Lecture 16: Query Processing & Optimization

CS348 Spring 2025:
Introduction to Database Management

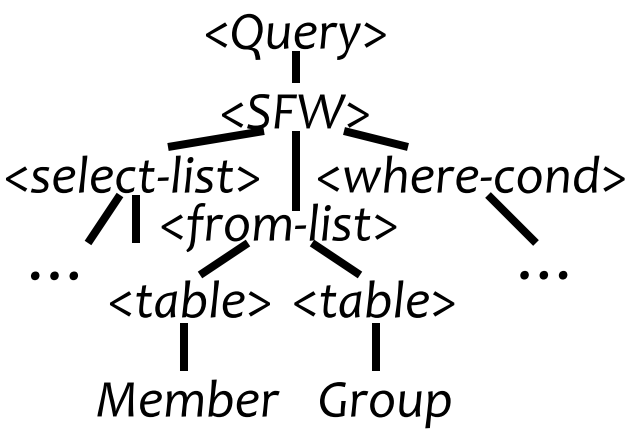
Instructor: **Xiao Hu**

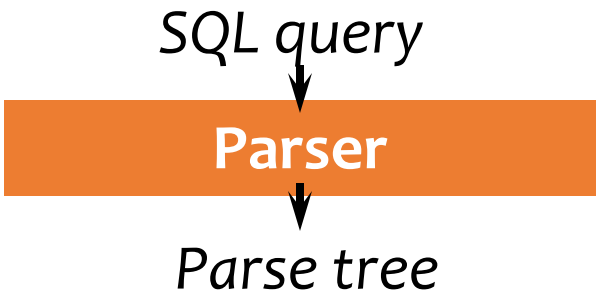
Sections: 001, 002, 003

Announcements

- Milestone 2 of group project
  - Due on next Tue, July 8
- Assignment 3 is released on Learn
  - Coverage: Lecture 13 – Lecture 20
  - Due on July 22
- Grading of Assignment 2 will be released by this week
  - See announcement on Piazza
  - Appealing period is one week after the release
- Grading of midterm is still in progress

A query's trip through the DBMS



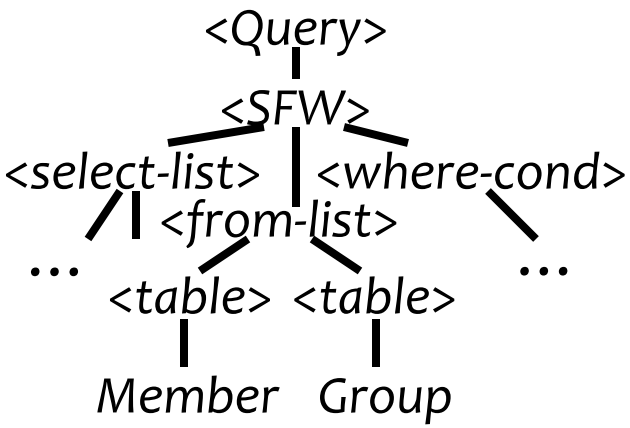


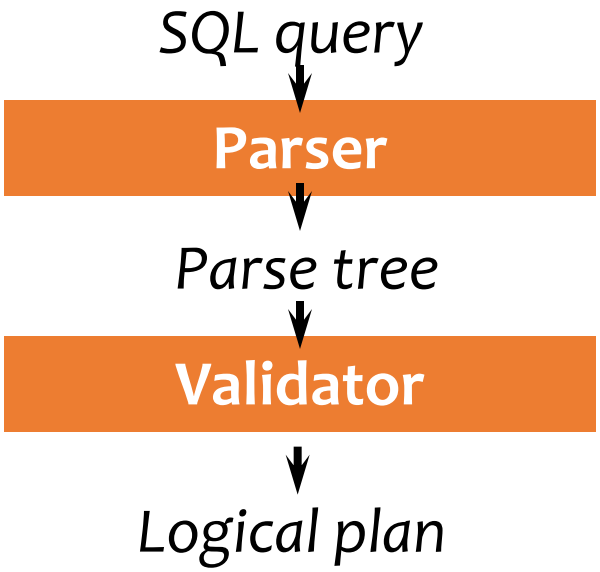

```
SELECT name, uid
FROM Member, Group
WHERE Member.gid =
    Group.gid;
```

Query parsing and validation

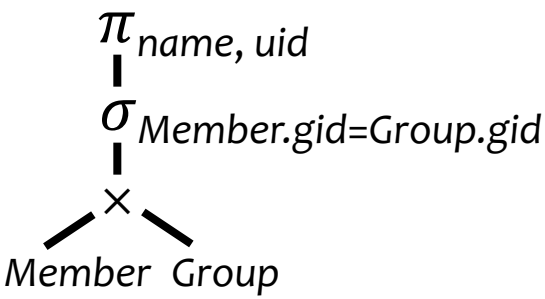
- Parser: SQL → parse tree
  - Detect and reject syntax errors

A query's trip through the DBMS





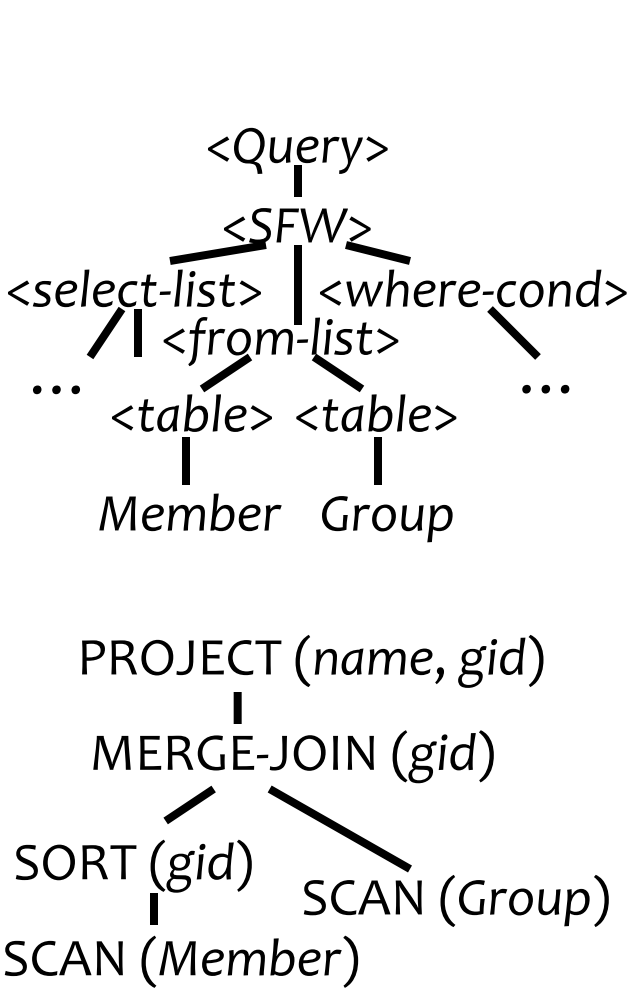

```
SELECT name, uid
FROM Member, Group
WHERE Member.gid =
    Group.gid;
```

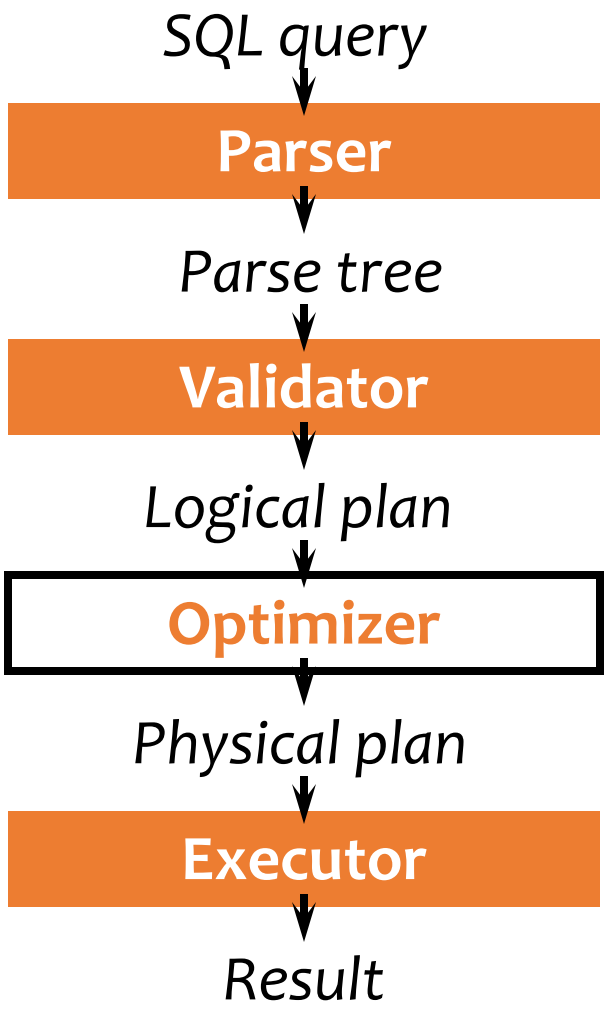


Query parsing and validation

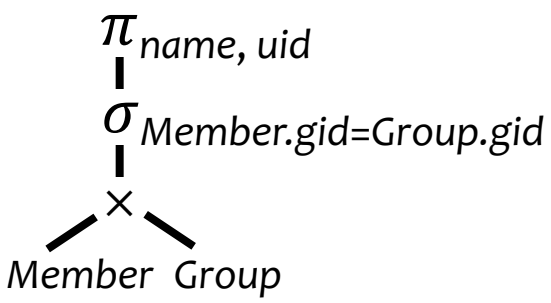
- Parser: SQL → parse tree
  - Detect and reject syntax errors
- Validator: parse tree → logical plan
  - Detect and reject semantic errors
    - Nonexistent tables/views/columns?
    - Insufficient access privileges?
    - Type mismatches?
      - AVG(name), name + pop, User UNION Member
  - Expand * and views
  - Information required for semantic checking is found in system catalog (which contains all schema information)

A query's trip through the DBMS






```
SELECT name, uid
FROM Member, Group
WHERE Member.gid =
    Group.gid;
```



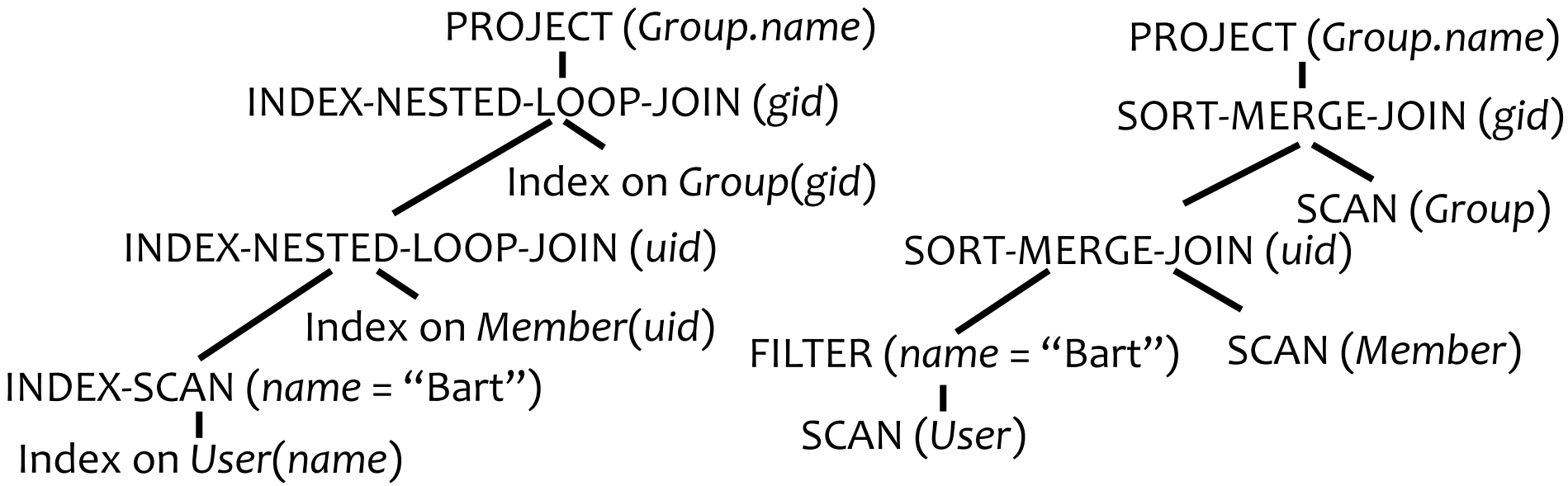
Physical plan

- A complex query may involve multiple tables and various query execution algorithms
  - E.g., table scan, basic & block nested-loop join, index nested-loop join, sort-merge join

- A physical plan for a query tells the DBMS query processor how to execute the query
  - A tree of physical plan operators
  - Each operator implements a query execution algorithm
  - Each operator accepts a number of input tables/streams and produces a single output table/stream

Examples of physical plans

```
SELECT Group.name
FROM User, Member, Group
WHERE User.name = 'Bart'
AND User.uid = Member.uid AND Member.gid = Group.gid;
```



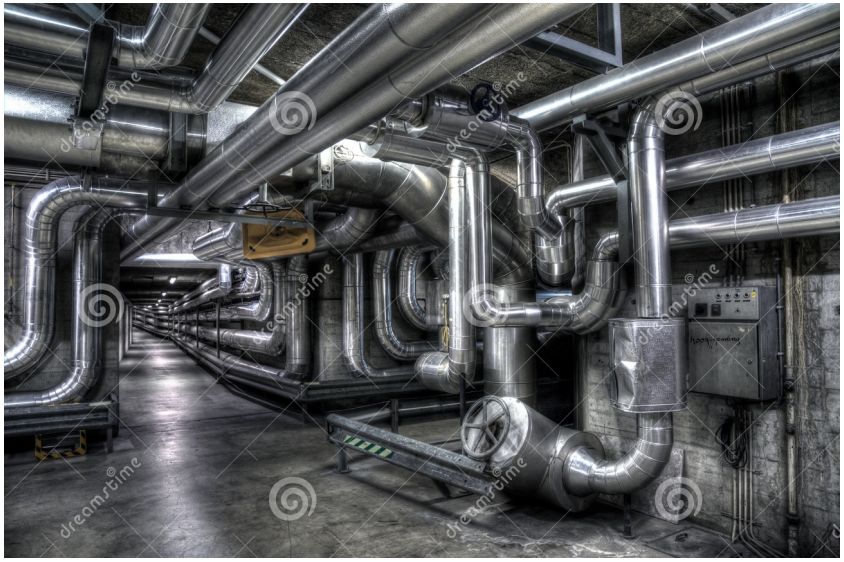
- Many physical plans for a single query

Execution of physical plans

What is the algorithm for each operator?

(focus of this lecture)

How are intermediate results passed from child to parent operators?

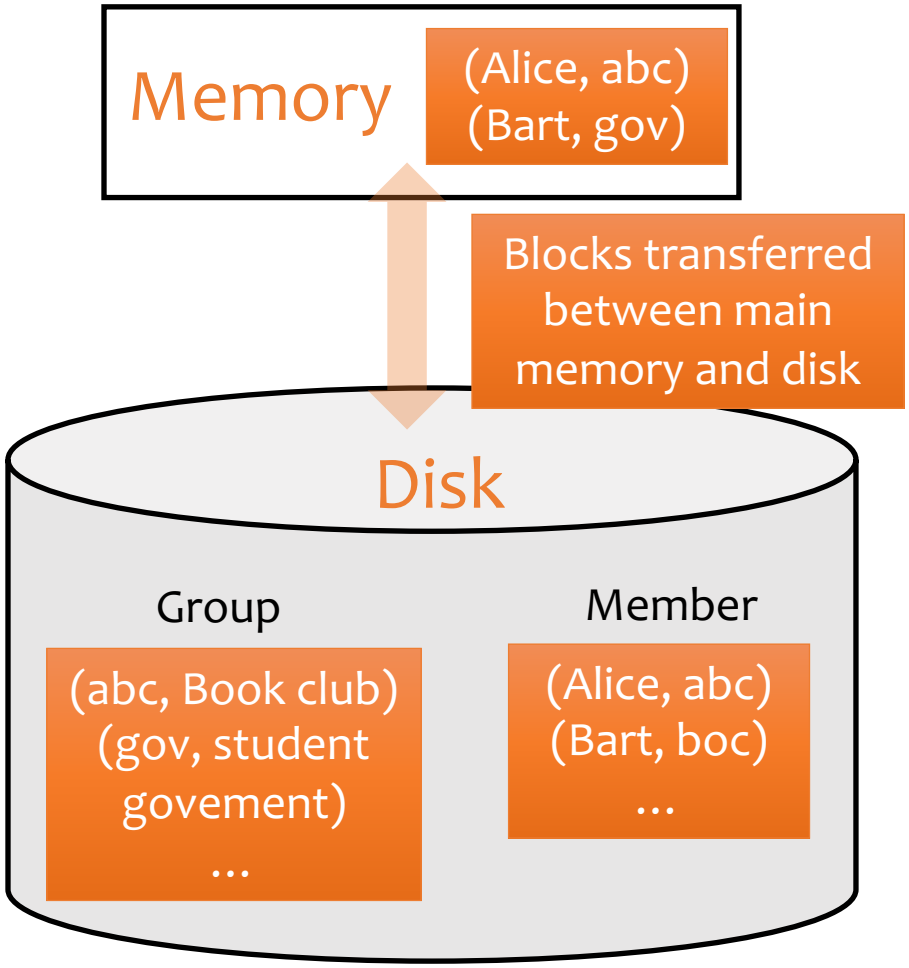


http://www.dreamstime.com/royalty-free-stock-image-basement-pipelines-grey-image25917236

- Temporary files
  - Compute the tree bottom-up
  - Children write intermediate results to temporary files
  - Parents read temporary files
- Iterators
  - Do not materialize the intermediate result
  - Children pipeline their results to parents

Outline for Today

- Scan
- Sort
- Hash
- Index



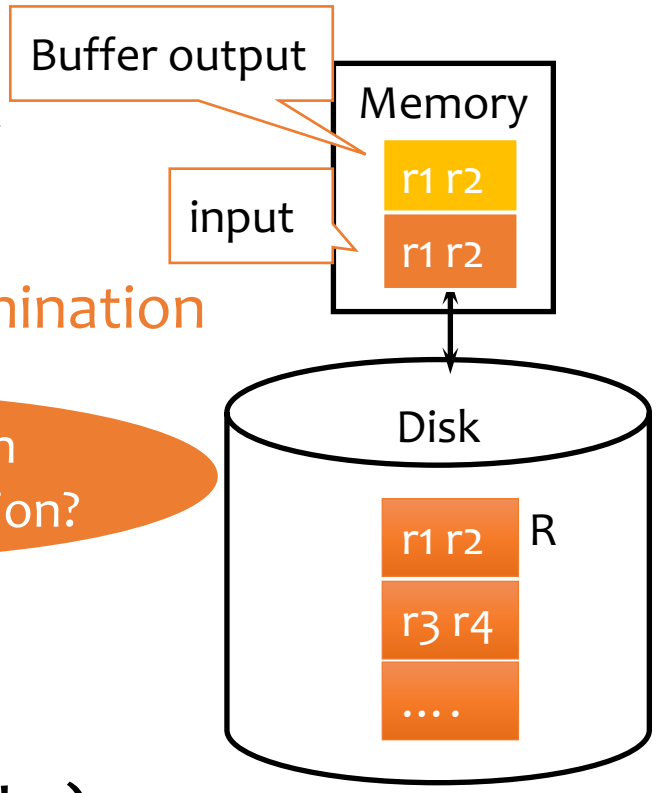
Notation and Assumption

- 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 (blocks transferred between memory and disk)
  - Memory requirement
- Not counting the cost of writing the result out
  - Same for any algorithm
  - Maybe not needed – results may be pipelined into downstream operator

Scanning-based algorithms

Table scan

- Scan table *R* and process the query
  - Selection over *R*
  - Projection of *R* without duplicate elimination



How about with duplicate elimination?

- I/O's: $B(R)$
  - Stop early if it is a lookup by key

- Memory requirement: $M \geq 2$ (blocks)
  - 1 for input, 1 for buffer output
  - Increase M does not improve I/O

Tuple-based Nested-loop join

$R \bowtie_p S$

- For each block of R, and for each r in the block:
  For each block of S, and for each s in the block:
  Output rs if p evaluates to true over r and s

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

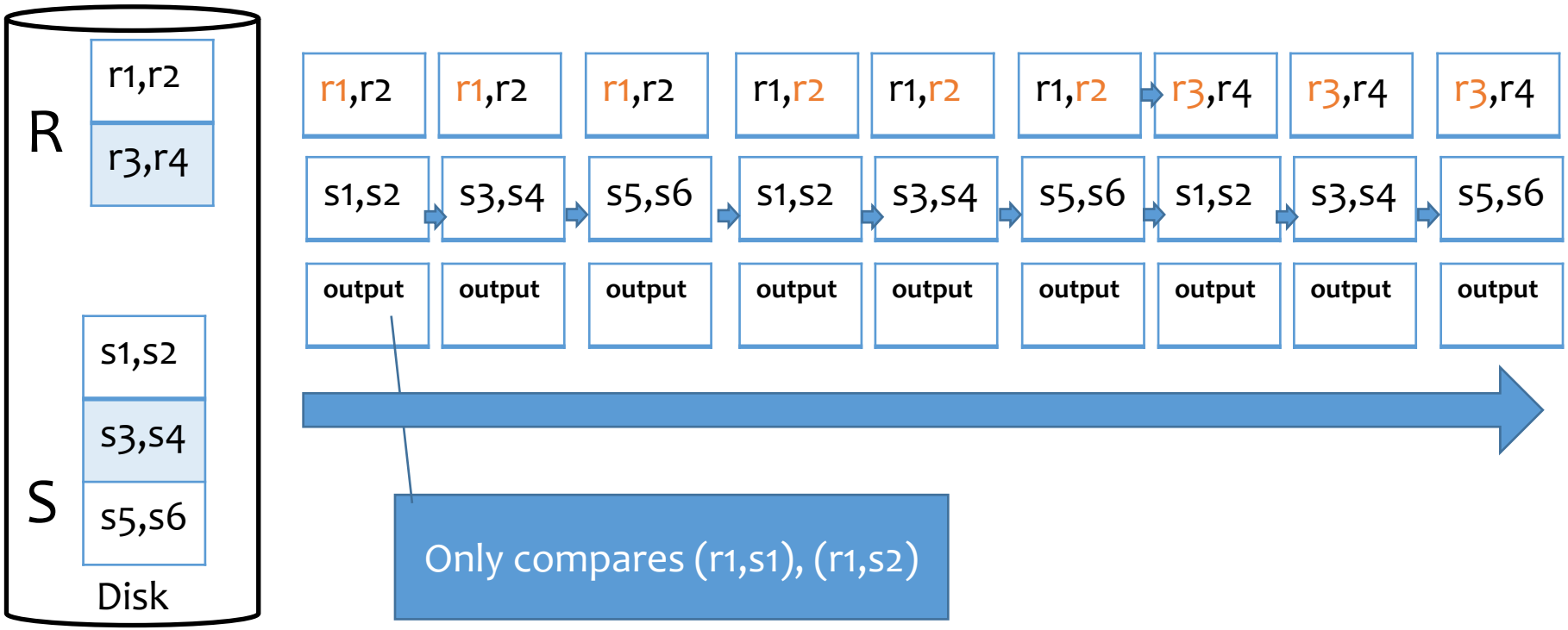
Blocks of R are moved into memory only once

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

- Memory requirement: $M \geq 3$ (blocks)

Tuple-based nested-loop join

- 1 block stores 2 tuples, 3 blocks in memory



- Number of I/Os: $B(R) + |R| * B(S) = 2 + 4 * 3 = 14$

Block-based nested-loop join

$R \bowtie_p S$

- For each block of R:
  For each block of S:
    For each r in the R block:
      For each s in the S block:
        Output rs if p evaluates to true over r and s
- I/O's: $B(R) + B(R) \cdot B(S)$

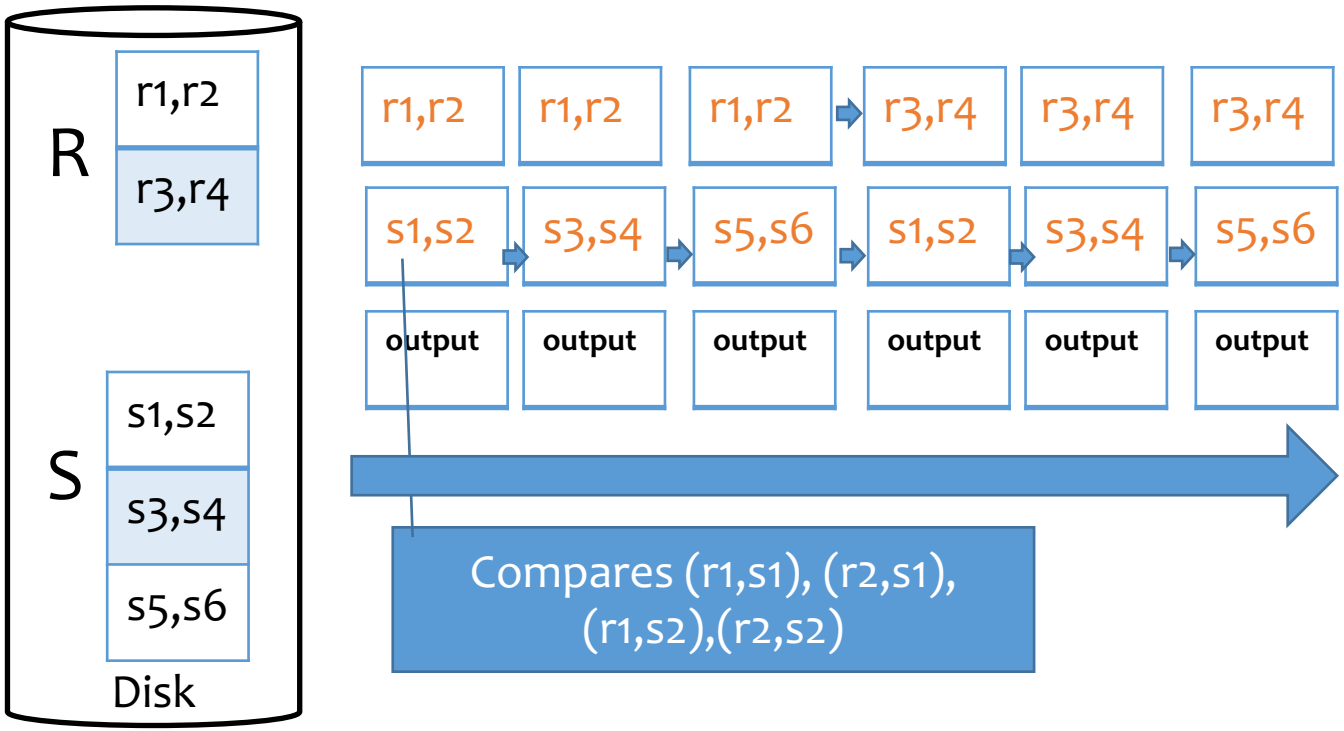
Blocks of R are moved into memory only once

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

- Memory requirement: same as before

Block-based nested loop join

- 1 block stores 2 tuples, 3 blocks in memory



- Number of I/Os: B(R) + B(R)* B(S) = 2 + 2 * 3 = 8

More improvements

- Stop early if the key of the inner table is being matched

- Make full use of available memory
  - Suppose M memory blocks are available
  - How to allocate memory blocks for outer and inner tables?
  - I/O's: $B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S)$ or, roughly: $B(R) \cdot B(S)/M$
  - Increase M improves I/O!

- Which table would you pick as the outer?

What about nested loop join?

- May be best if many tuples join
  - Cross-product
  - Non-equality joins that are not very selective

- Necessary for black-box predicates
  - Example: WHERE *user_defined_pred*($R.A, S.B$)

Outline

- Scan
  - Table scan
  - Selection, Duplicate-preserving projection
  - Nested-loop join
- Sort
- Hash
- Index

Sorting-based algorithms

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

External merge sort

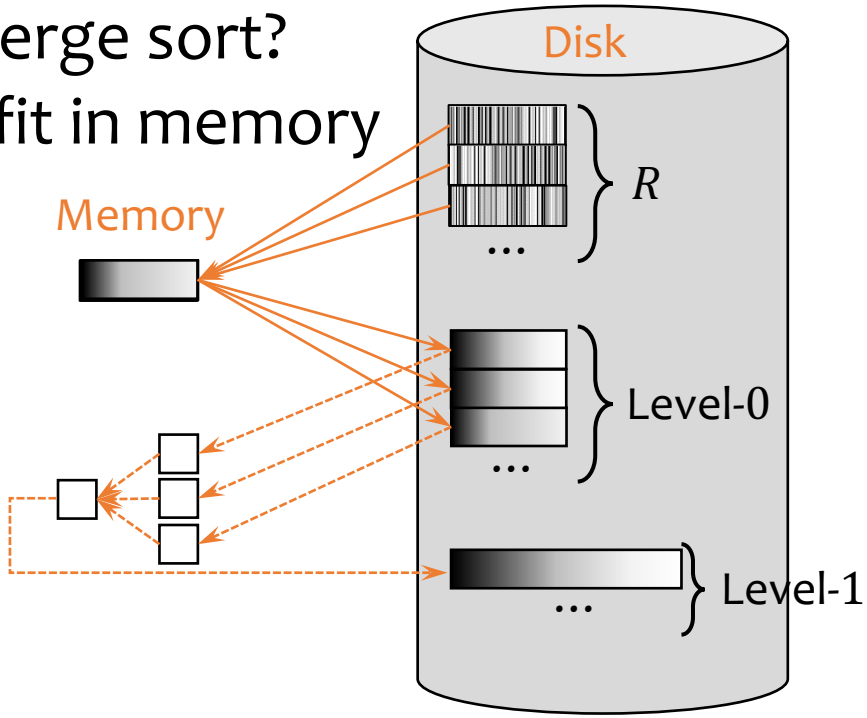
Remember (internal-memory) merge sort?

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

- Pass 0: read M blocks of R at a time, sort them, and write out a level-0 run
- Pass 1: merge $(M-1)$ level-0 runs at a time, and write out a level-1 run
- Pass 2: merge $(M-1)$ level-1 runs at a time, and write out a level-2 run

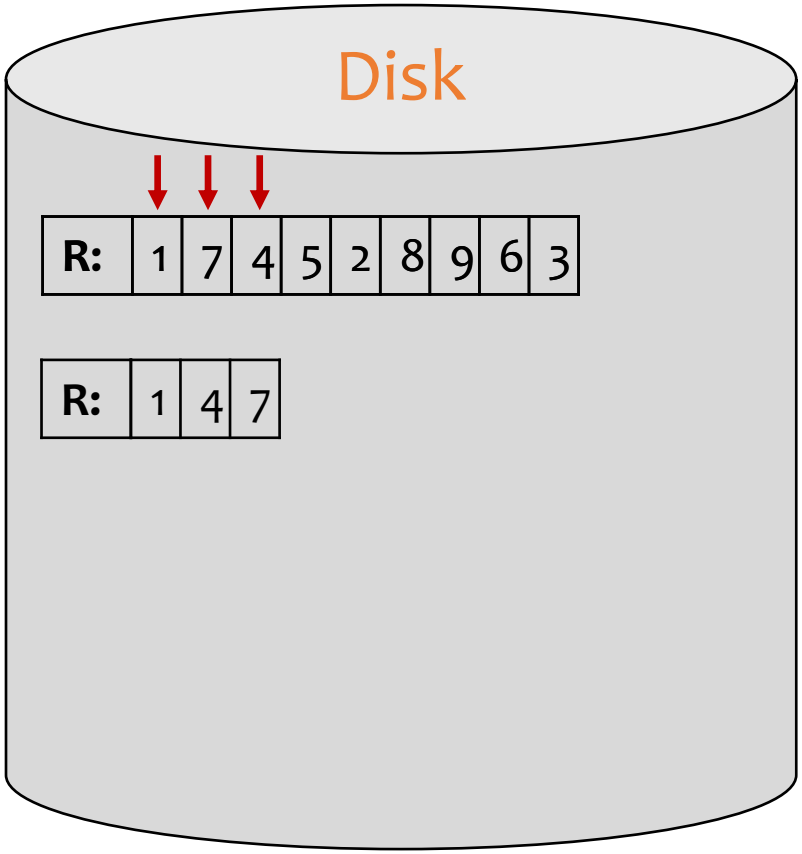
…

- Final pass produces one sorted run



Example of external merge-sort

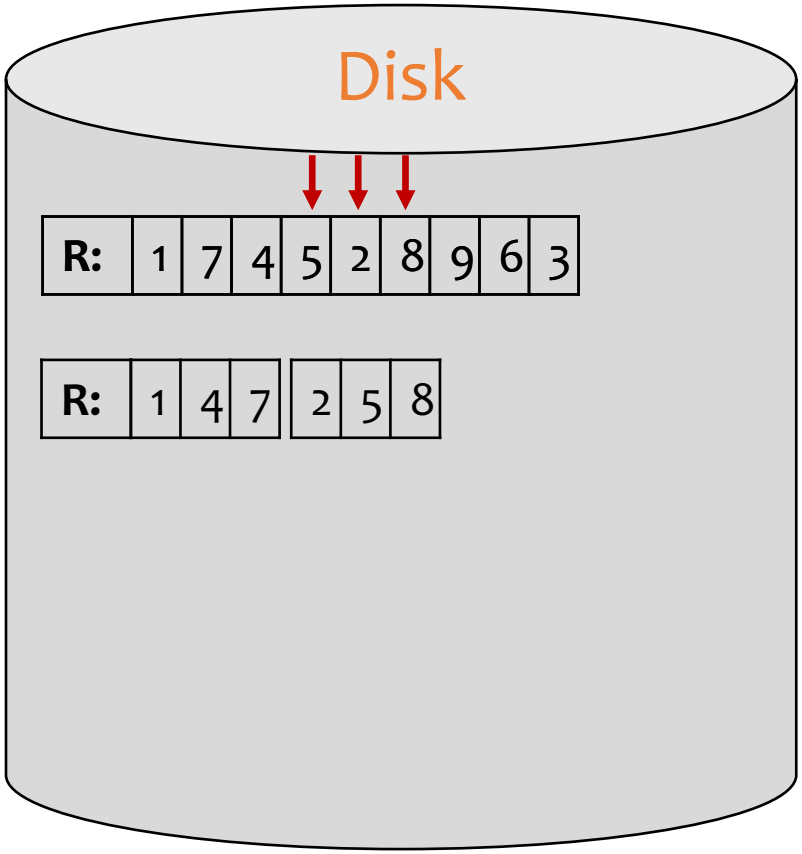
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:



Arrows indicate the blocks in memory

Example of external merge-sort

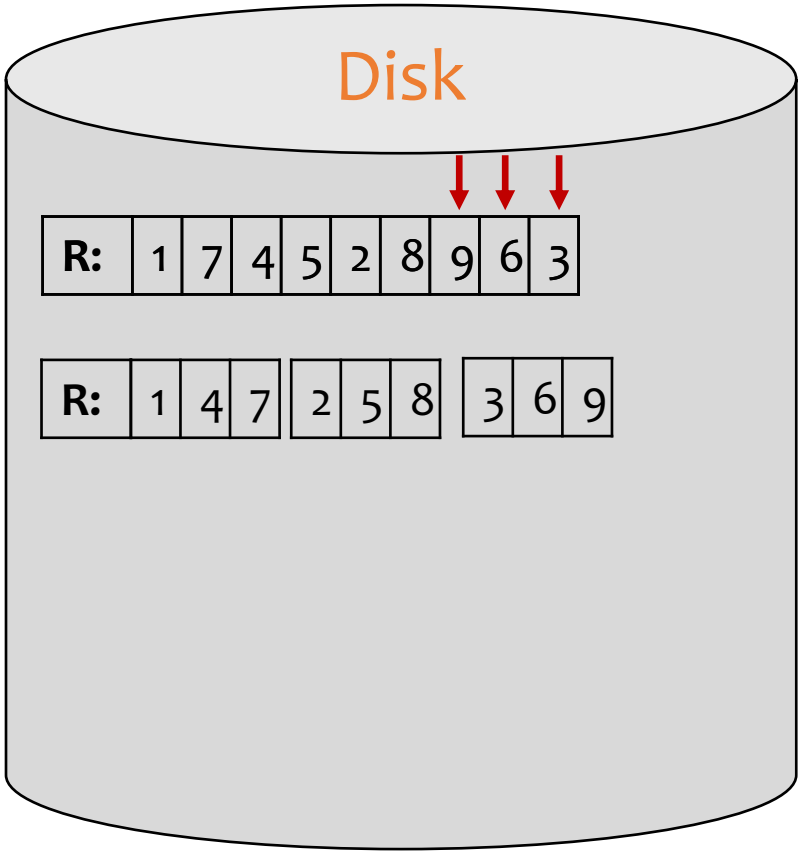
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:



Arrows indicate the blocks in memory

Example of external merge-sort

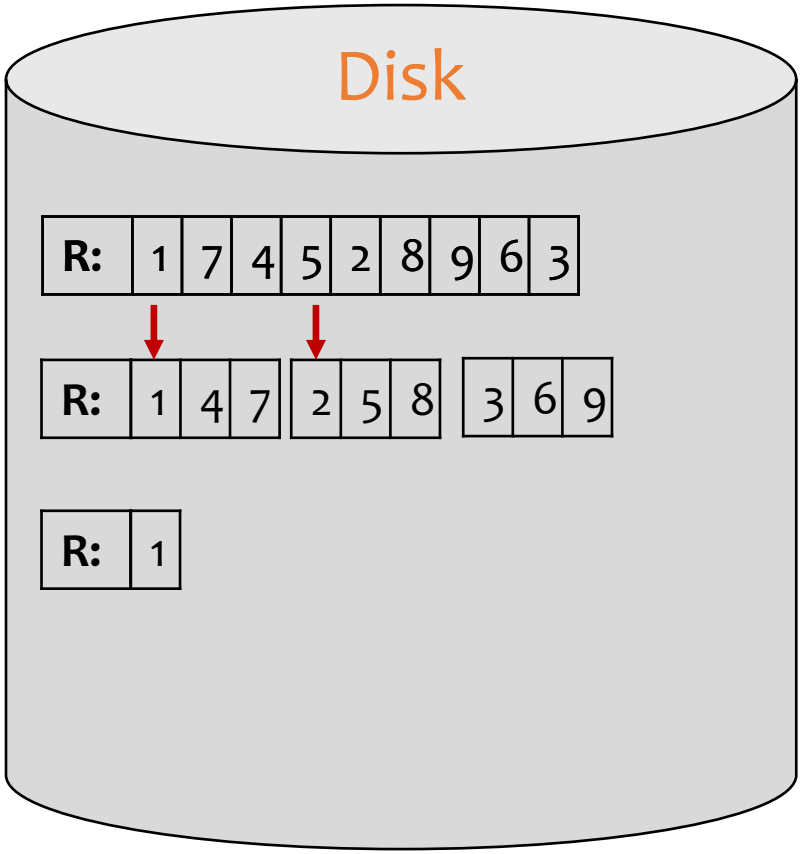
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:



Arrows indicate the blocks in memory

Example of external merge-sort

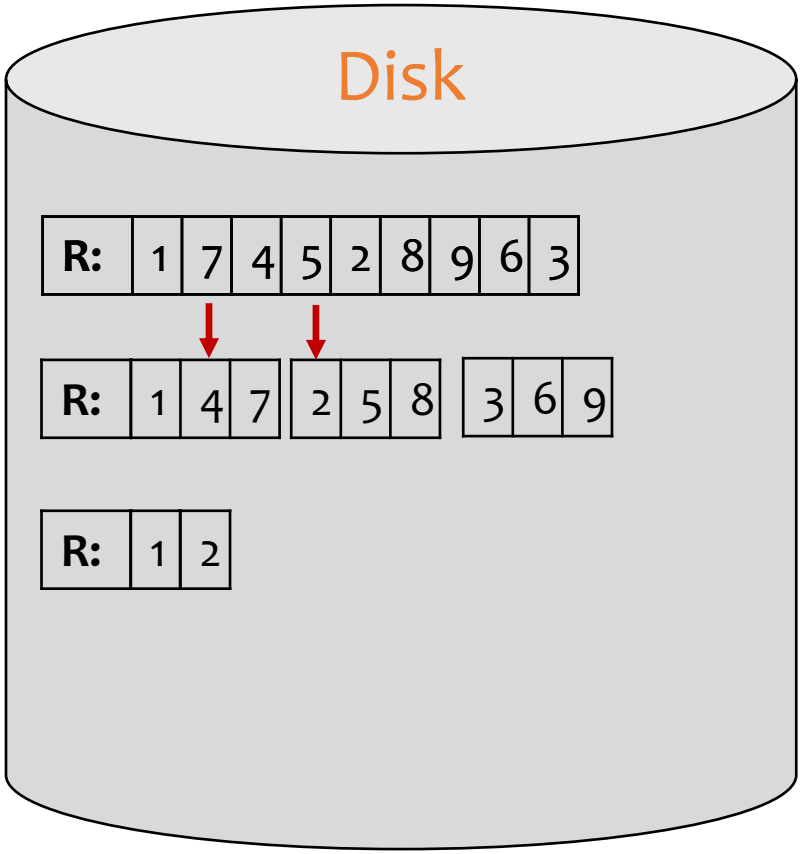
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

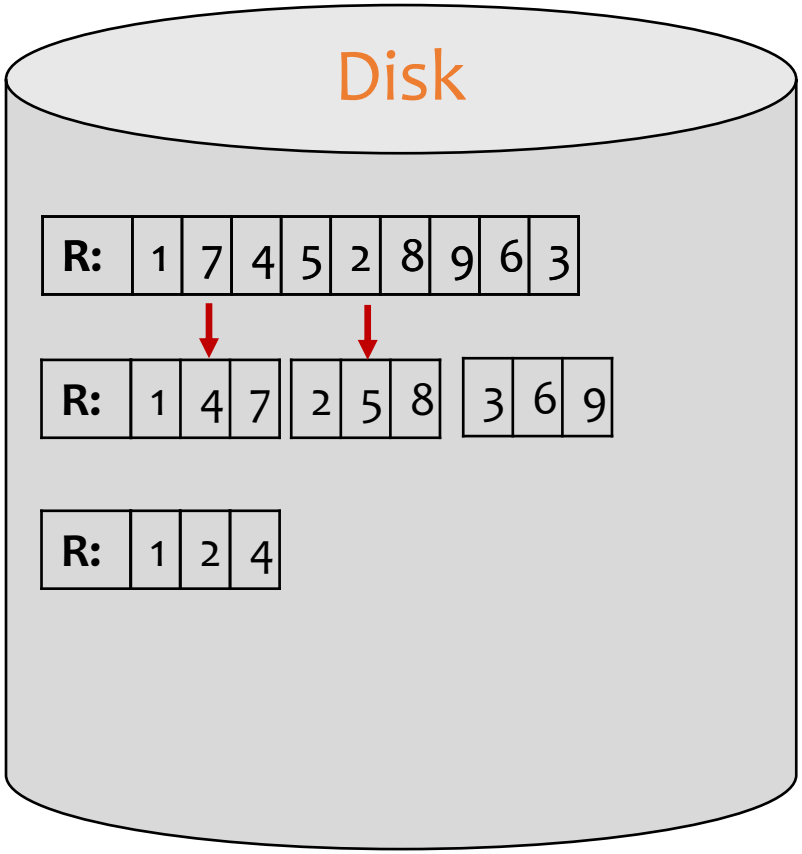
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

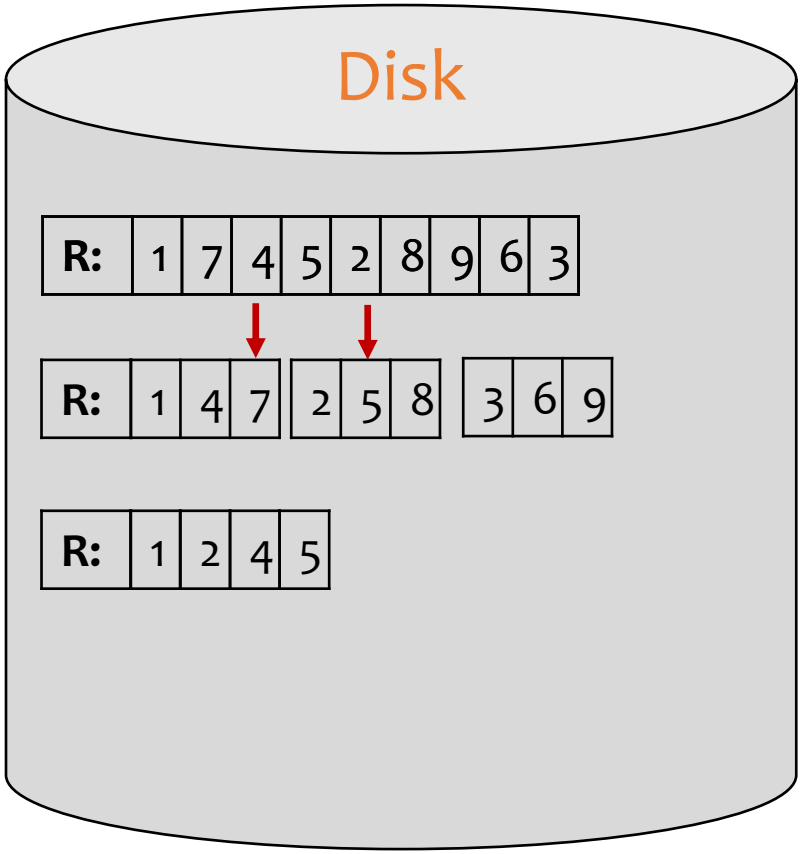
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

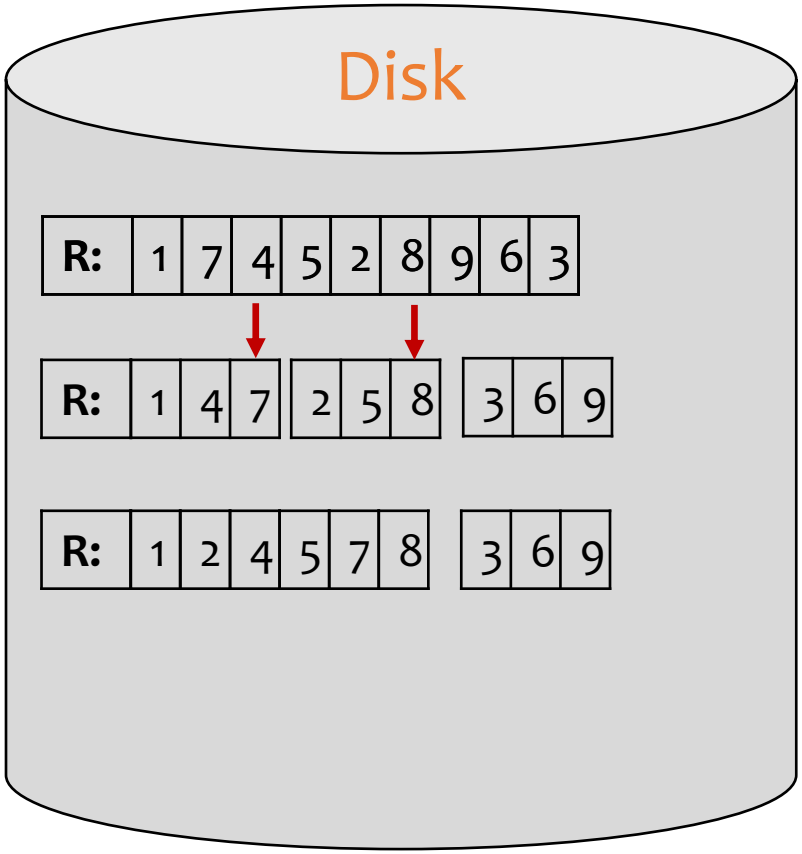
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

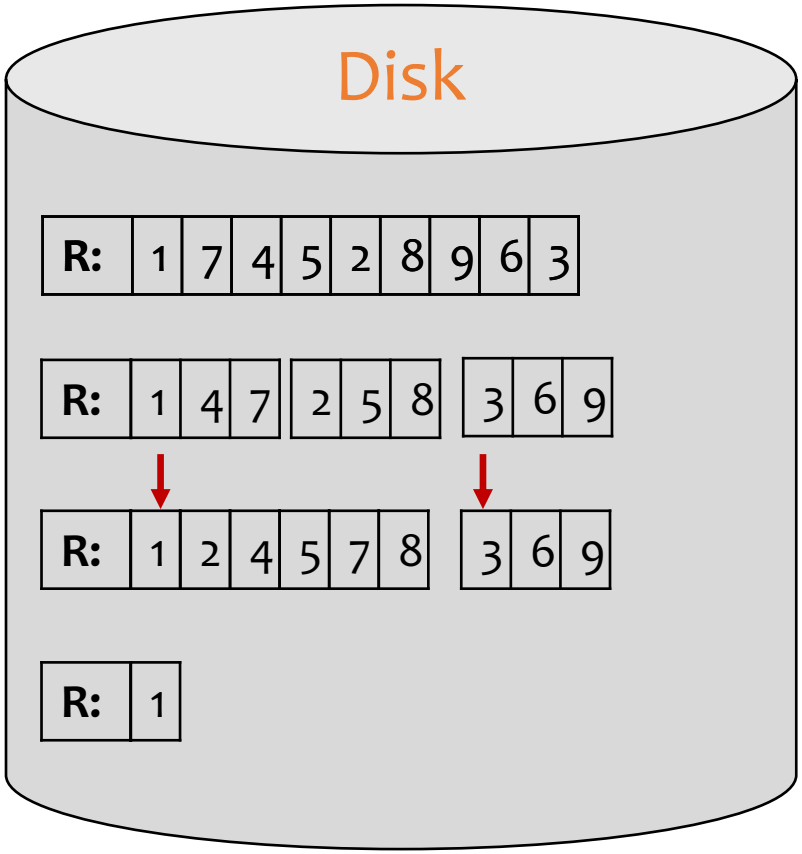
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

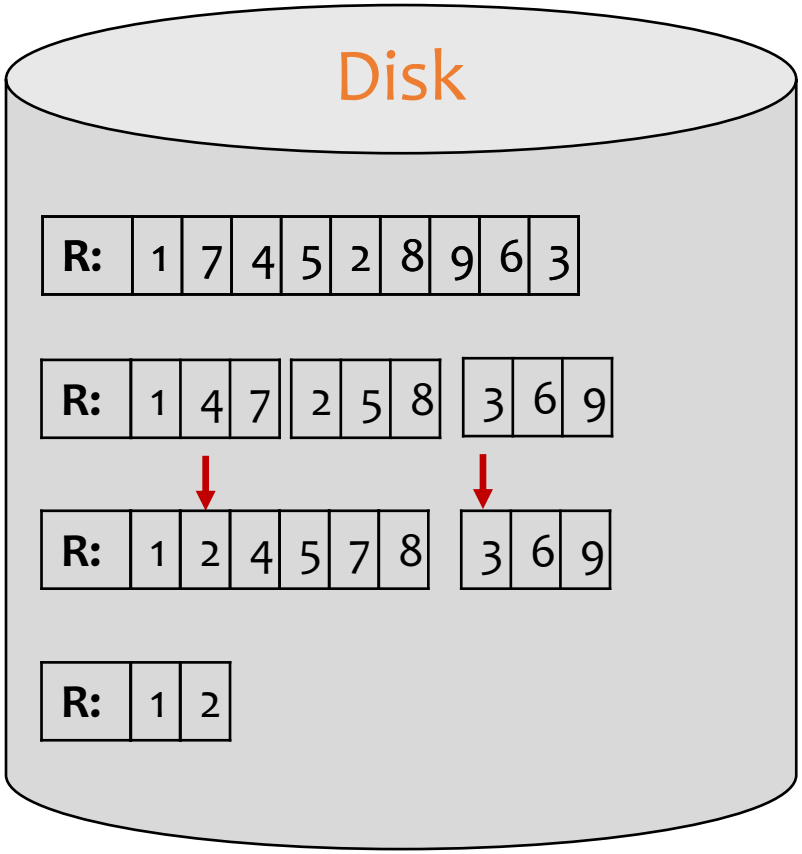
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

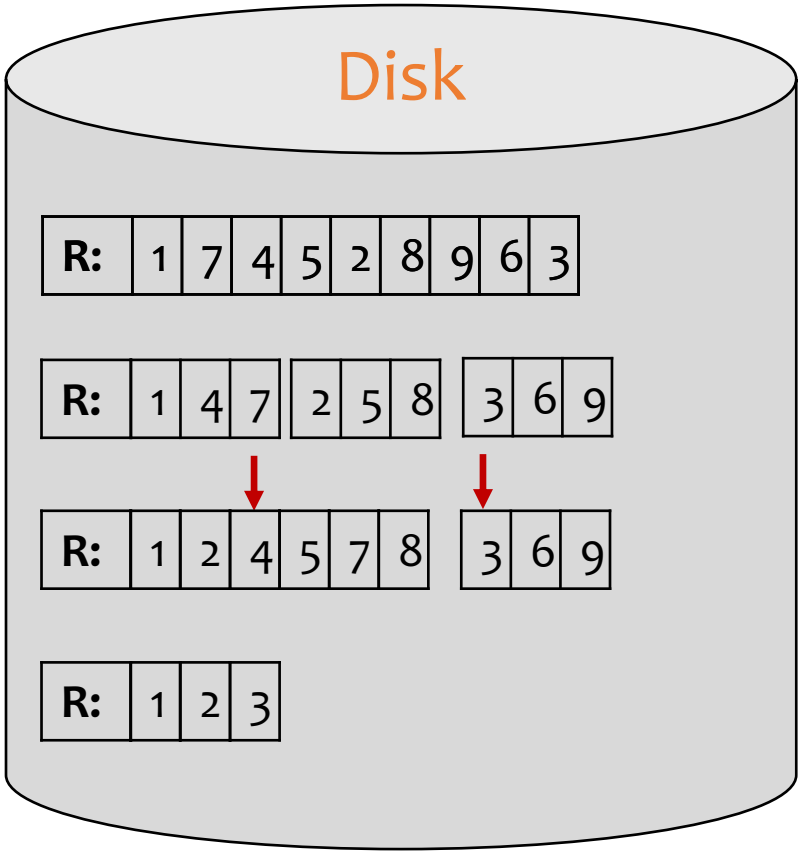
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

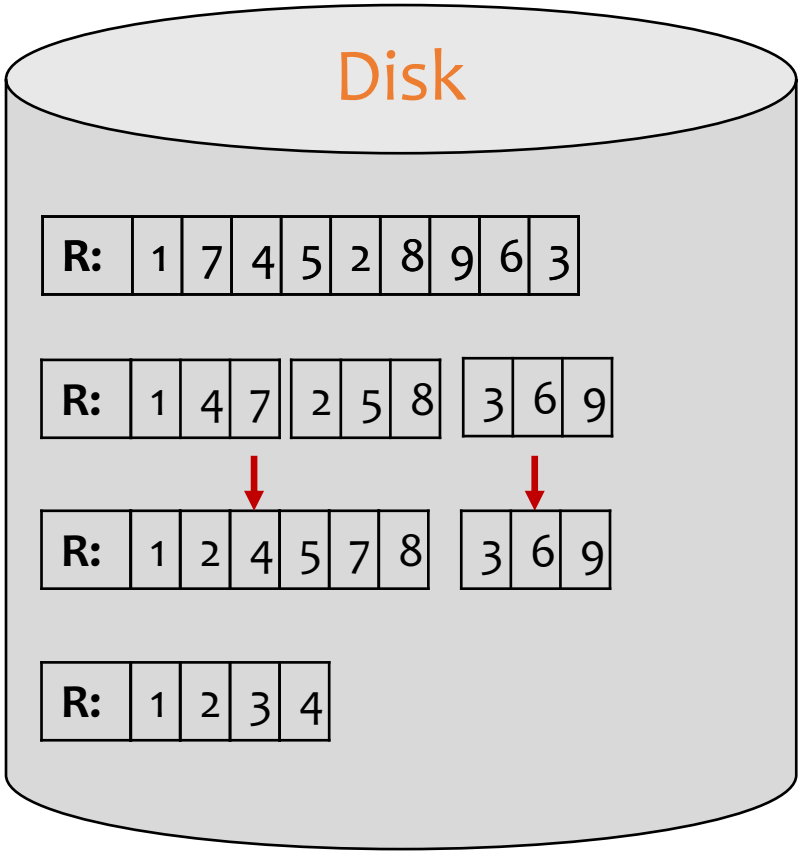
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

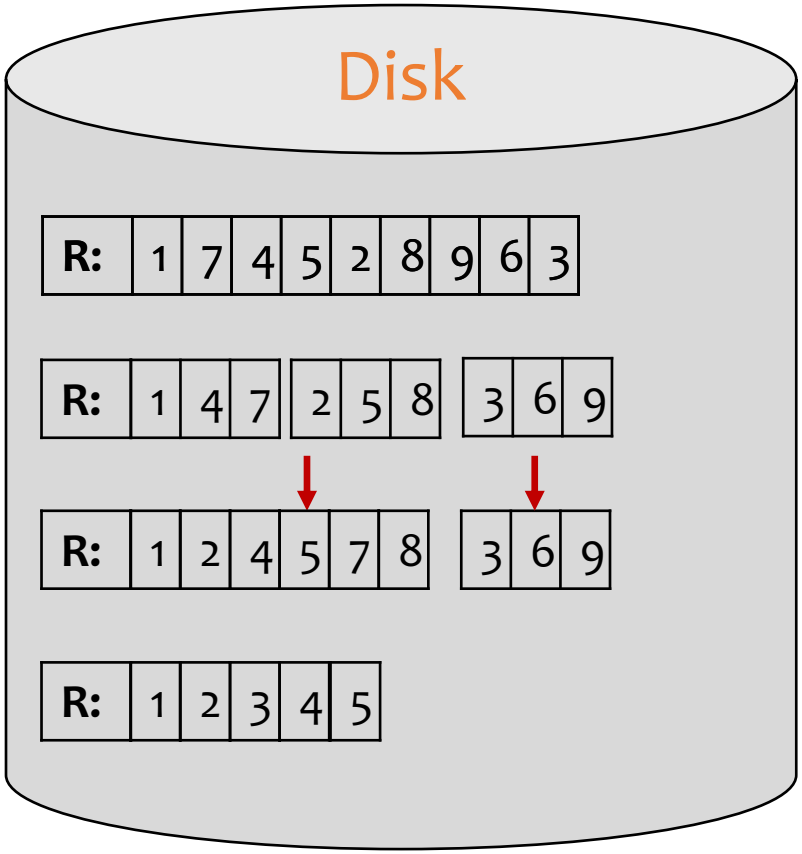
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

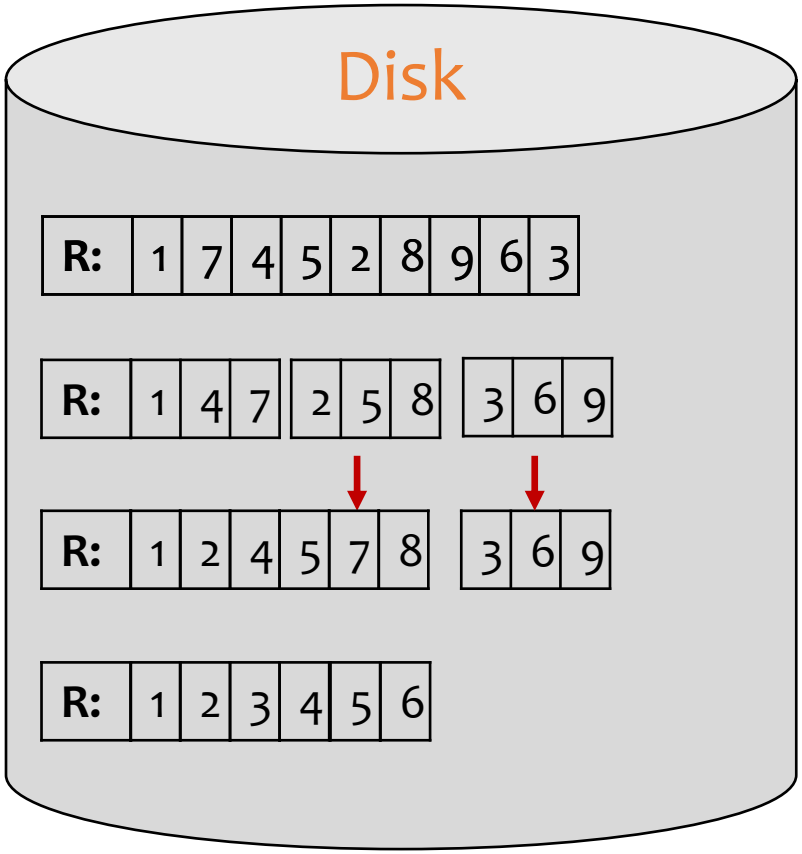
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

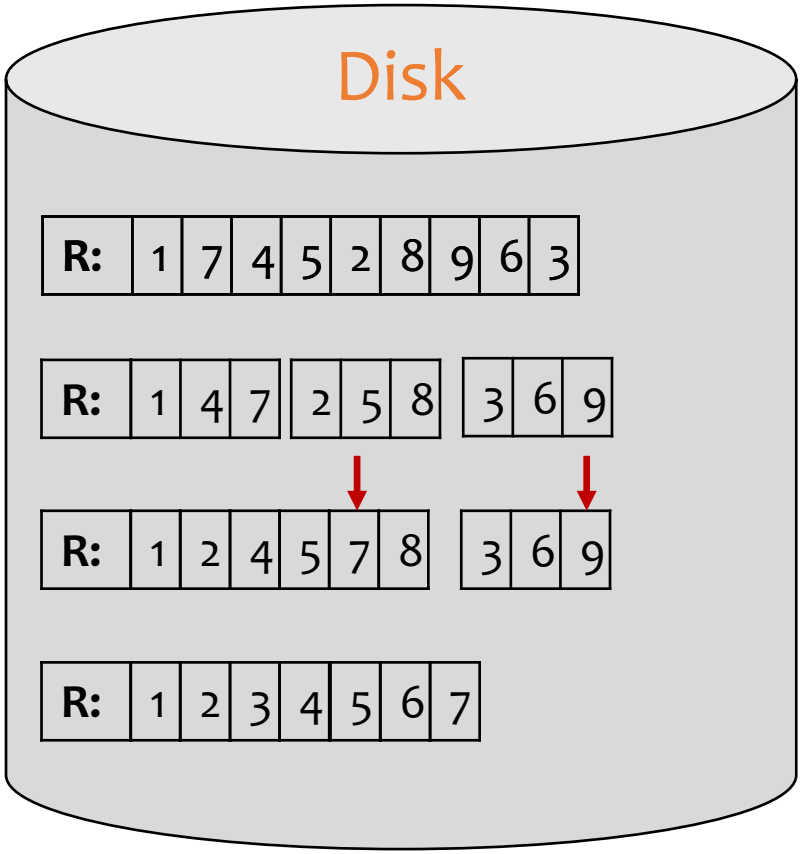
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

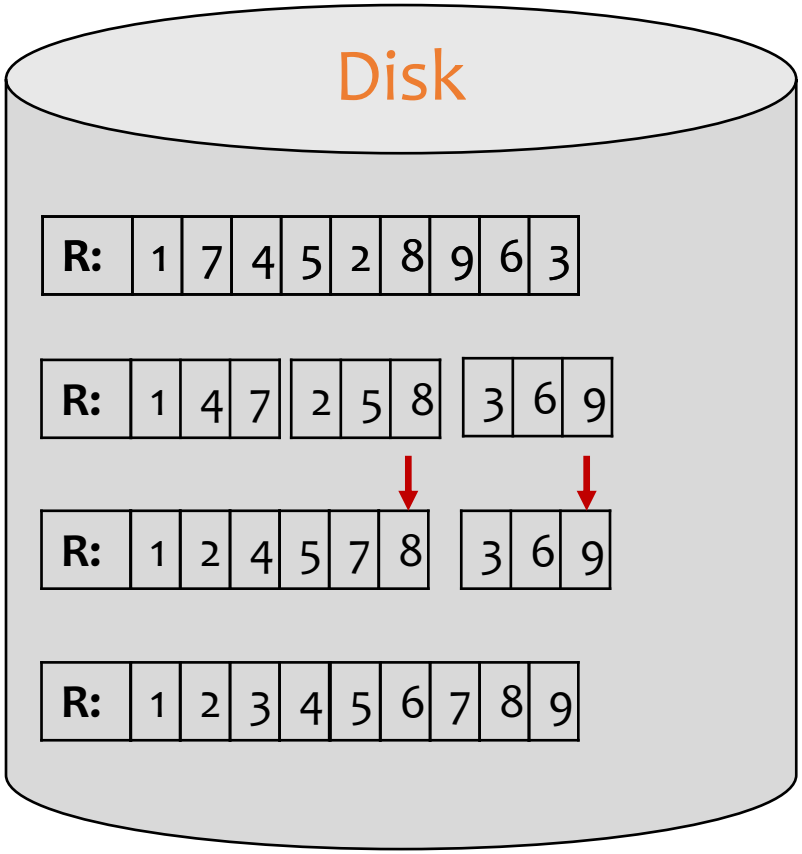
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Analysis

- Pass 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 $2 \cdot B(R)$

- Pass i: merge $(M-1)$ level-$(i-1)$ runs at a time, and write out a level-i run
  - # level-i runs = $\left\lceil \frac{\#\ \text{level}-(i-1)\ \text{runs}}{M-1} \right\rceil$

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

- Final pass produces one sorted run
  - if we don't count the output cost

I/O cost is $B(R)$

- $\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil$ number of such passes

Performance of external merge-sort

- Number of passes: I/O's
  - Multiply by $2 \cdot B(R)$: each pass reads the entire relation once and writes it once
  - Subtract $B(R)$ for the final pass
  - Roughly, this is

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

$$= O\left(B(R) \times \log_M B(R)\right)$$

- Memory requirement: M (blocks)

Sort-Merge join (SMJ)

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

- Sort R and S by their join attributes
- $r, s \leftarrow$ the first tuples in sorted R and S
- Repeat until one of R and S is exhausted:
  - If $r.A > s.B$, then $s \leftarrow$ next tuple in S
  - Else if $r.A < s.B$, then $r \leftarrow$ next tuple in R
  - Else

    output all matching tuples (if not all tuples are residing in memory, use block-based nested loop join)

    $r, s \leftarrow$ next tuples in R and S respectively

Example of merge in SMJ

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:

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

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

$r_1 s_1$

$r_2 s_3$

$r_2 s_4$

$r_3 s_3$

$r_3 s_4$

$r_7 s_5$

Performance of SMJ

User(uid) join with Member(uid, gid)

- I/O's: sorting cost + merge cost
  - Many practical cases are $O(B(R) + B(S))$ (e.g., join of key and foreign key)
  - Worst case is $O\left(\frac{B(R) \cdot B(S)}{M}\right)$: everything joins
    - Degenerates to blocked-based nested loop join
  - (Optional) In general, the cost is roughly

$$B(R) + B(S) + \sum_{a:|\sigma_{A=a}R| \geq MB \text{ or } |\sigma_{B=a}S| \geq MB} \frac{B(\sigma_{A=a}R) \cdot B(\sigma_{B=a}S)}{M}$$

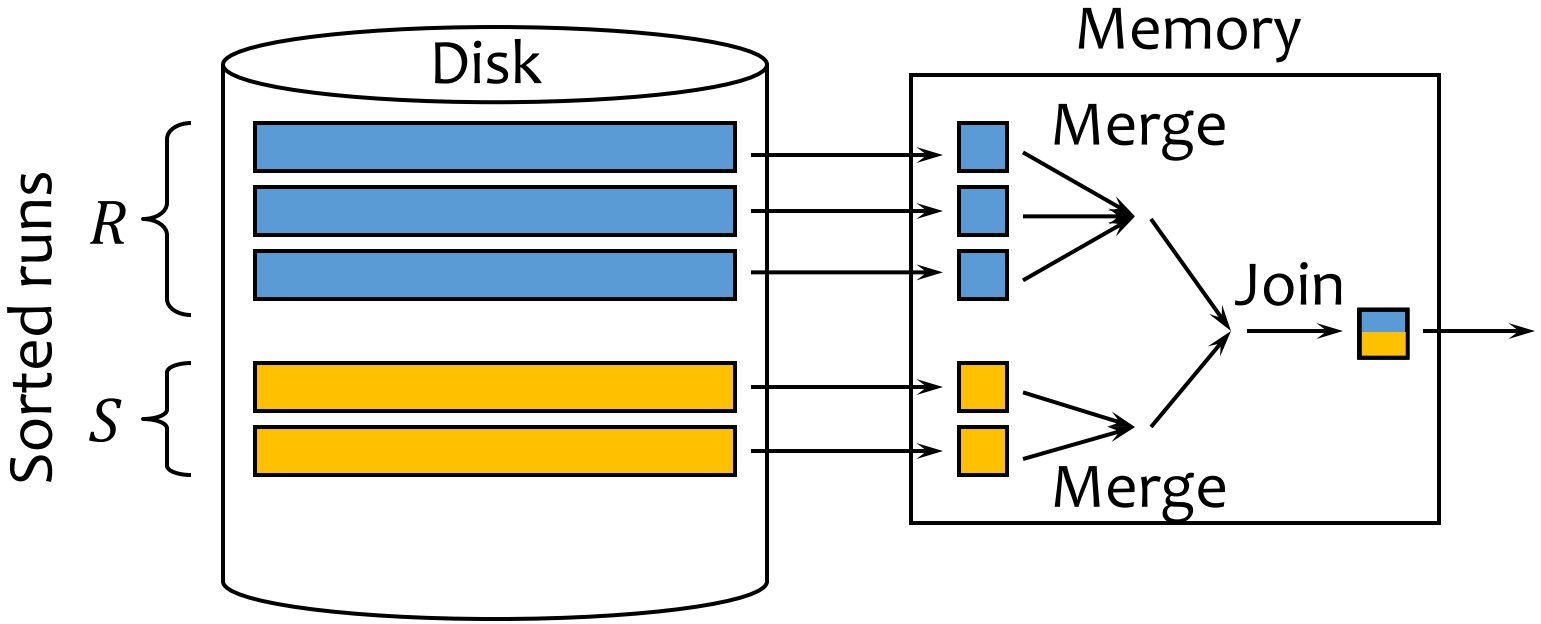
$$= O\left(B(R) + B(S) + \frac{B(R \bowtie S)}{MB}\right)$$

Theoretically, this is optimal

- Memory requirement: $M \geq 3$ (blocks)
  - Increase M improves I/O

Optimization of SMJ

- Idea: combine join with the (last) merge phase of merge-sort
  - Sort: produce sorted runs for R and S such that there are fewer than M of them total
  - Merge and join: merge the runs of R, merge the runs of S, and merge-join the result streams as they are generated!

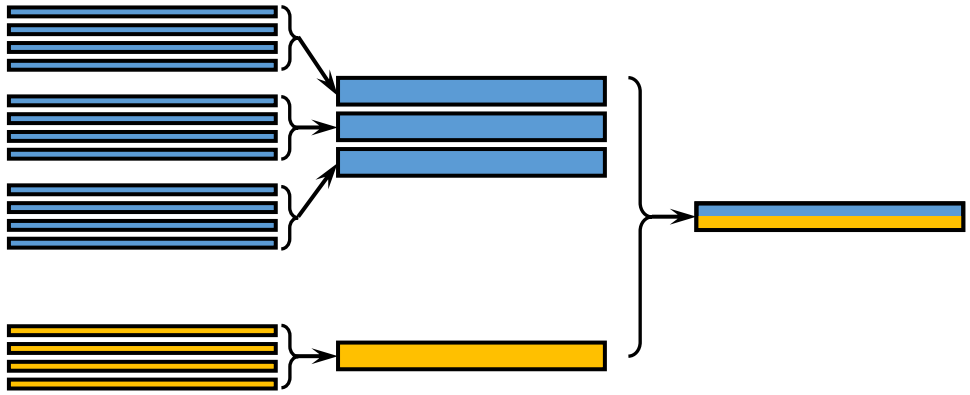


Performance of Optimized SMJ

- If SMJ completes in two passes:
  - I/O's: $3 \cdot \big(B(R) + B(S)\big)$
  - Memory requirement
    - We must have enough memory to accommodate one block from each run: $M > \left\lceil \frac{B(R)}{M} \right\rceil + \left\lceil \frac{B(S)}{M} \right\rceil$
    - Roughly $M > \sqrt{B(R) + B(S)}$

The first pass for sorting and the second pass for merge-join

- If SMJ cannot complete in two passes:
  - Repeatedly merge to reduce number of runs as necessary before final merge and join



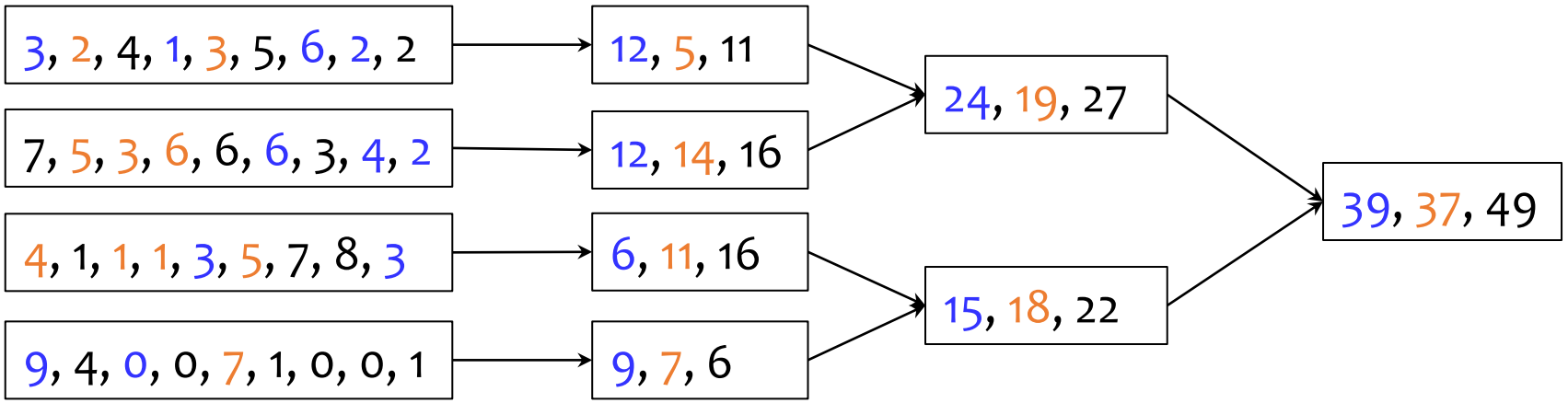
Other sort-based algorithms

- Union, difference, intersection
  - More or less like SMJ
- Duplication elimination
  - External merge sort
    - Eliminate duplicates in sort and merge
- Grouping and aggregation
  - External merge sort by group-by columns
    - Trick: produce "partial" aggregate values in each run, and combine them during merge

Example of Aggregation

Compute the sum of numbers for each color ● ● ● using partial aggregate values

3 memory blocks available
Each block holds 3 numbers



Beside SUM, the same trick works for COUNT, MIN, MAX but not COUNT(Distinct), Median etc.

What is next?

- Scan
  - Table scan
  - Selection, Duplicate-preserving projection
  - Nested-loop join
- Sort
  - External merge sort
  - Duplicate elimination, Grouping and Aggregation
  - SMJ, Union, Difference, Intersection
- Hash
- Index