Query Processing & Optimization

CS 377: Database Systems

Recap: File Organization & Indexing

- Physical level support for data retrieval
  - File organization: ordered or sequential file to find items using binary search
  - Index: data structures to help with some query evaluation (selection & range queries)
- Indexes may not always be useful even for selection queries
- What about join queries and other queries not supported by indices?

Query Processing Introduction

- Some database operations are expensive
- Performance can be improved by being “smart”
  - Clever implementation techniques for operators
  - Exploiting “equivalences” of relational operators
  - Using statistics and cost models to choose better plans

Basic Steps in Query Processing

- Parse and translate: convert to RA query
- Optimize RA query based on the different possible plans
- Evaluate the execution plan to obtain the query results

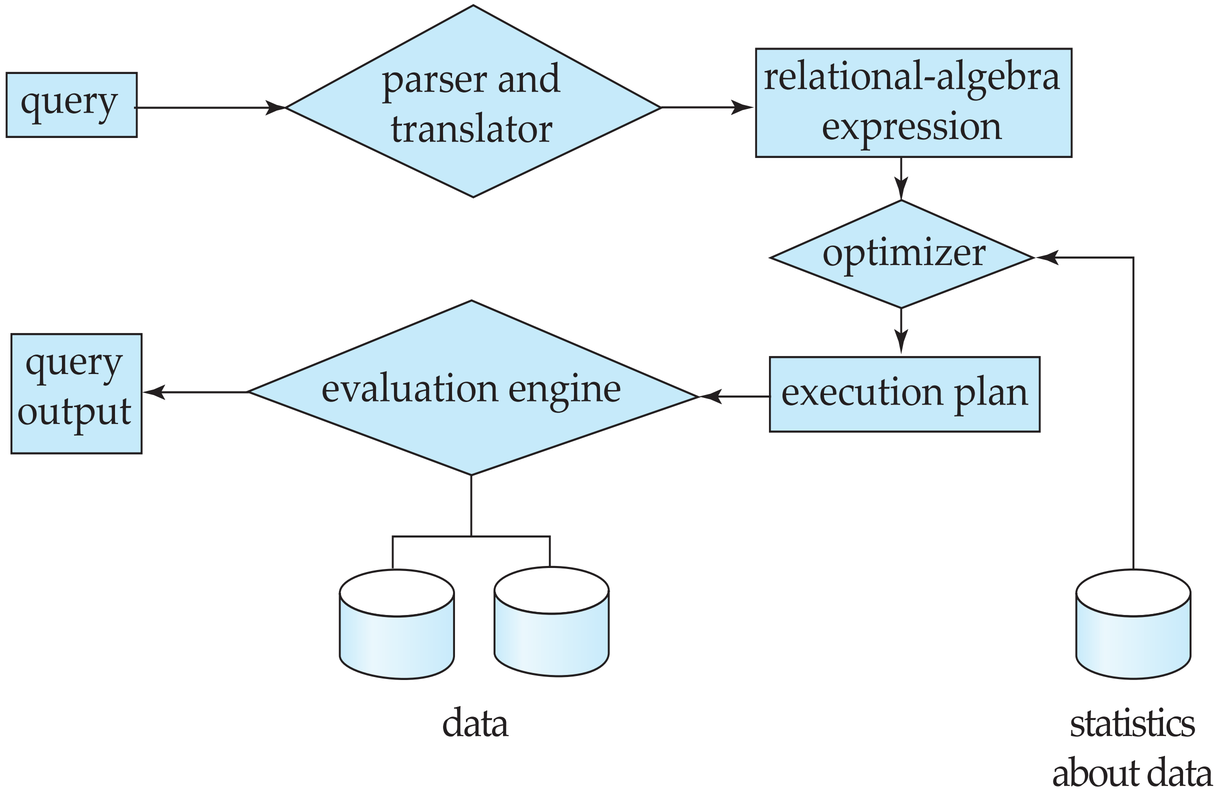
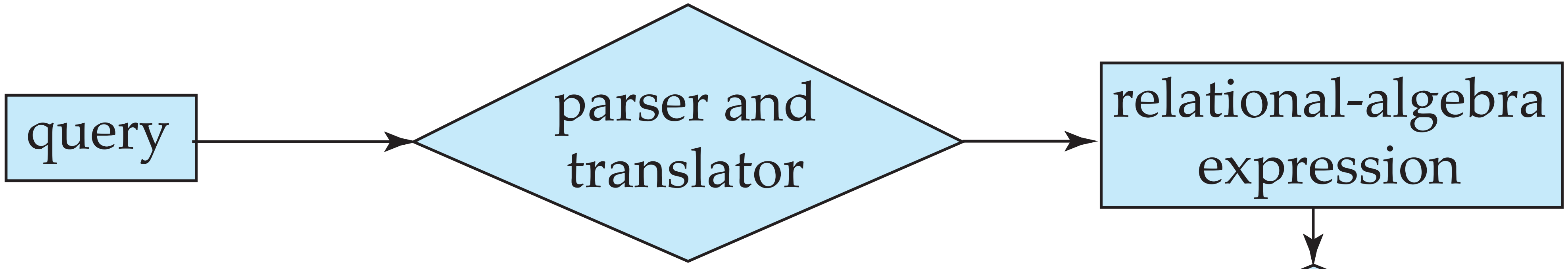


Figure 12.1 from Database System Concepts book

Query Processing Overview



WHY?

SQL

A **declarative** expression of the query result

Codd's theorem

RA

Operational description of a computation

Say what you want, not how to get it!

Systems optimize and execute RA query plan!

Example: SQL Query

Find movies with stars born in 1960

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name
AND birthdate LIKE ‘%1960’;
```

Example: SQL —> RA

Find movies with stars born in 1960

SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name
AND birthdate LIKE ‘%1960’;

$$\mathrm{H1} = (\mathrm{StarsIn} \times \mathrm{MovieStar})$$

$$\mathrm{H2} = \sigma_{\mathrm{starname\ =\ name\ AND\ birthdate\ like\ `\%1960'}}(\mathrm{H1})$$

$$\mathrm{Ans} = \pi_{\mathrm{movie\ title}}(\mathrm{H2})$$

Is this a good query plan?

Example: SQL —> RA Take II

Find movies with stars born in 1960

SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name
AND birthdate LIKE ‘%1960’;

Is this a better query plan?

$$\text{H1} = \sigma_{\text{birthdate like ‘\%1960’}}(\text{MovieStar})$$

$$\text{H2} = \sigma_{\text{starname} = \text{name}}(\text{H1} \times \text{StarsIn})$$

$$\text{Ans} = \pi_{\text{movie title}}(\text{H2})$$

Example: SQL —> RA Take III

Find movies with stars born in 1960

SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name
AND birthdate LIKE ‘%1960’;

Is this even better query plan?

$$H1 = \pi_{name}(\sigma_{\text{birthdate like ‘\%1960’}}(\text{MovieStar}))$$

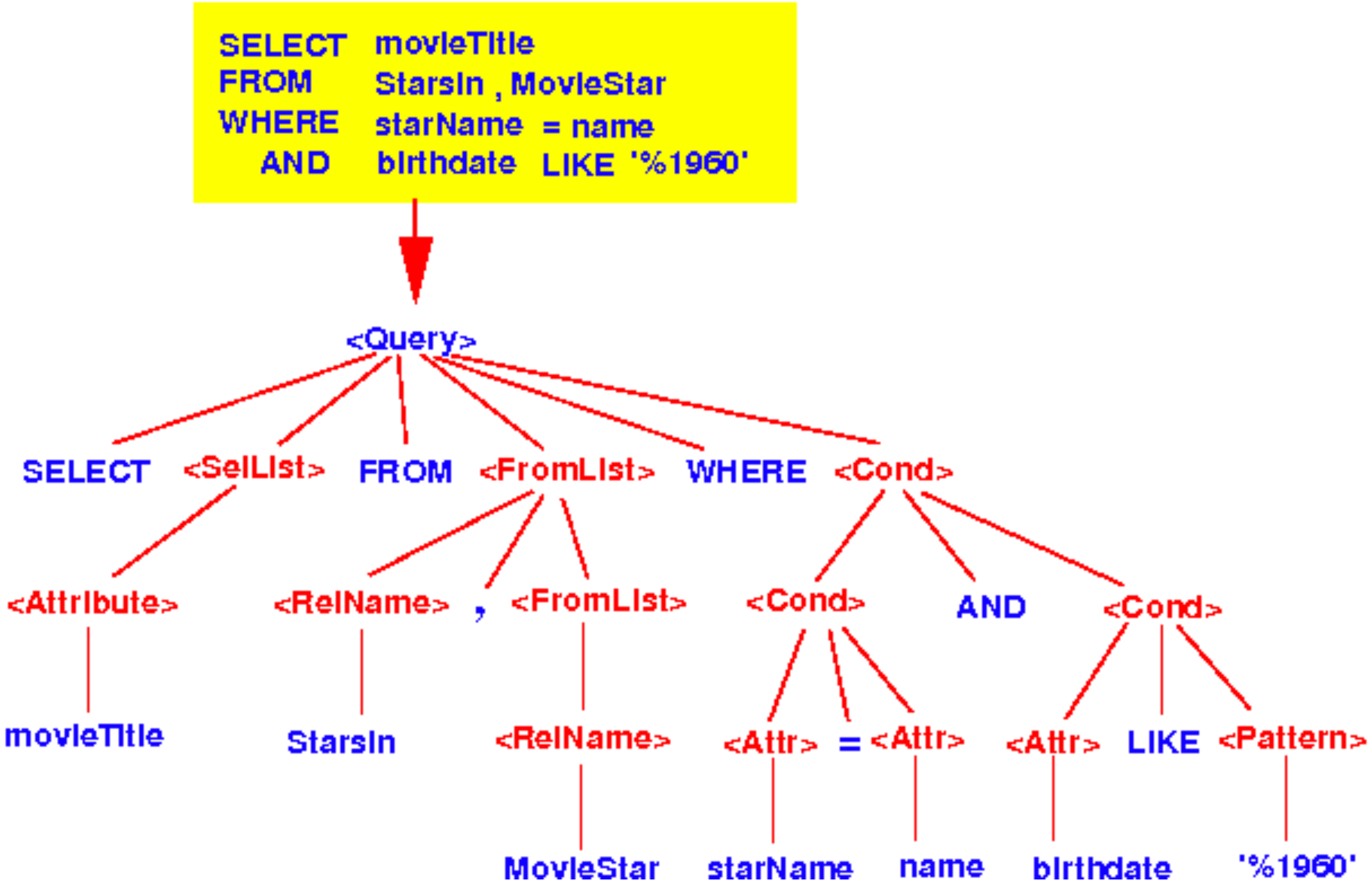
$$H2 = \sigma_{\text{starname = name}}(H1 \times \pi_{\text{starname, movieTitle}}(\text{StarsIn}))$$

$$\text{Ans} = \pi_{\text{movieTitle}}(H2)$$

SQL Optimization

- Step 1: Convert SQL query into a parse tree
- Step 2: Convert parse tree into initial logical query plan using RA expression
- Step 3: Transform initial plan into optimal query plan using some measure of cost to determine which plan is better
- Step 4: Select physical query operator for each relational algebra operator in the optimal query plan

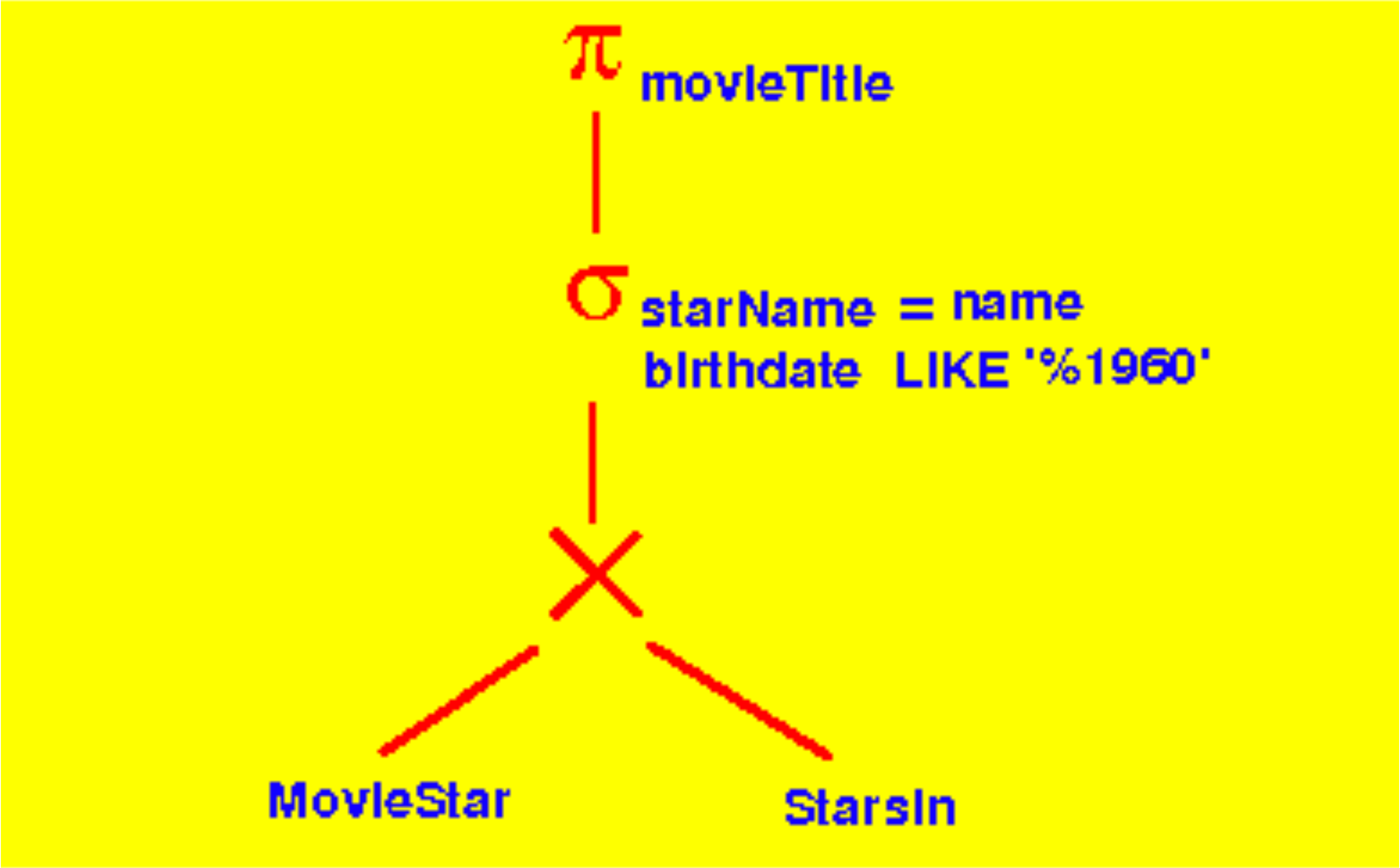
Example: SQL Query Step 1



```
SELECT  movieTitle
FROM    StarsIn , MovieStar
WHERE   starName  = name
  AND   birthdate  LIKE '%1960'
```

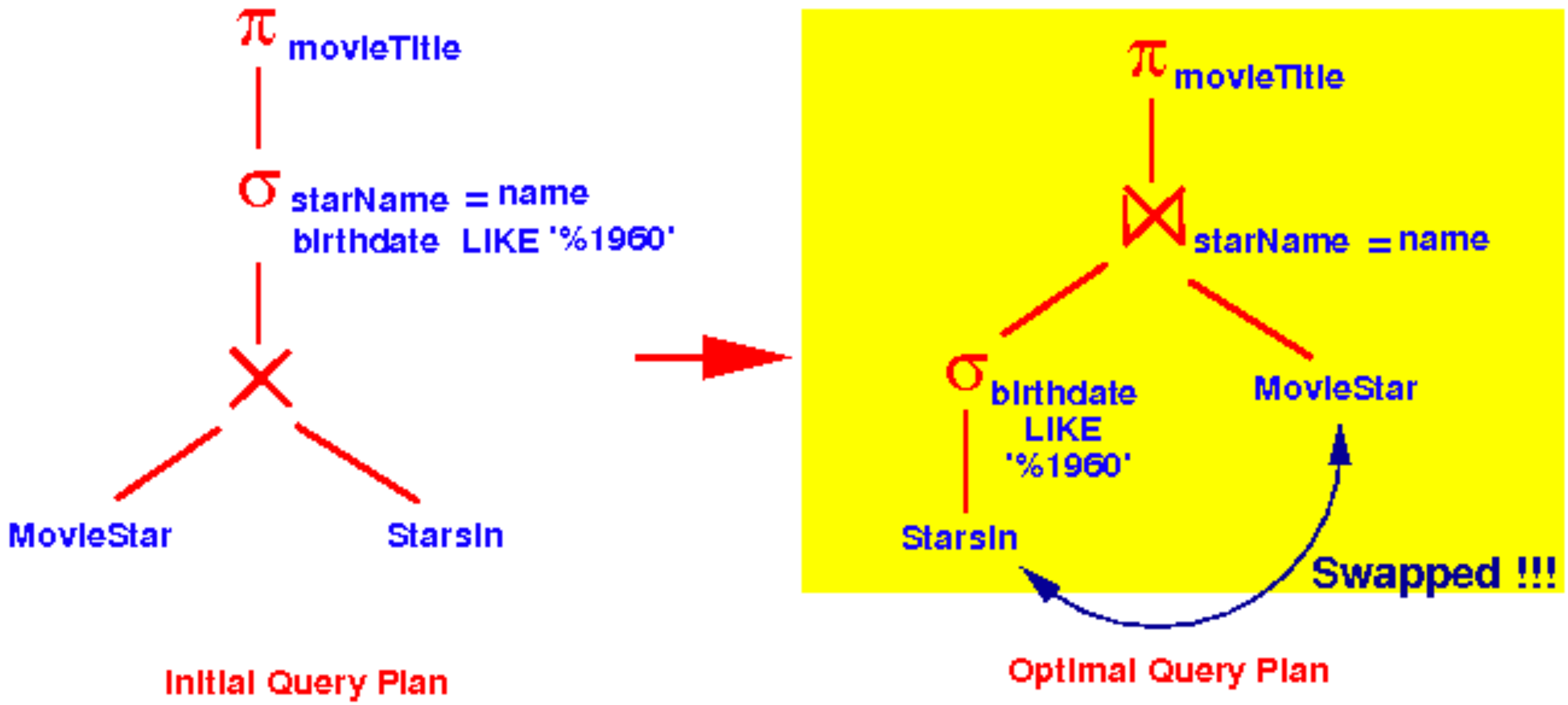
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/intro.html

Example: SQL Query Step 2



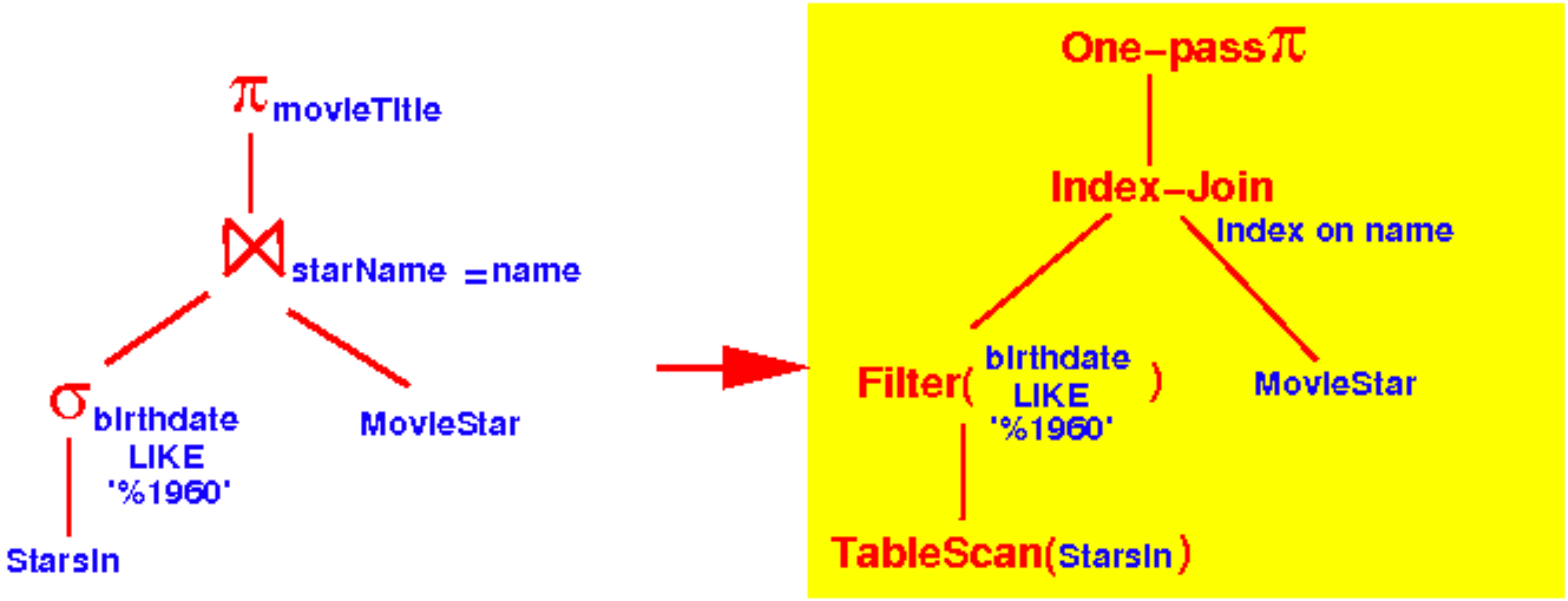
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/intro.html

Example: SQL Query Step 3



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

Example: SQL Query Step 4

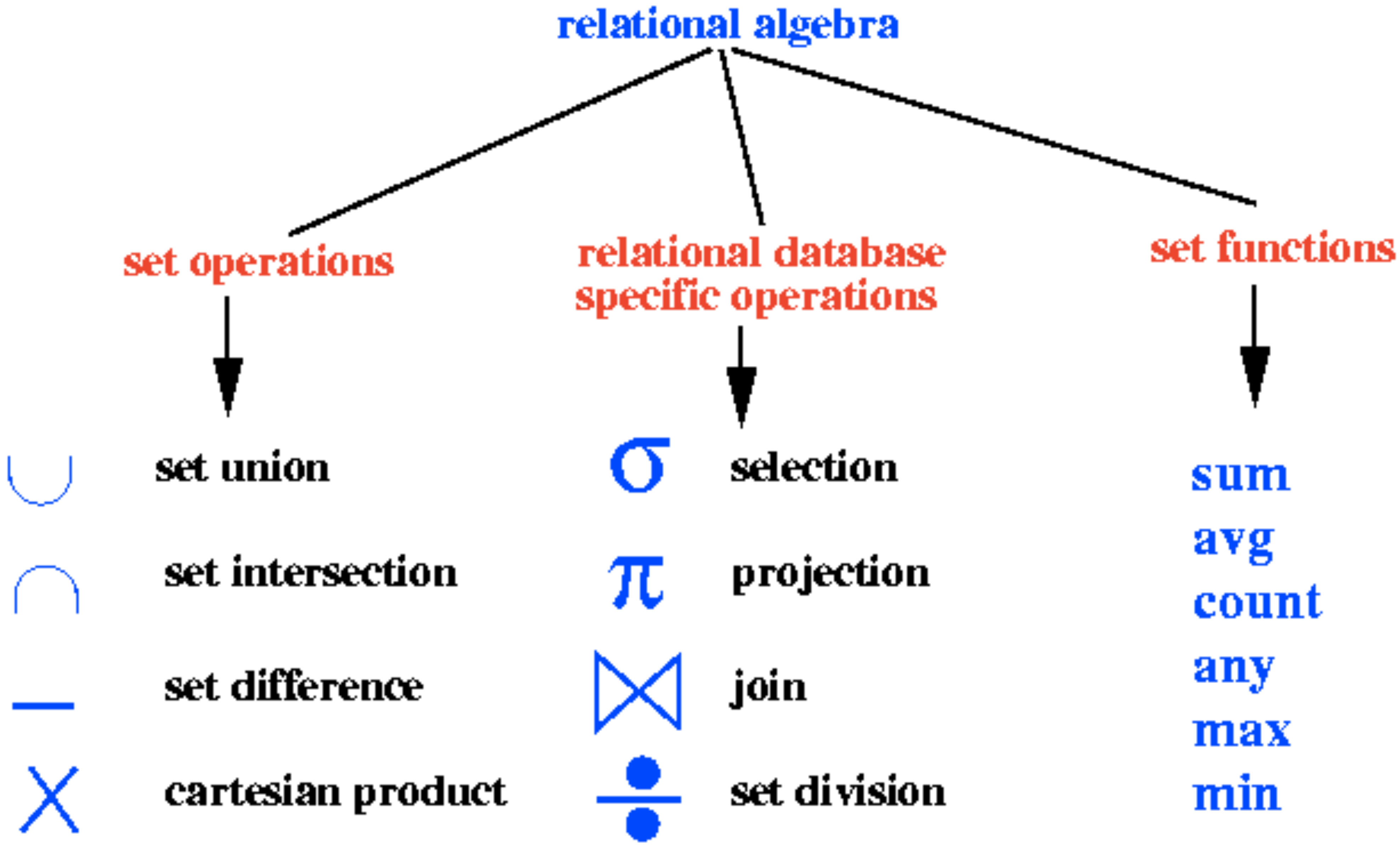


Optimal Logical Query Plan

Physical Query Plan

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

Recap: Relational Algebra



relational algebra

set operations

- ∪ set union
- ∩ set intersection
- − set difference
- × cartesian product

relational database specific operations

- σ selection
- π projection
- ⋈ join
- ÷ set division

set functions

- sum
- avg
- count
- any
- max
- min

Recap: SQL Query to RA

- How do you represent queries in RA?

- Database: Students(sid, sname, gpa)
  People(ssn, pname, address)

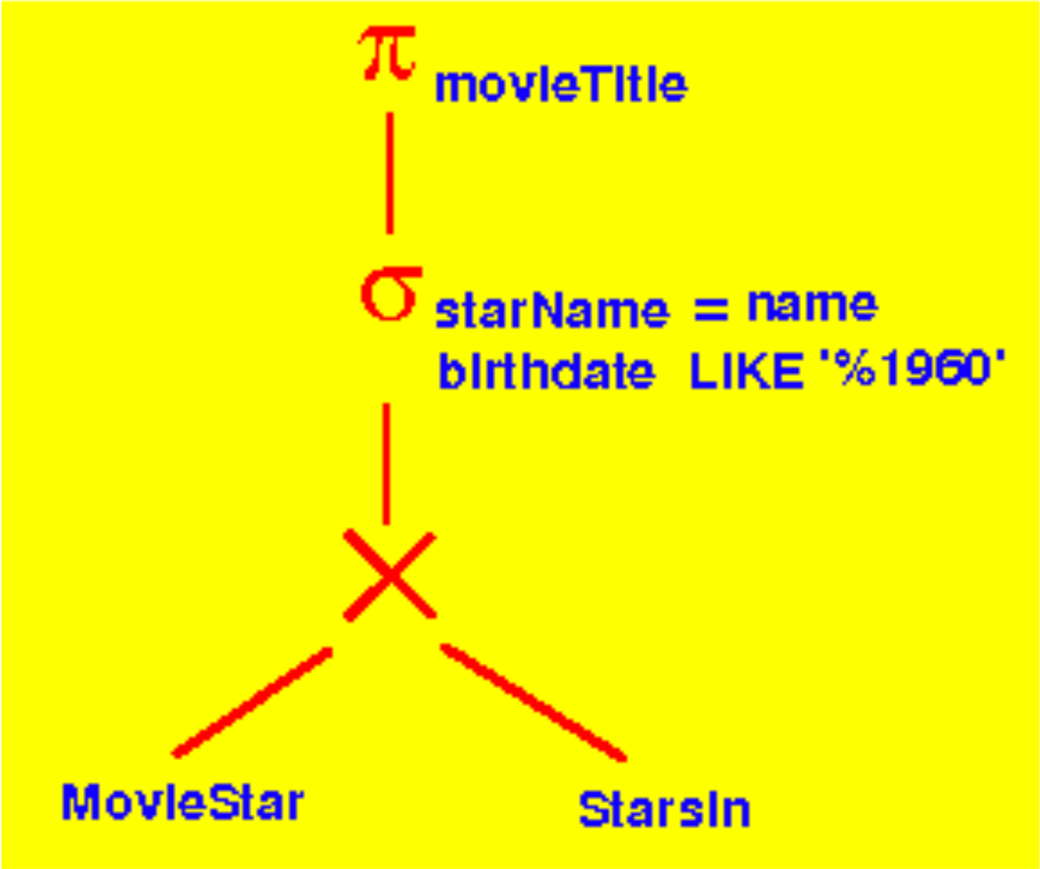
- SQL query:
  SELECT DISTINCT gpa, address
  FROM Students, People
  WHERE gpa > 3.5 AND sname = pname;

- RA query:

$$\pi_{\text{gpa,address}}(\sigma_{\text{gpa}>3.5}(\text{Students} \bowtie_{\text{sname=name}} \text{People}))$$

Query Tree (Plan)

- A tree data structure that corresponds to a relational algebra expression
  - Leaf nodes = input relations
  - Internal nodes = RA operations
- Execution of query tree
  - Start at the leaf nodes
  - Execute internal node whenever its operands are available and replace node by result



Query Optimization Heuristics

- Apply heuristic rules on standard initial query tree to find optimized equivalent query tree
- Main heuristic: Favor operations that reduce the size of intermediate results first
  - Apply SELECT and PROJECT operations before join or other set operations
  - Apply more selective SELECT and join first
- General transformation rules for relational algebra operators

RA Transformation Rules

- Selection cascade: conjunctive selection condition can be broken into sequence of individual operations

$$\sigma_{\text{c1 AND c2 AND} \cdots \text{AND cn}}(R) = \sigma_{\text{c1}}(\sigma_{\text{c2}}(\cdots(\sigma_{\text{cn}}(R))\cdots))$$

- Commutativity of selection

$$\sigma_{\text{c1}}(\sigma_{\text{c2}}(R)) = \sigma_{\text{c2}}(\sigma_{\text{c1}}(R))$$

- Cascade of projection: ignore all but the last one

$$\pi_A(\pi_{A,B}(R)) = \pi_A(R)$$

- Commuting selection and projection: if the selection condition c involves only attributes in the projection list commute the two

$$\pi_{\text{A, B}}(\sigma_{\text{c}}(R)) = \sigma_{\text{c}}(\pi_{\text{A, B}}(R))$$

RA Transformation Rules

- Commutativity of joins, cartesian product, union, intersection

$$R \ \theta \ S = S \ \theta \ R$$

- Associativity of join, cartesian product, union, intersection

$$(R \ \theta \ S) \ \theta \ T = R \ \theta \ (S \ \theta \ T)$$

- Selection and join: if attributes in the selection condition involves only attributes of one of the relations being joined

$$\sigma_c(R \bowtie S) = \sigma_c(R) \bowtie S$$

-

$$\sigma_c(R \bowtie S) = \sigma_{c1}(R) \bowtie \sigma_{c2}(S)$$

RA Transformation Rules

- Commuting projection with join: if join condition involves only attributes in the projection list, commute the operations

$$\pi_L(R \bowtie_c S) = (\pi_{L1}(R)) \bowtie_c (\pi_{L2}(S))$$

- Commuting selection with intersection, union, or difference

$$\sigma_c(R \; \theta \; S) = (\sigma_c(R)) \; \theta \; (\sigma_c(S))$$

- Several others in the book…

Query Optimization Heuristic Algorithm

- Break up any select operations with conjunctive conditions into cascade of select operations and move select operations as far down query tree as permitted
- Rearrange leaf nodes so leaf nodes with most restrictive select operations are executed first
- Combine cartesian product operation with a subsequent selection operation into join operation

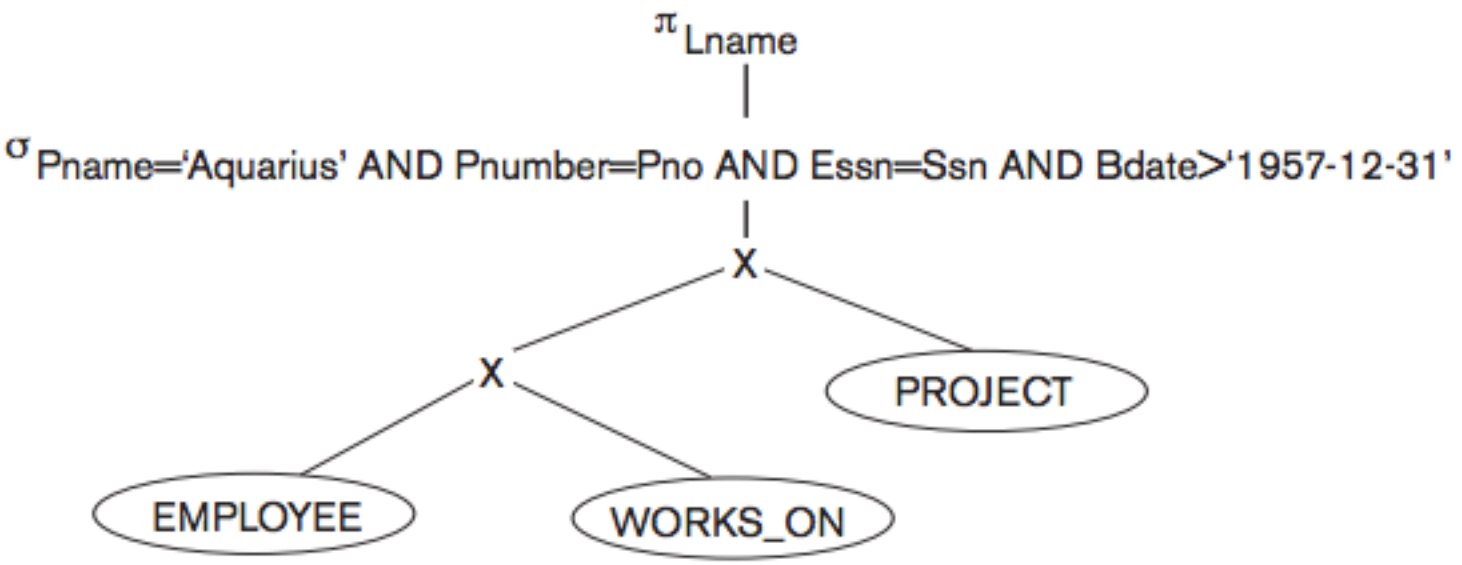
Query Optimization Heuristic Algorithm

- Break down and move lists of projection attributes down the tree as far as possible
- Identify subtrees that represent group of operations that can be executed as a single algorithm

Example: SQL Query Optimization

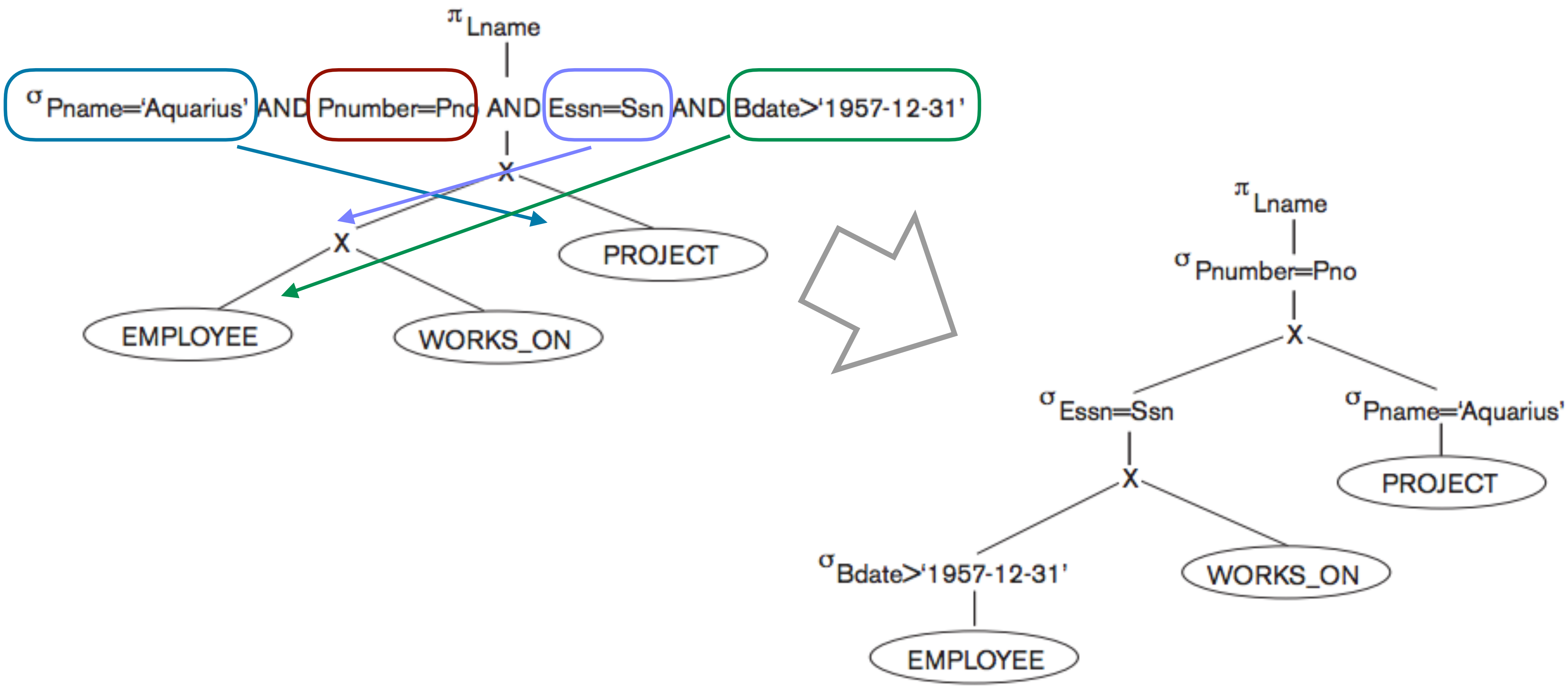
```
SELECT lname
FROM   employee, works_on, project
WHERE  pname = ‘Aquarius’ and pnumber = pno
AND    bdate > ‘1957-12-31’;
```

Initial query tree

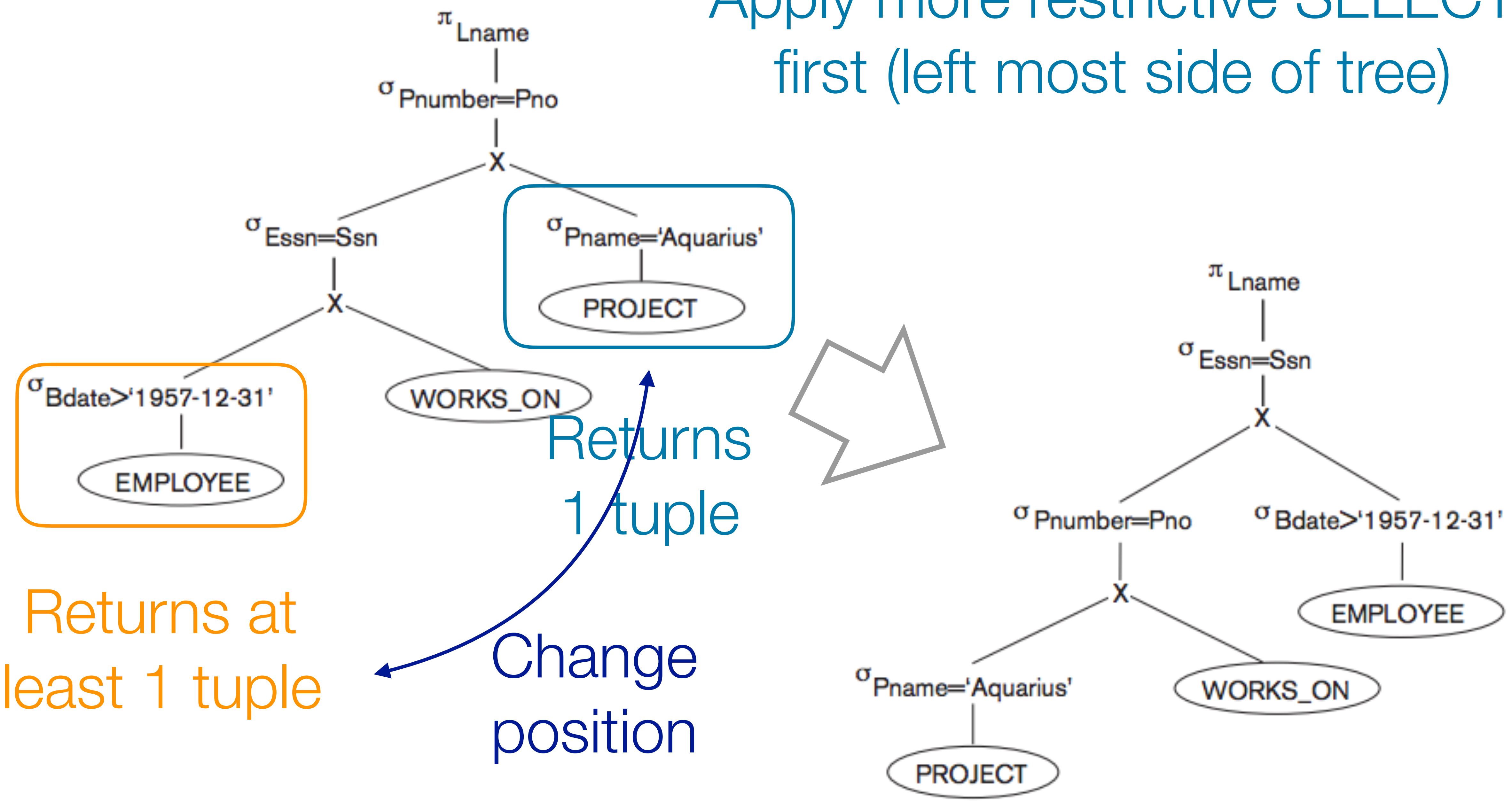


Example: SQL Query Optimization

Break up conjunctive select operations and move them down the tree



Example: SQL Query Optimization



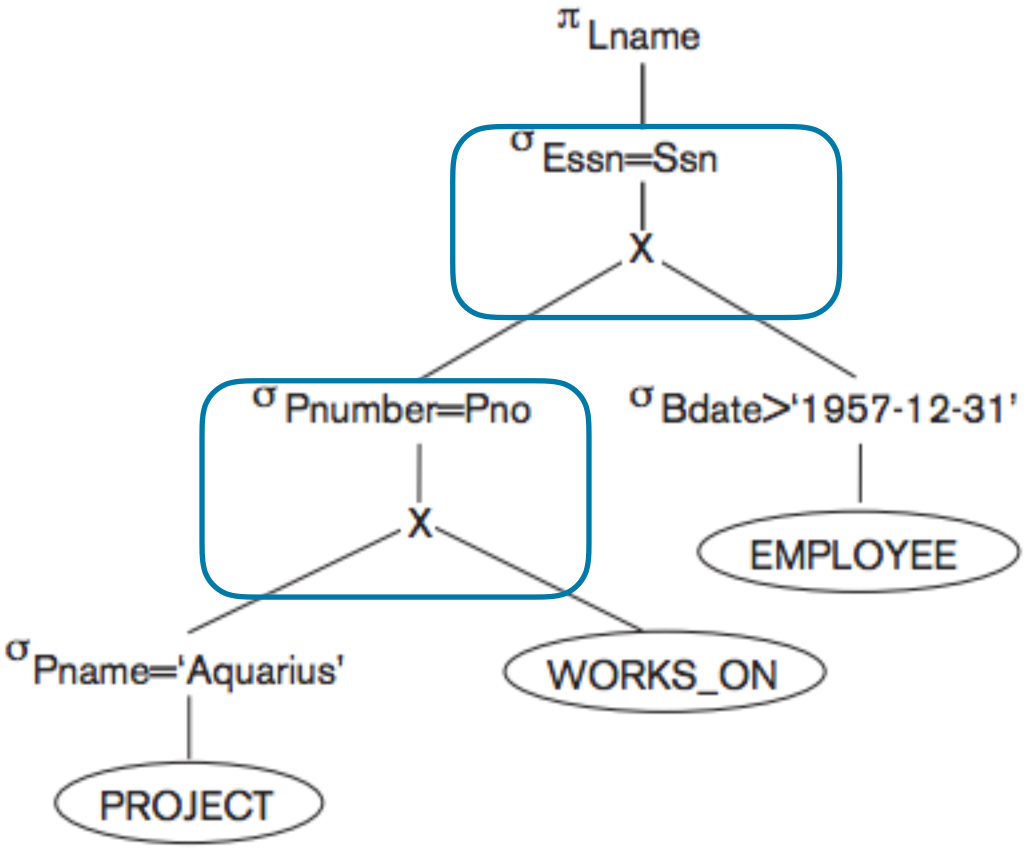
Apply more restrictive SELECT first (left most side of tree)

Returns 1 tuple

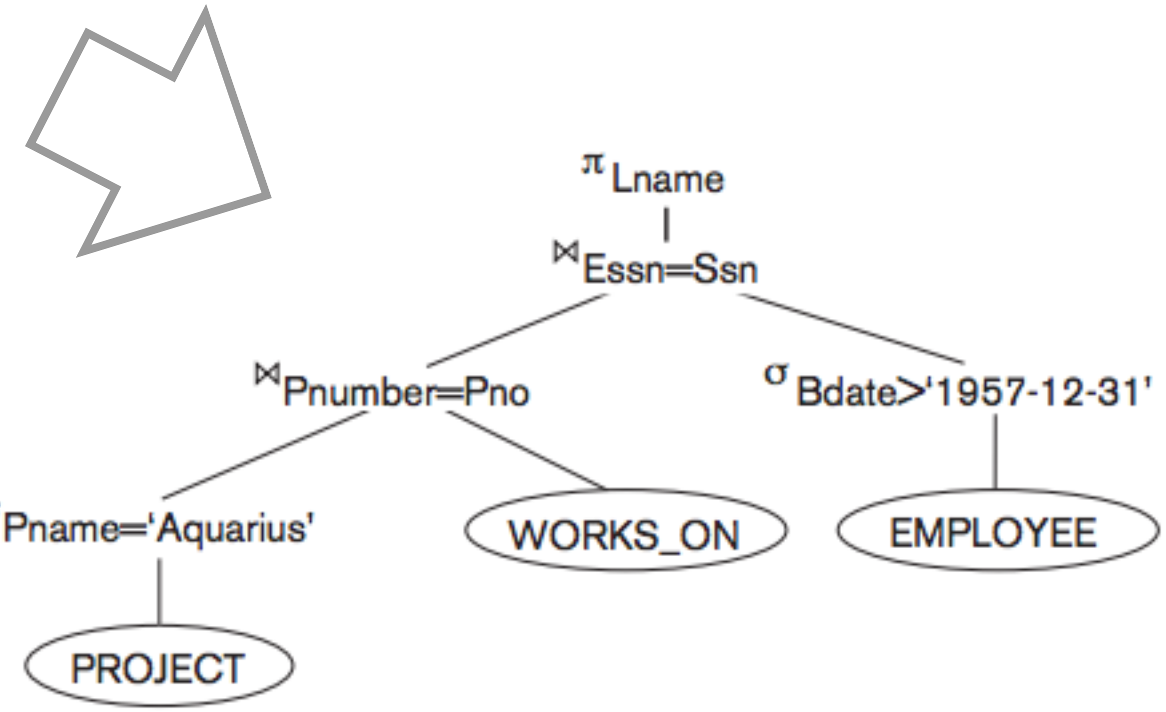
Returns at least 1 tuple

Change position

Example: SQL Query Optimization

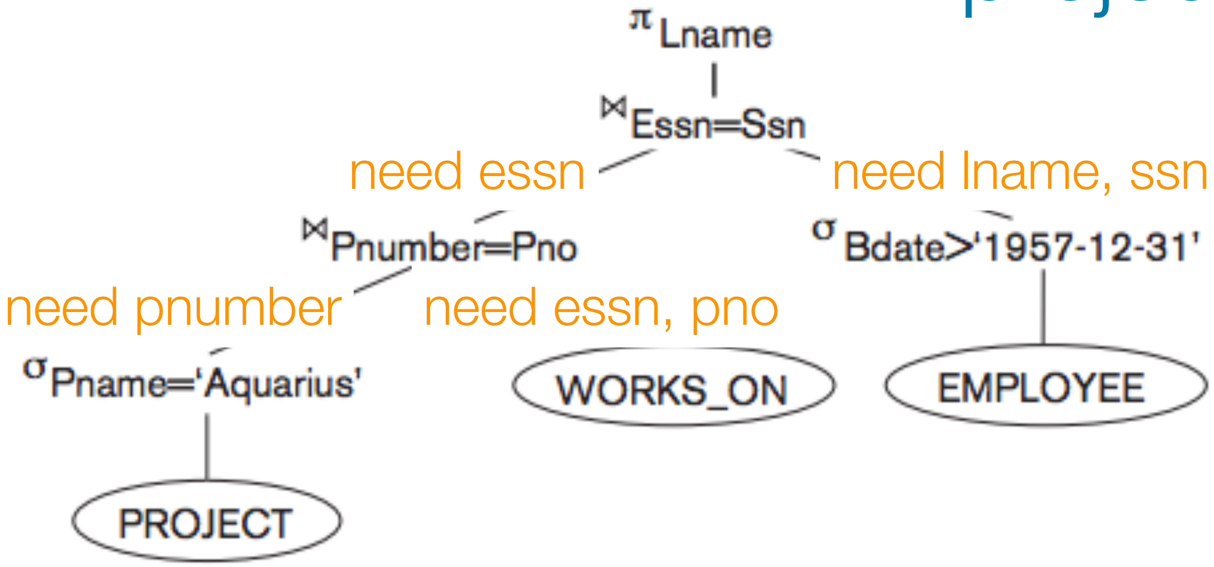


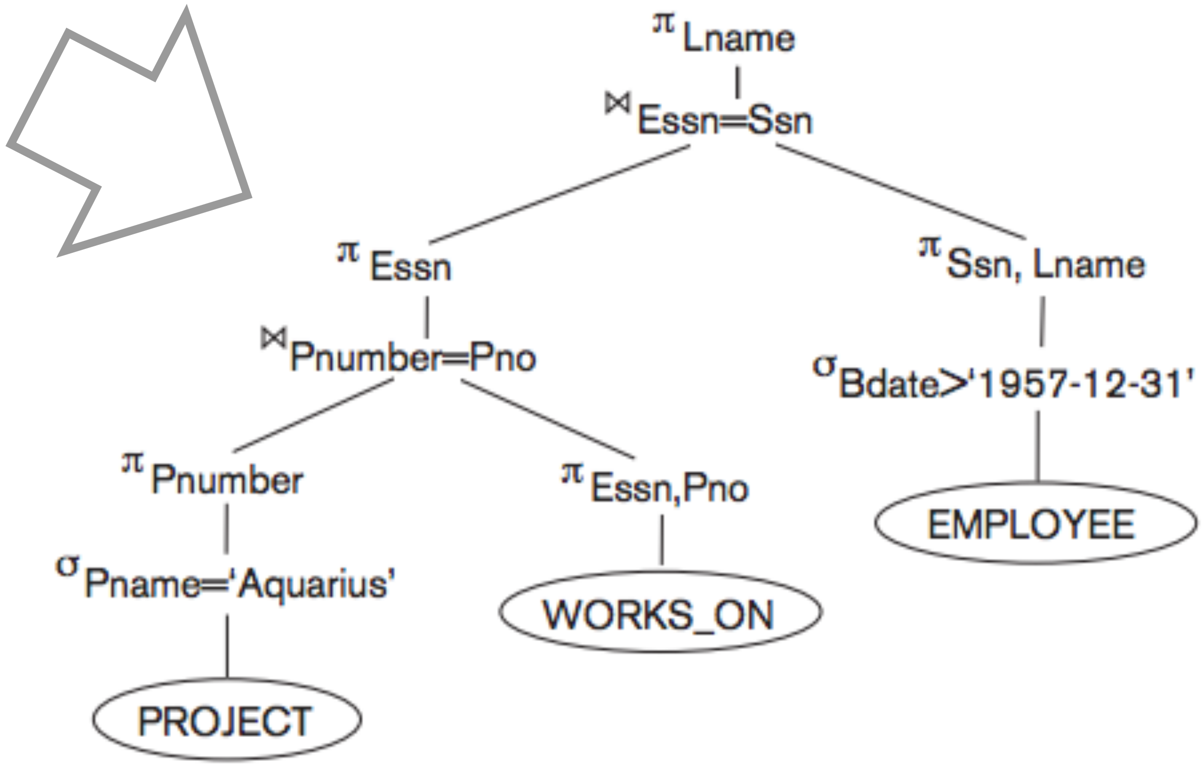
Replace cartesian product and select with join



Example: SQL Query Optimization

Break down and move lists of projection attributes down the tree if possible





Exercise: Query Optimization

Given three relations:
Course(cid, title, dname, credits)
Teaches(iid, cid, sid, semester, year)
Instructor(iid, name, dname, salary)

Query: Find the names of all instructors in the Music department who have taught a course in 2009, along with the titles of the courses that they taught

- What is the initial RA query?
- Transform the query into an "optimal" RA query

Query Optimization

- Logical level: heuristics based optimization to find a better RA query tree
    - SQL query —> initial logical query tree —> optimized query tree
- Physical level: cost-based optimization to determine “best” query plan
    - Optimized query tree —> query execution plans —> cost estimation —> “best” query plan

Cost-based Query Optimization

Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate

- Disk I/O cost
- Storage cost
- Computation cost
- Memory usage cost
- Communication cost (distributed databases)

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)]
  - …

Catalog Information for Index

- Average fan-out of internal nodes of index i for tree-structured indices [f_i]
- Number of levels in index i (i.e., height of index i) [HT_i]
  - Balanced tree on attribute A of relation r: $\lceil \log_{f_i} V(A, r) \rceil$
  - Hash index: 1
- Number of lowest-level index blocks in i (i.e., number of blocks at the leaf level of the index) [LB_i]

Example: Bank Schema

Account relation

- $f_{account}$ = 20 (20 tuples per block)
- V(bname, account) = 50 (50 branches)
- V(balance, account) = 500 (500 different balance values)
- $n_{account}$ = 10000 (10,000 tuples in account)
- $b_{account}$ = 10000 / 20 = 500

SELECT: Simple Algorithms

- Linear search (brute force): selection attribute is not ordered and no index on attribute
  - Cost: # blocks in relation = b_r
  - Account example: 500 I/Os
- Binary search: selection attribute is ordered and no index
  - Cost: $\underbrace{\lceil \log_2(b_r) \rceil}_{\text{locating first tuple}} + \underbrace{\lceil \mathrm{SC(att,\ r)}/f_r \rceil}_{\text{\# blocks with selection}} -1$

Example: Binary search

- How expensive is the following query if we assume Account is sorted by branch name?

$$\sigma_{\text{bname='Perryridge'}}(\text{Account})$$

- Ans:
  - # of tuples in the relation pertaining to Perryridge is total number of tuples divided by distinct values: 10000/50
  - Cost: $\lceil \log_2(500) \rceil + \lceil 200/20 \rceil - 1 = 18$

SELECT: Simple Algorithm with Index

- Index search: cost depends on the number of qualifying tuples, cost of retrieving the tuples and the type of query
  - Primary index
    - Equality search on candidate key: $HT_i + 1$
    - Equality search on nonkey: $HT_i + \lceil \mathrm{SC}(\mathrm{att}, r) / f_r \rceil$
    - Comparison search: $HT_i + \lceil c / f_r \rceil$

estimated number of tuples that satisfy condition

SELECT: Simple Algorithm with Index

- Secondary index
    - Equality search on candidate key: $HT_i + 1$
    - Equality search on nonkey: $HT_i + \mathrm{SC}(\mathrm{att}, r)$
    - Comparison search: $HT_i + \mathrm{LB}_i \times c/n_r + c$

Note that linear file scan maybe cheaper if the number of tuples satisfying the condition is large!

Example: Index search

- How expensive is the following query if we assume primary index on branch name?

$$\sigma_{\text{bname='Perryridge'}}(\text{Account})$$

- Ans:
  - 200 tuples relating to Perryridge branch => clustered index
  - Assume B$^+$-tree index stores 20 pointers per node, then index must have between 3 and 5 leaf nodes with a depth of 2
  - Cost: $2 + \lceil 200/20 \rceil = 12$

SELECT Algorithms: Cost

Search Type	Details	Cost
Linear		b_r
Binary		$\lceil \log_2 b_r \rceil + \lceil \text{SC(att, r)}/f_r \rceil - 1$
Primary index	candidate key	$HT_i + 1$
Primary index	nonkey	$HT_i + \lceil \text{SC(att, r)}/f_r \rceil$
Primary index	comparison	$HT_i + \lceil c/f_r \rceil$
Secondary index	candidate key	$HT_i + 1$
Secondary index	nonkey	$HT_i + \text{SC(att, r)}$
Secondary index	comparison	$HT_i + (LB_i c)/n_r + c$

Exercise: SELECT

- Employee relation with clustering index on salary:
  - nemployee = 10,000 (10,000 tuples in employee)
  - bemployee = 2,000 (2,000 blocks)
  - Secondary index (B+-Tree) on SSN (key attribute)
    - HTi = 4 levels
- What algorithm would be used for the following query and why? $\sigma_{\text{SSN}=123456789}(\text{Employee})$

Exercise: SELECT

Same employee relation with clustering index on salary:

- Secondary index (B^+-Tree) on DNO (non-key)
  - HTi = 2
  - LBi = 4 (4 first level index blocks)
  - V(DNO, employee) = 125
- What algorithm would be used for the following query and why?

$$\sigma_{\text{DNO}>5}(\text{Employee})$$

SELECT: Complex Algorithms

- Conjunctive selection (several conditions with AND)
  - Single index: retrieve records satisfying some attribute condition (with index) and check remaining conditions
  - Composite index
  - Intersection of multiple indexes
- Disjunctive selection (several conditions with OR)
  - Index/binary search if all conditions have access path and take union
  - Linear search otherwise

Example: Complex search

- How expensive if we want to find accounts where the branch name is Perryridge with a balance of 1200 if we assume there is a primary index on branch name and secondary on balance?
- Ans for using one index:
  - Cost for branch name: 12 block reads
  - Balance index is not clustered, so expected selection is 10,000 / 500 = 20 accounts
  - Cost for balance: 2 + 20 = 22 block reads
  - Thus use branch name index, even if it is less selective!

Example: Complex search

- Ans for using intersection of two indexes:
  - Use index on balance to retrieve set of S1 pointers: 2 reads
  - Use index on branch name to retrieve set of S2 pointers: 2 reads
  - Take intersection of the two
  - Estimate 1 tuple in 50 * 500 meets both conditions, so we estimate the intersection of two has one pointer
  - Estimated cost: 5 block reads

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

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

- 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 $\text{Depositor} \bowtie \text{Customer}$
- 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 $n_{\text{depositor}}$ = 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 outer relation

S is inner relation

Nested Loop Join Cost

- 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
```

Read in tuples of R: br

For every tuple in R read S: bs

Worst case: $b_r + n_r \times b_s$

Nested Loop Join Cost

- 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
```

Read in tuples of R: br

For every tuple in R read S: bs

If smaller block fits into memory, we can avoid the cost of re-reading relation S — cost = $b_r + b_s$

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
```

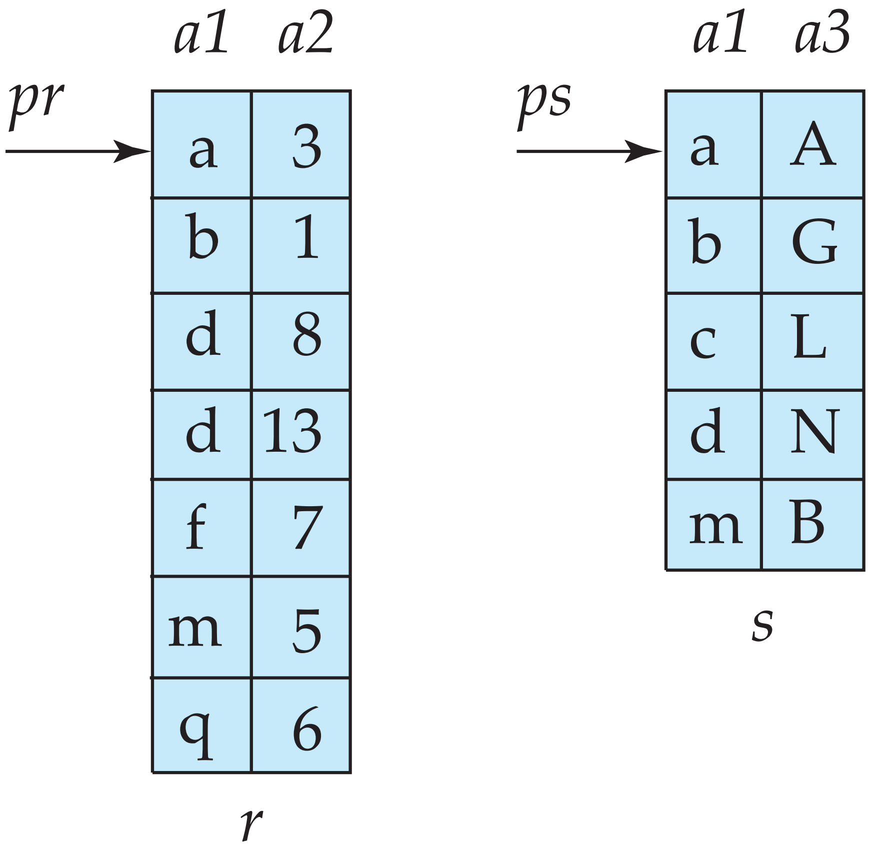
- Worst case: buffer only has space for one page of r and one page of index, estimated cost is $b_r + n_r * c$ (c is cost of single selection on s using join condition)
- If indices available on both relations, use one with fewer tuples as outer relation

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



r

a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

s

a1	*a3*
a	A
b	G
c	L
d	N
m	B

Figure 12.8 from Database System Concepts book

Sort-Merge Join

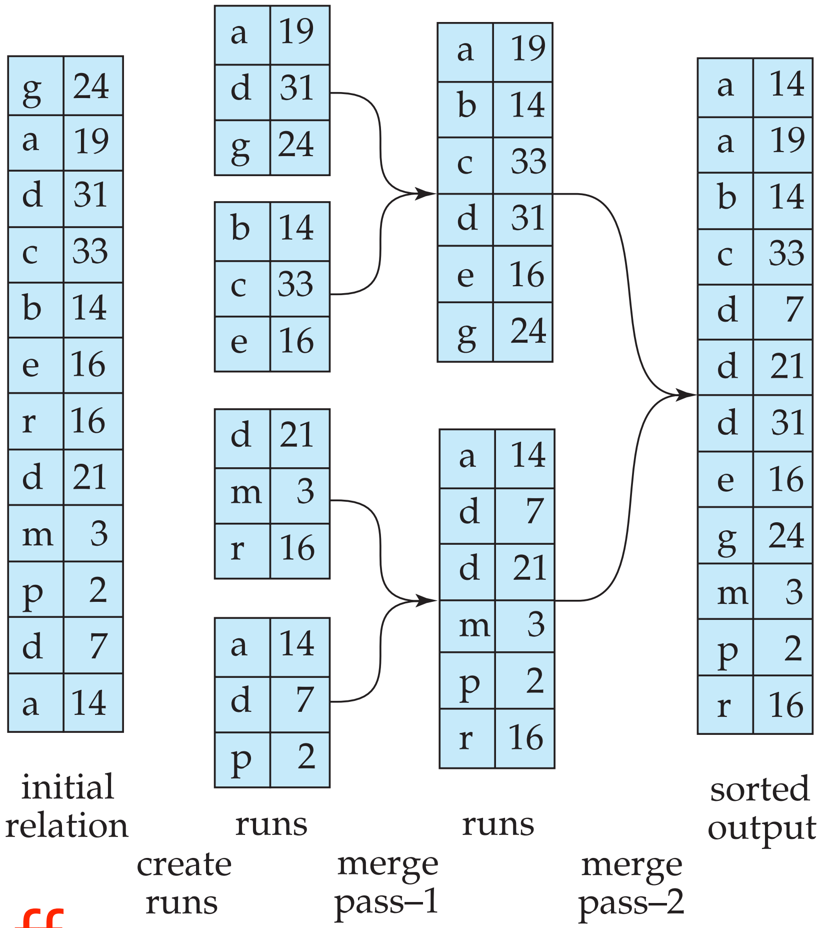
- Can only be used for equijoins and natural joins
- Each tuple needs to be read only once, and as a result, each block is also read only once
  cost = sorting cost + b_r + b_s
- If one relation is sorted, and other has secondary B+-tree index on join attribute, hybrid merge-joins are possible

External Sort Merge Algorithm

- Sort r records, stored in b file blocks with a total memory space of M blocks (relation is larger than memory)
- Total cost:

$$2b_r(\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

NOTE: that the previous slides were off by a factor of 2 for the second part!



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

a	19
d	31
g	24

b	14
c	33
e	16

d	21
m	3
r	16

a	14
d	7
p	2

a	19
b	14
c	33
d	31
e	16
g	24

a	14
d	7
d	21
m	3
p	2
r	16

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

Figure 12.4 from Database System Concepts book

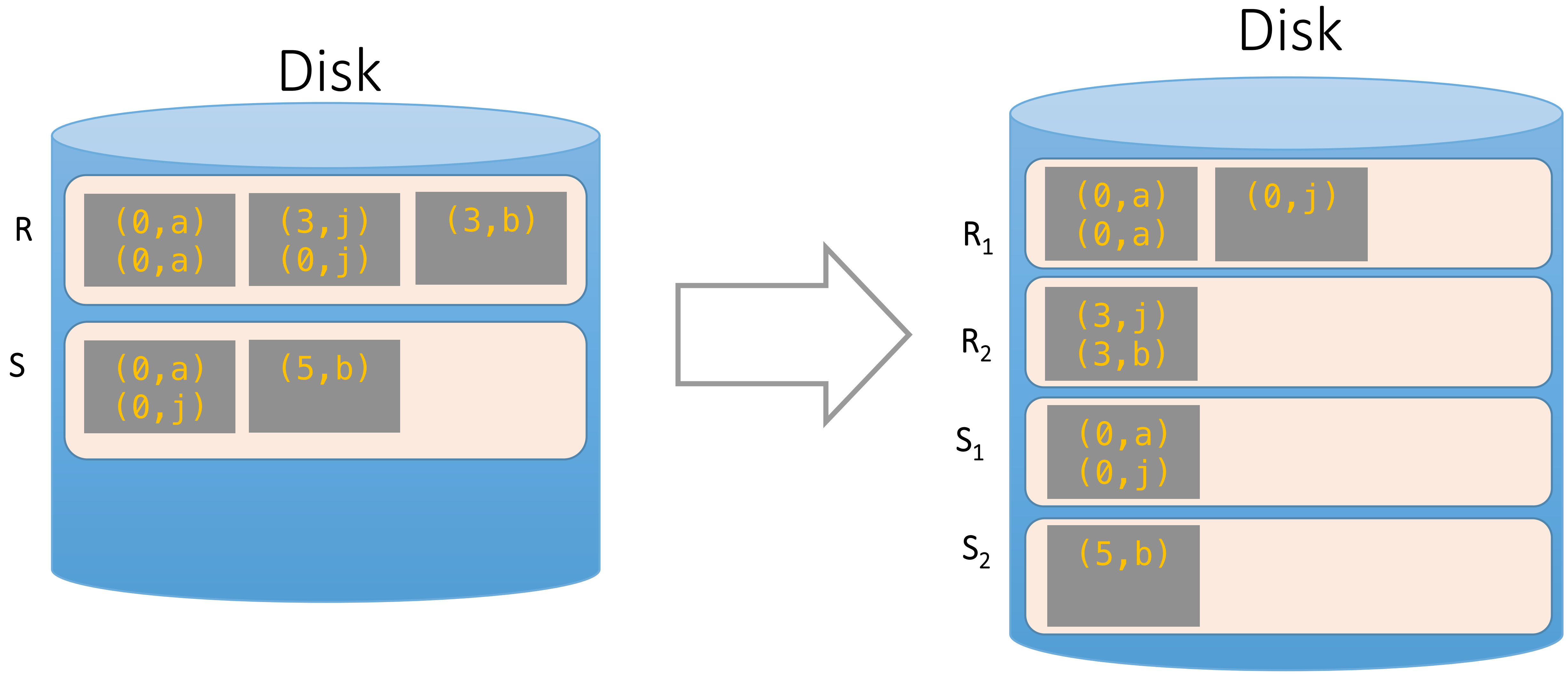
Sort-Merge vs Nested-Block

- Assume we have 100 blocks of memory, relation R has 1000 blocks and relation S has 500 blocks
- What is cost of nested-block?
- What is cost of sorted merge?
- What happens if we have only 35 blocks of memory?

Hash-Join

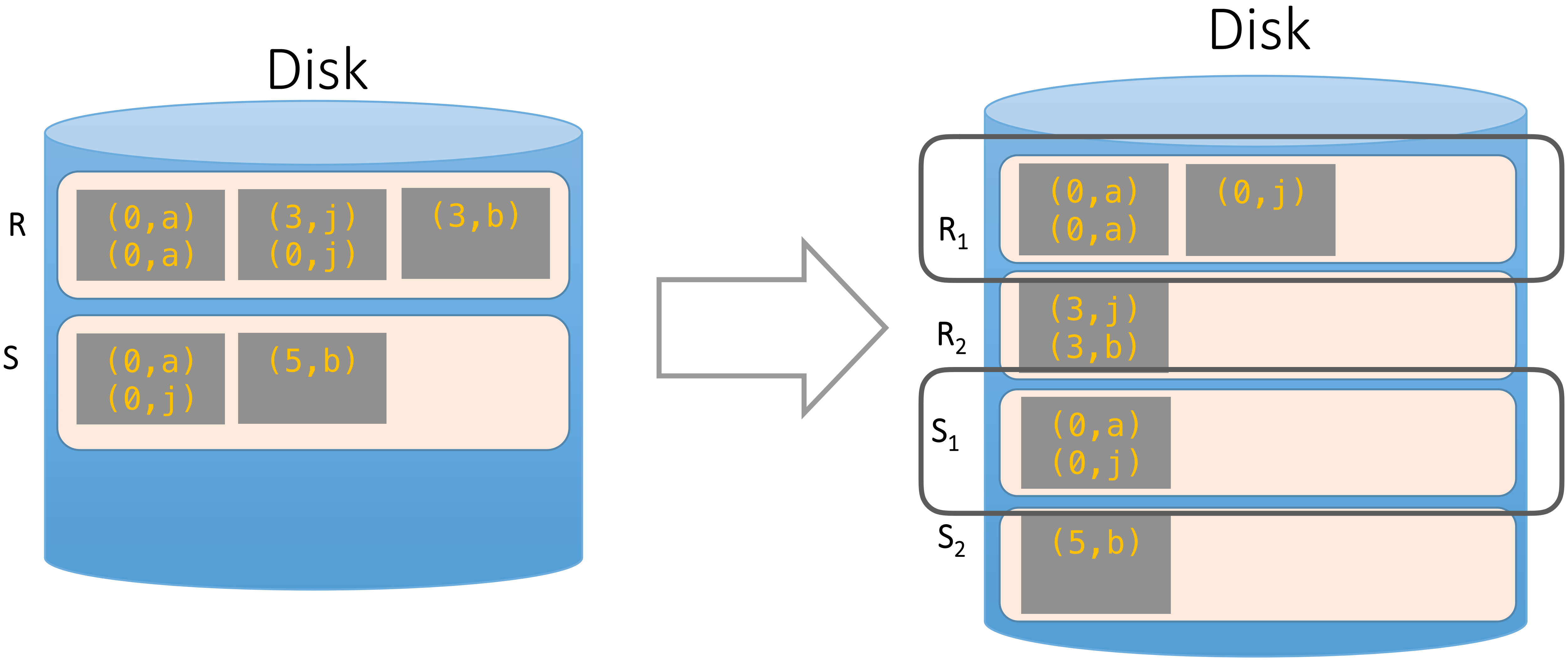
- Applicable for equijoins and natural joins
- A hash function, h, is used to partition tuples of both relations into sets that have same hash value on the join attributes
- Tuples in the corresponding same buckets just need to be compared with one another and not with all the other tuples in the other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

Example: Hash-Join



Step 1: Use hash function to partition into B buckets

Example: Hash-Join



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

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

JOIN Algorithms Cost

Type	Details	Cost
Nested loop		$n_R b_S + b_R$
Nested block		$b_R b_S + b_R$
Indexed nested loop		$b_R + n_R c$
Sort merge join		$\text{sort cost} + b_R + b_S$
Hash join	no recursive partitioning	$3b_R + 3b_S$
Hash join	recursive partitioning	$2(b_R + b_S)\lceil \log_{M-1}(b_S) - 1 \rceil$ $+ b_R + b_S$

if both fit in memory: $b_R + b_S$

Exercise: JOIN Operation

Employee and Department

- $n_{employee}$ = 10,000 (10,000 tuples in employee)
- $b_{employee}$ = 2,000 (2,000 blocks)
- $n_{department}$ = 125 (125 tuples in department)
- $b_{department}$ = 13 (13 blocks)
- Primary index dnumber in department with HTi = 1
- Secondary index mgr_ssn in department with HTi = 2

Exercise: JOIN Operation

Employee and Department

- What join algorithm makes sense for joining Employee and Department on department number?
- What join algorithm makes sense for joining Employee and Department on manager ssn?

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

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

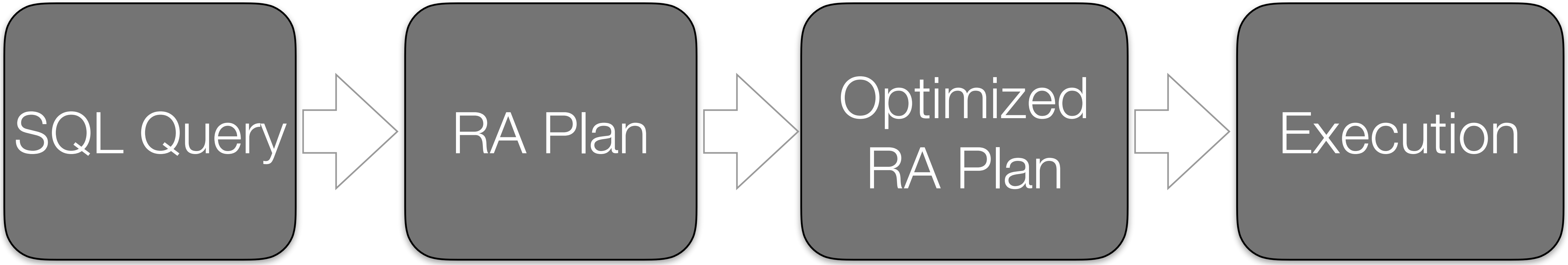
Similar to duplicate elimination

- 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

Recap: Query Processing


SQL Query

RA Plan

Optimized RA Plan

Execution

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

- Query processing
- Cost-based optimization
  - SELECT, JOIN algorithms
  - Other operation algorithms (PROJECT, SET, Aggregate)