Lecture 16: Query Processing & Optimization

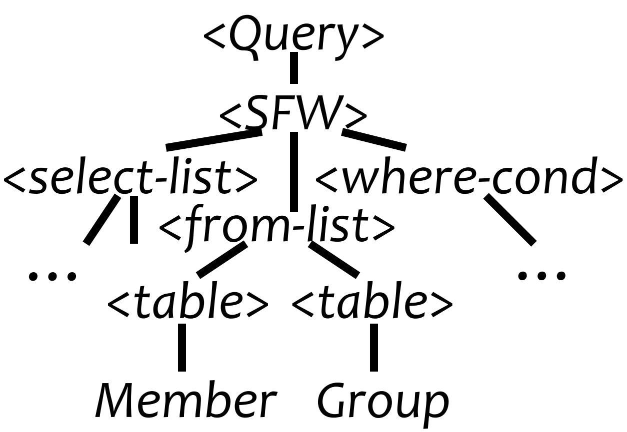
CS348 Spring 2025:
Introduction to Database Management

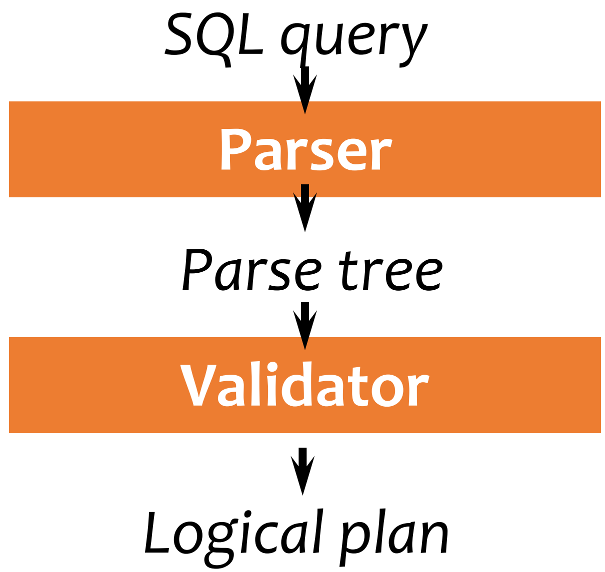
Instructor: **Xiao Hu**
Sections: 001, 002, 003

Announcements

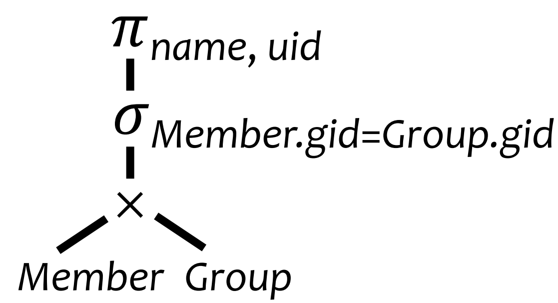
- Milestone 2 of group project
  - Due on July 8!

A query's trip through the DBMS






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

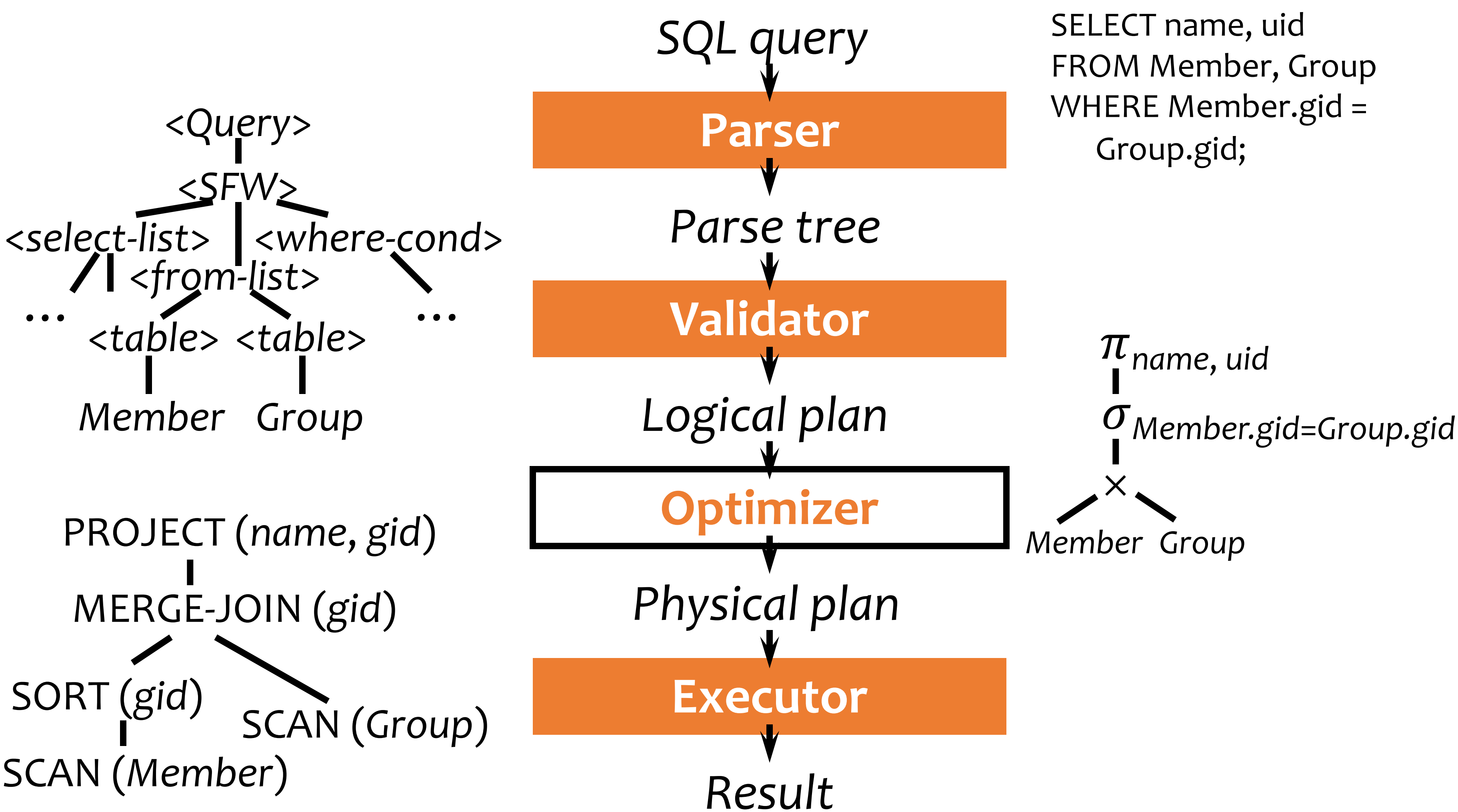


Query parsing and validation

- 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



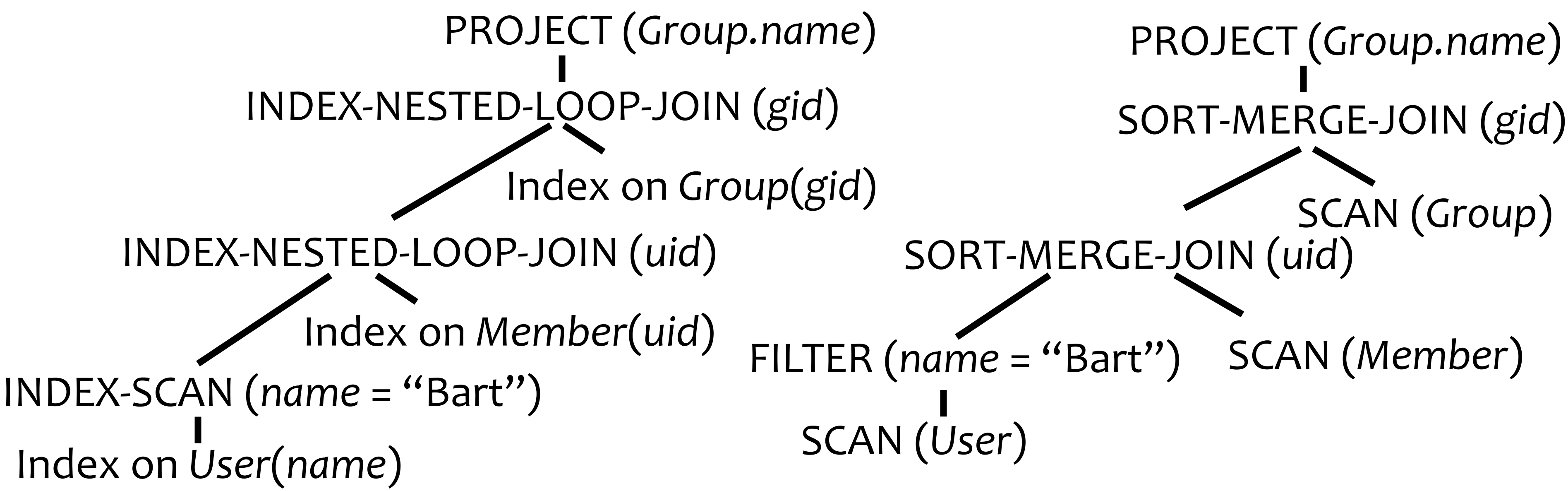
Physical plan

- 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 processing 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;

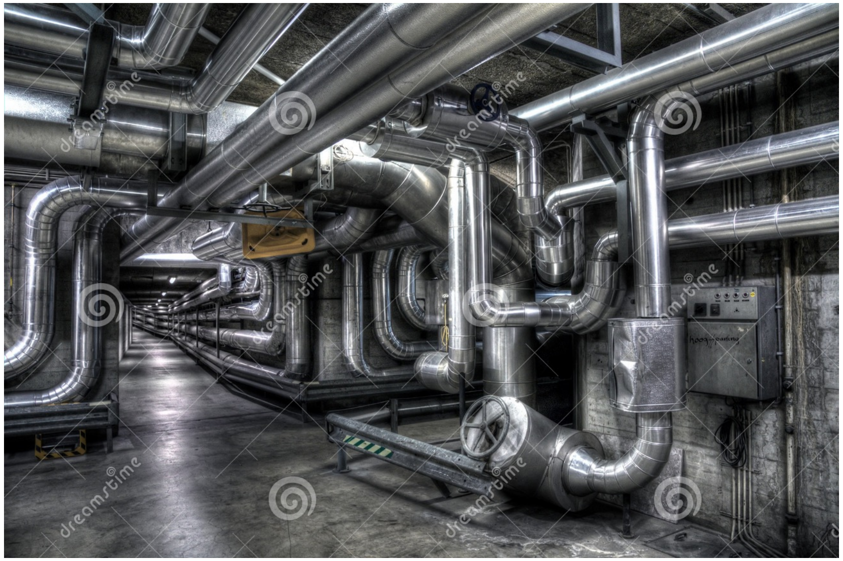


- Many physical plans for a single query

Execution of physical plans

What is the algorithm for each operator?

How are intermediate results passed from child operators to parent operators?

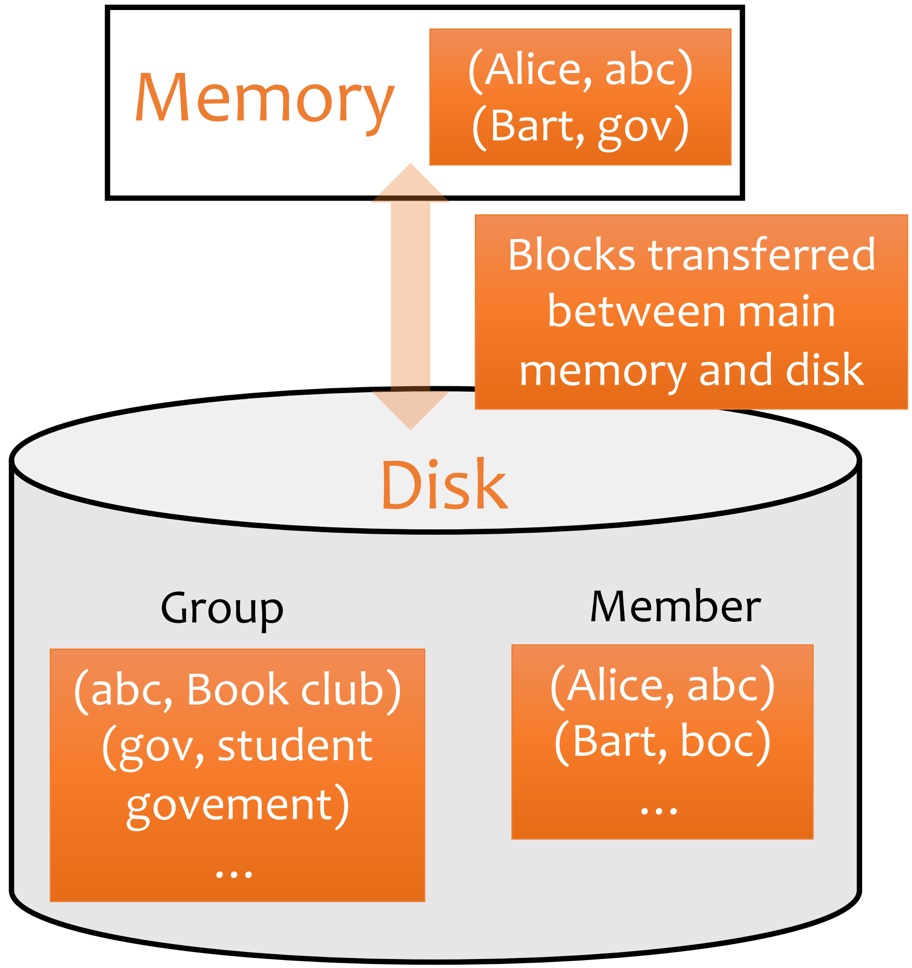


http://www.dreamstime.com/royalty-free-stock-image-basement-pipelines-grey-image25917236

- 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



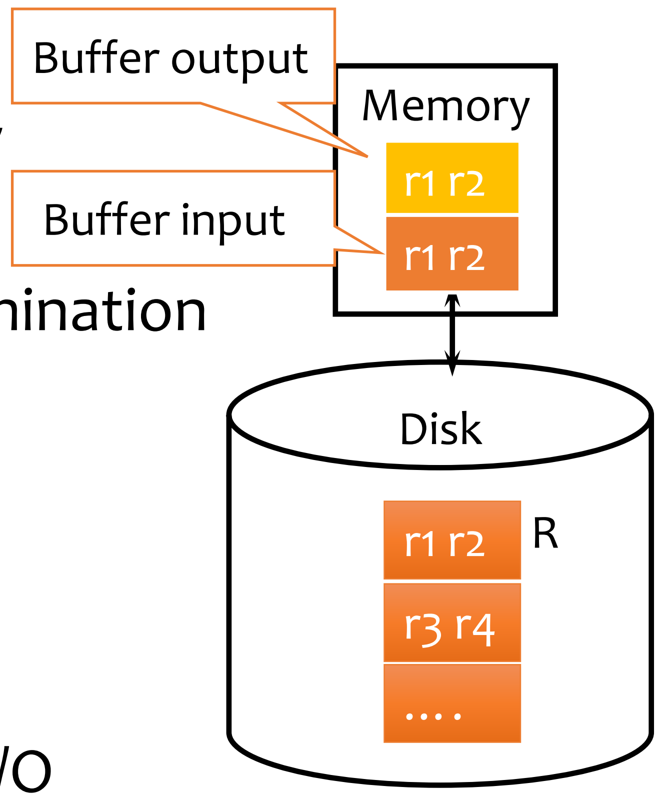
Notation and Assumption

- 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
  - Memory requirement
- Not counting the cost of writing the result out
  - Same for any algorithm!
  - Consumed by subsequent operators

Scanning-based algorithms

Table scan

- Scan table *R* and process the query
  - Selection over *R*
  - Projection of *R* without duplicate elimination
- I/O's: $B(R)$
  - 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
- Selection, Duplication-preserving projection



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

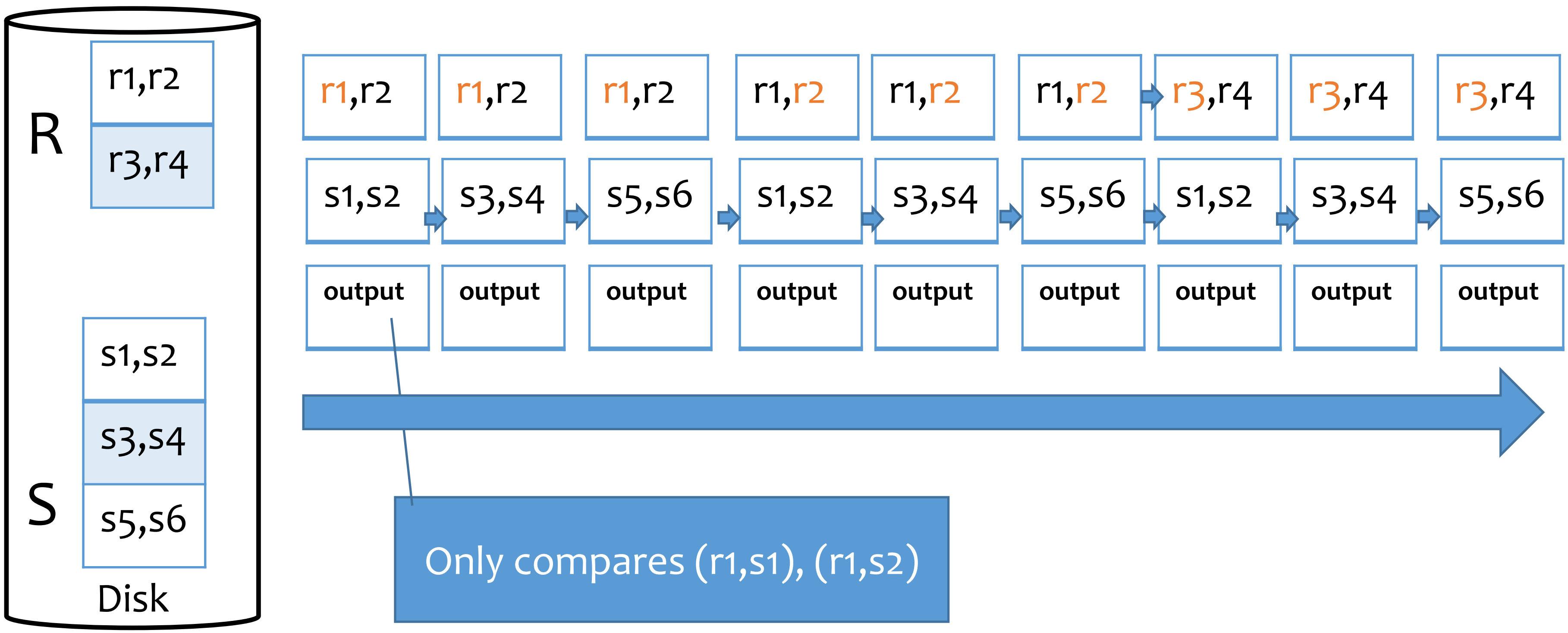
Blocks of R are moved into memory only once

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

- Memory requirement: 3

Tuple-based nested-loop join

- one block stores two 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: $B(R) + B(R) \cdot B(S)$

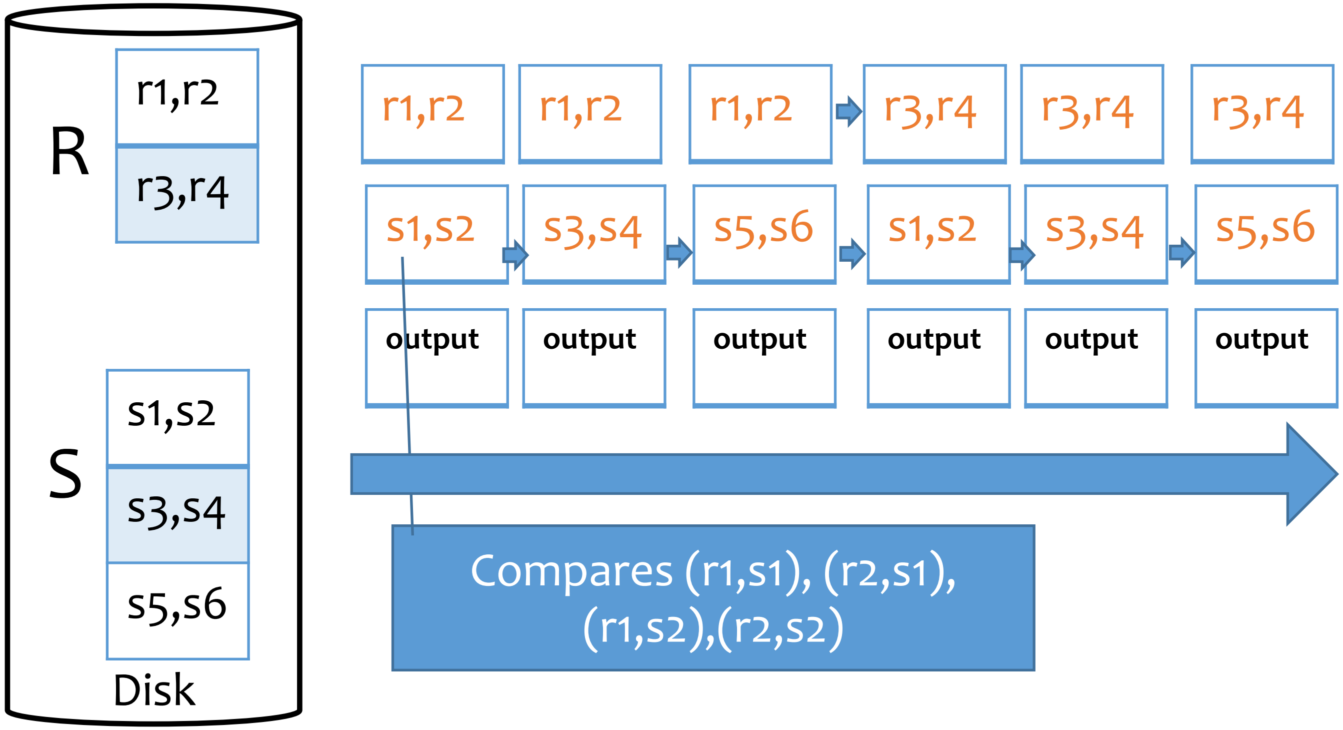
Blocks of R are moved into memory only once

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

- Memory requirement: same as before

Block-based nested loop join

- one block stores two 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 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)

What about nested-loop join?

- May be best if many tuples join
    - Example: 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

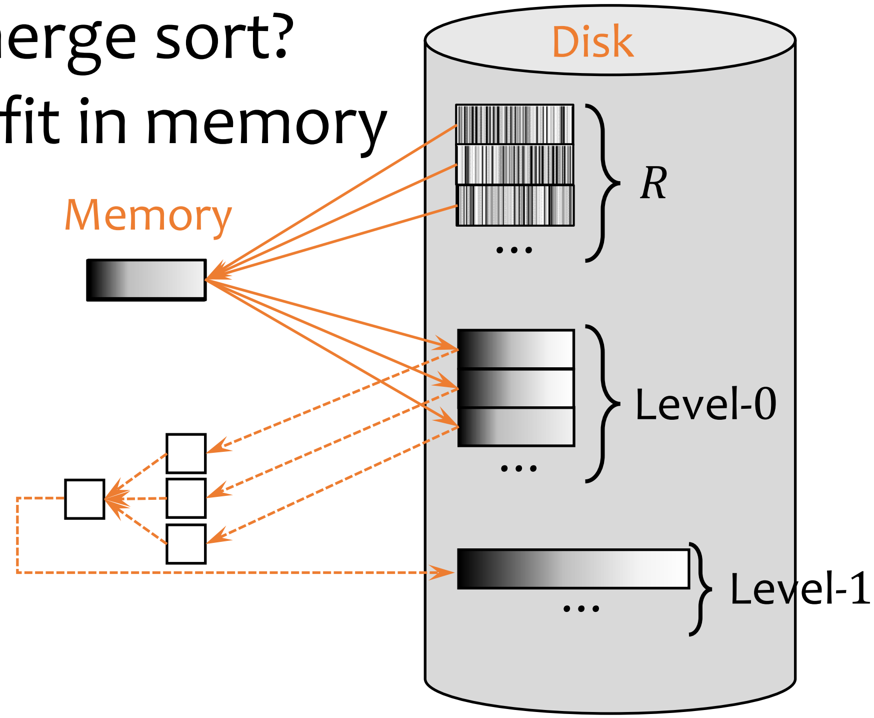
Remember (internal-memory) merge sort?

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

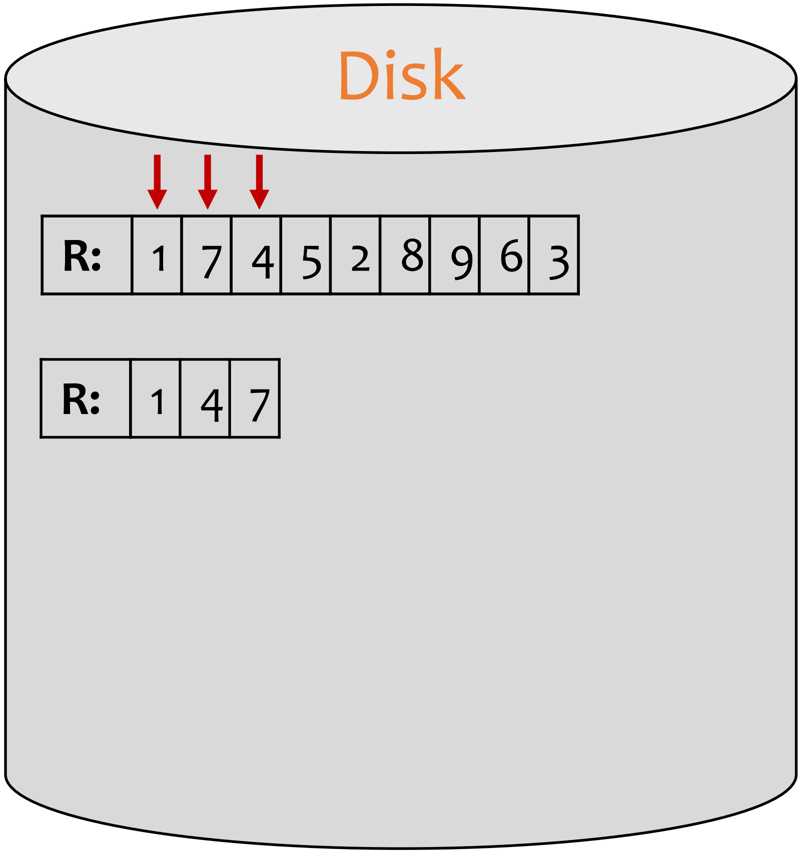
…

- Final pass produces one sorted run



Example of external merge-sort

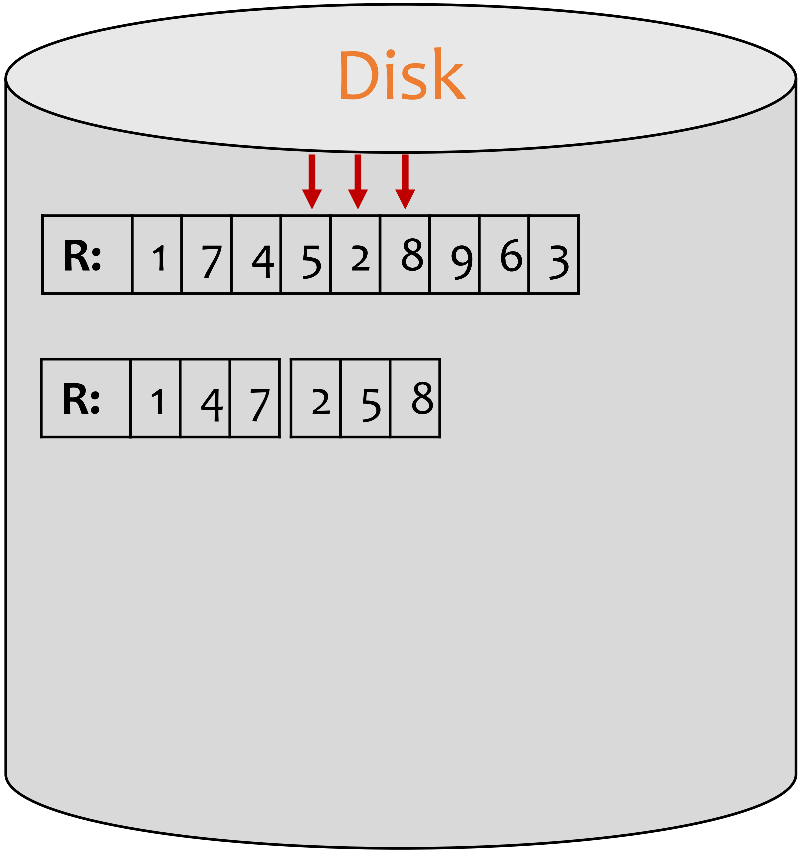
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:



Arrows indicate the blocks in memory

Example of external merge-sort

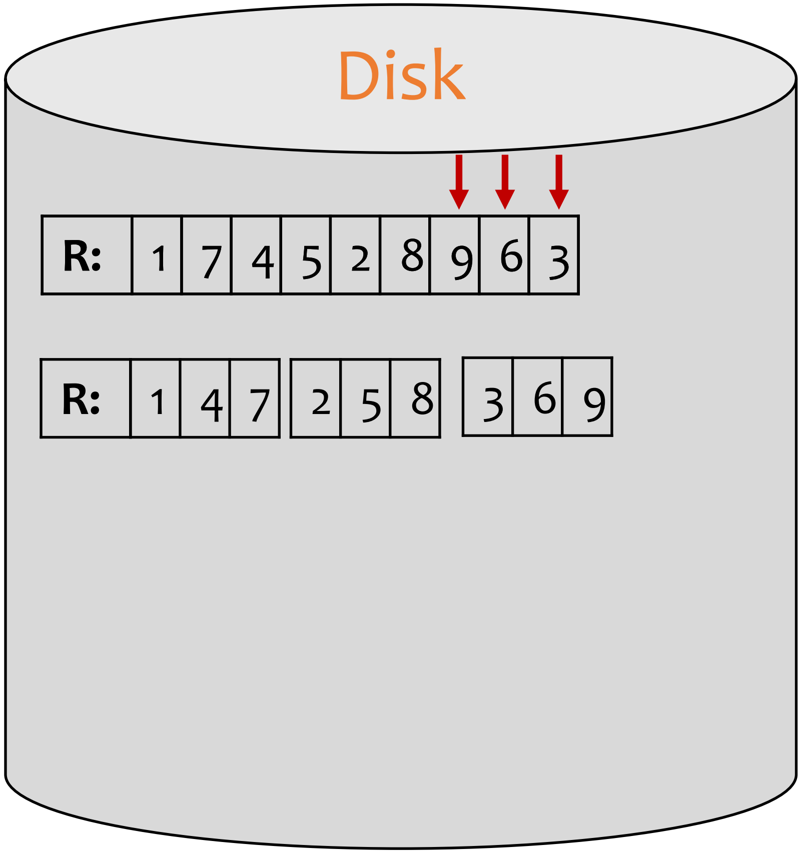
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:



Arrows indicate the blocks in memory

Example of external merge-sort

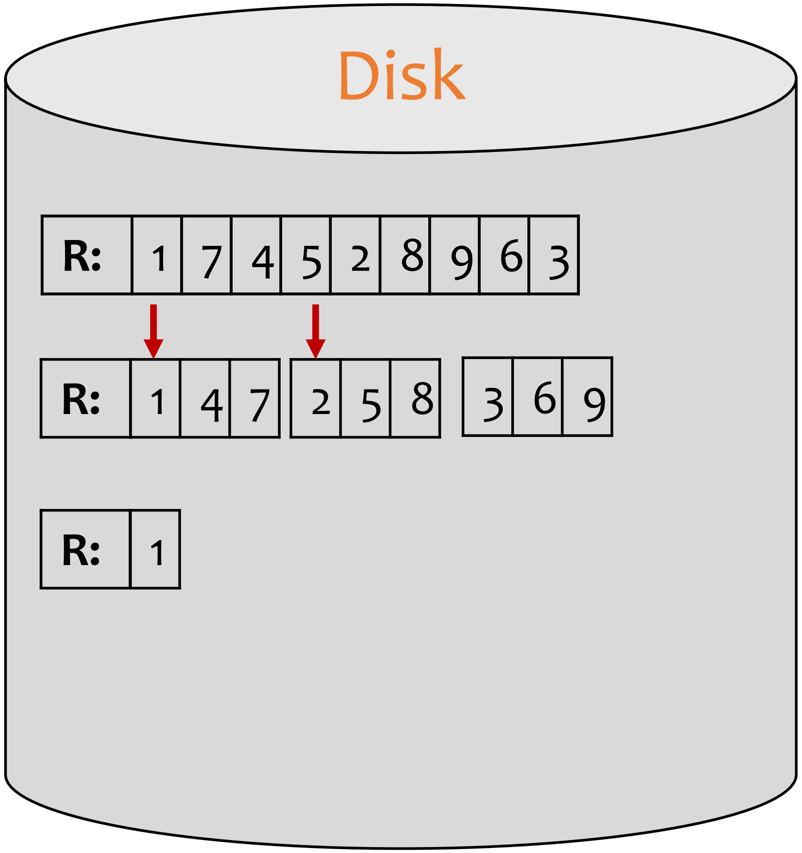
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:



Arrows indicate the blocks in memory

Example of external merge-sort

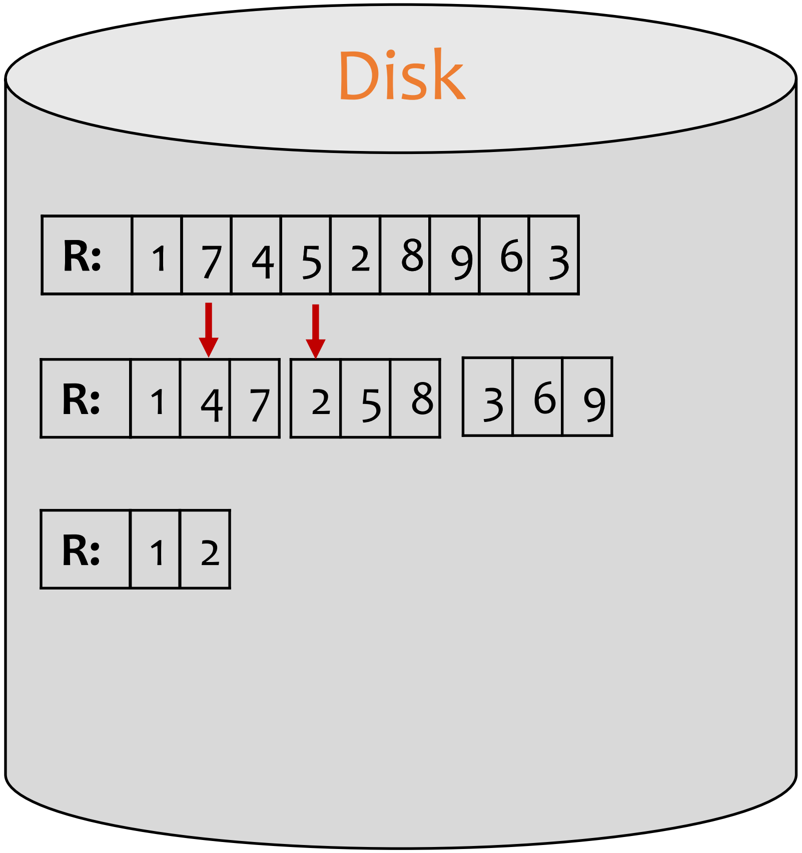
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

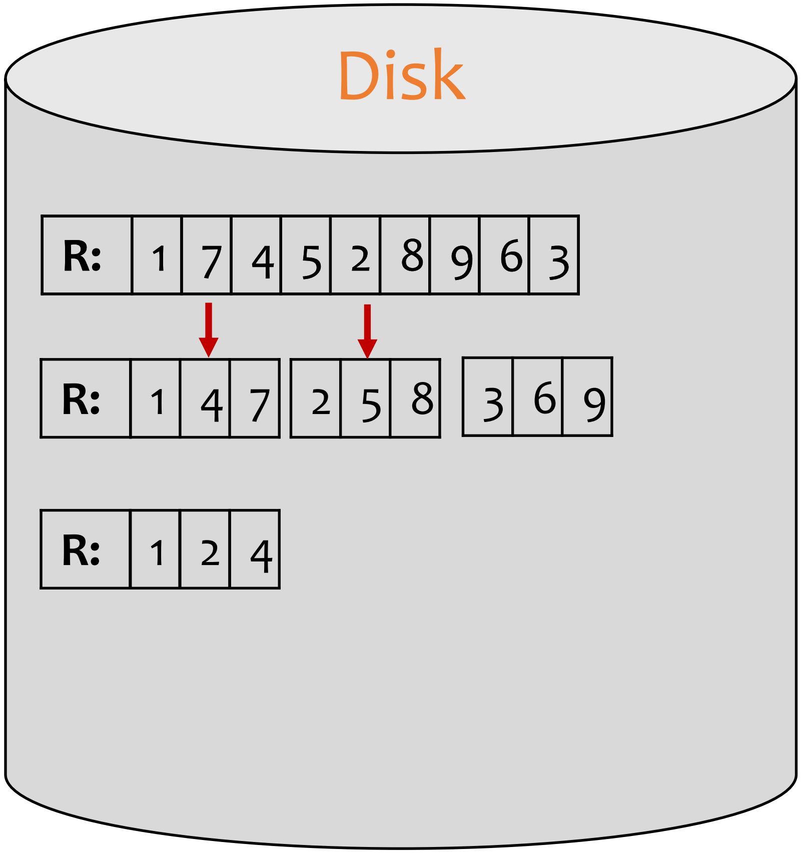
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

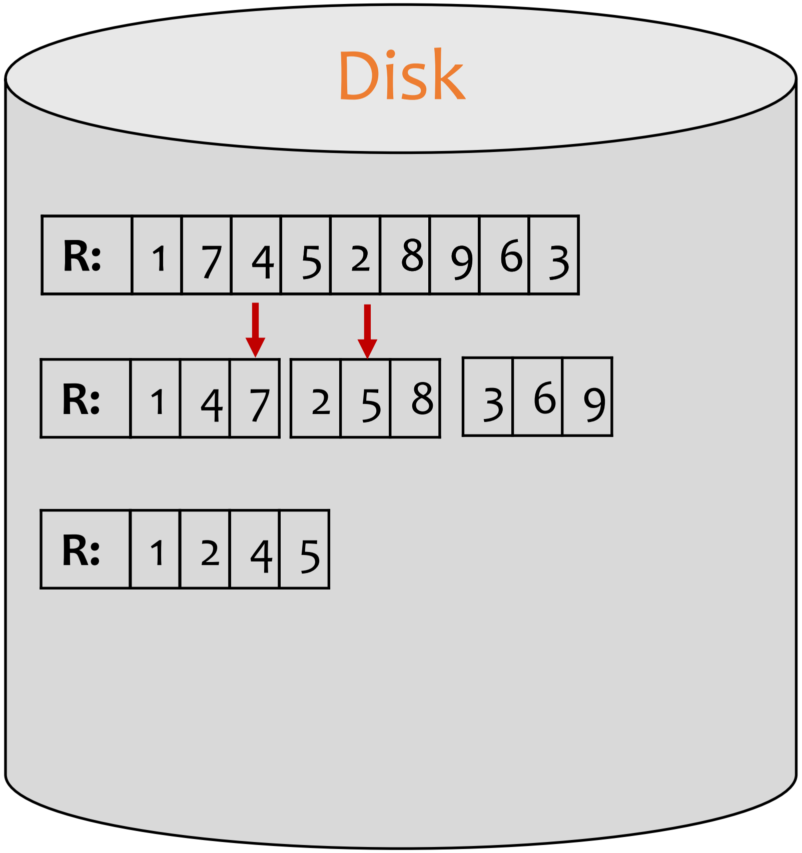
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

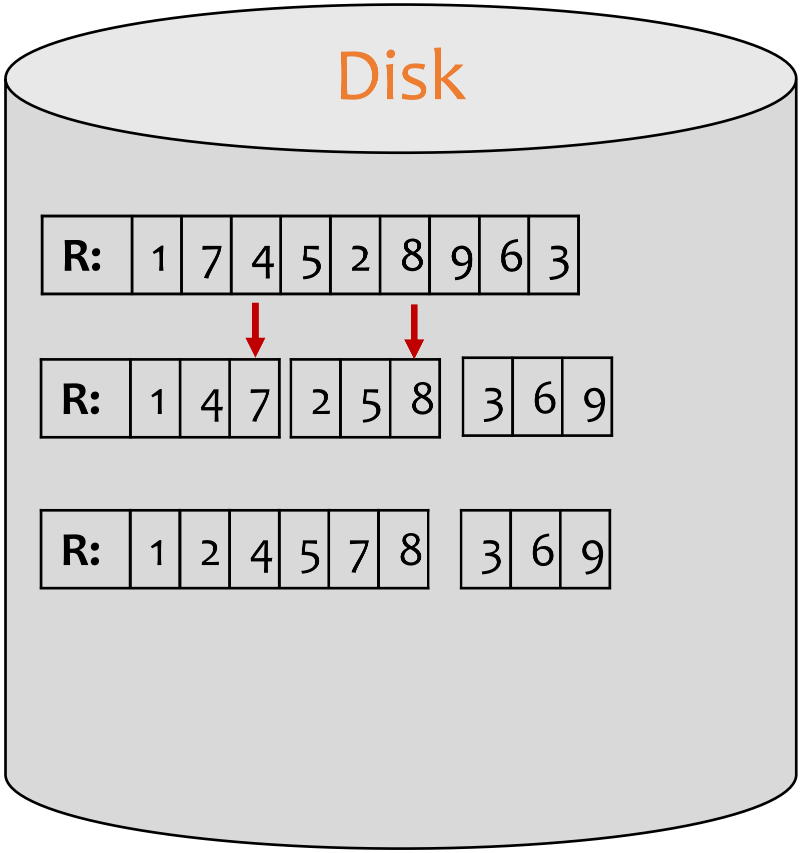
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

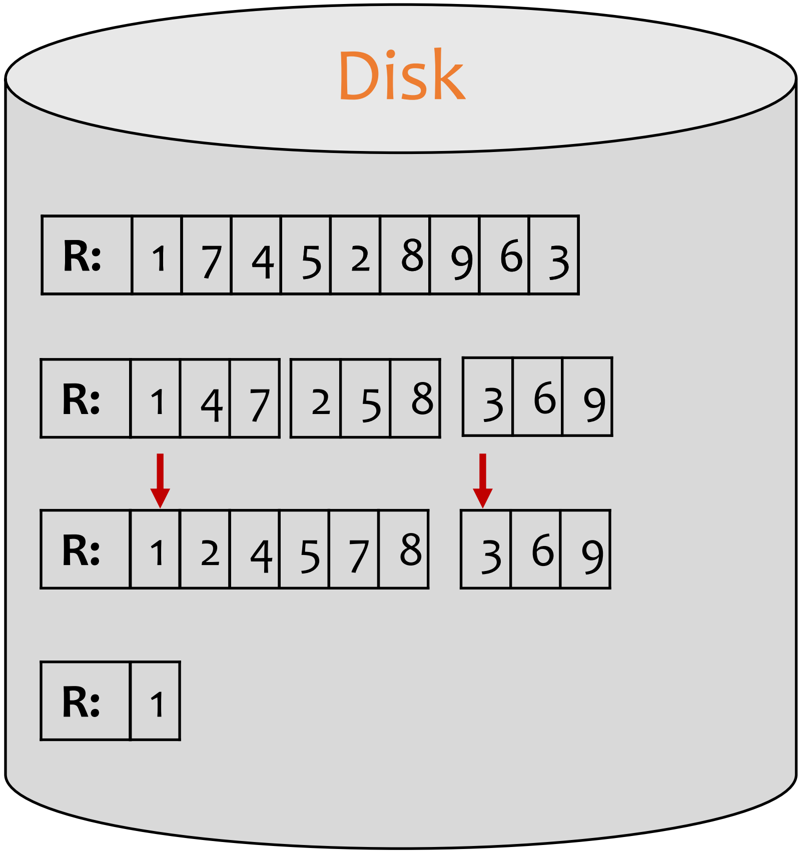
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:



Arrows indicate the blocks in memory

Example of external merge-sort

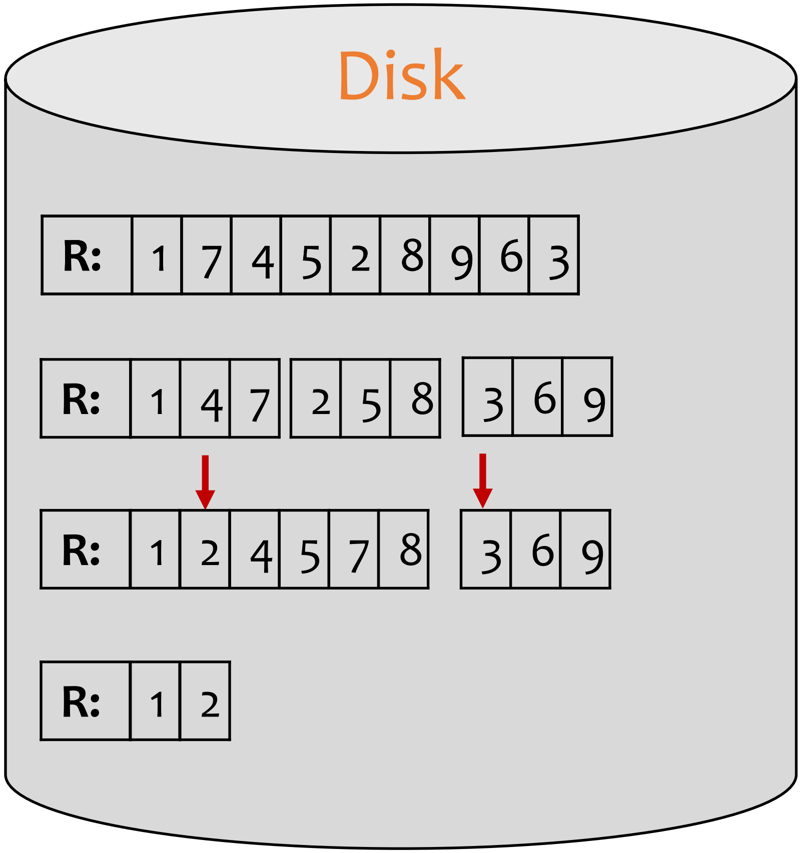
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

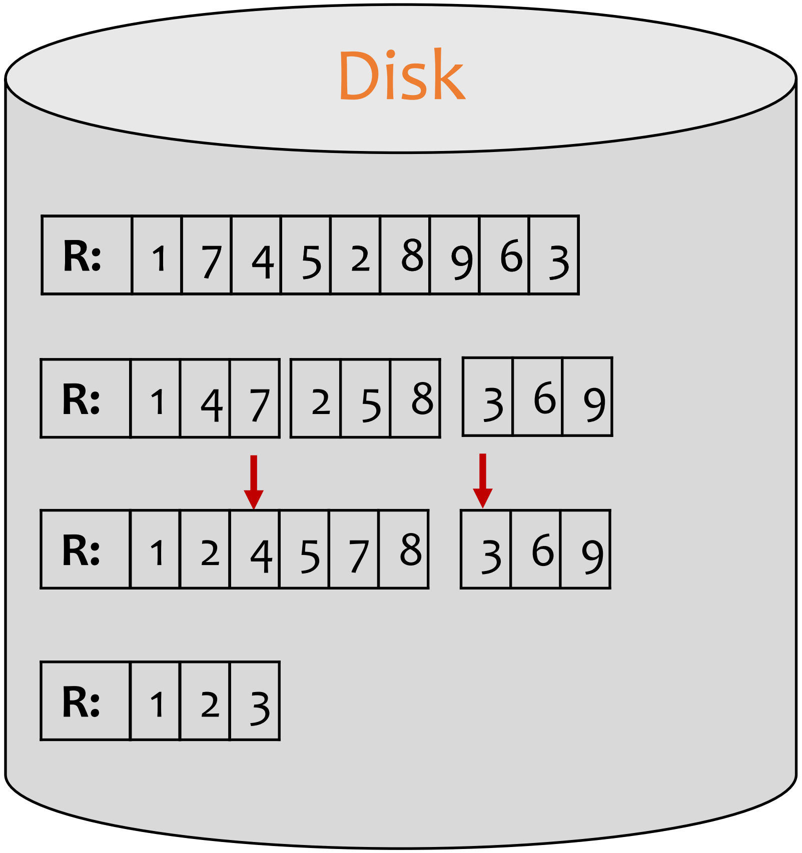
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

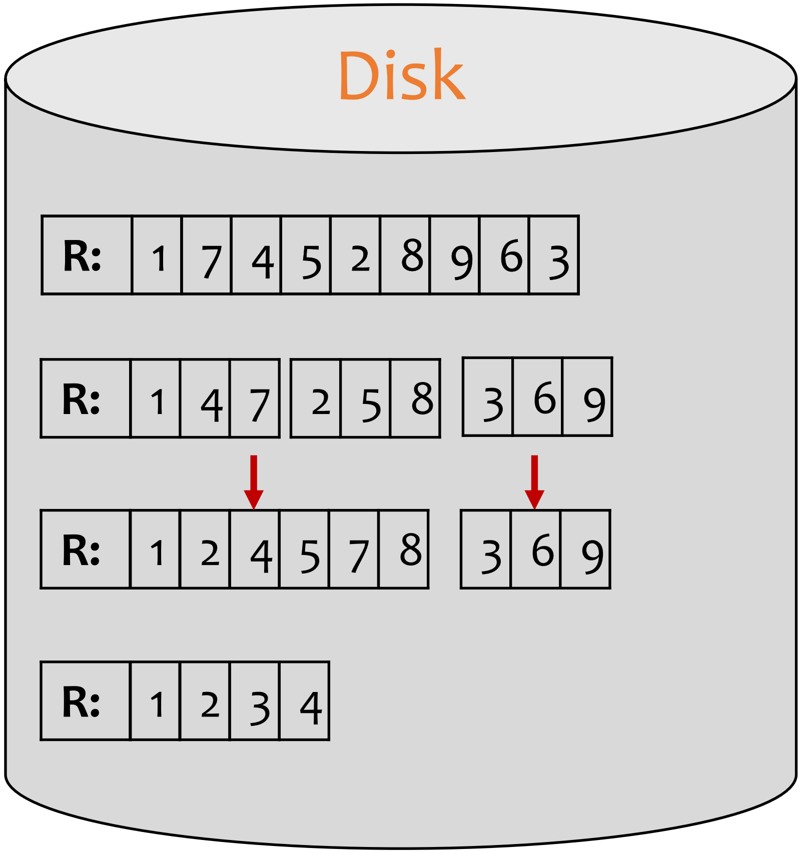
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

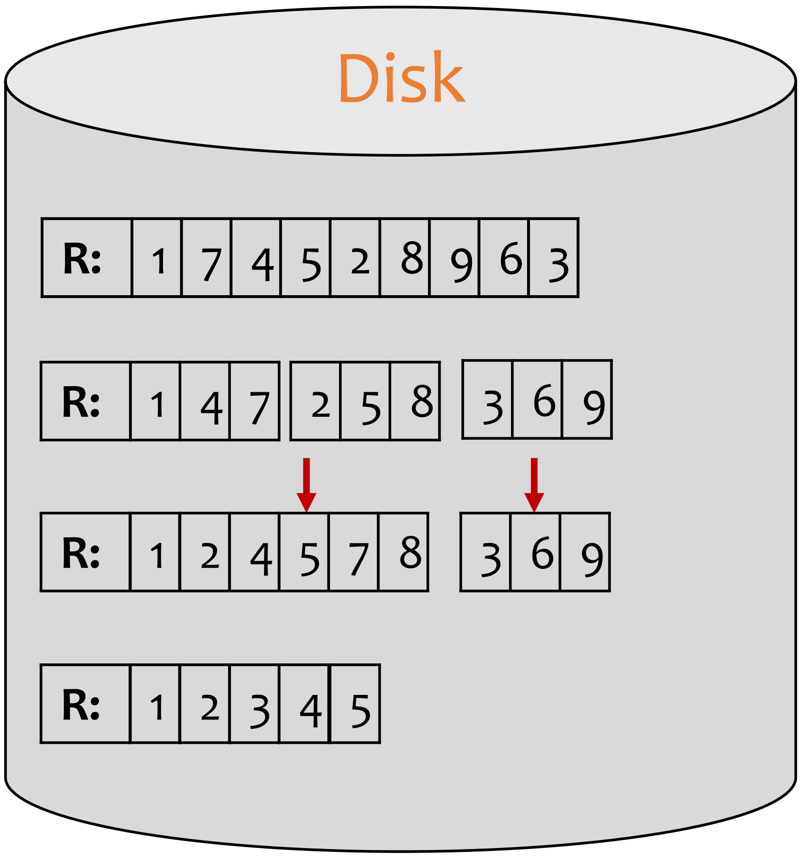
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

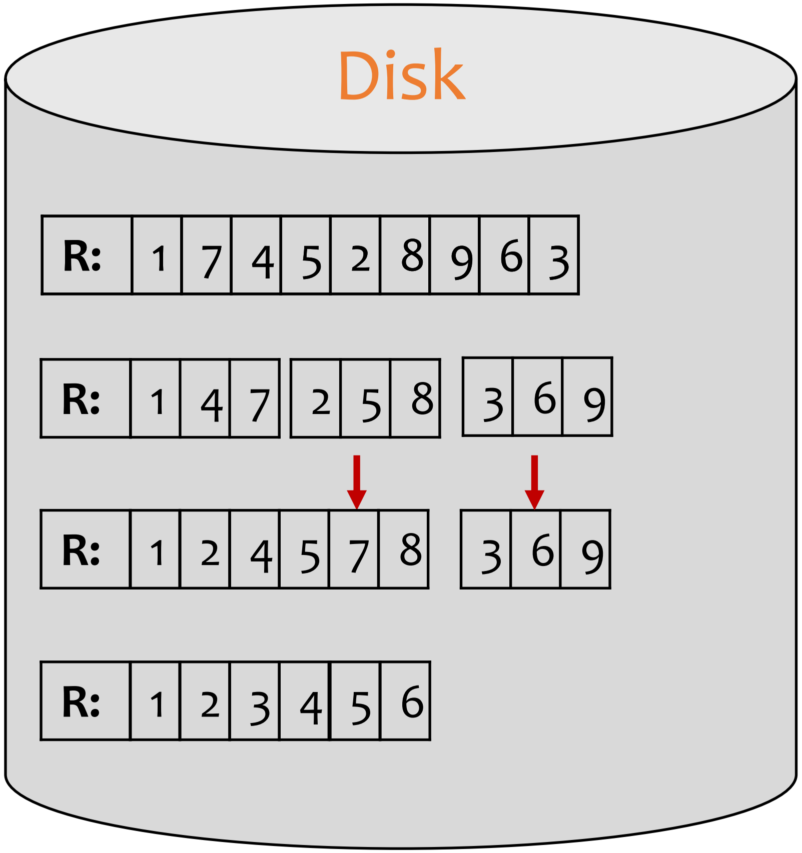
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

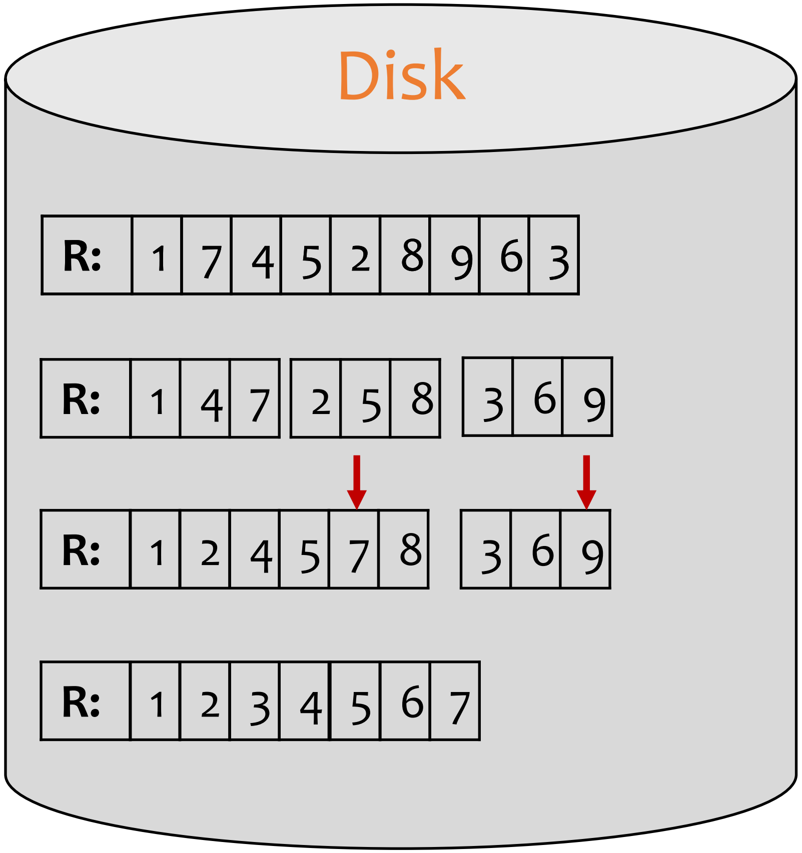
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



Arrows indicate the blocks in memory

Example of external merge-sort

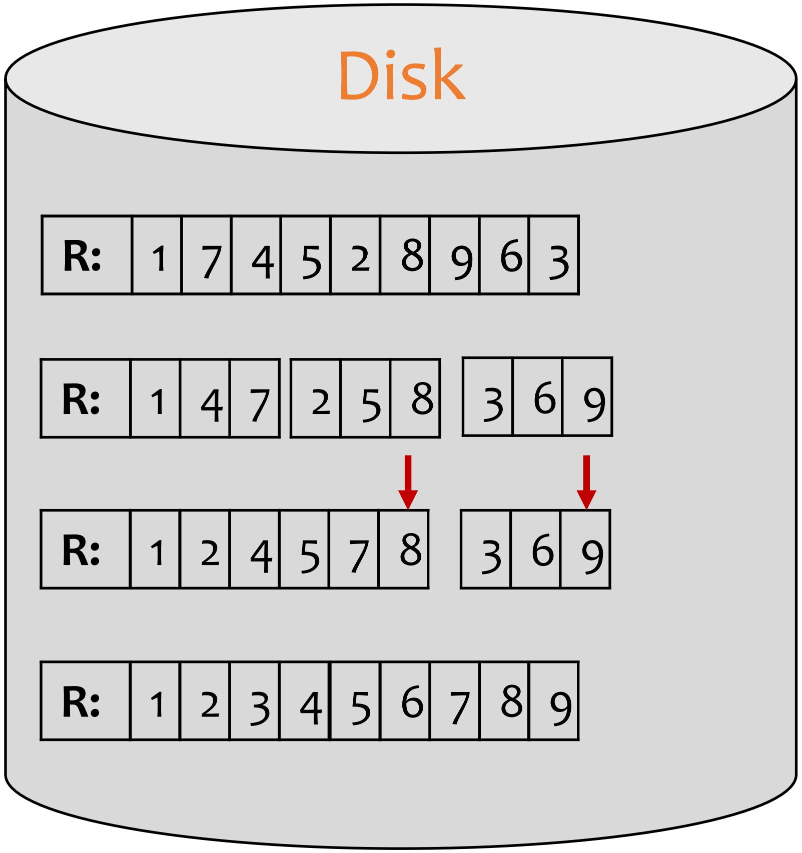
- 3 memory blocks available
- Each block holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Pass 0:
- Pass 1:
- Pass 2:



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
  - $(M-1)$ 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 $2 \cdot B(R)$ times # of phases

Subtract $B(R)$ for the final pass

Performance of external merge-sort

- Number of passes: $\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil + 1$
- 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 $O\left(B(R) \times \log_M B(R)\right)$
- Memory requirement: M (as much as possible)

Sort-merge join

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

- 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 $(r.A = s.B)$

    output all matching tuples;
    $r, s \leftarrow$ next tuples in R and S respectively
- If R is not exhausted, output remaining tuples in R
- If S is not exhausted, output remaining tuples in S

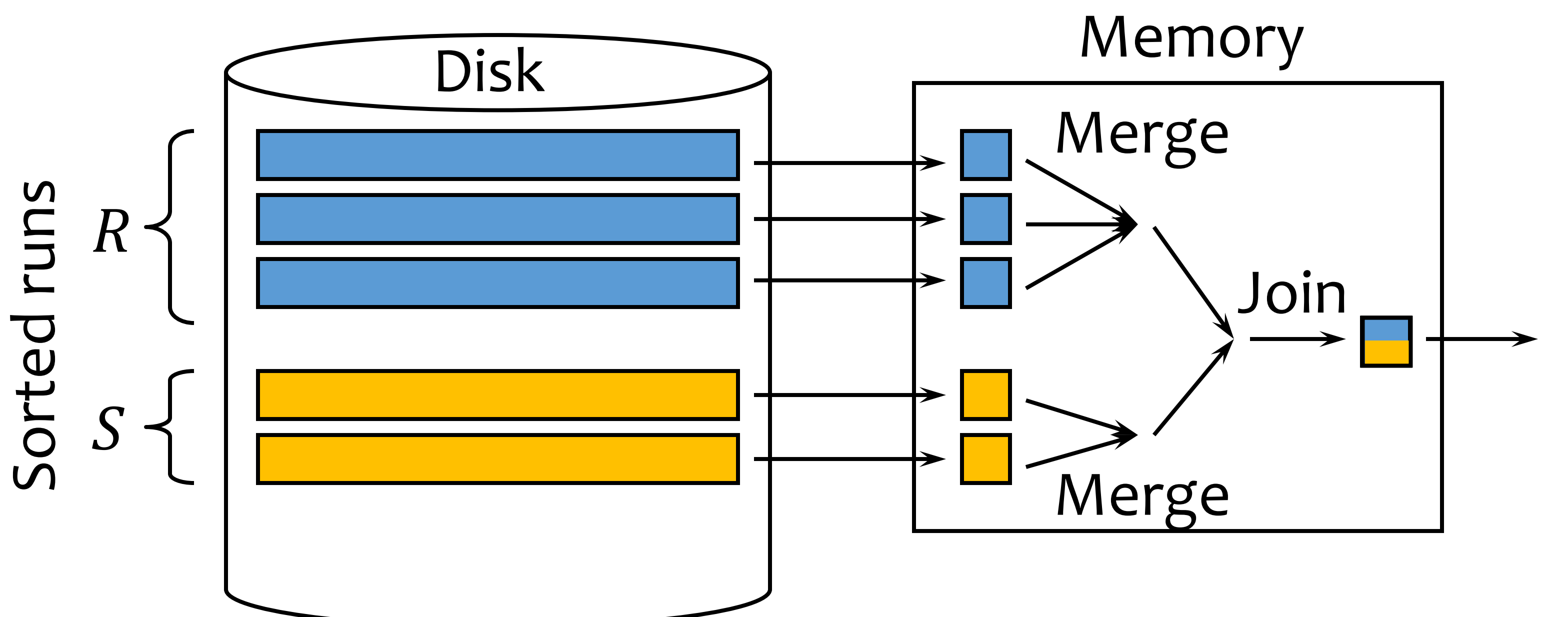
Example of merge join

R:	S:	$R \bowtie_{R.A=S.B} S$:
➡ $r_1.A = 1$	➡ $s_1.B = 1$	$r_1 s_1$
➡ $r_2.A = 3$	➡ $s_2.B = 2$	$r_2 s_3$
$r_3.A = 3$	➡ $s_3.B = 3$	$r_2 s_4$
➡ $r_4.A = 5$	$s_4.B = 3$	$r_3 s_3$
➡ $r_5.A = 7$	➡ $s_5.B = 8$	$r_3 s_4$
➡ $r_6.A = 7$		$r_7 s_5$
➡ $r_7.A = 8$		

- 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

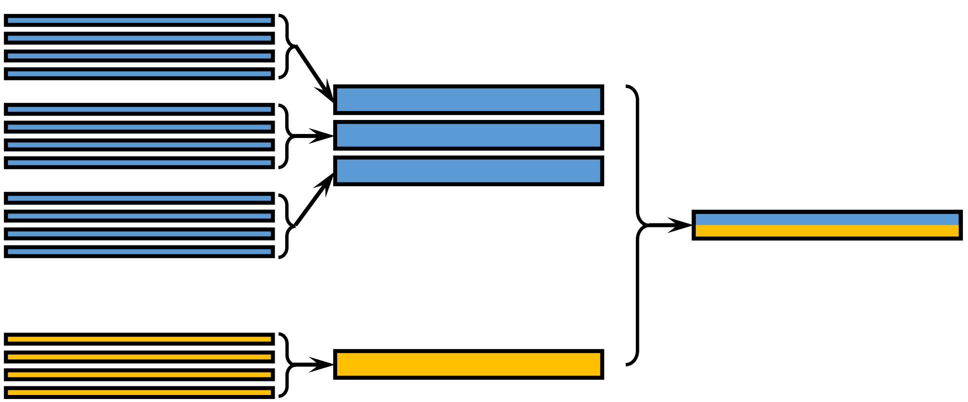
Optimization of SMJ

- 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 SMJ

- If SMJ completes in two passes:
  - I/O's: $3 \cdot \left(B(R) + B(S)\right)$
  - 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 $M > \sqrt{B(R) + B(S)}$
- If SMJ cannot complete in two passes:
  - Repeatedly merge to reduce number of runs as necessary before final merge and join



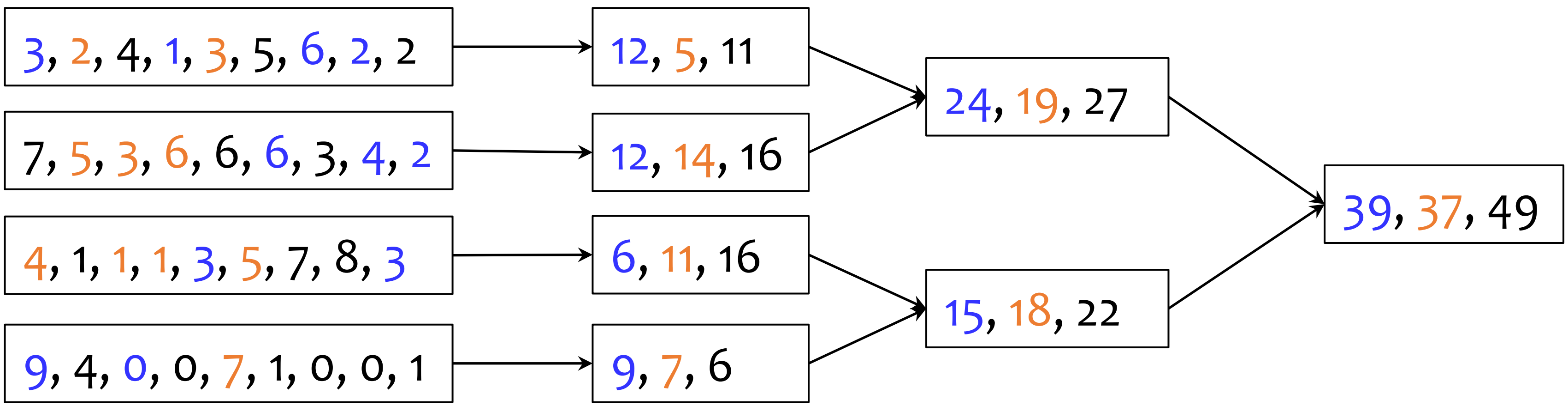
Other sort-based algorithms

- Union (set), 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
And also AVG, STDEV, with some little twists

Truly "holistic" aggregates?

E.g., SUM(DISTINCT …)

- Sort by the input expression (within each group)
- Example:
  SELECT SUM(DISTINCT B)
  FROM R GROUP BY A;
  - Sort R by (A, B)

☞But if there are multiple holistic aggregates with different input expressions, a single global order won't work — a group may need to be resorted

☞Instead of sorting, specialized algorithms exist for some aggregates, e.g.: median/quantile

What is next?

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