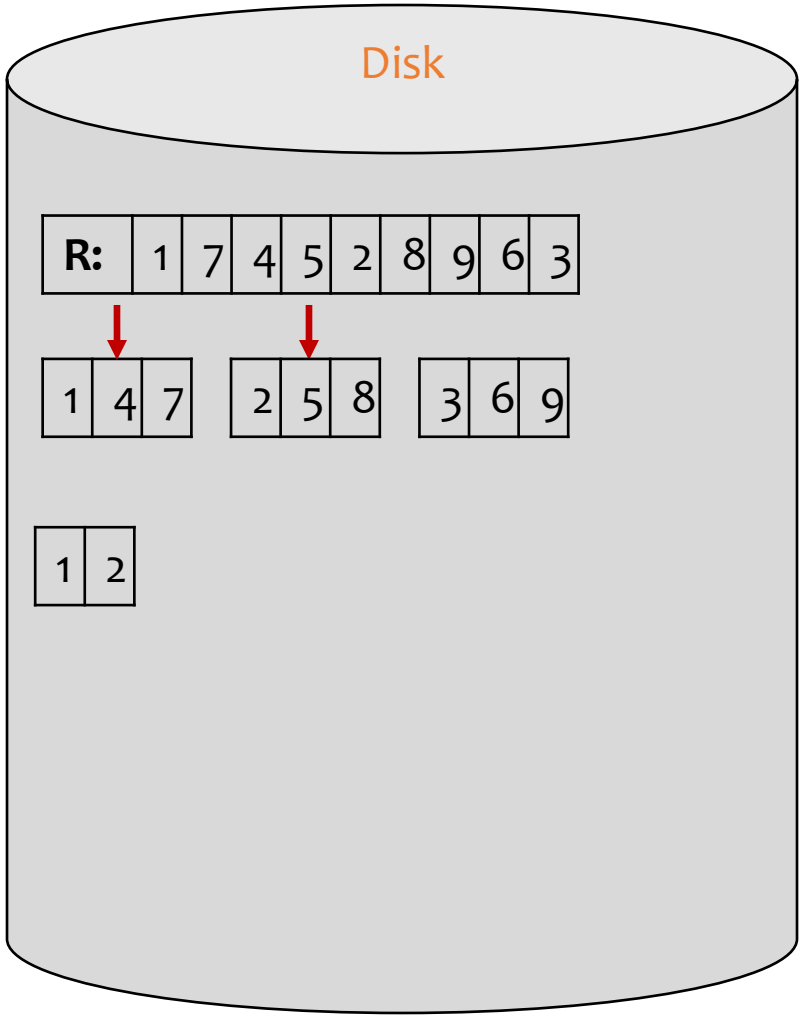
Arrows indicate the blocks in memory



Example

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

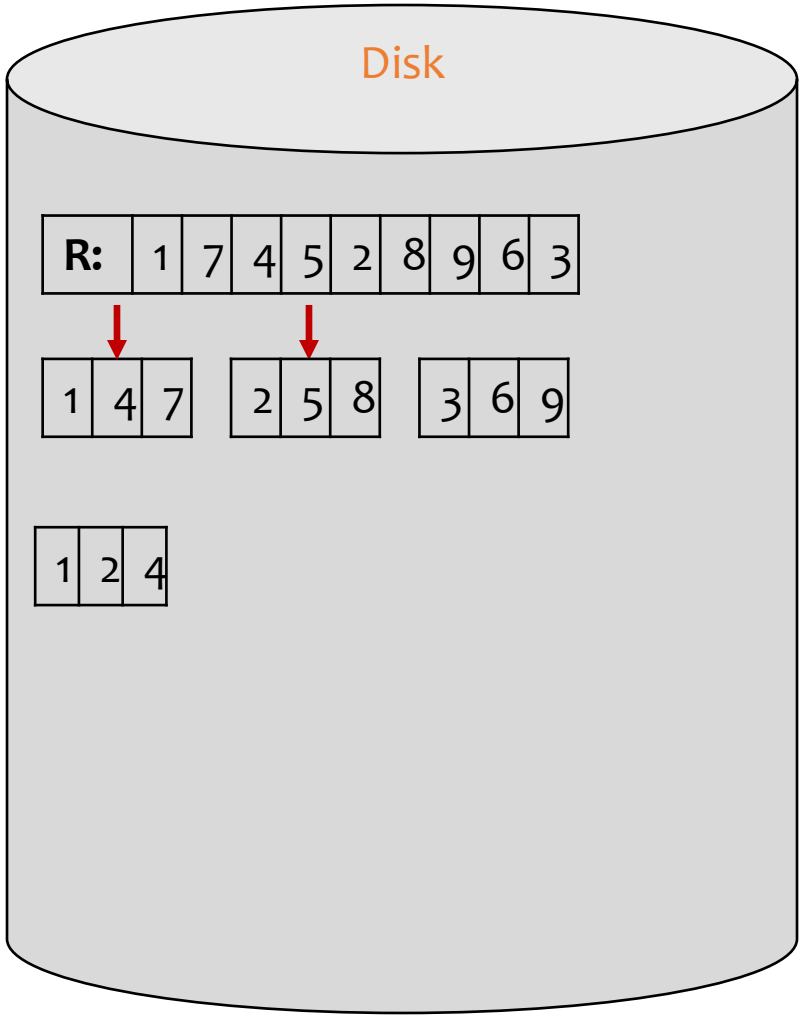
Arrows indicate the blocks in memory



Example

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

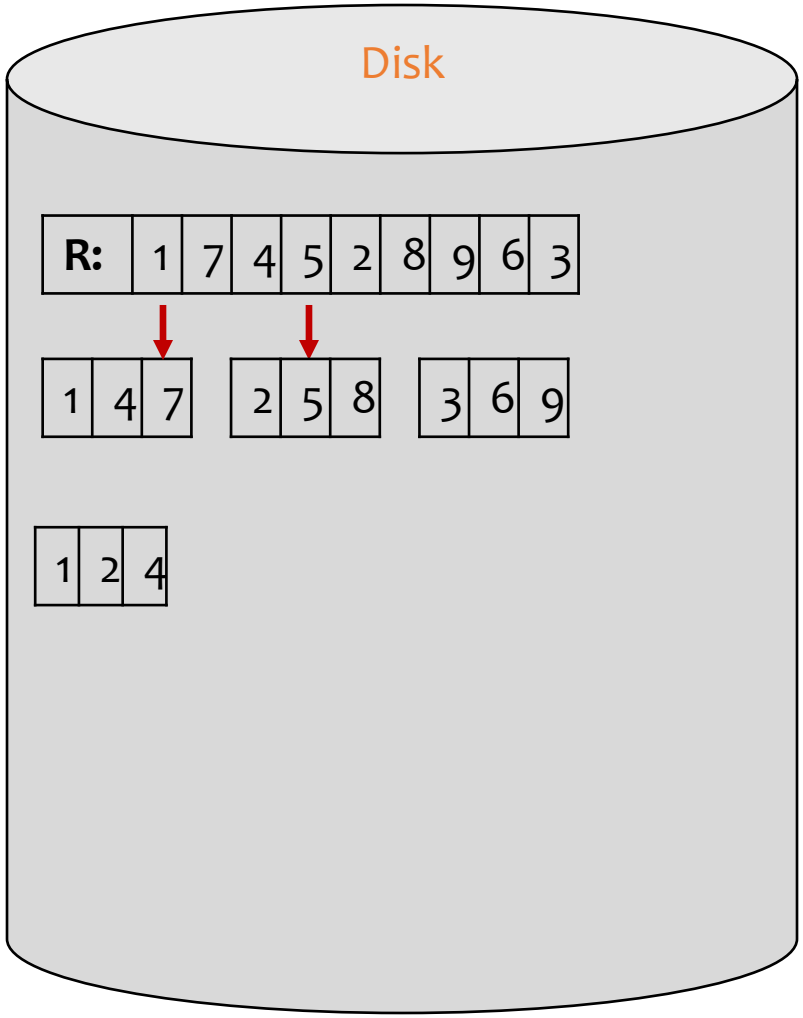
Arrows indicate the blocks in memory



Example

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

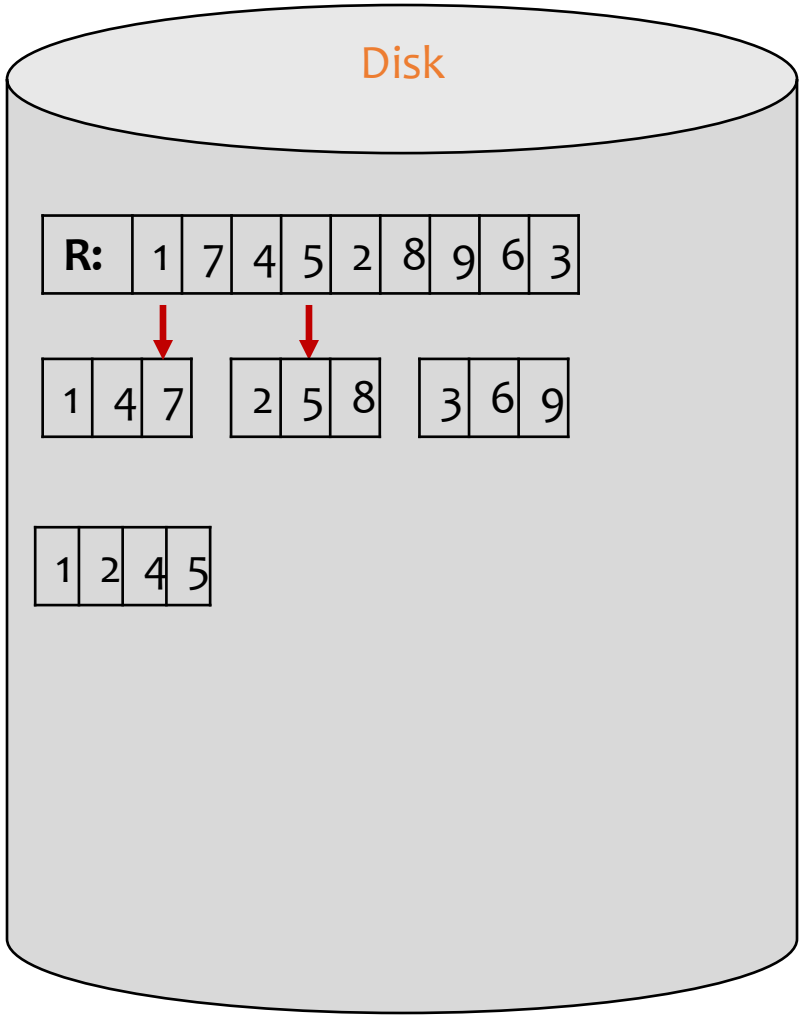
Arrows indicate the blocks in memory



Example

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

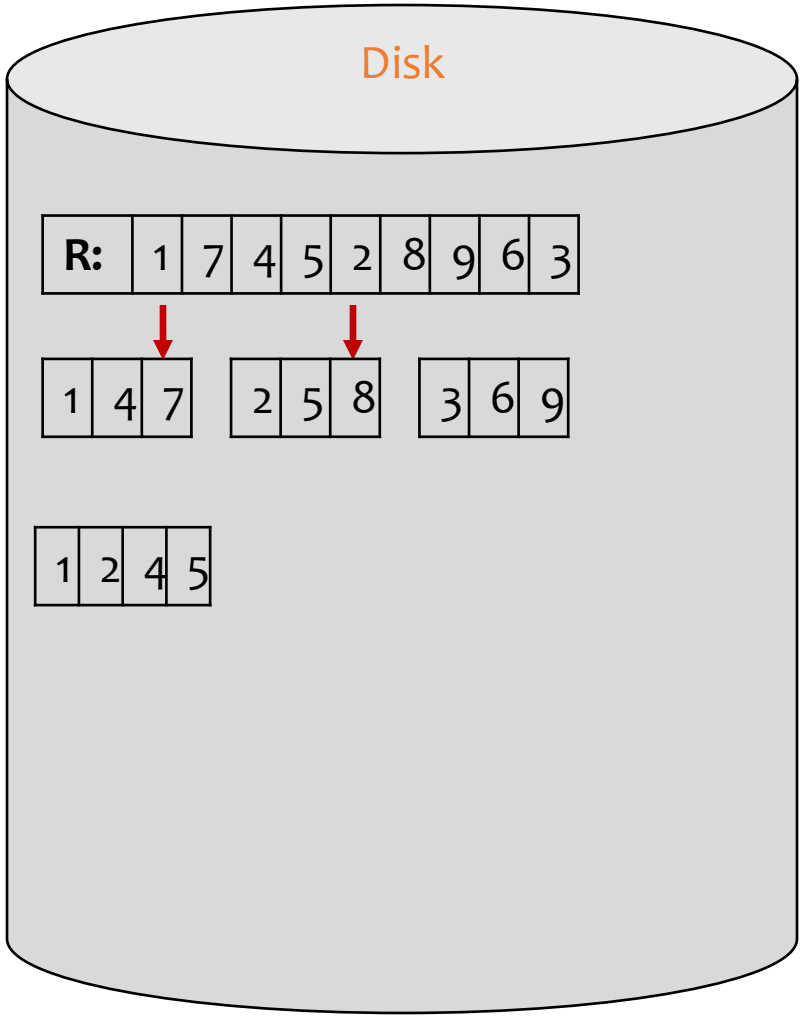
Arrows indicate the blocks in memory



Example

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

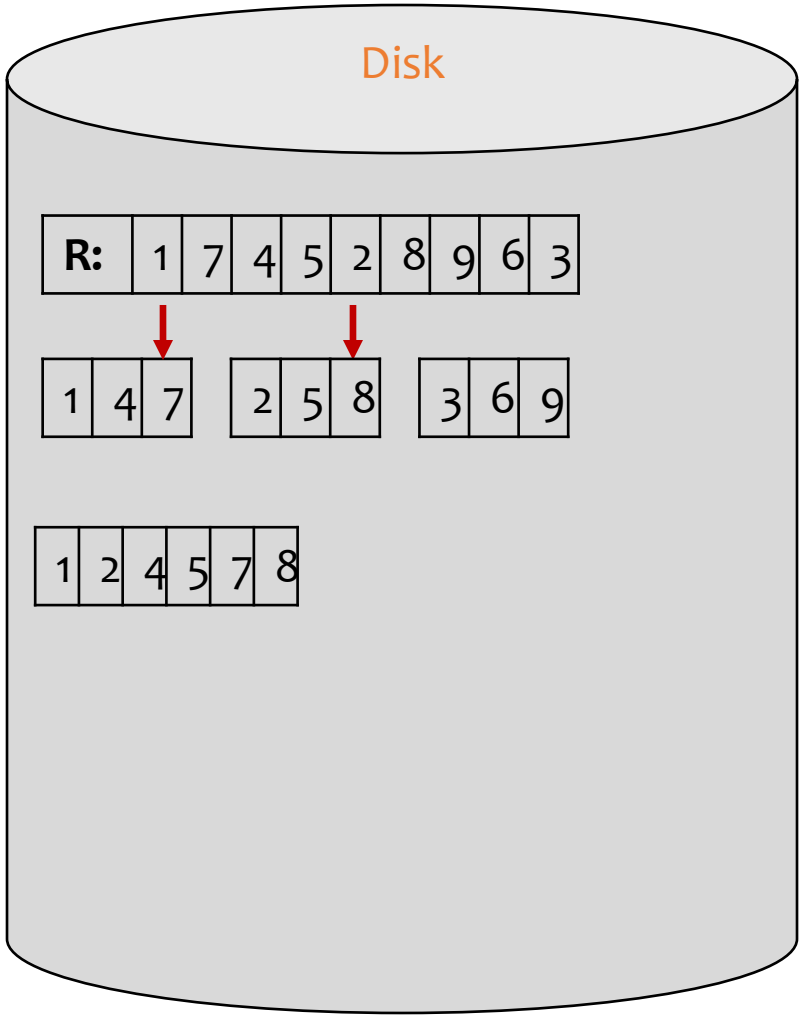
Arrows indicate the blocks in memory



Example

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

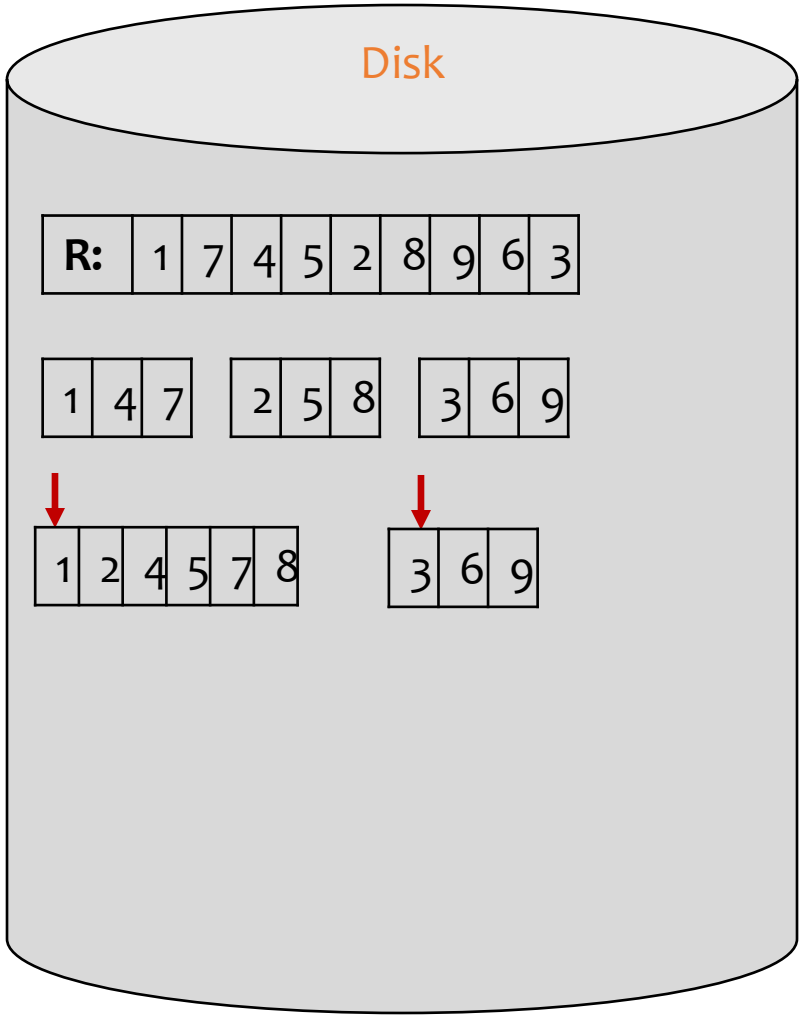
Arrows indicate the blocks in memory



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)

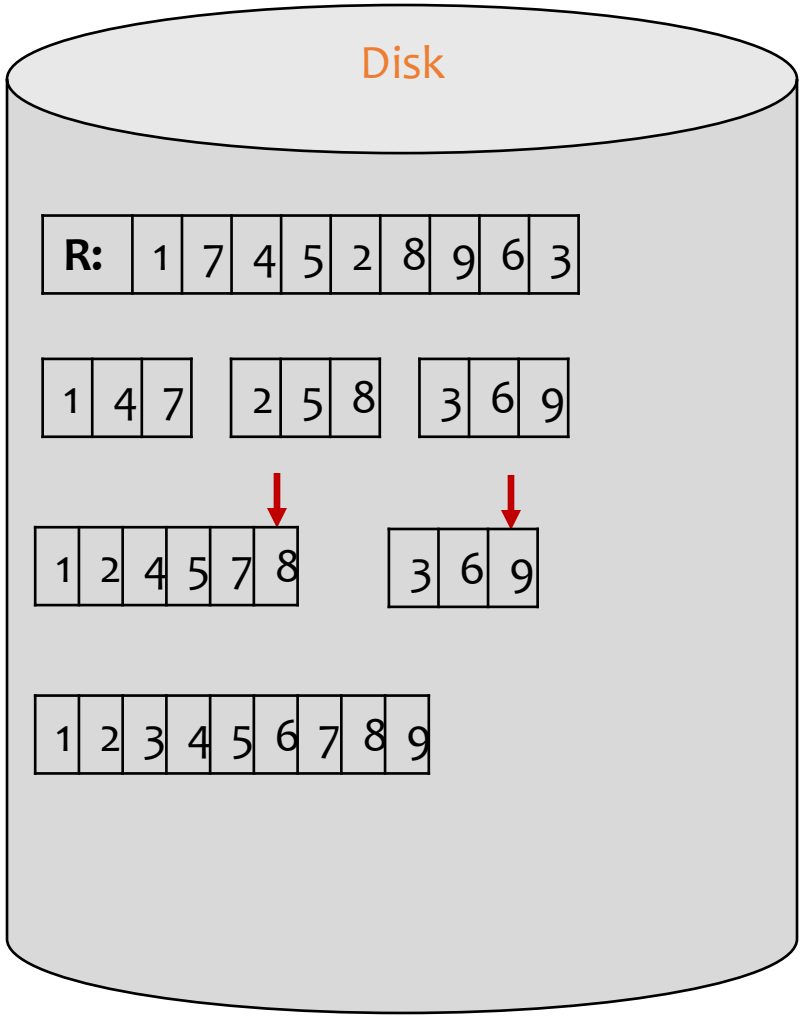
Arrows indicate the blocks in memory



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 (optional)

- 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

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

- 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 = $\left\lceil \frac{number \text{ of level}-(i-1) \text{ runs}}{M-1} \right\rceil$
  - $\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil$ number of such phases
  - Final pass produces one sorted run

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

Subtract $B(R)$ for the final pass

Performance of external merge sort

- I/O's
  - $2\mathrm{B(R)} \cdot \left(1 + \left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil\right) - \mathrm{B(R)}$
  - Roughly, this is $\boxed{O\left(B(R) \times \log_M B(R)\right)}$

What we will use in class

- 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(User)=1000/10=100; B(Group)=100/10=10; B(Member)=50000/10=5k
- Q3: select * from User order by age asc;
  - I/O cost using external merge sort?

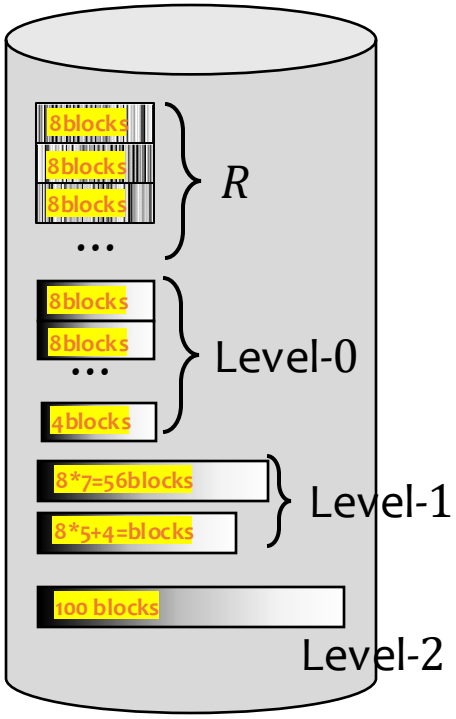
Case study (optional):

- 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(User)=1000/10=100; B(Group)=100/10=10; B(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 => ceil(100/8)=13 runs
    - Phase 1: merge 7 runs at a time => ceil(13/7)=2 runs
    - Phase 2: merge last 2 runs into a single run

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

Phase 0: read B(user)=100 blocks, write B(User)=100 blocks (temporary result)
Phase 1: read B(user)=100 blocks, write B(User)=100 blocks (temporary result)
Phase 2: read B(user)=100 blocks, write B(User)=100 blocks (final result, don't count)

Total: $2B(User) * 3 - B(User) = 5B(user) = 500$



Operators That Use Sorting

- Pure Sort: e.g., ORDER BY
- Set Union, Difference, Intersection, or Join on 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:

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

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

$r_1 s_1$

$r_2 s_3$

$r_2 s_4$

$r_3 s_3$

$r_3 s_4$

$r_7 s_5$

Outline

- Scan
  - Selection, duplicate-preserving projection, nested-loop join
- Sort
  - External merge sort, sort-merge-join
- Hash
  - Hash join, point selection, group by and aggregations
- Index
  - Selection, index nested-loop join

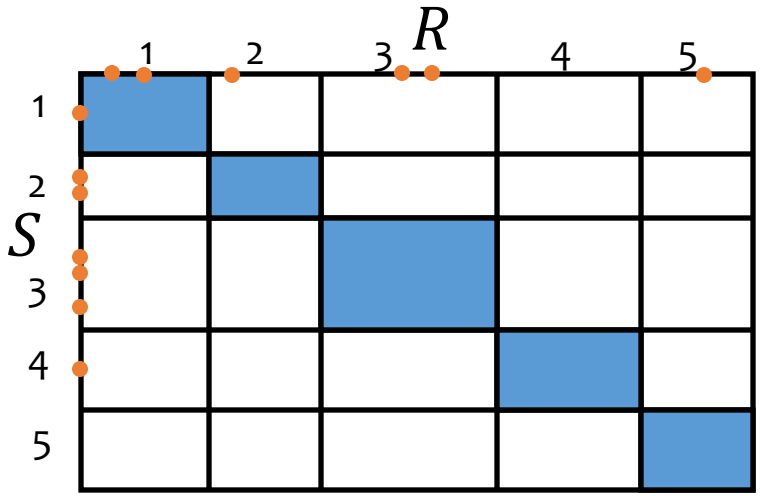
Hashing-based algorithms

https://recipesfromapantry.com/frozen-hash-browns-in-oven/

Hash join

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

- Main idea
  - Partition R and S by hashing their join attributes, and then consider corresponding partitions of R and S
  - If $r.A$ and $s.B$ get hashed to different partitions, they don't join

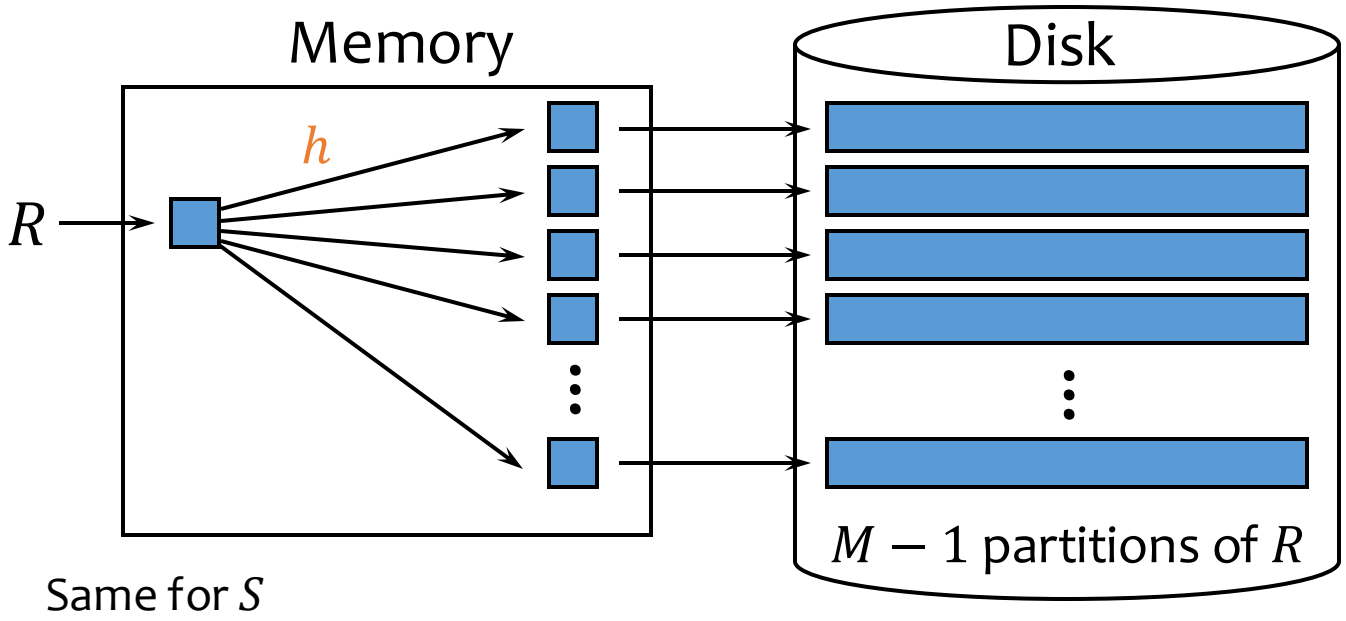


Nested-loop join considers all slots

Hash join considers only those along the diagonal!

Partitioning phase

- Partition R and S according to the same hash function on their join attributes

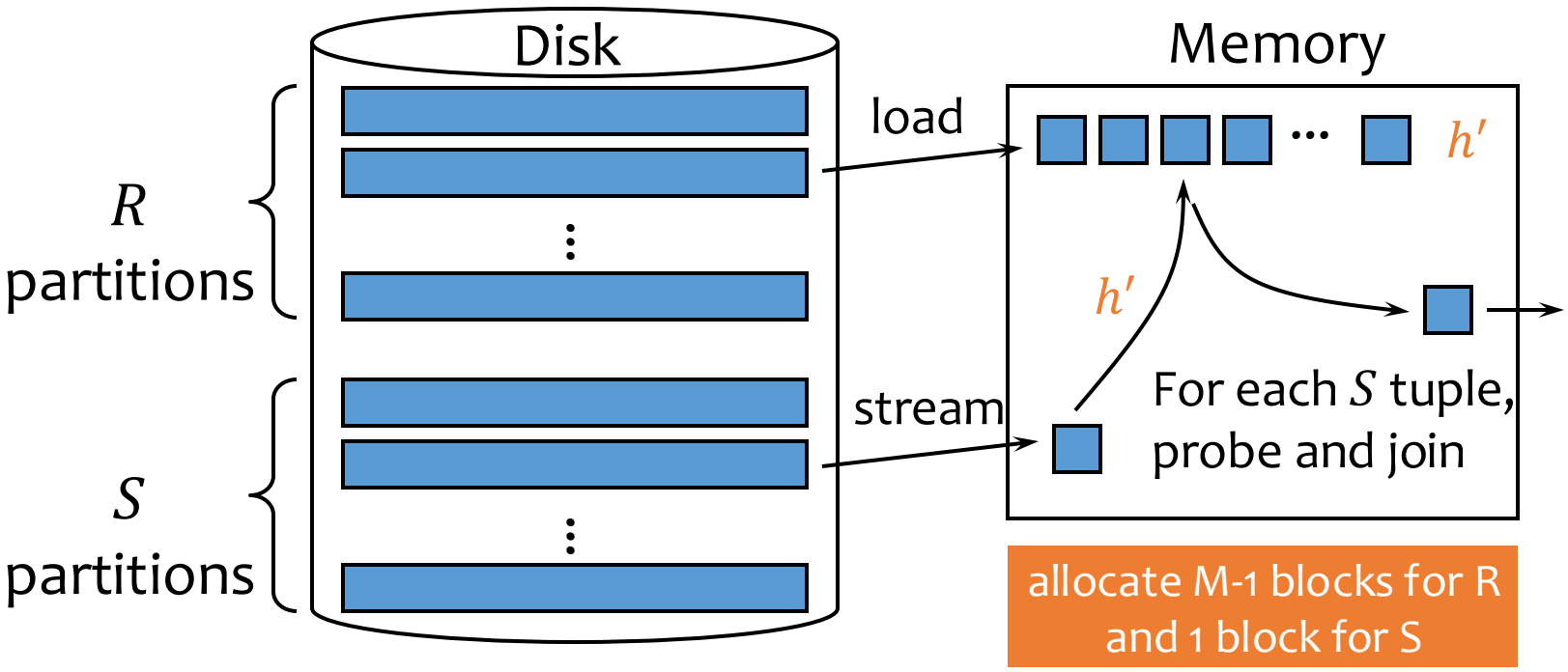


allocate 1 for input and M-1 for output buffers

If the hash function is good, each partition has a size of B(R)/(M-1)

Probing phase

- Read in each partition of R, stream in the corresponding partition of S, join
  - Typically build a hash table for the partition of R
    - Not the same hash function used for partition, of course!



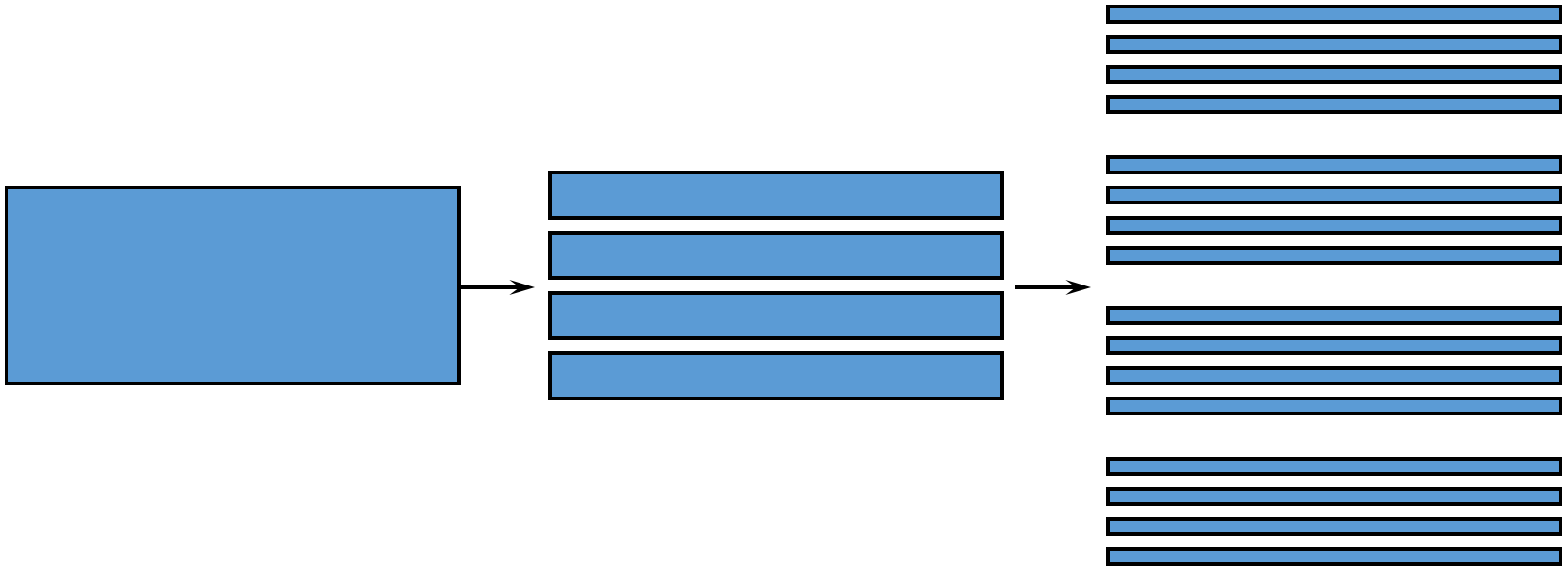
Performance of (two-pass) hash join

- If hash join completes in two phases:
  - I/O's: $3 \cdot \big(B(R) + B(S)\big)$ or $O\big(B(R) + B(S)\big)$
    - 1st phase: read B(R) + B(S) into memory to partition and write partitioned B(R) +B(S) to disk
    - 2nd phase: read B(R) + B(S) into memory to merge and join
  - Memory requirement:
    - In the probing phase, we should have enough memory to fit one partition of *R*: $M - 1 > \frac{B(R)}{M-1}$
    - $M > \sqrt{B(R)} + 1$
    - We can always pick R to be the smaller relation, so:

$$M > \sqrt{\min\big(B(R), B(S)\big)} + 1$$

Generalizing for larger inputs

- What if a partition is too large for memory?
  - Read it back in and partition it again!
  - Re-partition $O\left(\log_M B(R)\right)$ times



Other hash-based algorithms

- Union (set), difference, intersection

- Duplicate elimination
  - Check for duplicates within each partition/bucket

- Grouping and aggregation
  - Apply the hash functions to the group-by columns

Outline

- Scan
  - Selection, duplicate-preserving projection, nested-loop join
- Sort
  - External merge sort, sort-merge-join
- Hash
  - Hash join, point selection, group by and aggregations
- Index
  - Selection, index nested-loop join

Index-based algorithms

http://i1.trekearth.com/photos/28820/p2270994.jpg

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: lookup + 20% $|R|$
  - Table scan wins if a block contains more than 5 tuples!
    - B(R) = |R|/5 < 20%|R|+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: $B(R) + |R| \cdot (\text{index\_lookup} + \text{I/O for record fetch})$
  - 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)

Summary of techniques

- Scan
  - Selection, duplicate-preserving projection, nested-loop join
- Sort
  - External merge sort, sort-merge join, union (set), difference, intersection, duplicate elimination, grouping and agg.
- Hash
  - Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Index
  - Selection, index nested-loop join, zig-zag join

Another view of techniques

- Selection
  - Scan without index (linear search): $O(B(R))$
  - Scan with index – selection condition must be on search-key of index
    - B+ index: $O(\log(B(R))$
    - Hash index: $O(1)$
- Projection
  - Without duplicate elimination: $O(B(R))$
  - With duplicate elimination
    - Sorting-based: $O\left(B(R) \cdot \log_M B(R)\right)$
    - Hash-based: $O(B(R) + t)$where t is the result of the hashing phase
- Join
  - Block-based nested loop join (scan table): $O(B(R) \cdot \frac{B(S)}{M})$
  - Sort-merge join $O\left(B(R) \cdot \log_M B(R) + B(S) \cdot \log_M B(S)\right)$
  - Hash join $O\left(B(R) + B(S)\right)$
  - Index nested loop join $O(B(R) + |R| \cdot (\text{index lookup}))$