Query Optimization: Sorting & Joining

CS 377: Database Systems

Recap: Query Processing

- Some database operations are expensive
- Performance can be improved by being "smart"
- RA expressions can be optimized via heuristics
- Cost-based optimization to determine "best" query plan

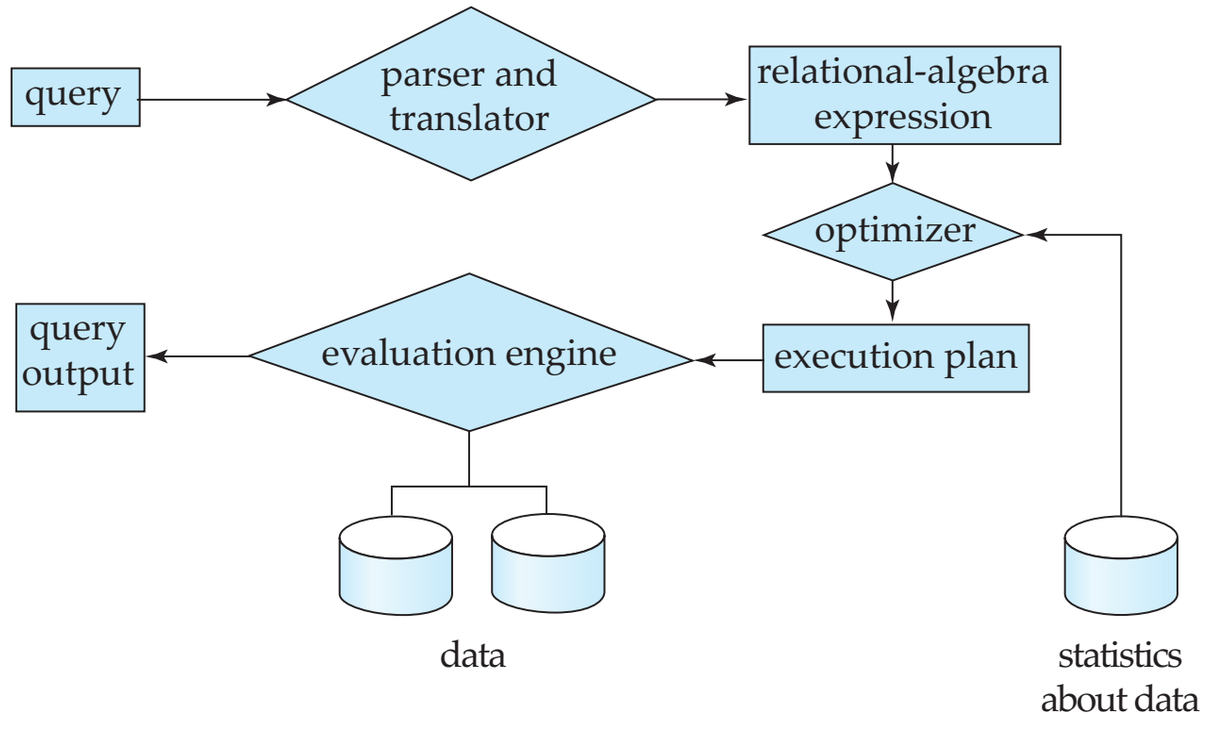
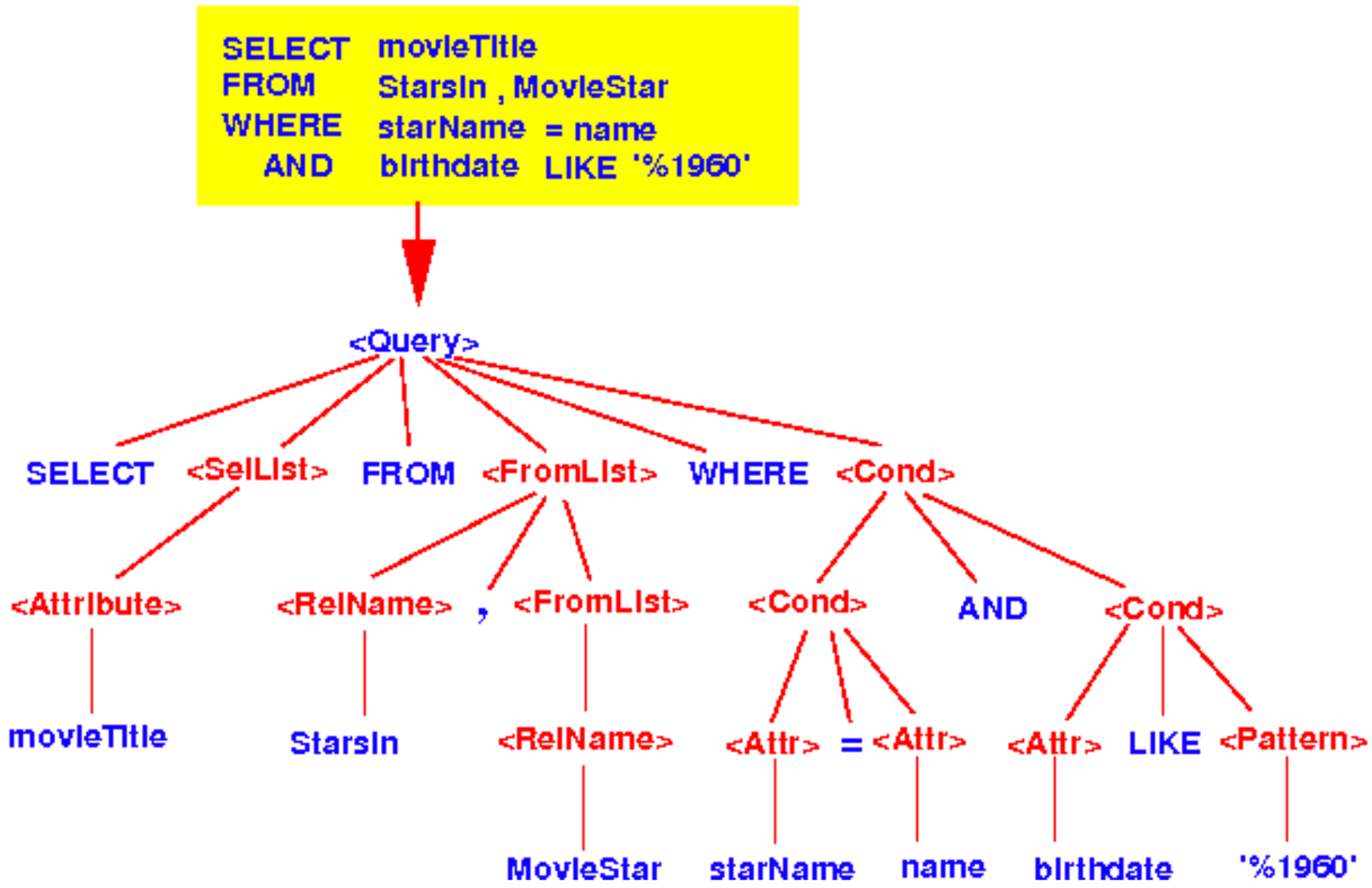


Figure 12.1 from Database System Concepts book

Example: SQL Query Step 1

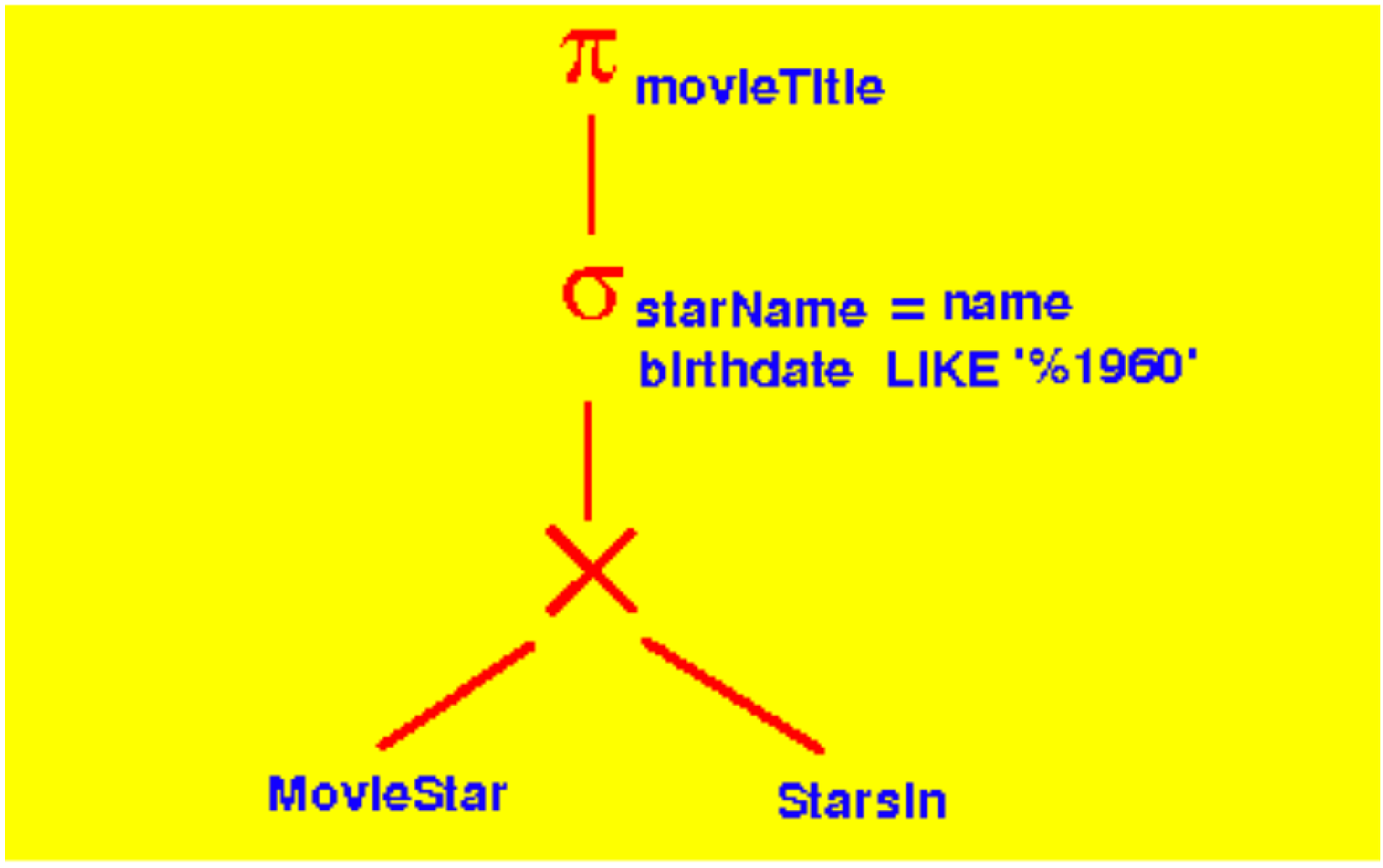
Step 1: Convert SQL query into a parse tree



http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/intro.html

Example: SQL Query Step 2

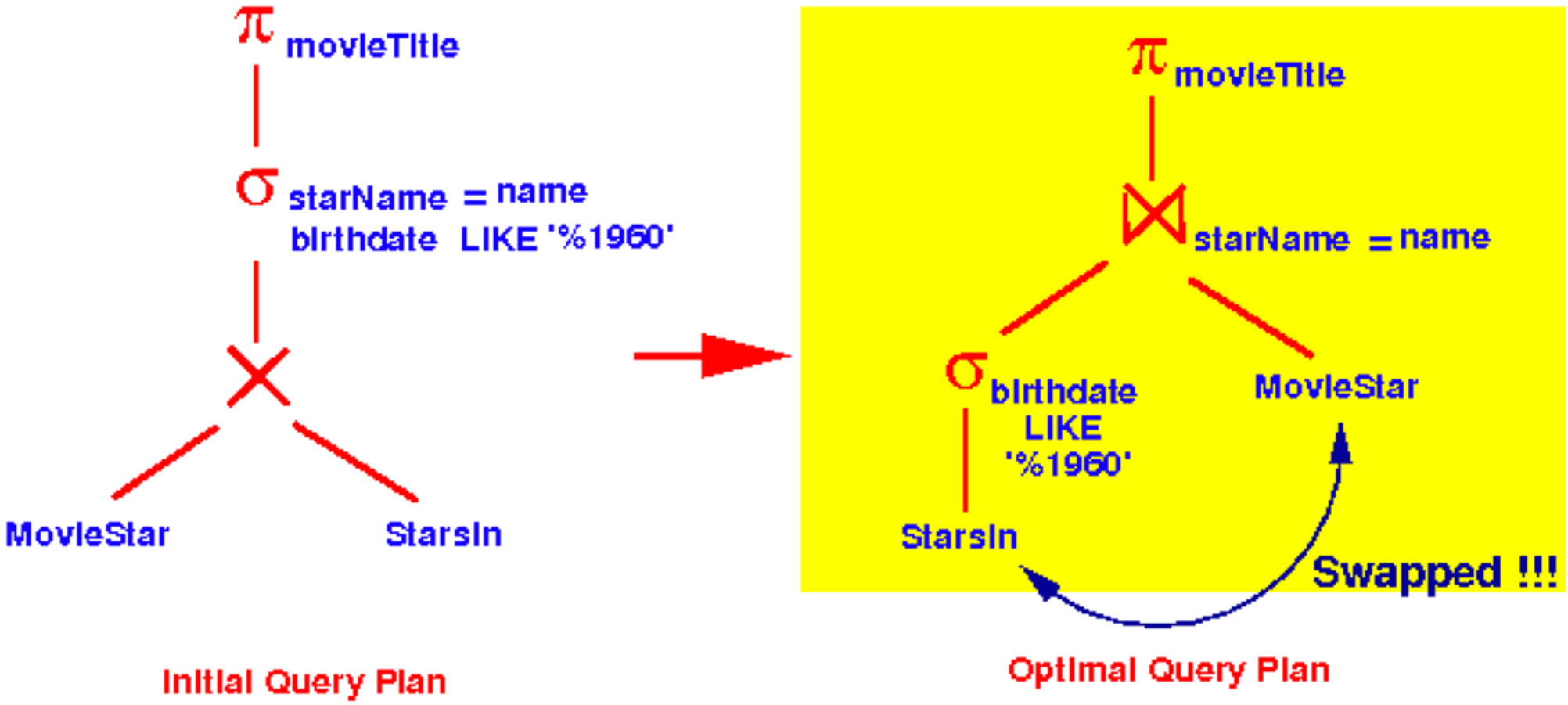
Step 2: Convert parse tree into initial logical query plan using RA expression



http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/intro.html

Example: SQL Query Step 3

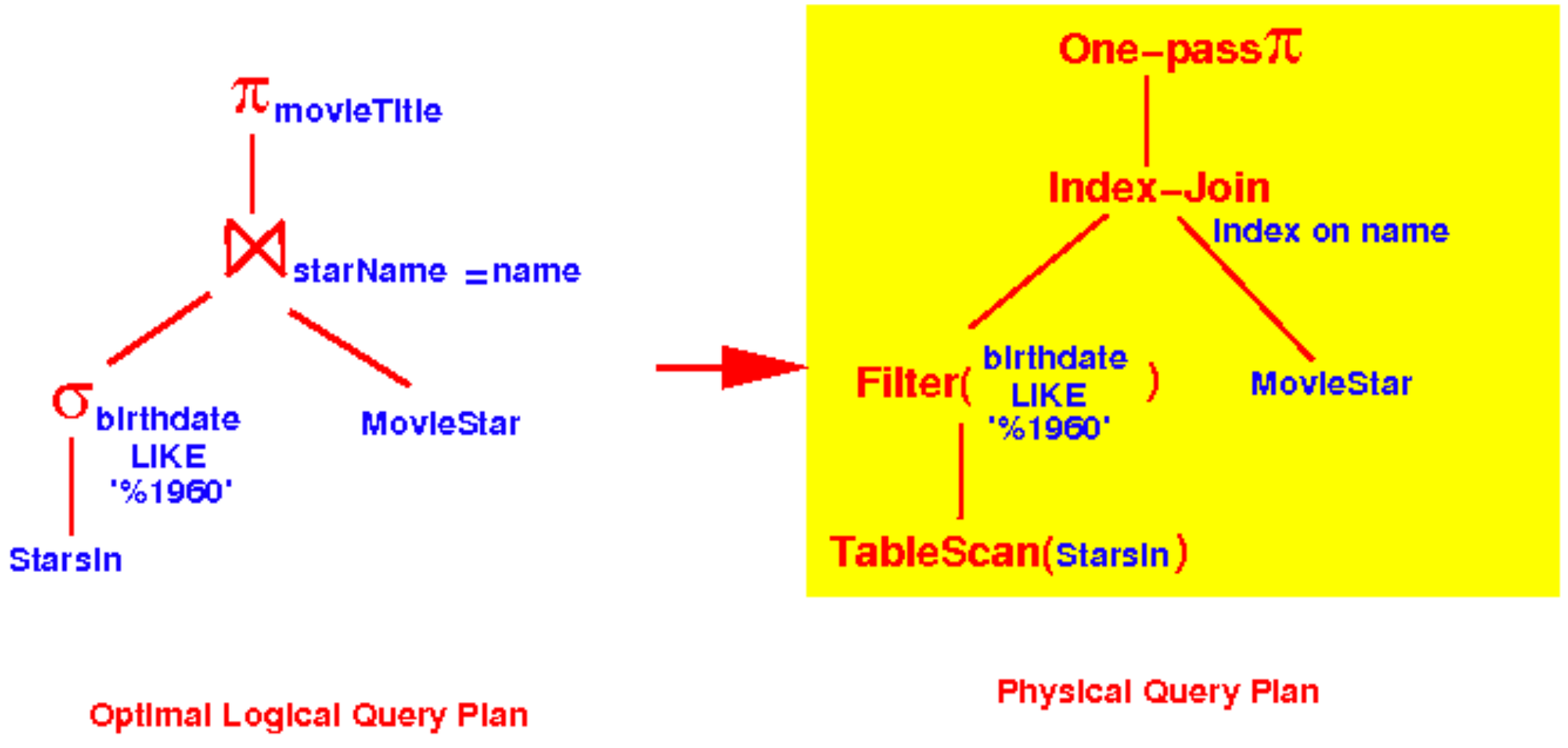
Step 3: Transform initial plan into optimal query plan using some measure of cost to determine which plan is better



http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/intro.html

Example: SQL Query Step 4

Step 4: Select physical query operator for each relational algebra operator in the optimal query plan



http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/intro.html

Recap: Catalog Information

Database maintains statistics about each relation

- Size of file: number of tuples [n_r], number of blocks [b_r], tuple size [s_r], number of tuples or records per block [f_r], etc.
- Information about indexes and indexing attributes
  - Attribute values - number of distinct values [V(att, r)]
  - Selection cardinality - expected size of selection given value [SC(att, r)]
  - …

Recap: Cost-based Optimization

SELECT algorithms

- Linear search
- Binary search
- Index search

Different costs depending on the file organization and indexes

Sorting

- One of the primary algorithms used for query processing
  - ORDER BY
  - DISTINCT
  - JOIN
- Relations that fit in memory — use techniques like quicksort, merge sort, bubble sort
- Relations that don’t fit in memory — external sort-merge

External Sort-Merge Algorithm

- Problem: Sort r records, stored in b file blocks with a total memory space of M blocks

- Create sorted runs with i = 0

  - Read M blocks of relation into memory

  - Sort the in-memory blocks

  - Write sorted data to run Ri, increment i

External Sort-Merge Algorithm (2)

- Merge the sorted runs: merge subfiles until 1 remains
  - Select the first record in sort order from each of the buffers
  - Write the record to the output
  - Delete the record from the buffer page, and read the next block if empty
- Total cost: $b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$

Example: External Merge Sort

Sort fragments of file in memory using internal sort — where each run size is the size of the block

For this example, use block size = 3 tuples

g	24
a	19
d	31
c	33
b	14
e	16
r	16
d	21
m	3
p	2
d	7
a	14

initial relation

run 1

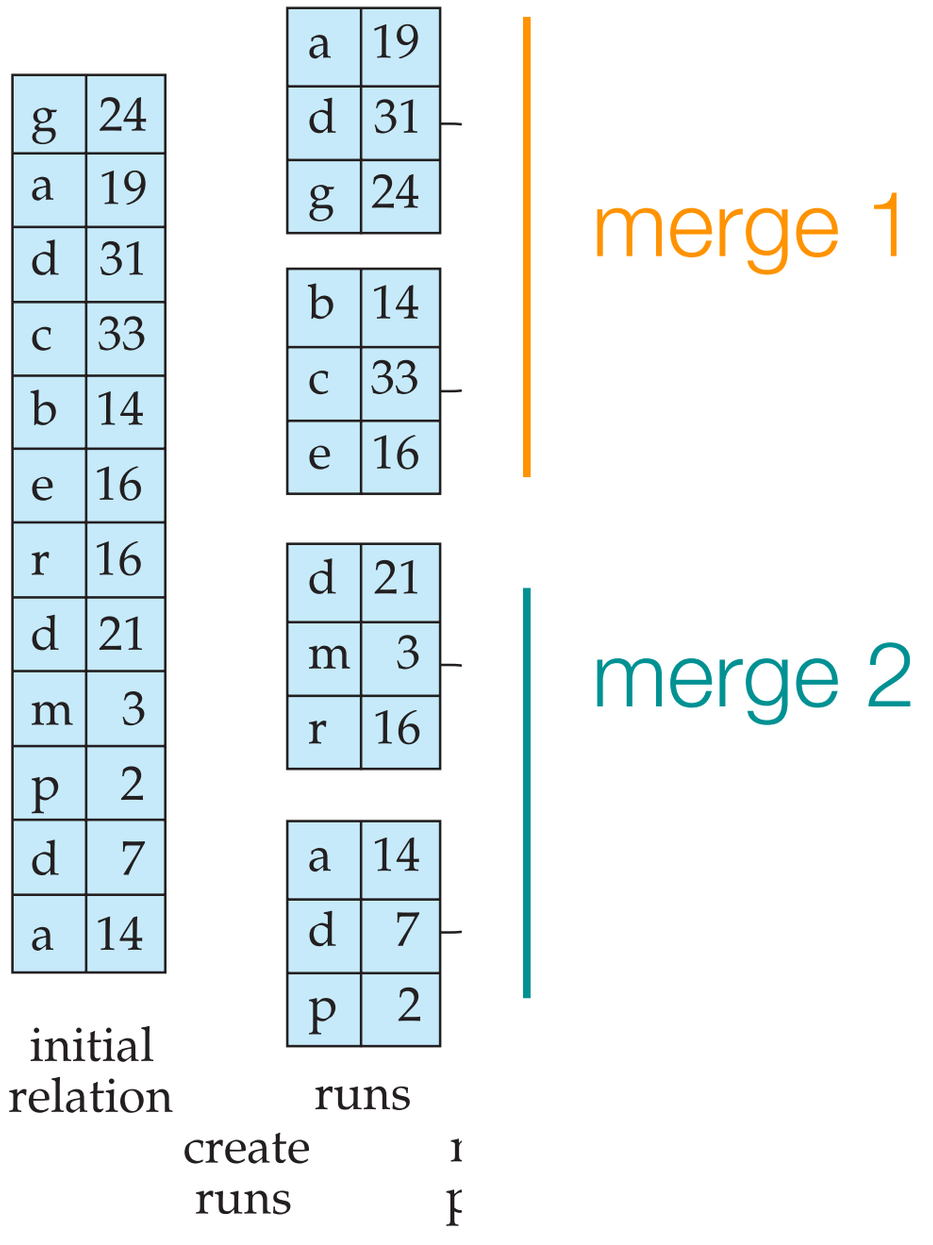
run 2

run 3

run 4

Figure 12.4 from Database System Concepts book

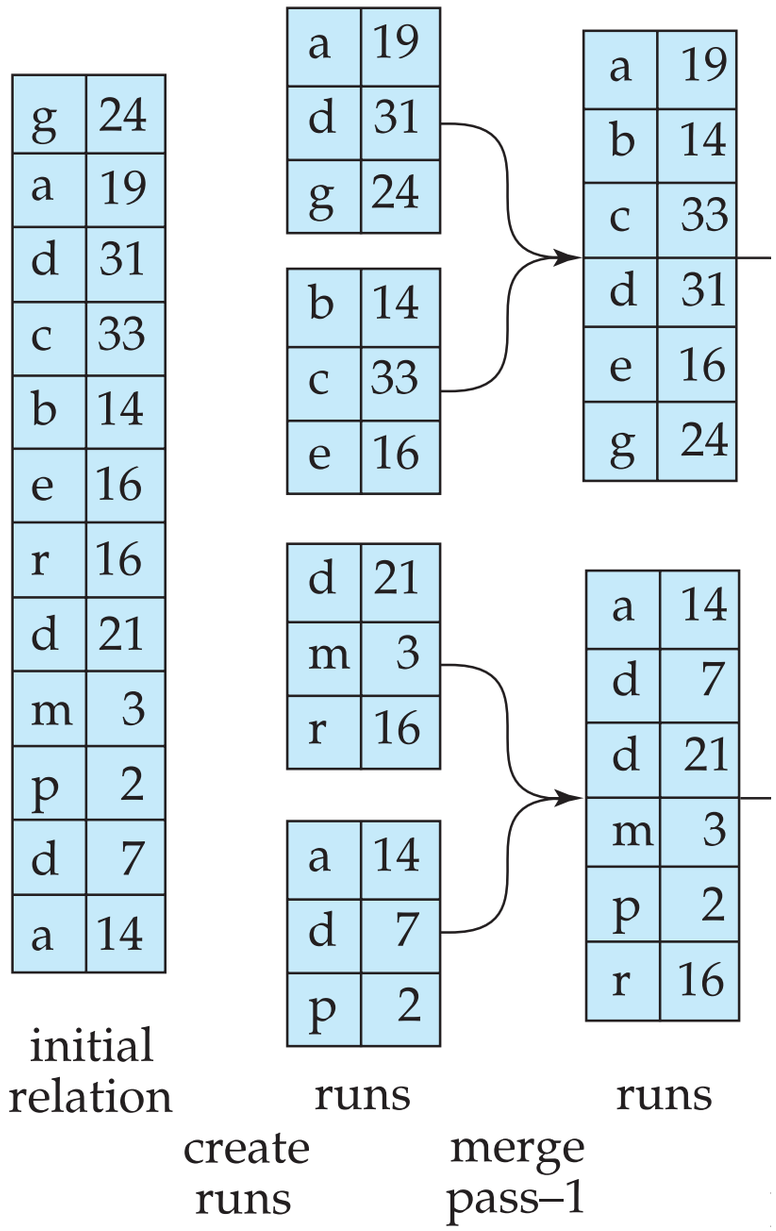
Example: External Merge Sort (2)



Once each run is sorted, we will merge two runs together at a time

Figure 12.4 from Database System Concepts book

Example: External Merge Sort (3)



Another layer of sorted runs, so again merge 2 runs at a time…

Figure 12.4 from Database System Concepts book

Example: External Merge Sort (4)

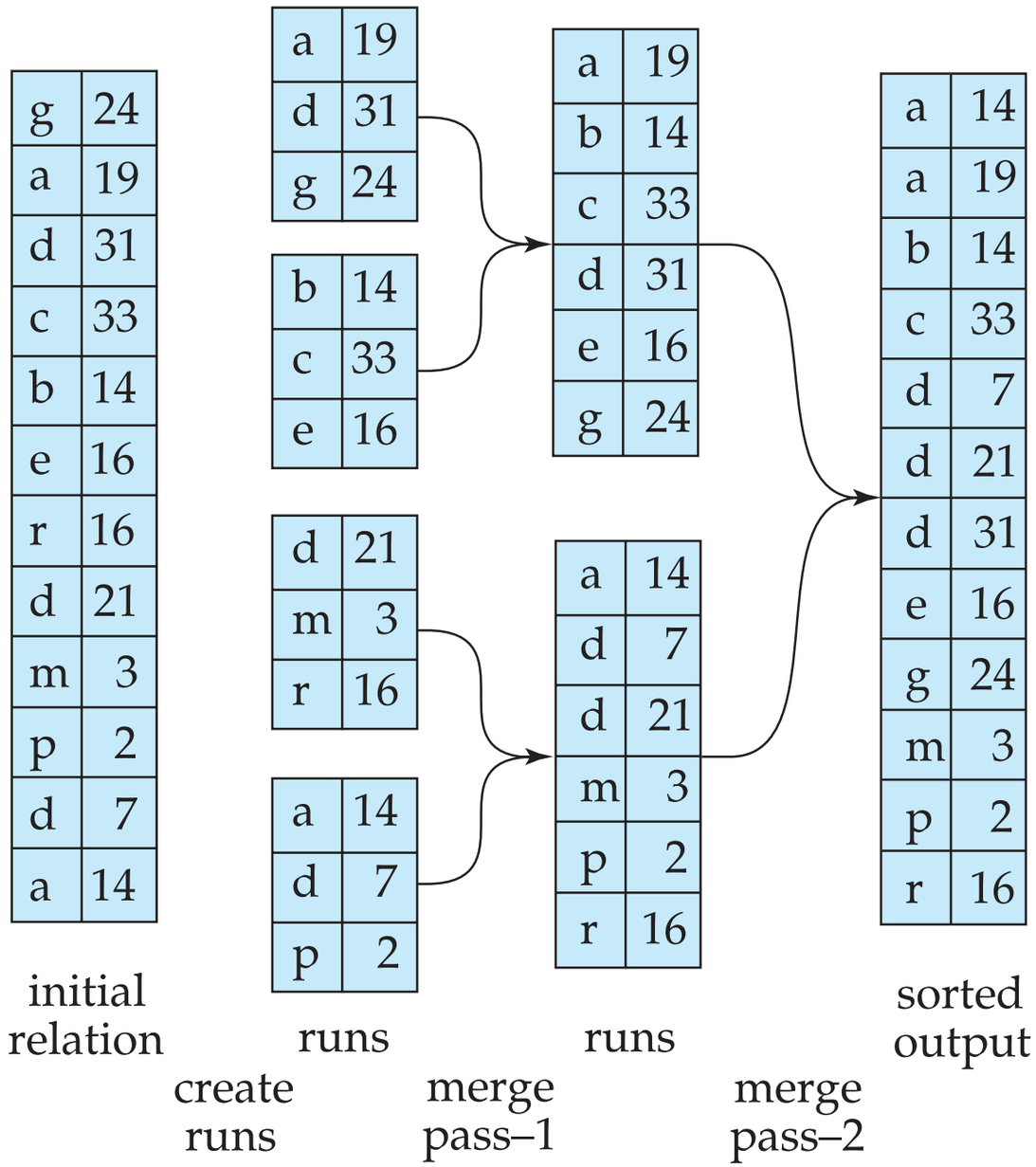


Figure 12.4 from Database System Concepts book

JOIN

- One of the most time-consuming operations
- EQUIJOIN & NATURAL JOIN varieties are most prominent — focus on algorithms for these
  - Two way join: join on two files
  - Multi-way joins: joins involving more than two files

JOIN Performance

Factors that affect performance

- Tuples of relation stored physically together
- Relations sorted by join attribute
- Existence of indexes

JOIN Algorithms

- Several different algorithms to implement joins
  - Nested loop join
  - Nested-block join
  - Indexed nested loop join
  - Sort-merge join
  - Hash-join
- Choice is based on cost estimate

Example: Bank Schema

- Join depositor and customer tables
- Catalog information for both relations:
  - $n_{customer}$ = 10000
  - $f_{customer}$ = 25 => $b_{customer}$ = 10000/25 = 400
  - $n_{depositor}$ = 5000
  - $f_{depositor}$ = 50 => $b_{depositor}$ = 5000/50 = 100
  - V(cname, depositor) = 2500 (each customer on average has 2 accounts)
- Cname in depositor is a foreign key of customer

Cardinality of Join Queries

- Cartesian product or two relations R x S contains n_R * n_S tuples with each tuple occupying s_R + s_S bytes

- If $R \cap S = \emptyset$, then $R \bowtie S$ is the same as $R \times S$

- If $R \cap S$ is a key in R, then a tuple of s will join with one tuple from R => the number of tuples in the join will be no greater than the number of tuples in S

- If $R \cap S$ is a foreign key in S referencing R, then the number of tuples is exactly the same number as S

Cardinality of Join Queries (2)

- If $R \cap S = \{A\}$ and A is not a key of R or S there are two estimates that can be used
  - Assume every tuple in R produces tuples in the join, number of tuples estimated:

$$\frac{n_R * n_s}{V(A, s)}$$

- Assume every tuple in S produces tuples in the join, number of tuples estimated:

$$\frac{n_R * n_s}{V(A, r)}$$

- Lower of two estimates is probably more accurate

Example: Cardinality of Join

- Estimate the size of $\mathrm{Depositor} \bowtie \mathrm{Customer}$
- Assuming no foreign key:
  - V(cname, depositor) = 2500 =>
    5000 * 10000 / 2500 = 20,000
  - V(cname, customer) = 10000 =>
    5000 * 10000 / 10000 = 5000
- Since cname in depositor is foreign key of customer, the size is exactly $n_{depositor} = 5000$

Nested Loop Join

- Default (brute force) algorithm
- Requires no indices and can be used with any join condition
- Algorithm:
  for each tuple t_r in r do
    for each tuple t_s in s do
      test pair (t_r, t_s) to see if condition satisfied
      if satisfied, output (t_r, t_s) pair
- R is the outer relation and S is the inner relation

Nested Loop Join Cost

- Expensive as it examines every pair of tuples in the two relations

  - If smaller relation fits entirely in main memory, use that relation as inner relation

- Worst case: only enough memory to hold one block of each relation, estimated cost is $n_r * b_s + b_r$

- Best case: smaller relation fits in memory, estimated cost is $b_r + b_s$ disk access

Example: Nested Loop Join

- Worst case memory scenario:
  - Depositor as outer relation: 5000 * 400 + 1000 = 2,000,100 I/Os
  - Customer as outer relation: 10000 * 100 + 400 = 1,000,400 I/Os
- Best case memory scenario (depositor fits in memory)
  - 100 + 400 = 500 I/Os

Nested-Block Join

- Instead of individual tuple basis, join one block at a time together

- Algorithm:
  for each block in r do
      for each block in s do
          use nested loop join algorithm on blocks
          to output matching pairs

- Worst case: each block in the inner relation s is only read once for each block in the outer relation, so estimated cost is $b_r * b_s + b_r$

- Best case: same as nested loop with cost $b_r + b_s$

Nested-Block vs Nested Loop Join

Assume worst memory case

- Nested loop join with depositor as inner relation: 10000 * 100 + 400 = 1,000,400 I/Os
- Nested-block join with depositor as inner relation: 400 * 100 + 400 = 40400 I/Os

What if a disk speed is 360K I/Os per hour?

- Nested loop join ~= 2.78 hours
- Nested-block join ~= 0.11 hours

A very small change can make a huge difference in speed!

Indexed Nested-Loop Join

- Index is available on inner loop's join attribute — use index to compute the join

- Algorithm:
  for each tuple t_r in r do
    retrieve tuples from s using index search

- Worst case: buffer only has space for one page of r and one page of index, estimated cost is $b_r + n_r * c$ (c is cost of single selection on s using join condition)

- If indices available on both relations, use one with fewer tuples as outer relation

Example: Index Nested Loop Join

- Assume customer has primary B^+-tree index on customer name, which contains 20 entries in each node
- Since customer has 10,000 tuples, height of tree is 4
- Using depositor as outer relation, estimated cost: 100 + 5000 * (4 + 1) = 25,100 disk accesses
- Block nested-loop join cost: 100 * 400 + 100 = 40,100 I/Os
- Cost is lower with index nested loop than block nested-loop join

Sort-Merge Join

- Sort the relations based on the join attributes (if not already sorted)
- Merge similar to the external sort-merge algorithm with the main difference in handling duplicate values in the join attribute — every pair with same value on join attribute must be matched

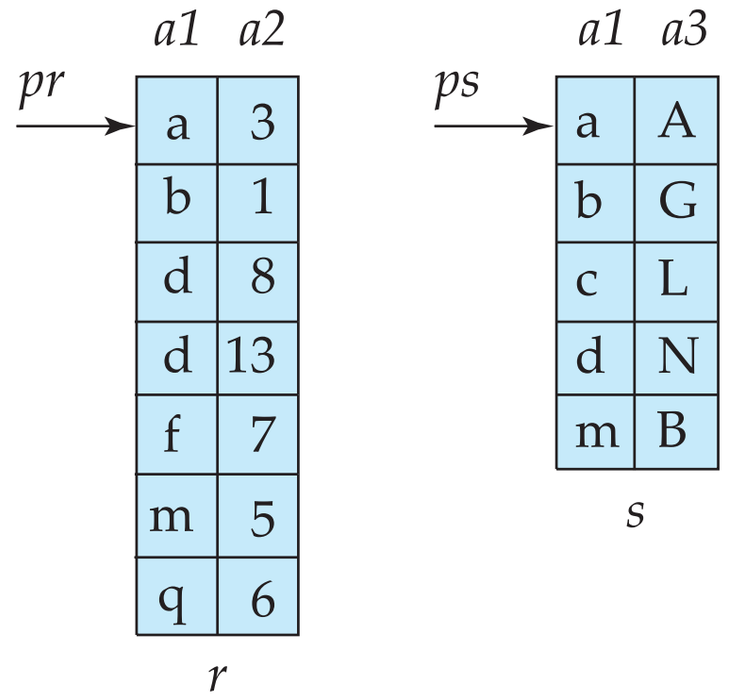


Figure 12.8 from Database System Concepts book

Sort-Merge Join (2)

- Can only be used for equijoins and natural joins

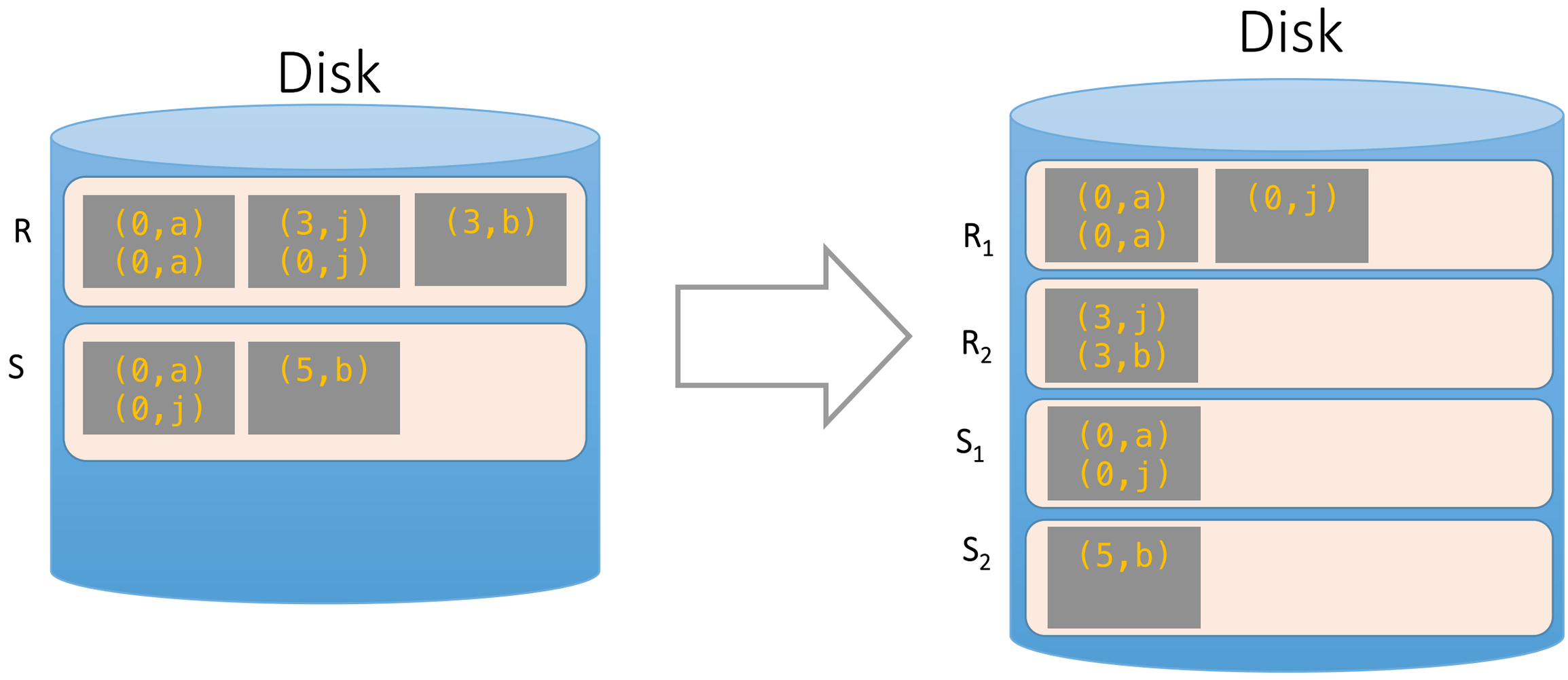
- Each tuple needs to be read only once, and as a result, each block is also read only once
  cost = sorting cost + b_r + b_s

- If one relation is sorted, and other has secondary B+-tree index on join attribute, hybrid merge-joins are possible

Hash-Join

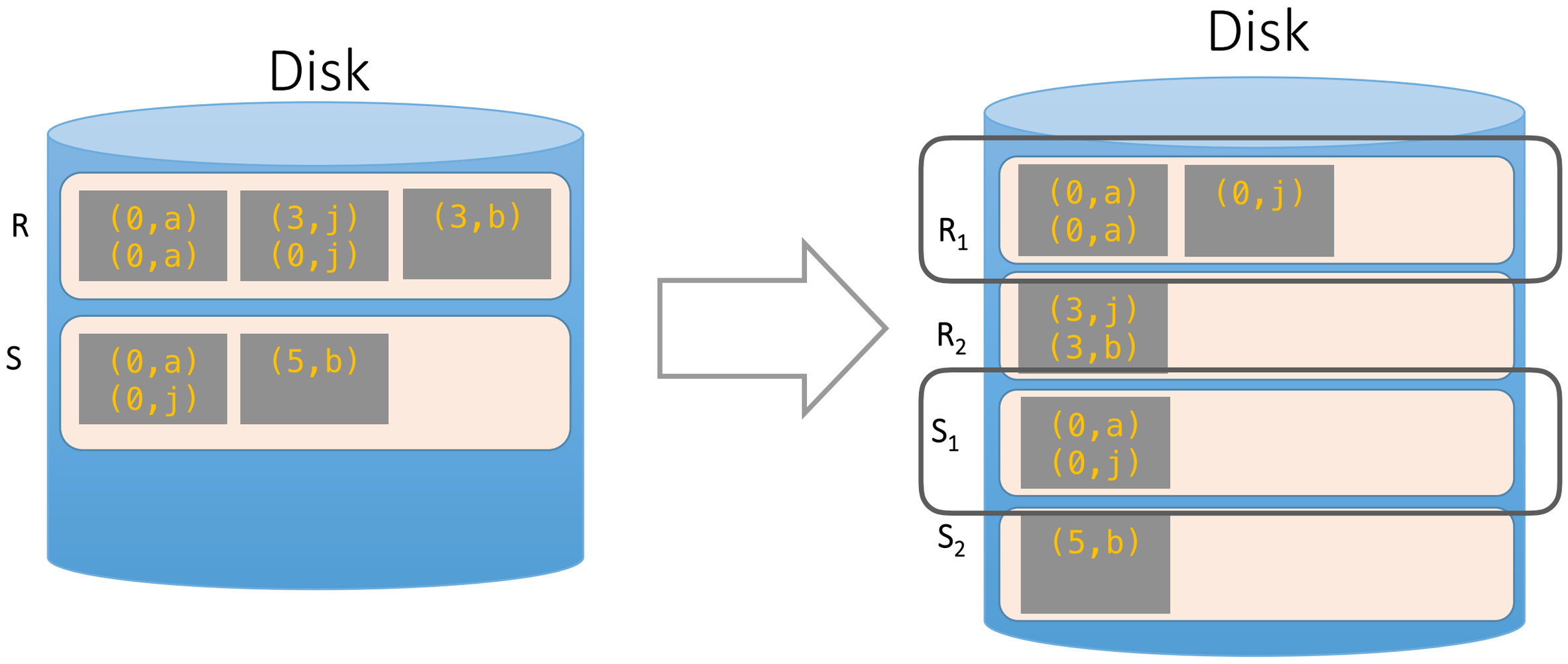
- Applicable for equijoins and natural joins
- A hash function, h, is used to partition tuples of both relations into sets that have same hash value on the join attributes
- Tuples in the corresponding same buckets just need to be compared with one another and not with all the other tuples in the other buckets

Example: Hash-Join



Step 1: Use hash function to partition into B buckets

Example: Hash-Join (2)



Step 2: Join matching buckets

Hash-Join Algorithm

- Partitioning phase
  - 1 block for reading and M-1 blocks for hashed partitions
    - Hash R tuples into k buckets (partitions)
    - Hash S tuples into k buckets (partitions)
- Joining phase (nested block join for each pair of partitions)
  - M-2 blocks for R partition, 1 block for S partition

Hash-Join Algorithm

- Hash function h and the number of buckets are chosen such that each bucket should fit in memory

- Recursive partitioning required if number of buckets is greater than number of pages M of memory

- Hash-table overflow occurs if each bucket does not fit in memory

Hash-Join Cost

- If recursive partitioning is not required:
  - Partitioning phase: $2b_R + 2b_S$
  - Joining phase: $b_R + b_S$
  - Total: $3b_R + 3b_S$
- If recursive partitioning is required:
  - Number of passes required to partition: $\lceil \log_{M-1}(b_S) - 1 \rceil$
  - Total cost: $2(b_R + b_S)\lceil \log_{M-1}(b_S) - 1 \rceil + b_R + b_S$

Example: Hash-Join

- Assume memory size is 20 blocks
- What is cost of joining customer and depositor?
- Since depositor has less total blocks, we will use it to partition into 5 buckets, each of size 20 blocks
- Customer is also partitioned into 5 buckets, each of size 80 blocks
- Total cost: 3(100 + 400) = 1500 block transfers

Hybrid Hash-Join

- Useful when memory sizes are relatively large and the smallest relation is bigger than memory

- Idea: Keep first partition in memory to avoid disk I/O for reading and writing the first block

- Assume we have a slightly larger memory size of 25 blocks (compared to previous example) - keep the first partition of depositor in memory (20 blocks)

- Cost: 3(80 + 320) + 20 + 80 = 1300 block transfers

Hash Join vs Sorted Join

- Sorted join advantages
  - Good if input is already sorted, or need output to be sorted
  - Not sensitive to data skew or bad hash functions
- Hash join advantages
  - Can be cheaper due to hybrid hashing
  - Dependent on size of smaller relation — good for different relation sizes
  - Good if input already hashed or need output hashed

Complex Join

- What about joins with conjunctive (AND) conditions?
  - Compute the result of one of the simpler joins
  - Final result consists of tuples in intermediate results that satisfy remaining conditions
  - Test these conditions as tuples are generated
- What about joins with disjunctive (OR) conditions?
  - Compute as the union of the records in individual joins

Example: Complex Join

What if we did a join on loan, depositor, and customer?

- Strategy 1: Compute depositor joins customer and then use that to compute the join with loans
- Strategy 2: Compute loan joins depositor first then use that to join with customer

Example: Complex Join (2)

What if we did a join on loan, depositor, and customer?

- Strategy 3: Perform pair of joins at once, build an index on loan for lID and on customer for cname

- For each tuple t in depositor, lookup corresponding tuples in customer and corresponding tuples in loan

- Each tuple of depositor is examined exactly once

PROJECT Algorithms

- Extract all tuples from R with only attributes in attribute list of projection operator & remove tuples

- By default, SQL does not remove duplicates (unless DISTINCT keyword is included)

- Duplicate elimination
  - Sorting
  - Hashing (duplicates in same bucket)

Aggregation Algorithms

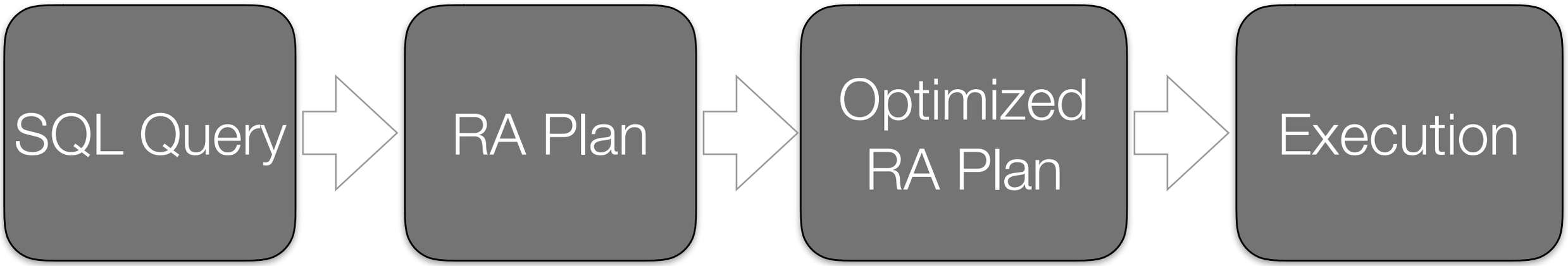
Similar to duplicate elimination

- Sort or hash to group same tuples together
- Apply aggregate functions to each group

Set Operation Algorithms

- CARTESIAN PRODUCT
  - Nested loop - expensive and should avoid if possible
- UNION, INTERSECTION, SET DIFFERENCE
  - Sort-merge
  - Hashing

Query Processing Recap



Declarative user query

Translate to RA expression

Find logically equivalent but more efficient RA expression

Select physical algorithm with lowest IO cost to execute the plan

DBMS's Query Execution Plan

- Most commercial RDBMS can produce the query optimizer's execution plan to try to understand the decision made by the optimizer

- Common syntax is EXPLAIN <SQL query> (used by MySQL)

- Good DBAs (database administrators) understand query optimizers VERY WELL!

Why Should I Care?

- If query runs slower than expected, check the plan — DBMS may not be executing a plan you had in mind
  - Selections involving null values
  - Selections involving arithmetic or string operations
  - Complex subqueries
  - Selections involving OR conditions
- Determine if you should build another index, or if index needs to be re-clustered or if statistics are too old

Query Tuning Guidelines

- Minimize the use of DISTINCT — don't need if duplicates are acceptable or if answer already has a key
- Minimize use of GROUP BY and HAVING
- Consider DBMS use of index when using math
  - E.age = 2 * D.age might only match index on E.age
- Consider using temporary tables to avoid "double-dipping" into a large table
- Avoid negative searches (can't utilize indexes)

Query Optimization: Recap

- External sort-merge
- JOIN algorithms
  - Nested loop join
  - Nested-block join
  - Indexed nested-loop join
  - Sort-merge join
  - Hash-join
- Other operation algorithms (PROJECT, SET, Aggregate)