Principles of Distributed Database Systems

M. Tamer Özsu
Patrick Valduriez

Outline

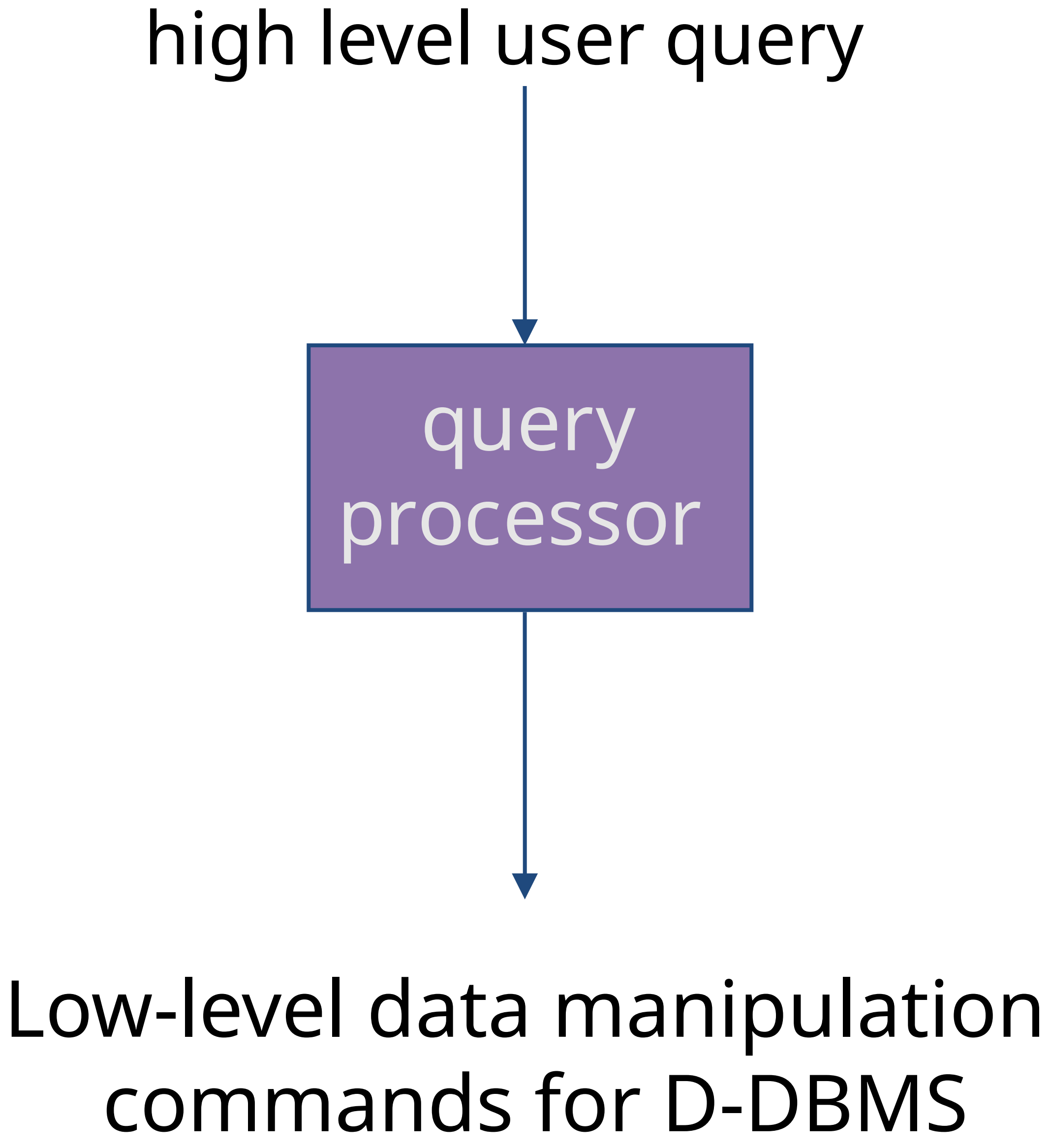
- Introduction
- Distributed and parallel database design
- Distributed data control
- Distributed Query Processing
- Distributed Transaction Processing
- Data Replication
- Database Integration – Multidatabase Systems
- Parallel Database Systems
- Peer-to-Peer Data Management
- Big Data Processing
- NoSQL, NewSQL and Polystores
- Web Data Management

Outline

- Distributed Query Processing
    - Introduction
    - Query Decomposition and Localization
    - Introduction to QO
    - Centralized query optimization
    - Join Ordering
    - Distributed Query Optimization
    - Adaptive Query Processing

- Slides of the 3rd Edition of the textbook !

Query Processing in a DDBMS



high level user query

query processor

Low-level data manipulation commands for D-DBMS

Query Processing Components

- Query language that is used

  SQL: “intergalactic dataspeak”

- Query execution methodology

  The steps that one goes through in executing high-level (declarative) user queries.

- Query optimization

  How do we determine the “best” execution plan?

- We assume a homogeneous D-DBMS

Selecting Alternatives

```
SELECTENAME
FROM    EMP,ASG
WHERE   EMP.ENO = ASG.ENO
AND   RESP = "Manager"
```

Strategy 1

$\Pi_{ENAME}(\sigma_{RESP=\text{“Manager”} \wedge EMP.ENO=ASG.ENO}(EMP \times ASG))$

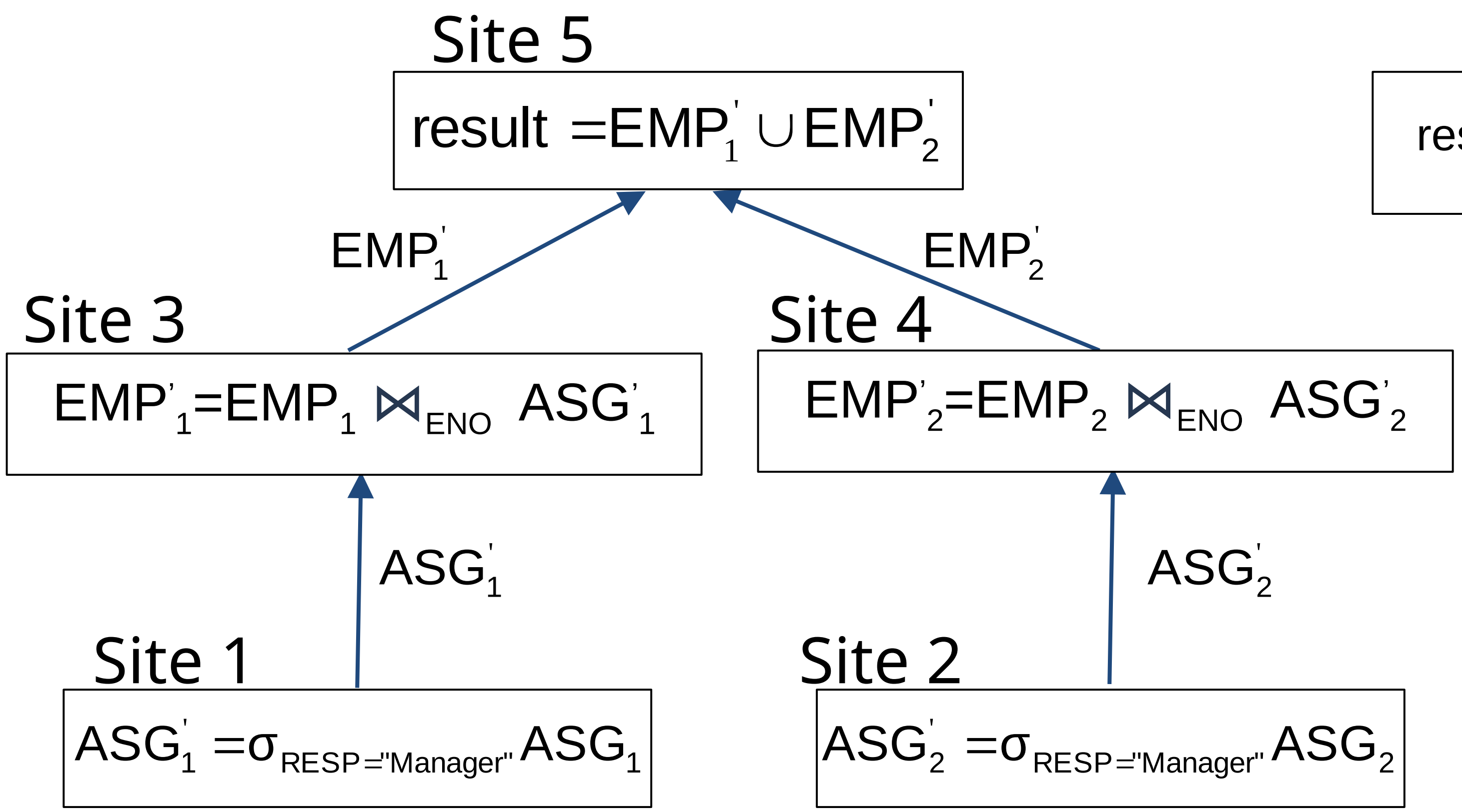
Strategy 2

$\Pi_{ENAME}(EMP \bowtie_{ENO} (\sigma_{RESP=\text{“Manager”}}(ASG))$

Strategy 2 avoids Cartesian product, so may be “better”

What is the Problem?

Site 1	Site 2	Site 3	Site 4	Site 5
$ASG_1=\sigma_{ENO\leq\text{“E3”}}(ASG)$	$ASG_2=\sigma_{ENO>\text{“E3”}}(ASG)$	$EMP_1=\sigma_{ENO\leq\text{“E3”}}(EMP)$	$EMP_2=\sigma_{ENO>\text{“E3”}}(EMP)$	Result



Site 5

$result = EMP'_1 \cup EMP'_2$

EMP'_1

EMP'_2

Site 3

$EMP'_1=EMP_1 \bowtie_{ENO} ASG'_1$

Site 4

$EMP'_2=EMP_2 \bowtie_{ENO} ASG'_2$

ASG'_1

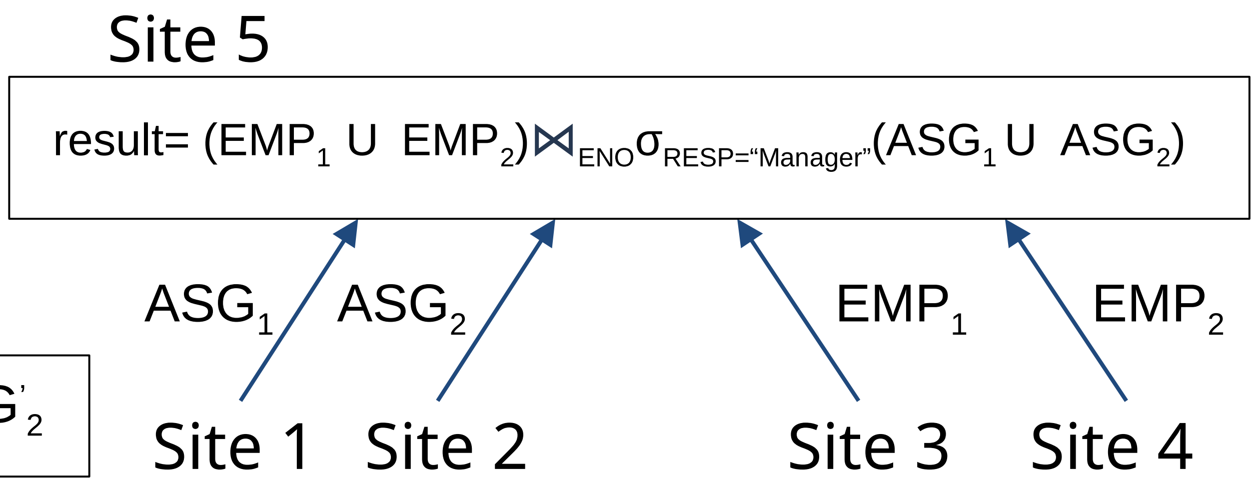
ASG'_2

Site 1

$ASG'_1 = \sigma_{RESP=\text{"Manager"}} ASG_1$

Site 2

$ASG'_2 = \sigma_{RESP=\text{"Manager"}} ASG_2$



Site 5

$result = (EMP_1 \cup EMP_2) \bowtie_{ENO} \sigma_{RESP=\text{“Manager”}}(ASG_1 \cup ASG_2)$

ASG_1

ASG_2

EMP_1

EMP_2

Site 1 Site 2 Site 3 Site 4

Cost of Alternatives

- Assume

  size(EMP) = 400, *size*(ASG) = 1000

  tuple access cost = 1 unit; tuple transfer cost = 10 units

  ASG and EMP are locally clustered on attributes RESP and ENO

- Strategy 1

  produce ASG': assume 20 managers; (10+10) ∗ tuple access cost = 20
  transfer ASG' to the sites of EMP: (10+10) ∗ tuple transfer cost = 200
  produce EMP': (10+10) ∗ tuple access cost ∗ 2 = 40
  transfer EMP' to result site: (10+10) ∗ tuple transfer cost = 200

Total Cost 460

- Strategy 2

  transfer EMP to site 5: 400 ∗ tuple transfer cost = 4,000
  transfer ASG to site 5: 1000 ∗ tuple transfer cost = 10,000
  produce ASG': 1000 ∗ tuple access cost = 1,000
  join EMP and ASG': 400 ∗ 20 ∗ tuple access cost = 8,000

Total Cost 23,000

Query Optimization Objectives

- Minimize a cost function
  - I/O cost + CPU cost + communication cost
  - These might have different weights in different distributed environments
- Wide area networks

  communication cost may dominate or vary much

  - ✦ bandwidth
  - ✦ speed
  - ✦ high protocol overhead
- Local area networks

  communication cost not that dominant

  total cost function should be considered
- Can also maximize throughput

Complexity of Relational Operations

- Assume

  relations of cardinality n

  sequential scan

Operation	Complexity
Select Project (without duplicate elimination)	$O(n)$
Project (with duplicate elimination) Group	$O(n \ast \log n)$
Join Semi-join Division Set Operators	$O(n \ast \log n)$
Cartesian Product	$O(n^2)$

Query Optimization Issues – Types Of Optimizers

- Exhaustive search
  - Cost-based
  - Optimal
  - Combinatorial complexity in the number of relations
  - Dynamic programming
- Heuristics
  - Not optimal
  - Restrict the solution space
  - Regroup common sub-expressions
  - Perform selection, projection first
  - Replace a join by a series of semijoins
  - Reorder operations to reduce intermediate relation size
  - Optimize individual operations

Query Optimization Issues – Optimization Granularity

- Single query at a time
    - Cannot use common intermediate results
- Multiple queries at a time
    - Efficient if many similar queries
    - Decision space is much larger

Query Optimization Issues – Optimization Timing

- Static
  - Compilation optimize prior to the execution
  - Difficult to estimate the size of the intermediate results⇒error propagation
  - Can amortize over many executions
  - R*
- Dynamic
  - Run time optimization
  - Exact information on the intermediate relation sizes
  - Have to reoptimize for multiple executions
  - Distributed INGRES
- Hybrid
  - Compile using a static algorithm
  - If the error in estimate sizes > threshold, reoptimize at run time
  - Mermaid

Query Optimization Issues – Statistics

- Relation
  - Cardinality
  - Size of a tuple
  - Fraction of tuples participating in a join with another relation
- Attribute
  - Cardinality of domain
  - Actual nuymber of distinct values
- Common assumptions
  - Independence between different attribute values
  - Uniform distribution of attribute values within their domain

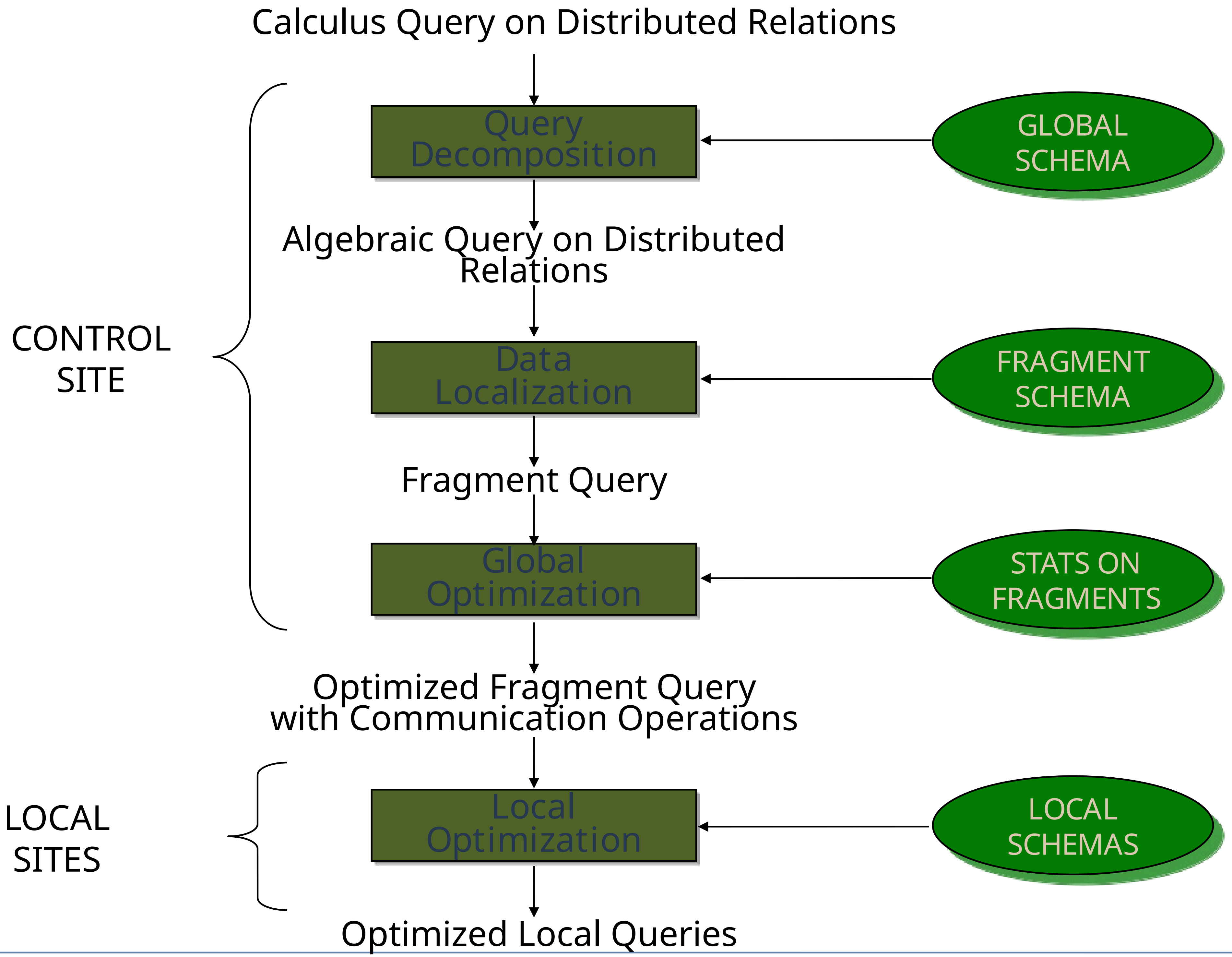
Query Optimization Issues – Decision Sites

- Centralized
  - Single site determines the “best” schedule
  - Simple
  - Need knowledge about the entire distributed database
- Distributed
  - Cooperation among sites to determine the schedule
  - Need only local information
  - Cost of cooperation
- Hybrid
  - One site determines the global schedule
  - Each site optimizes the local subqueries

Query Optimization Issues – Network Topology

- Wide area networks (WAN) – point-to-point
    - Characteristics
        - Low bandwidth
        - Low speed
        - High protocol overhead
    - Communication cost will dominate; ignore all other cost factors
    - Global schedule to minimize communication cost
    - Local schedules according to centralized query optimization
- Local area networks (LAN)
    - Communication cost not that dominant
    - Total cost function should be considered
    - Broadcasting can be exploited (joins)
    - Special algorithms exist for star networks

Distributed Query Processing Methodology



Outline

- Distributed Query Processing
    - Introduction
    - Query Decomposition and Localization
    - Introduction to query optimization
    - Centralized query optimization
    - Join Ordering
    - Distributed Query Optimization
    - Adaptive Query Processing

Query Decomposition

Input : Calculus query on global relations

- Normalization
  - manipulate query quantifiers and qualification
- Analysis
  - detect and reject “incorrect” queries
  - possible for only a subset of relational calculus
- Simplification
  - eliminate redundant predicates
- Restructuring
  - calculus query → algebraic query
  - more than one translation is possible
  - use transformation rules

Normalization

- Lexical and syntactic analysis
  - check validity (similar to compilers)
  - check for attributes and relations
  - type checking on the qualification
- Put into normal form
  - Conjunctive normal form

    $$(p_{11} \lor p_{12} \lor \ldots \lor p_{1n}) \land \ldots \land (p_{m1} \lor p_{m2} \lor \ldots \lor p_{mn})$$

  - Disjunctive normal form

    $$(p_{11} \land p_{12} \land \ldots \land p_{1n}) \lor \ldots \lor (p_{m1} \land p_{m2} \land \ldots \land p_{mn})$$

  - OR's mapped into union
  - AND's mapped into join or selection

Normalization - example

```
SELECT ENAME
FROM   EMP, ASG
WHERE  EMP.ENO = ASG.ENO
AND    ASG.PNO = "P1"
AND    DUR = 12 OR DUR = 24
```

$$\text{EMP.ENO} = \text{ASG.ENO} \land \text{ASG.PNO} = \text{“P1”} \land (\text{DUR} = 12 \lor \text{DUR} = 24)$$

$$\begin{aligned}&(\text{EMP.ENO} = \text{ASG.ENO} \land \text{ASG.PNO} = \text{“P1”} \land \text{DUR} = 12) \lor \\ &(\text{EMP.ENO} = \text{ASG.ENO} \land \text{ASG.PNO} = \text{“P1”} \land \text{DUR} = 24)\end{aligned}$$

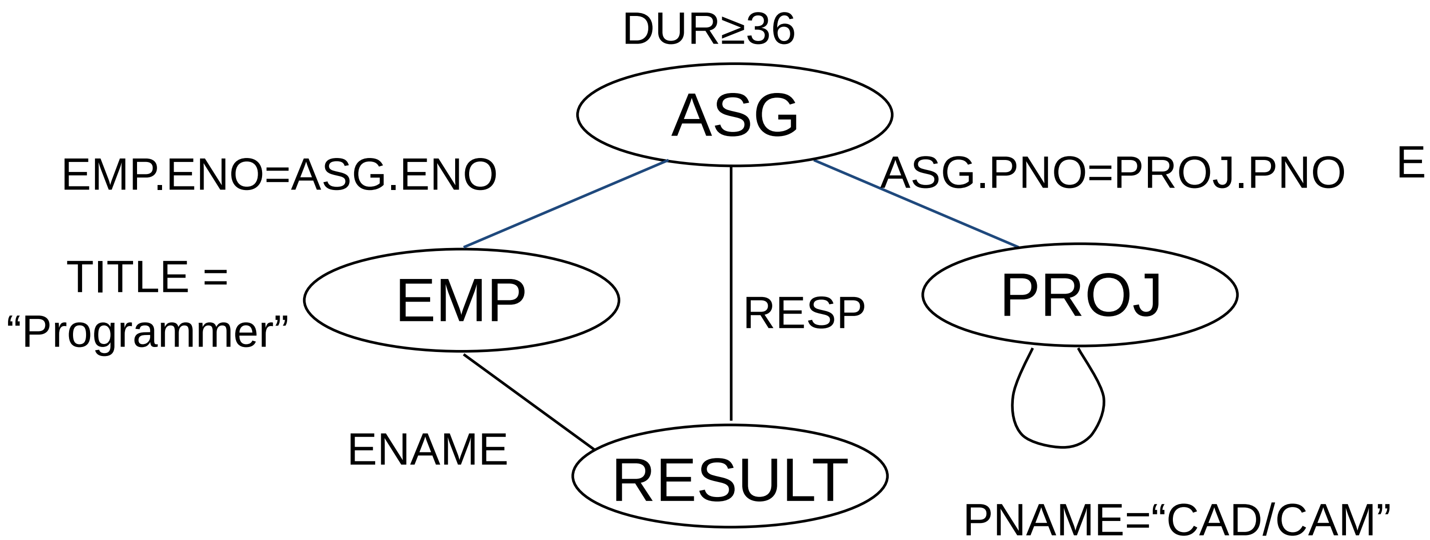
Analysis

- Refute incorrect queries
- Type incorrect
  - If any of its attribute or relation names are not defined in the global schema
  - If operations are applied to attributes of the wrong type
- Semantically incorrect
  - Components do not contribute in any way to the generation of the result
  - Only a subset of relational calculus queries can be tested for correctness
  - Those that do not contain disjunction and negation
  - To detect
    - connection graph (query graph)
    - join graph

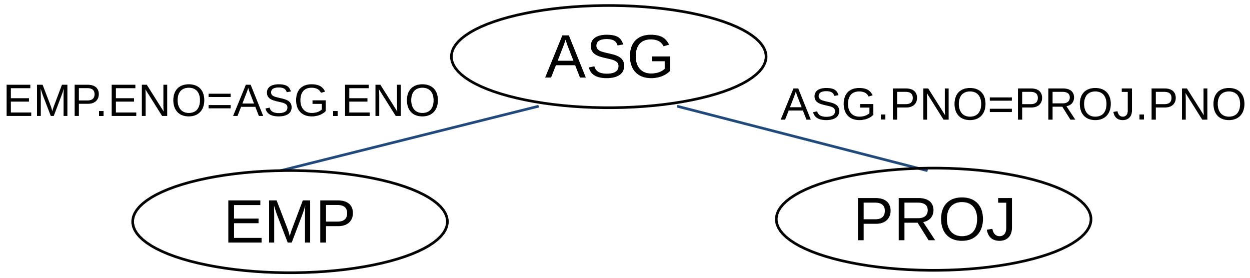
Analysis – Example

```
SELECT  ENAME,RESP
FROM EMP, ASG, PROJ
WHERE   EMP.ENO = ASG.ENO
AND  ASG.PNO = PROJ.PNO
AND  PNAME = "CAD/CAM"
AND  DUR ≥ 36
AND  TITLE = "Programmer"
```

Query graph



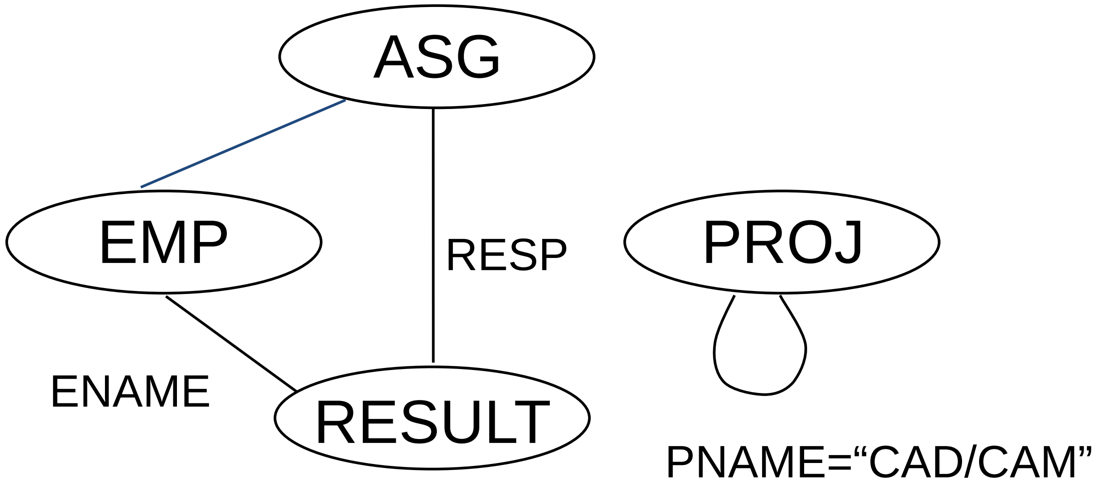
Join graph



Analysis

If the query graph is not connected, the query may be wrong or use Cartesian product

```
SELECT ENAME,RESP
FROM EMP, ASG, PROJ
WHERE  EMP.ENO = ASG.ENO
AND  PNAME = "CAD/CAM"
AND  DUR > 36
AND  TITLE = "Programmer"
```



Simplification

- Why simplify?
    - Remember the example
- How? Use transformation rules
    - Elimination of redundancy
        - idempotency rules

          $p_1 \wedge \neg(p_1) \Leftrightarrow$ false

          $p_1 \wedge (p_1 \vee p_2) \Leftrightarrow p_1$

          $p_1 \wedge$ false $\Leftrightarrow p_1$

          …
    - Application of transitivity
    - Use of integrity rules

Simplification – Example

```
SELECT TITLE
FROM EMP
WHERE  EMP.ENAME = "J. Doe"
OR (NOT(EMP.TITLE = "Programmer")
AND  (EMP.TITLE = "Programmer"
OR EMP.TITLE = "Elect. Eng.")
AND  NOT(EMP.TITLE = "Elect. Eng."))
```

□

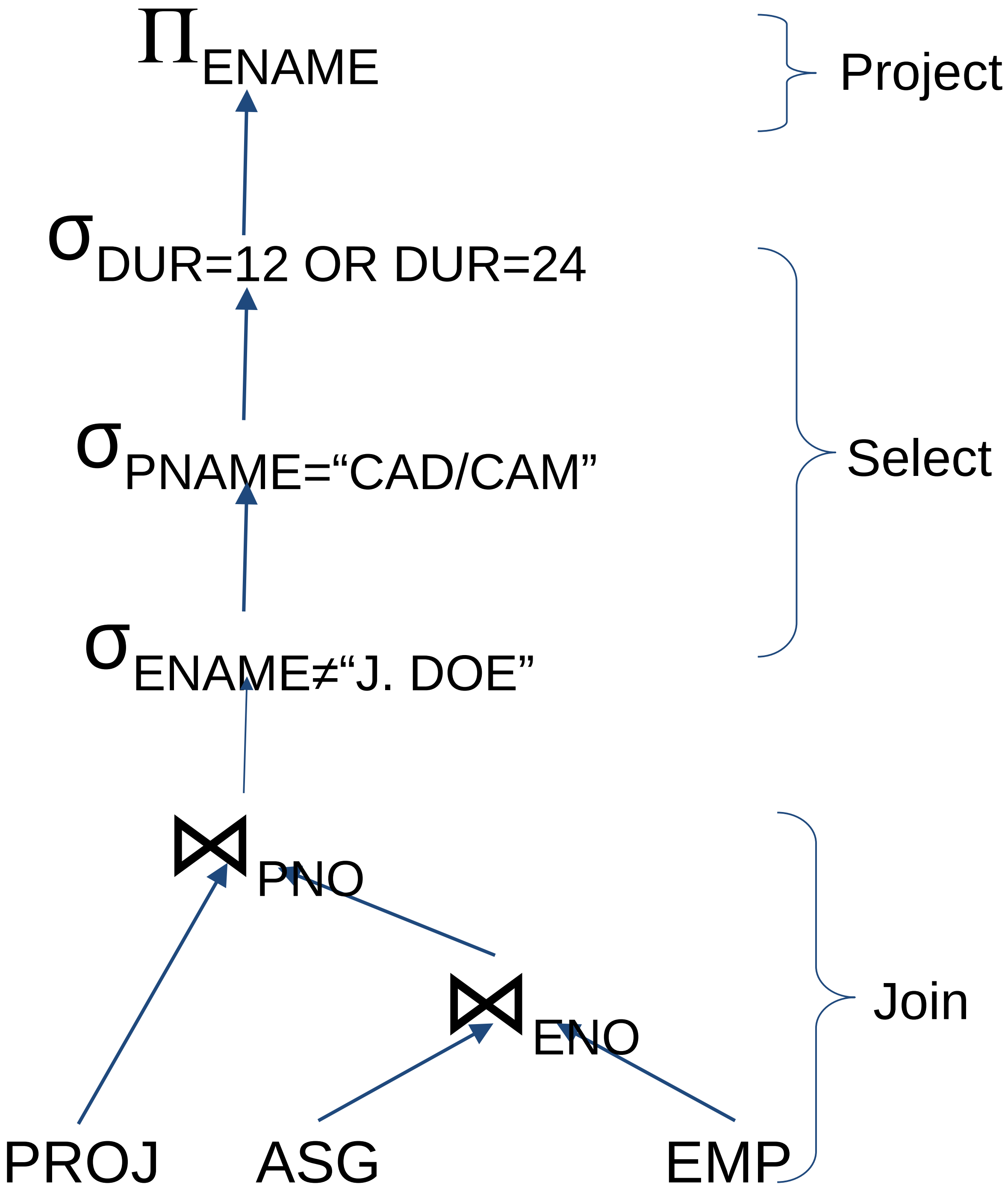
```
SELECT TITLE
FROM EMP
WHERE  EMP.ENAME = "J. Doe"
```

Restructuring

- Convert relational calculus to relational algebra
- Make use of query trees
- Example

Find the names of employees other than J. Doe who worked on the CAD/CAM project for either 1 or 2 years.

```
SELECT ENAME
FROM EMP, ASG, PROJ
WHERE  EMP.ENO = ASG.ENO
AND  ASG.PNO = PROJ.PNO
AND  ENAME≠ "J. Doe"
AND  PNAME = "CAD/CAM"
AND  (DUR = 12 OR DUR = 24)
```



Π_{ENAME}

Project

$\sigma_{\text{DUR=12 OR DUR=24}}$

$\sigma_{\text{PNAME=“CAD/CAM”}}$

$\sigma_{\text{ENAME≠“J. DOE”}}$

Select

$\bowtie_{\text{PNO}}$

$\bowtie_{\text{ENO}}$

PROJ ASG EMP

Join

Restructuring –Transformation Rules

- Commutativity of binary operations

$R \times S \Leftrightarrow S \times R$

$R \bowtie S \Leftrightarrow S \bowtie R$

$R \cup S \Leftrightarrow S \cup R$

- Associativity of binary operations

$(R \times S) \times T \Leftrightarrow R \times (S \times T)$

$(R \bowtie S) \bowtie T \Leftrightarrow R \bowtie (S \bowtie T)$

- Idempotence of unary operations

$\Pi_{A'}(\Pi_{A''}(R)) \Leftrightarrow \Pi_{A'}(R)$

$\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) \Leftrightarrow \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$

where $R[A]$ and $A' \subseteq A$, $A'' \subseteq A$ and $A' \subseteq A''$

- Commuting selection with projection

Restructuring – Transformation Rules

- Commuting selection with binary operations

$$\sigma_{p(A)}(R \times S) \Leftrightarrow (\sigma_{p(A)}(R)) \times S$$

$$\sigma_{p(A_i)}(R \bowtie_{(A_j,B_k)} S) \Leftrightarrow (\sigma_{p(A_i)}(R)) \bowtie_{(A_j,B_k)} S$$

$$\sigma_{p(A_i)}(R \cup T) \Leftrightarrow \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$$

where A_i belongs to R and T

- Commuting projection with binary operations

$$\Pi_C(R \times S) \Leftrightarrow \Pi_{A'}(R) \times \Pi_{B'}(S)$$

$$\Pi_C(R \bowtie_{(A_j,B_k)} S) \Leftrightarrow \Pi_{A'}(R) \bowtie_{(A_j,B_k)} \Pi_{B'}(S)$$

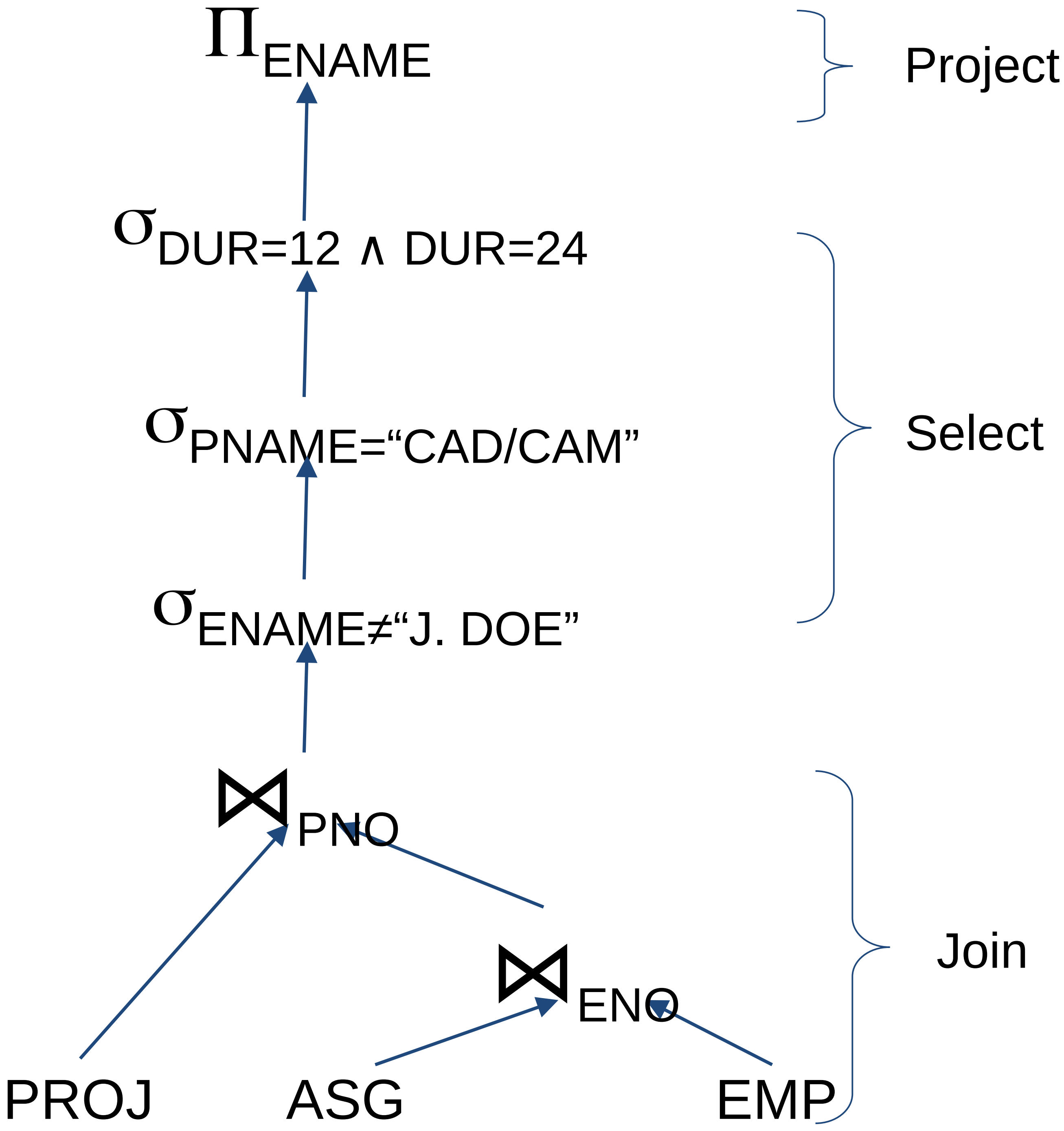
$$\Pi_C(R \cup S) \Leftrightarrow \Pi_C(R) \cup \Pi_C(S)$$

where $R[A]$ and $S[B]$; $C = A' \cup B'$ where $A' \subseteq A$, $B' \subseteq B$

Example

Recall the previous example:
Find the names of employees other than J. Doe who worked on the CAD/CAM project for either one or two years.

```
SELECT ENAME
FROM PROJ, ASG, EMP
WHERE  ASG.ENO=EMP.ENO
AND  ASG.PNO=PROJ.PNO
AND  ENAME ≠ "J. Doe"
AND  PROJ.PNAME="CAD/CAM"
AND  (DUR=12 OR DUR=24)
```



Π_{ENAME}

Project

$\sigma_{\text{DUR=12} \wedge \text{DUR=24}}$

$\sigma_{\text{PNAME="CAD/CAM"}}$

$\sigma_{\text{ENAME}\neq\text{"J. DOE"}}$

Select

$\bowtie_{\text{PNO}}$

$\bowtie_{\text{ENO}}$

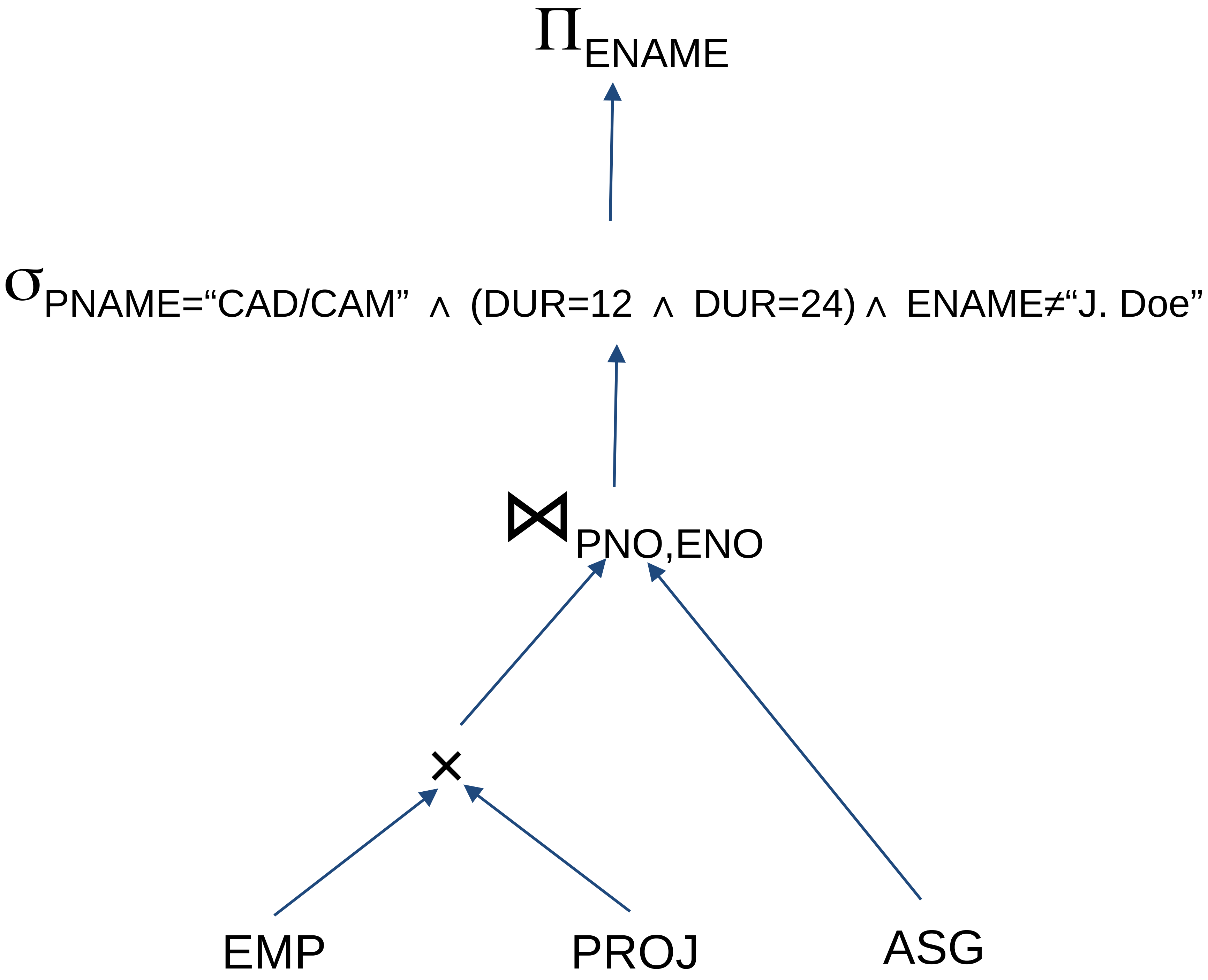
PROJ

ASG

EMP

Join

Equivalent Query



Π_{ENAME}

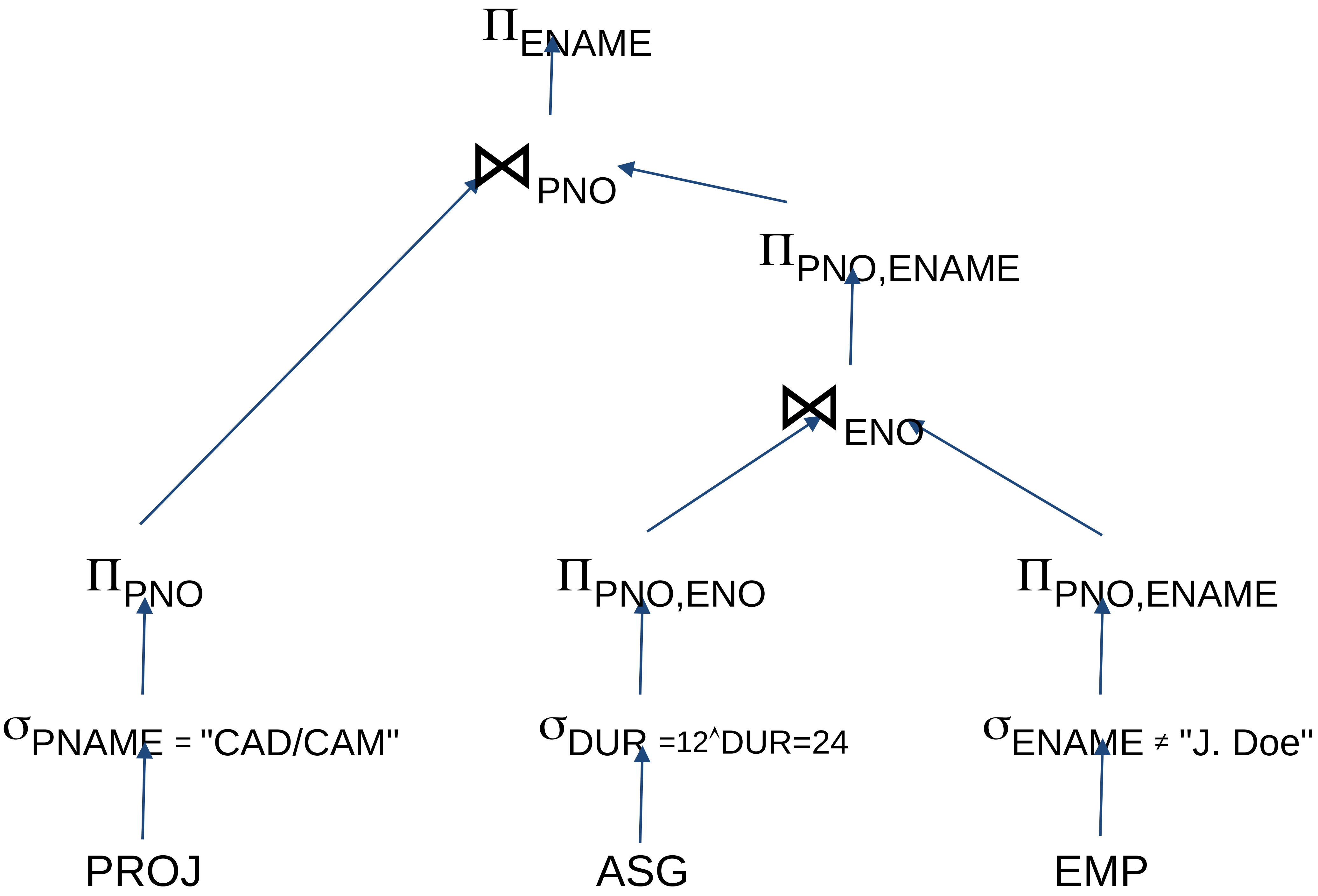
$\sigma_{\text{PNAME=“CAD/CAM”} \wedge (\text{DUR=12} \wedge \text{DUR=24}) \wedge \text{ENAME} \neq \text{“J. Doe”}}$

$\bowtie_{\text{PNO,ENO}}$

$\times$

EMP PROJ ASG

Restructuring



Π_{ENAME}

$\bowtie_{\text{PNO}}$

$\Pi_{\text{PNO,ENAME}}$

$\bowtie_{\text{ENO}}$

Π_{PNO} $\Pi_{\text{PNO,ENO}}$ $\Pi_{\text{PNO,ENAME}}$

$\sigma_{\text{PNAME = "CAD/CAM"}}$ $\sigma_{\text{DUR =12}\wedge\text{DUR=24}}$ $\sigma_{\text{ENAME} \neq \text{"J. Doe"}}$

PROJ ASG EMP

Outline

- Distributed Query Processing
  - Introduction
  - Query Decomposition and Localization
  - Introduction to query optimization
  - Centralized query optimization
  - Join Ordering
  - Distributed Query Optimization
  - Adaptive Query Processing

Data Localization

Input: Algebraic query on distributed relations

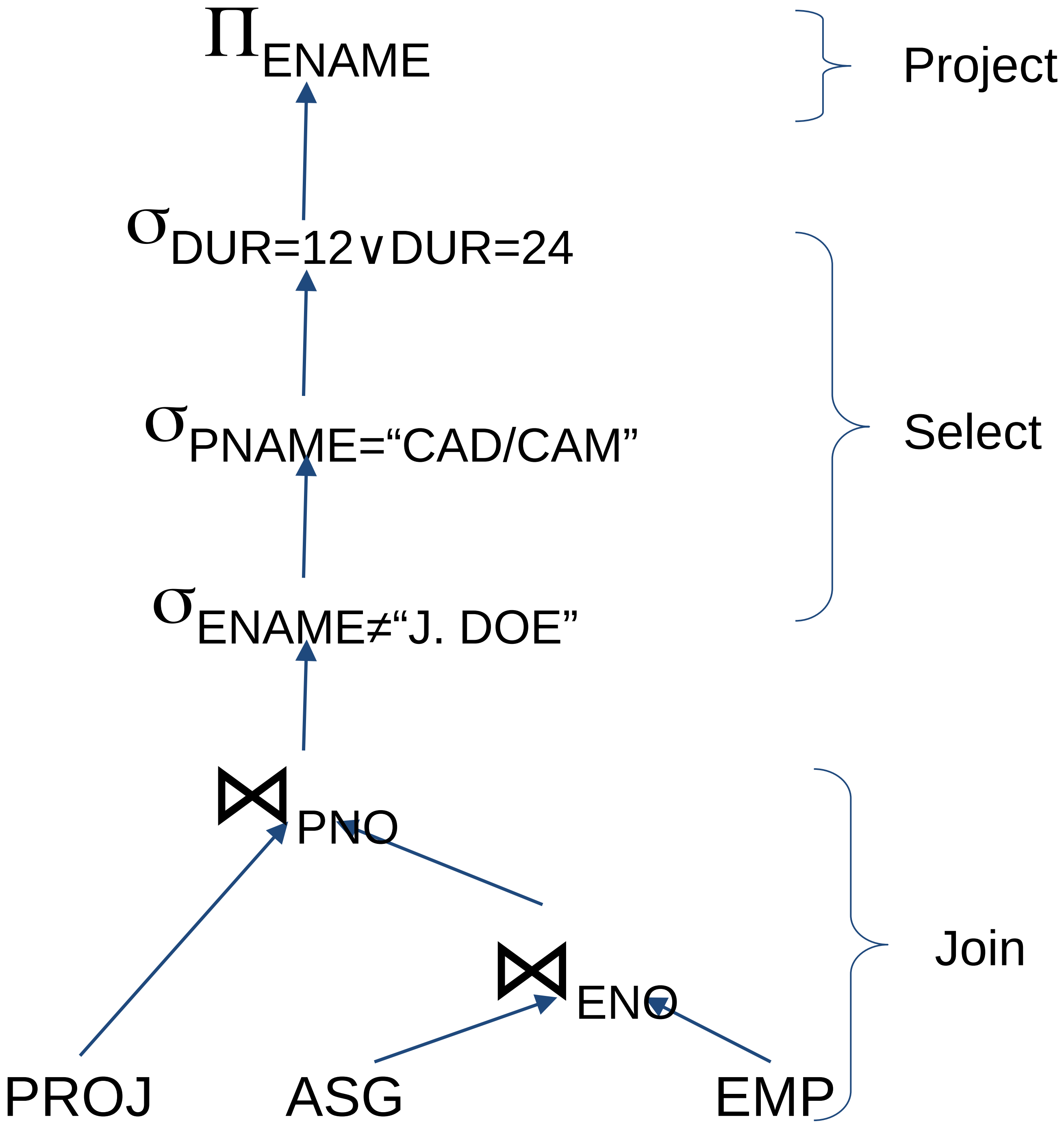
- Determine which fragments are involved
- Localization program
    - substitute for each global query its materialization program
    - optimize

Example

Recall the previous example:

Find the names of employees other than J. Doe who worked on the CAD/CAM project for either one or two years.

```
SELECT ENAME
FROM PROJ, ASG, EMP
WHERE  ASG.ENO=EMP.ENO
AND  ASG.PNO=PROJ.PNO
AND  ENAME ≠ "J. Doe"
AND  PROJ.PNAME="CAD/CAM"
AND  (DUR=12 OR DUR=24)
```



Example

Assume

EMP is fragmented into EMP_1, EMP_2, EMP_3 as follows:

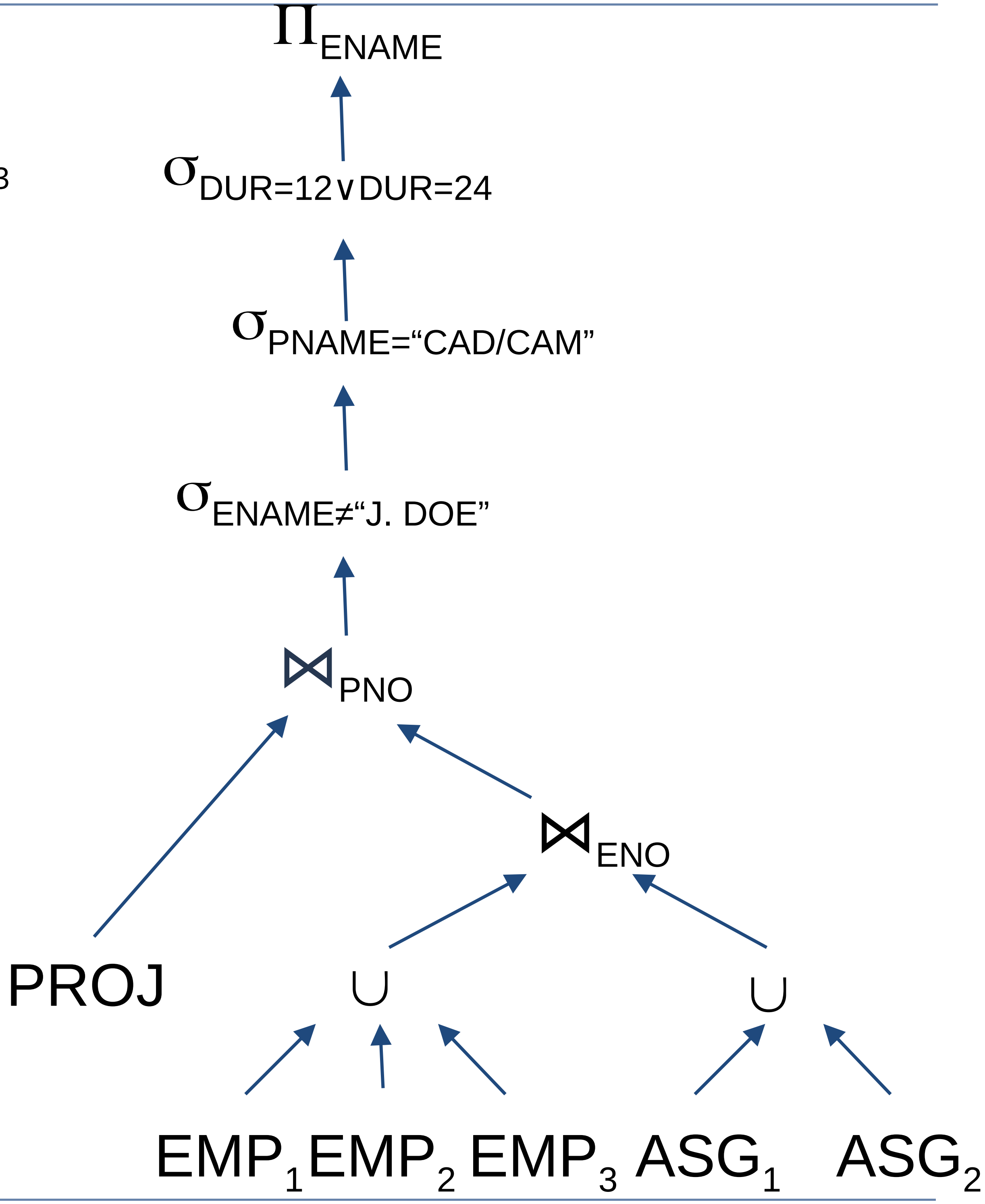
- $EMP_1 = \sigma_{ENO \leq \text{“E3”}}(EMP)$
- $EMP_2 = \sigma_{\text{“E3”} < ENO \leq \text{“E6”}}(EMP)$
- $EMP_3 = \sigma_{ENO > \text{“E6”}}(EMP)$

ASG fragmented into ASG_1 and ASG_2 as follows:

- $ASG_1 = \sigma_{ENO \leq \text{“E3”}}(ASG)$
- $ASG_2 = \sigma_{ENO > \text{“E3”}}(ASG)$

Conditions p_i are defined on the common join key

Replace EMP by ($EMP_1 \cup EMP_2 \cup EMP_3$) and ASG by ($ASG_1 \cup ASG_2$) in any query



Π_{ENAME}

$\sigma_{DUR=12 \vee DUR=24}$

$\sigma_{PNAME=\text{“CAD/CAM”}}$

$\sigma_{ENAME \neq \text{“J. DOE”}}$

$\bowtie_{PNO}$

$\bowtie_{ENO}$

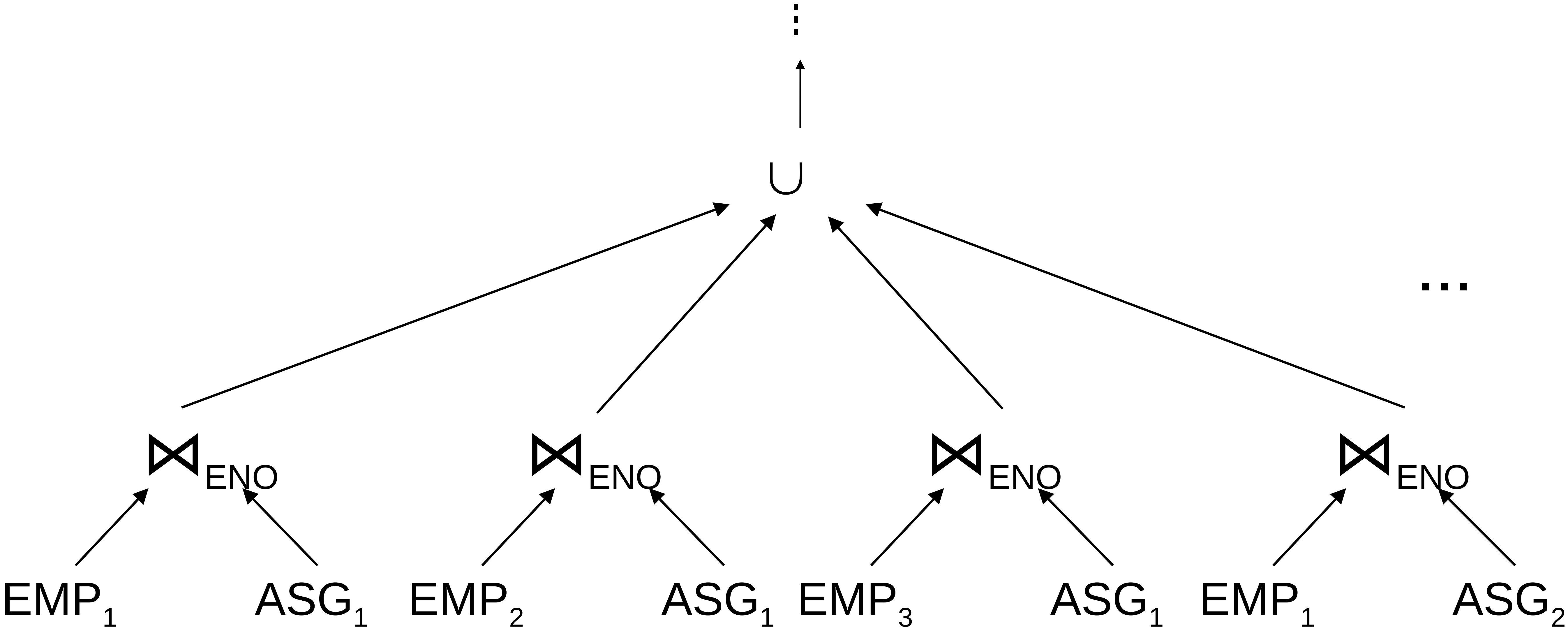
PROJ

$\cup$

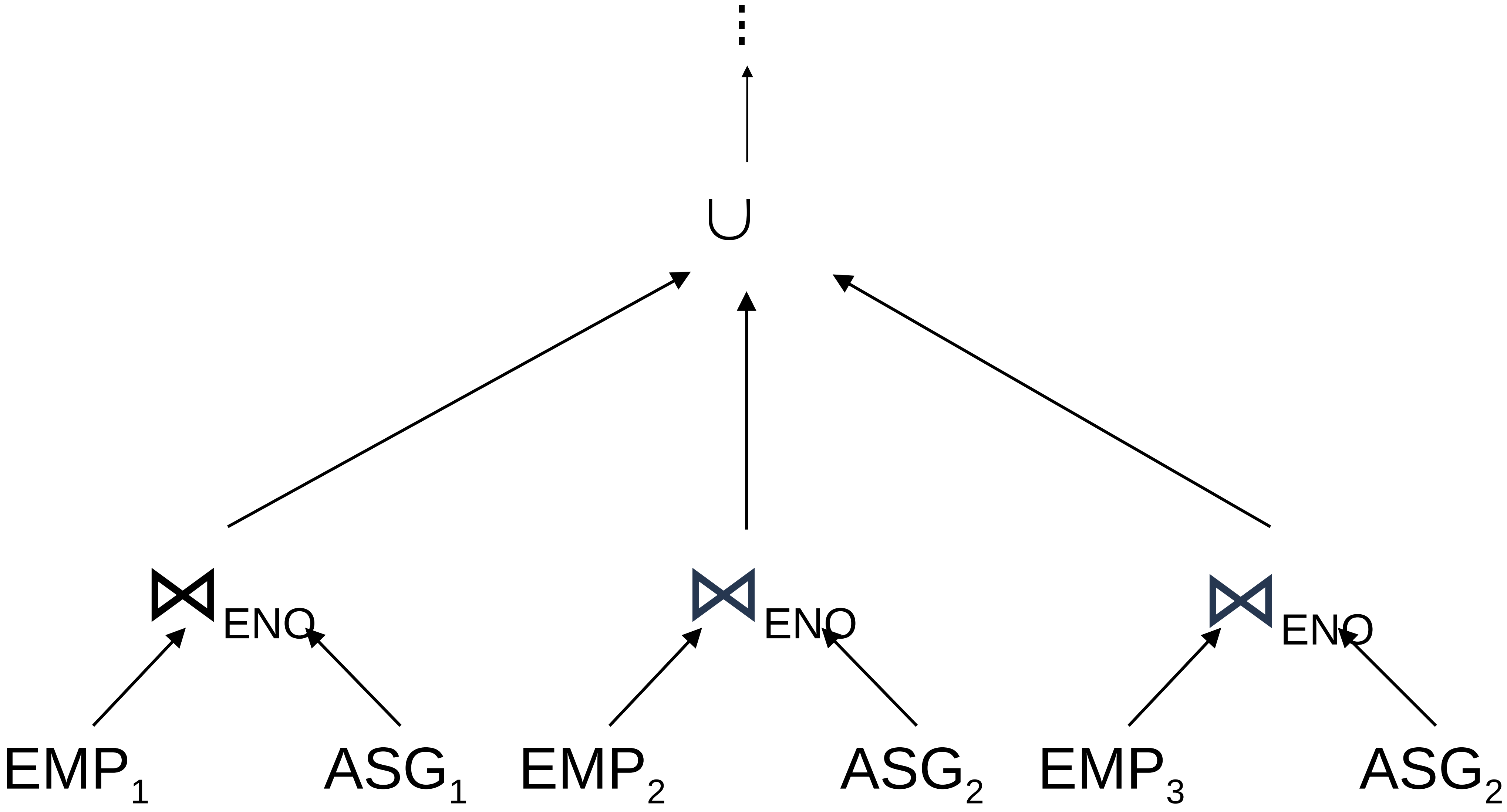
$\cup$

EMP_1 EMP_2 EMP_3 ASG_1 ASG_2

Provides Parallellism



Eliminates Unnecessary Work



$\vdots$

$\cup$

$\bowtie_{ENO}$ $\bowtie_{ENO}$ $\bowtie_{ENO}$

EMP_1 ASG_1 EMP_2 ASG_2 EMP_3 ASG_2

Reduction for PHF

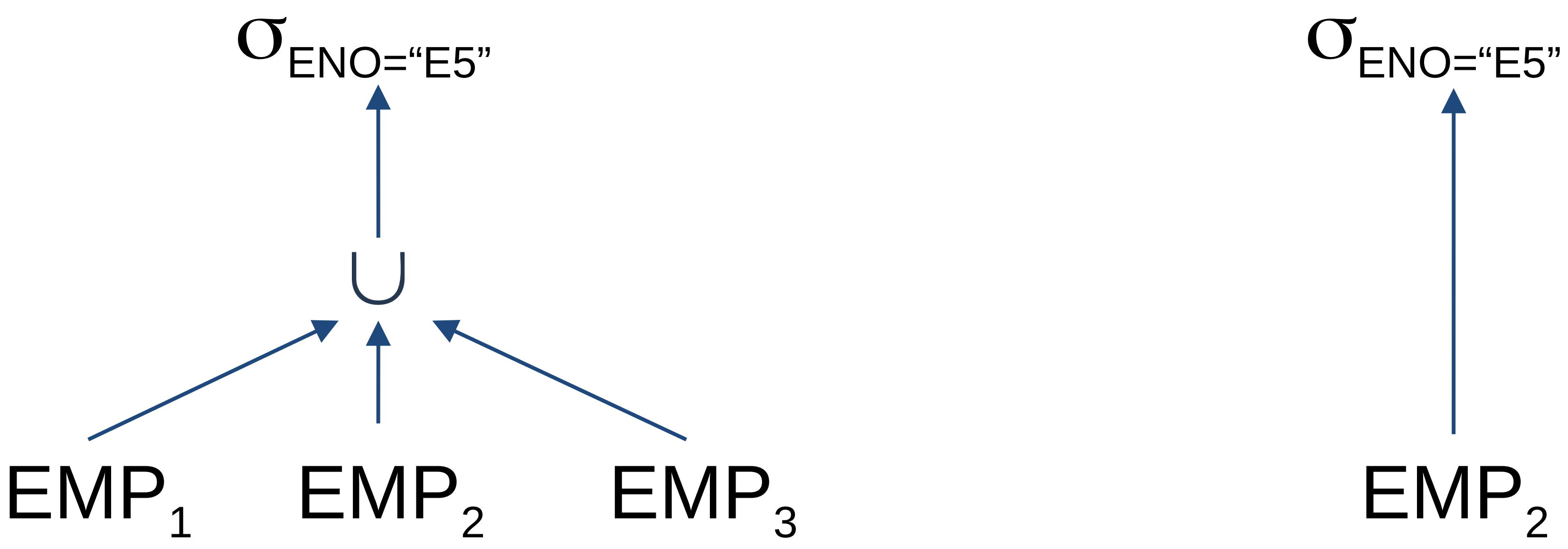
- Reduction with selection

  Relation R and $F_R=\{R_1,\ R_2,\ \ldots,\ R_w\}$ where $R_j=\sigma_{p_j}(R)$

  Rule 1: $\sigma_{p_i}(R_j)=\emptyset$ if $\forall\ x$ in R: $\neg\ (p_i(x) \wedge\ p_j(x))$

  Example

```
SELECT  *
FROM    EMP
WHERE   ENO="E5"
```



$\sigma_{ENO=\text{“E5”}}$

$\cup$

EMP_1 EMP_2 EMP_3

$\sigma_{ENO=\text{“E5”}}$

EMP_2

Reduction for PHF

- Reduction with join
  - Possible if fragmentation is done on join attribute
  - Distribute join over union

$$(R_1 \cup R_2) \bowtie S \Leftrightarrow (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

Reduction for PHF

- Reduction with join

  Possible if fragmentation is done on join attribute

  Distribute join over union

$$(R_1 \cup R_2) \bowtie S \Leftrightarrow (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

Given $R_i = \sigma_{p_i}(R)$ and $R_j = \sigma_{p_j}(R)$

Rule 2: $R_i \bowtie R_j = \emptyset$ if $\forall\ x$ in R_i, $\forall\ y$ in R_j: $\neg (p_i(y) \wedge p_j(x))$

Reduction for PHF

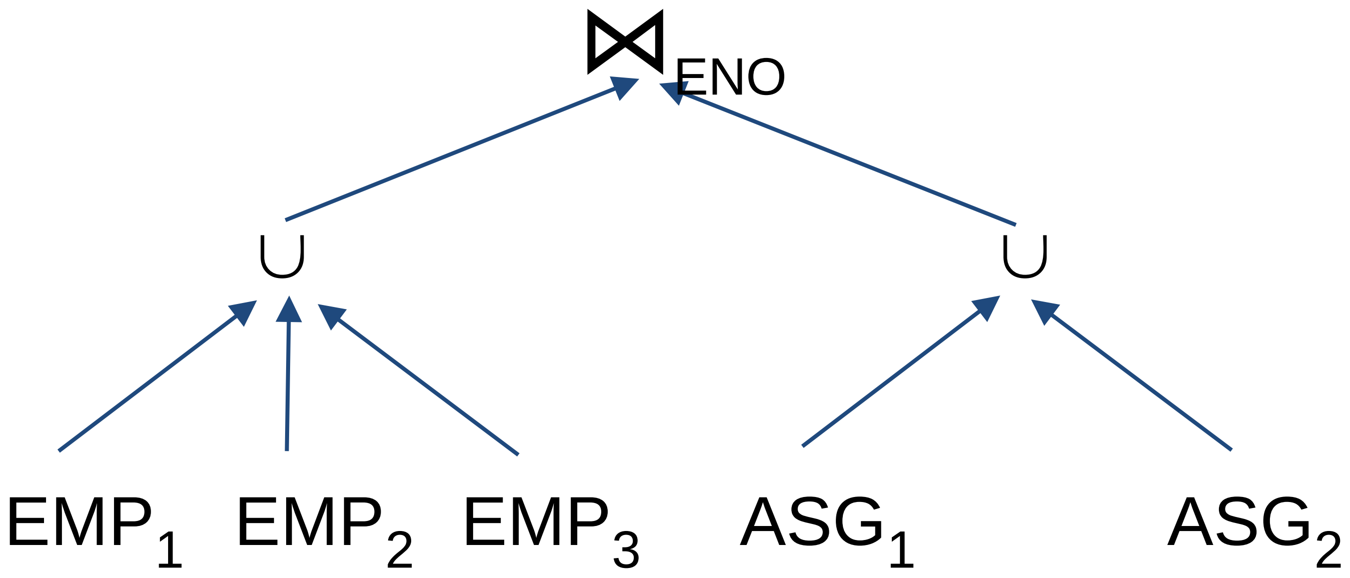
- Assume EMP is fragmented as before and

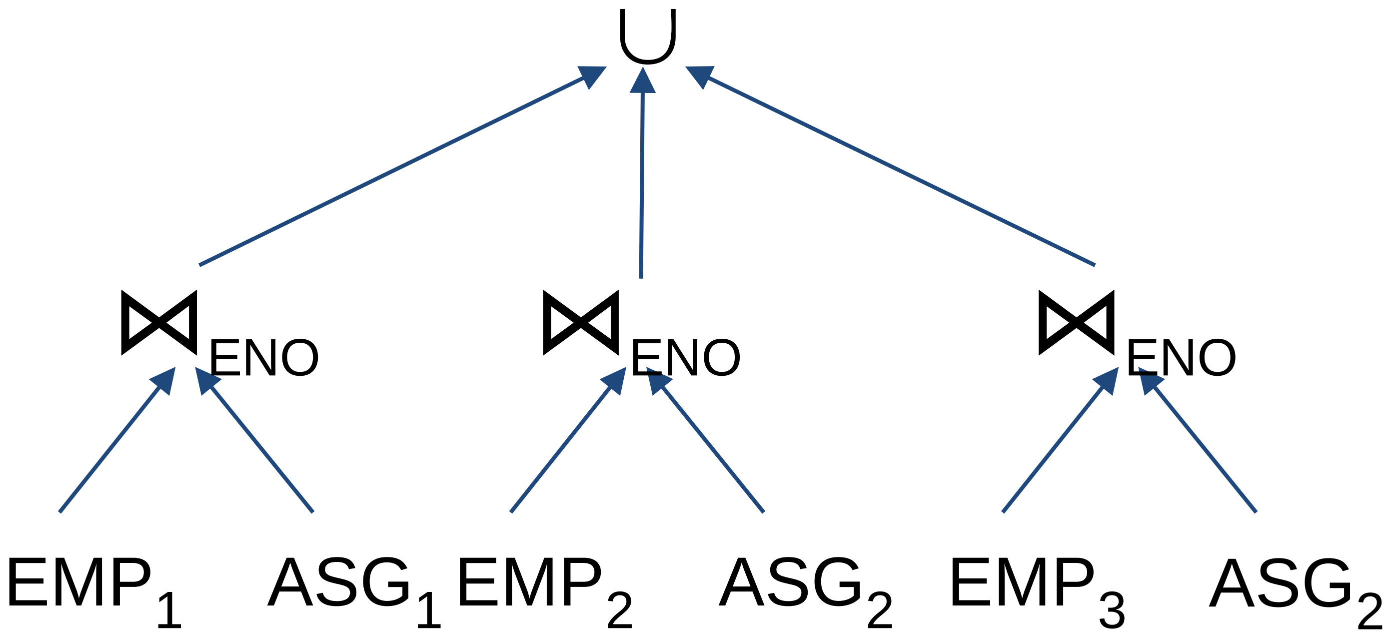
  ASG_1: $\sigma_{ENO \leq "E3"}(ASG)$

  ASG_2: $\sigma_{ENO > "E3"}(ASG)$
- Consider the query

```
SELECT *
FROM   EMP,ASG
WHERE  EMP.ENO=ASG.ENO
```

- Distribute join over unions
- Apply the reduction rule





Reduction for VF

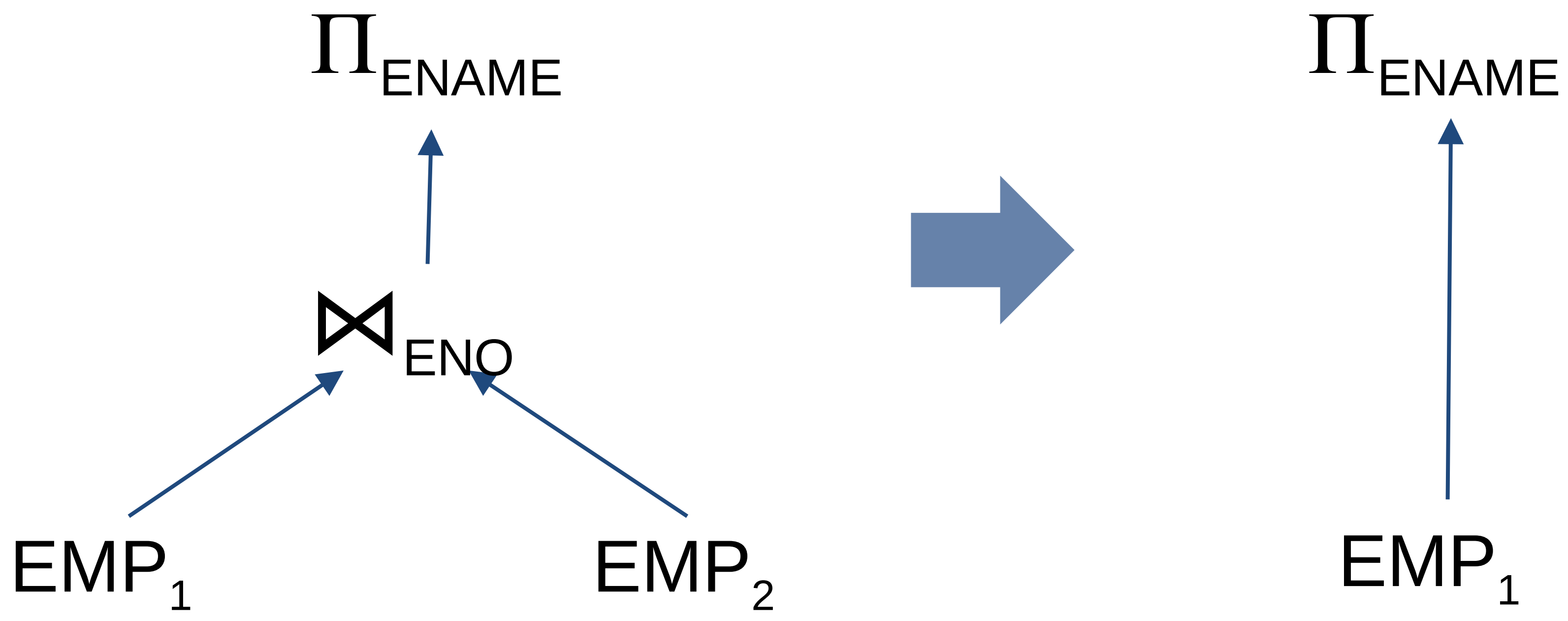
- Find useless (not empty) intermediate relations

Relation R defined over attributes $A = \{A_1, ..., A_n\}$ vertically fragmented as $R_i = \Pi_{A'}(R)$ where $A' \subseteq A$:

$\Pi_{D,K}(R_i)$ is useless if the set of projection attributes D is not in A'

Example: $EMP_1 = \Pi_{ENO,ENAME}(EMP)$; $EMP_2 = \Pi_{ENO,TITLE}(EMP)$

```
SELECT  ENAME
 FROM   EMP
```



Reduction for DHF

- Rule :

  Distribute joins over unions

  Apply the join reduction for horizontal fragmentation
- Example

  $ASG_1: ASG \ltimes_{ENO} EMP_1$

  $ASG_2: ASG \ltimes_{ENO} EMP_2$

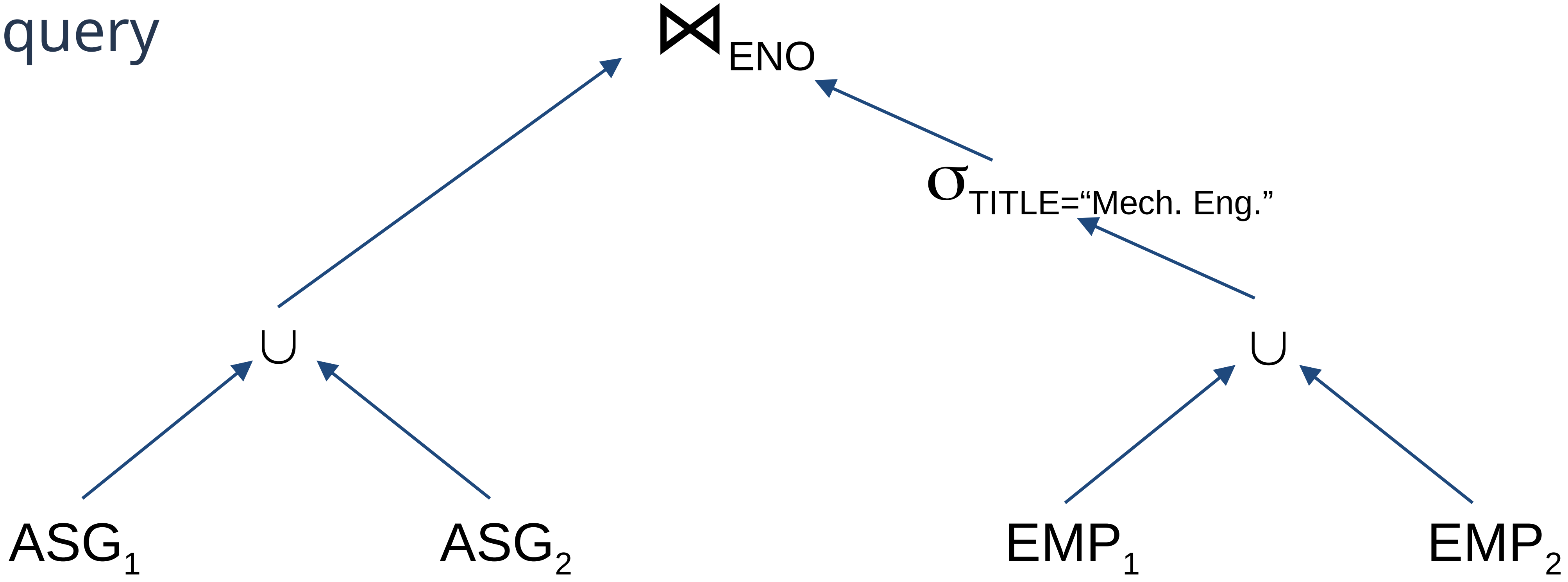
  $EMP_1: \sigma_{TITLE=\text{“Programmer”}}(EMP)$

  $EMP_2: \sigma_{TITLE\neq\text{“Programmer”}}(EMP)$
- Query

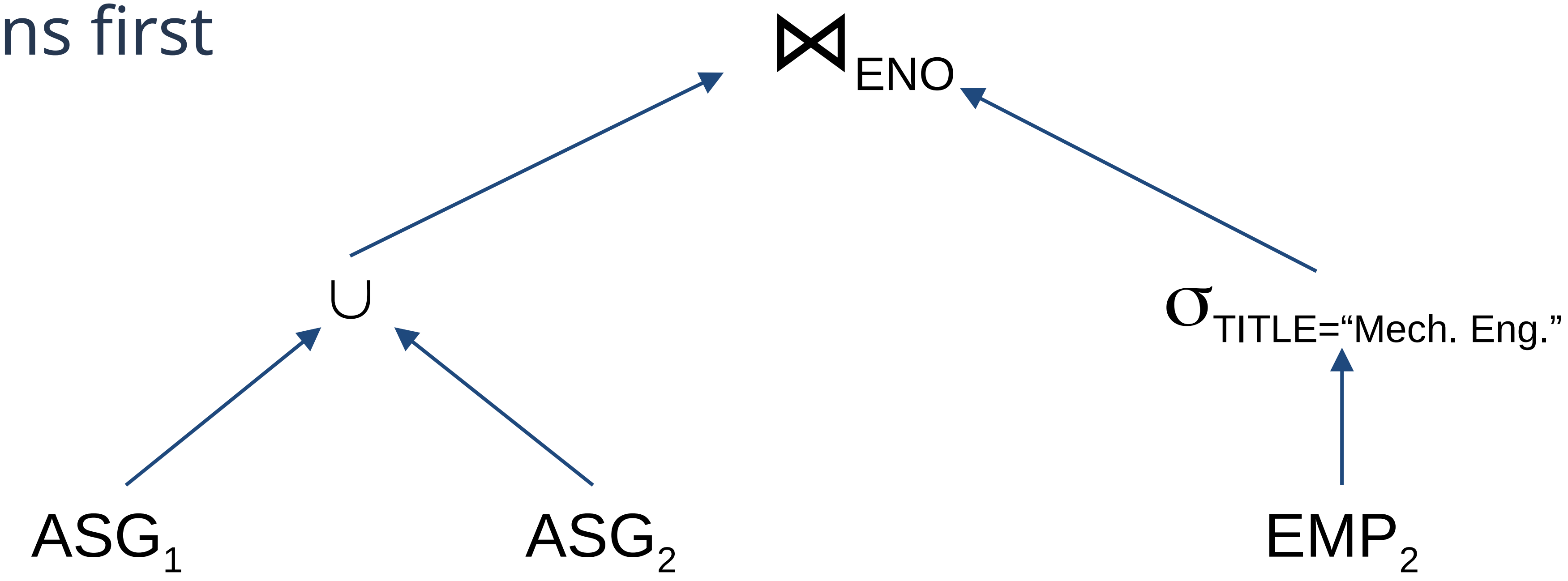
```
SELECT   *
FROM     EMP, ASG
WHERE    ASG.ENO = EMP.ENO
AND      EMP.TITLE = "Mech. Eng."
```

Reduction for DHF

Generic query

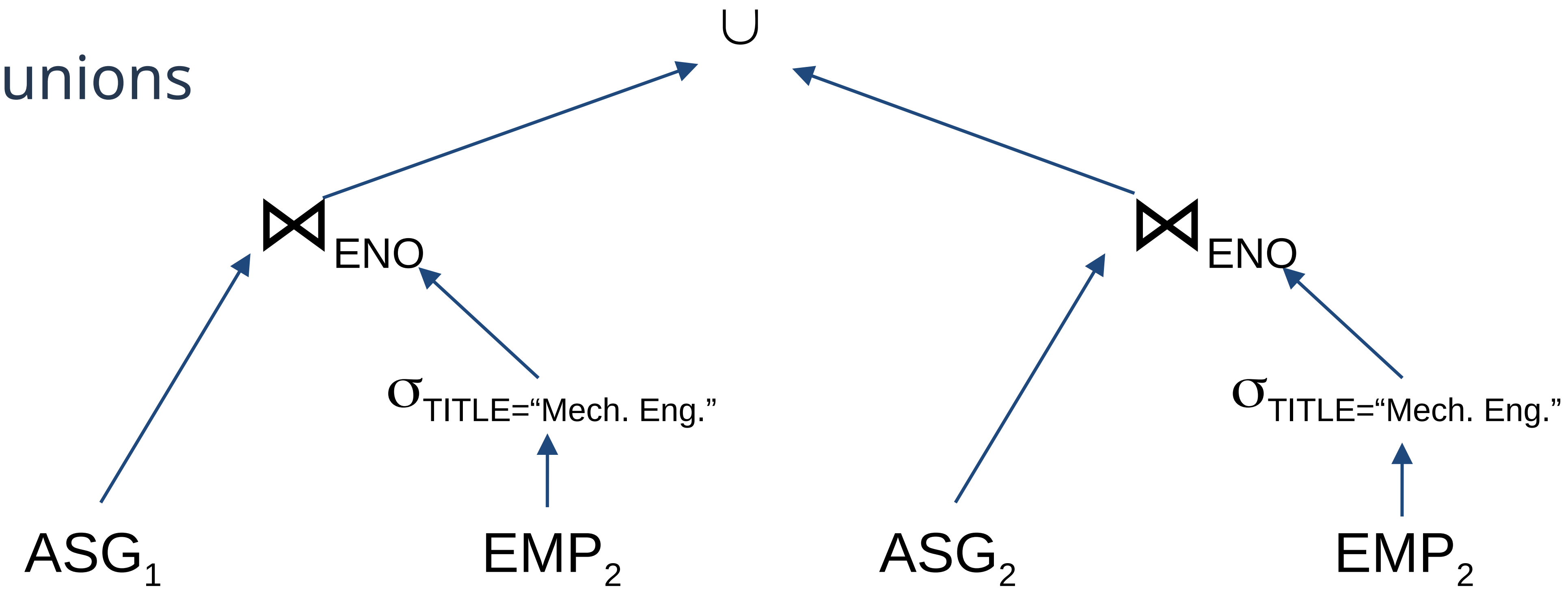


Selections first



Reduction for DHF

Joins over unions



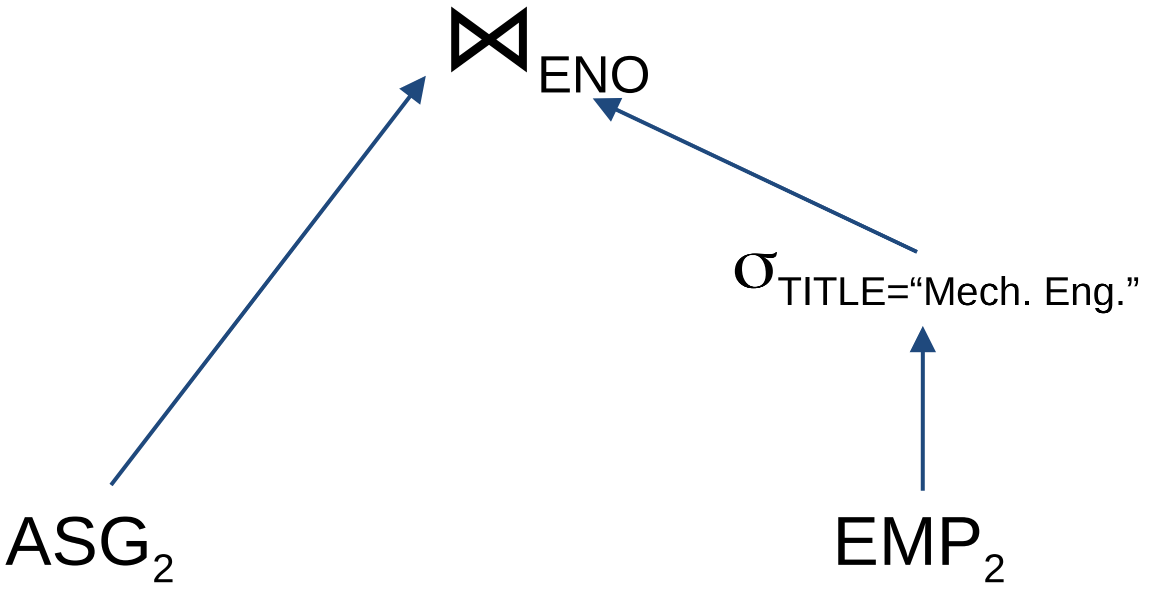
$\cup$

$\bowtie_{ENO}$ $\bowtie_{ENO}$

$\sigma_{TITLE=\text{“Mech. Eng.”}}$ $\sigma_{TITLE=\text{“Mech. Eng.”}}$

ASG$_1$ EMP$_2$ ASG$_2$ EMP$_2$

Elimination of the empty intermediate relations (left sub-tree)



$\bowtie_{ENO}$

$\sigma_{TITLE=\text{“Mech. Eng.”}}$

ASG$_2$ EMP$_2$

Reduction for Hybrid Fragmentation

- Combine the rules already specified:
    - Remove empty relations generated by contradicting selections on horizontal fragments;
    - Remove useless relations generated by projections on vertical fragments;
    - Distribute joins over unions in order to isolate and remove useless joins.

Reduction for HF

Example

Consider the following hybrid fragmentation:

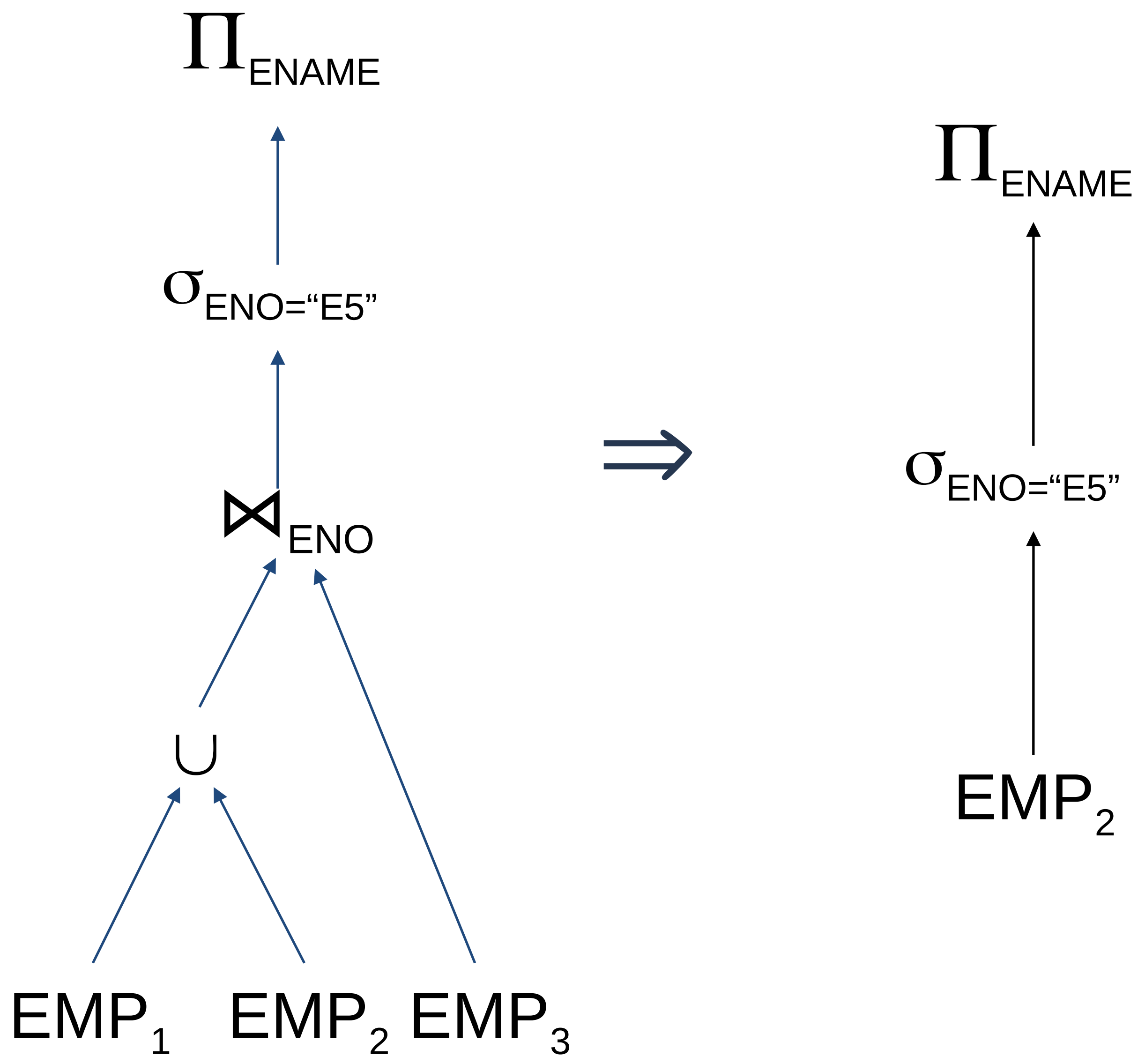
$EMP_1 = \sigma_{ENO \leq "E4"} (\Pi_{ENO,ENAME} (EMP))$

$EMP_2 = \sigma_{ENO > "E4"} (\Pi_{ENO,ENAME} (EMP))$

$EMP_3 = \sigma_{ENO,TITLE} (EMP)$

and the query

```
SELECT   ENAME
FROM EMP
WHERE    ENO="E5"
```



Outline

- Distributed Query Processing
  - Introduction
  - Query Decomposition and Localization
  - Introduction to QO
  - Centralized query optimization
  - Join Ordering
  - Distributed Query Optimization
  - Adaptive Query Processing

Global Query Optimization

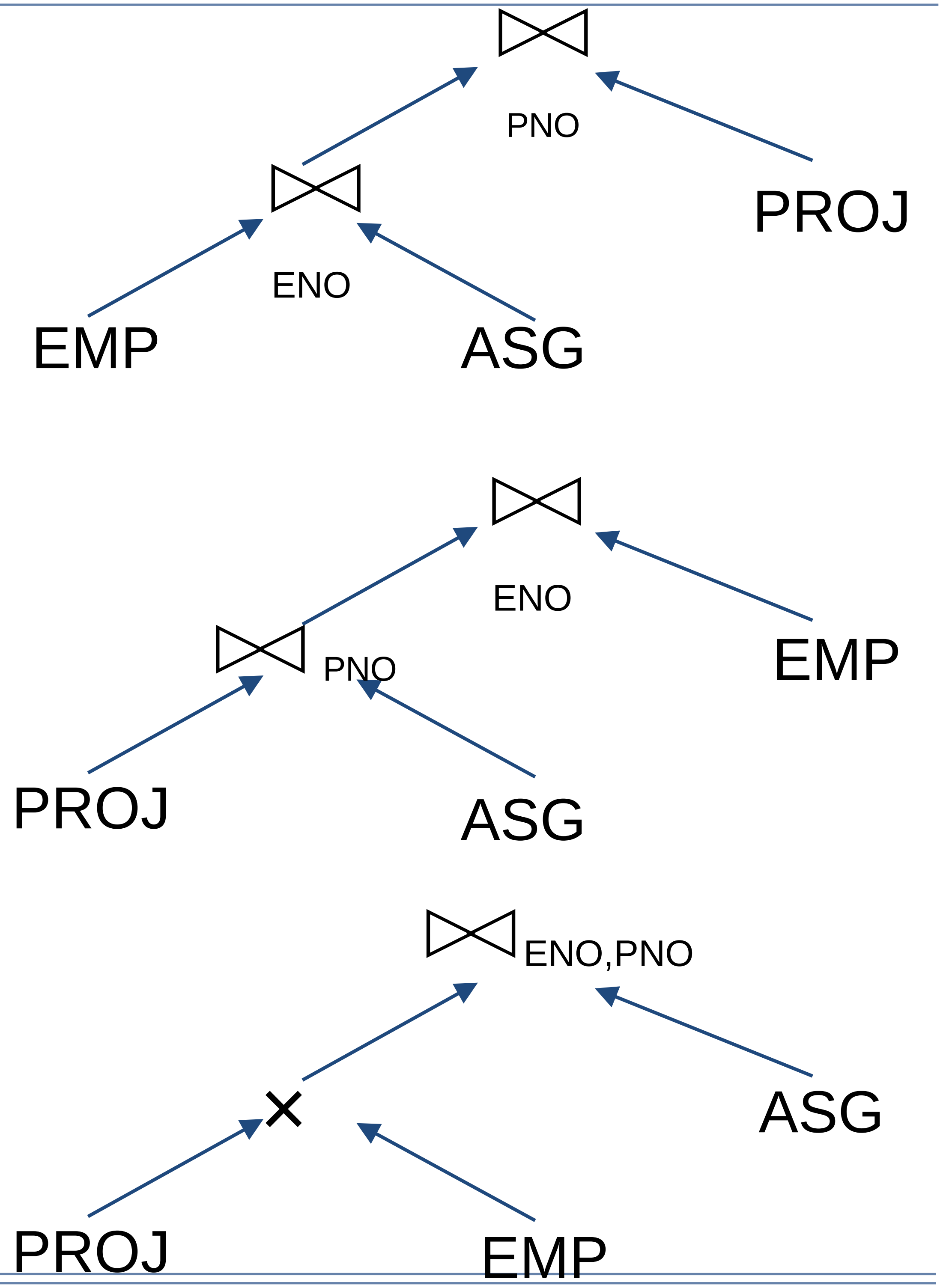
Input: Fragment query

- Find the *best* (not necessarily optimal) global schedule
  - Minimize a cost function
  - Distributed join processing
    - Bushy vs. linear trees
    - Which relation to ship where?
    - Ship-whole vs ship-as-needed
  - Decide on the use of semijoins
    - Semijoin saves on communication at the expense of more local processing.
  - Join methods
    - nested loop vs ordered joins (merge join or hash join)

Search Space

- Search space characterized by alternative execution
- Focus on join trees
- For *N* relations, there are O(*N*!) equivalent join trees that can be obtained by applying commutativity and associativity rules

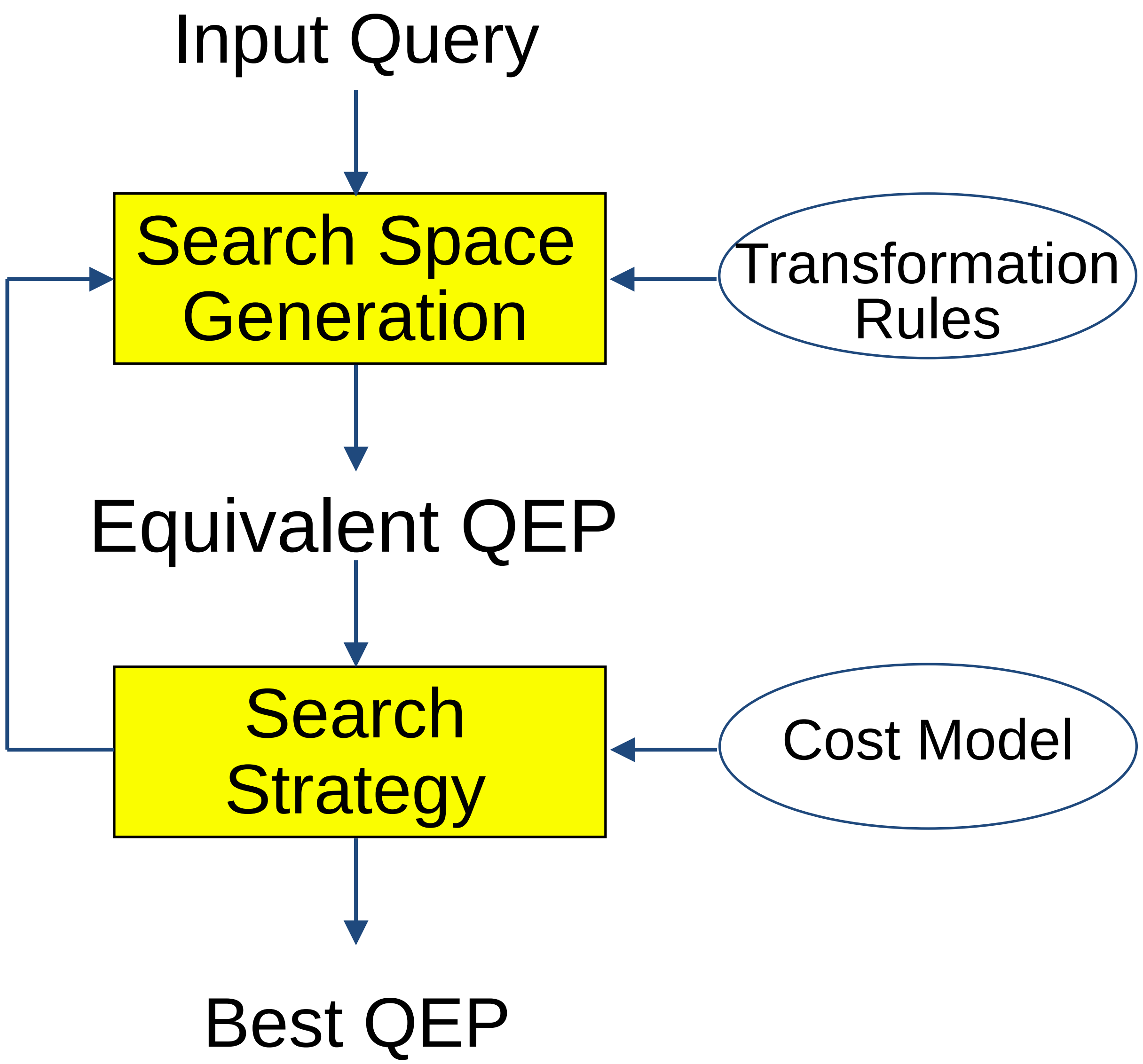
```
SELECT ENAME,RESP
FROM EMP, ASG,PROJ
WHERE  EMP.ENO=ASG.ENO
AND  ASG.PNO=PROJ.PNO
```



Cost-Based Optimization

- Solution space
  - The set of equivalent algebra expressions (query trees).
- Cost function (in terms of time)
  - I/O cost + CPU cost + communication cost
  - These might have different weights in different distributed environments (LAN vs WAN).
  - Can also maximize throughput
- Search algorithm
  - How do we move inside the solution space?
  - Exhaustive search, heuristic algorithms (iterative improvement, simulated annealing, genetic,…)

Query Optimization Process

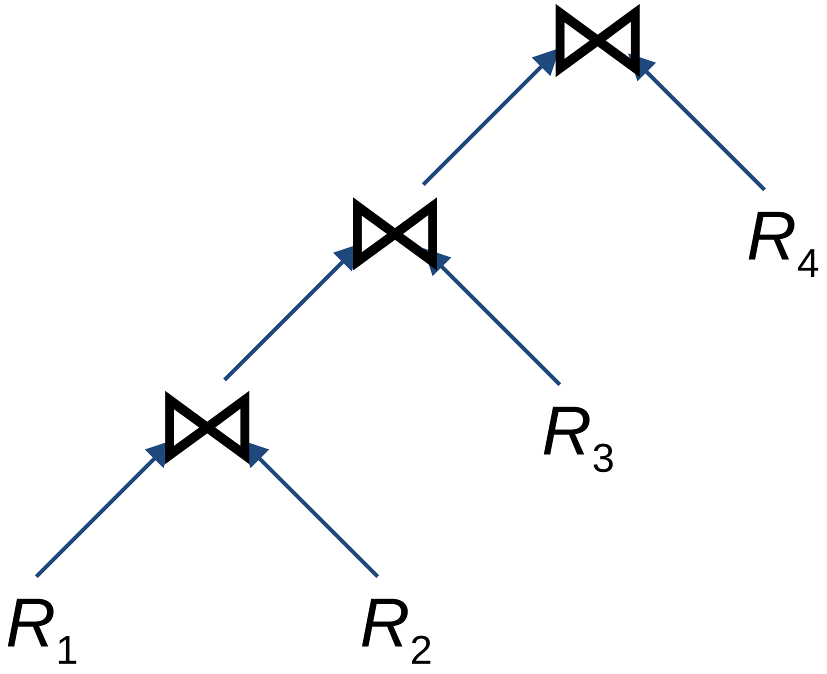


Search Space

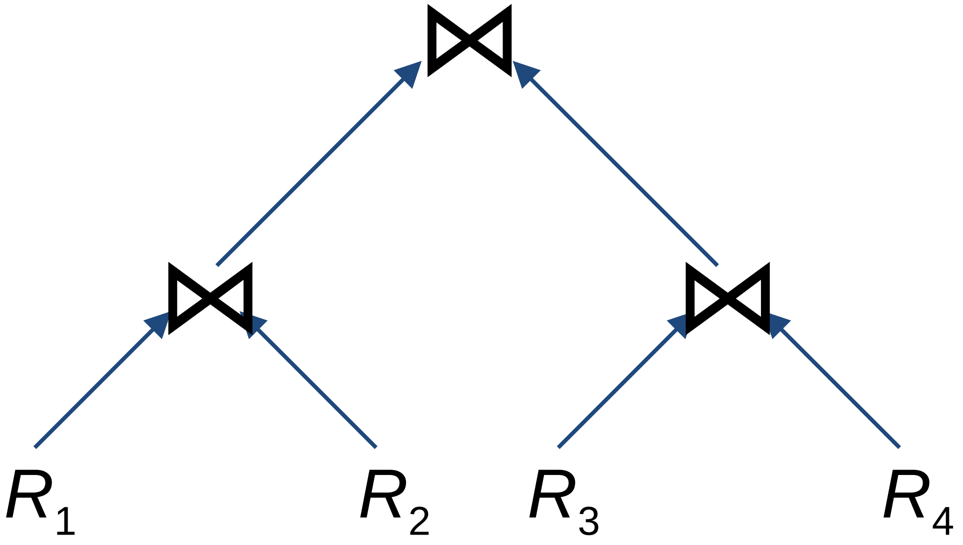
- Restrict by means of heuristics
  - Perform unary operations before binary operations
  - …
- Restrict the shape of the join tree

  Consider only linear trees, ignore bushy ones

Linear Join Tree



Bushy Join Tree

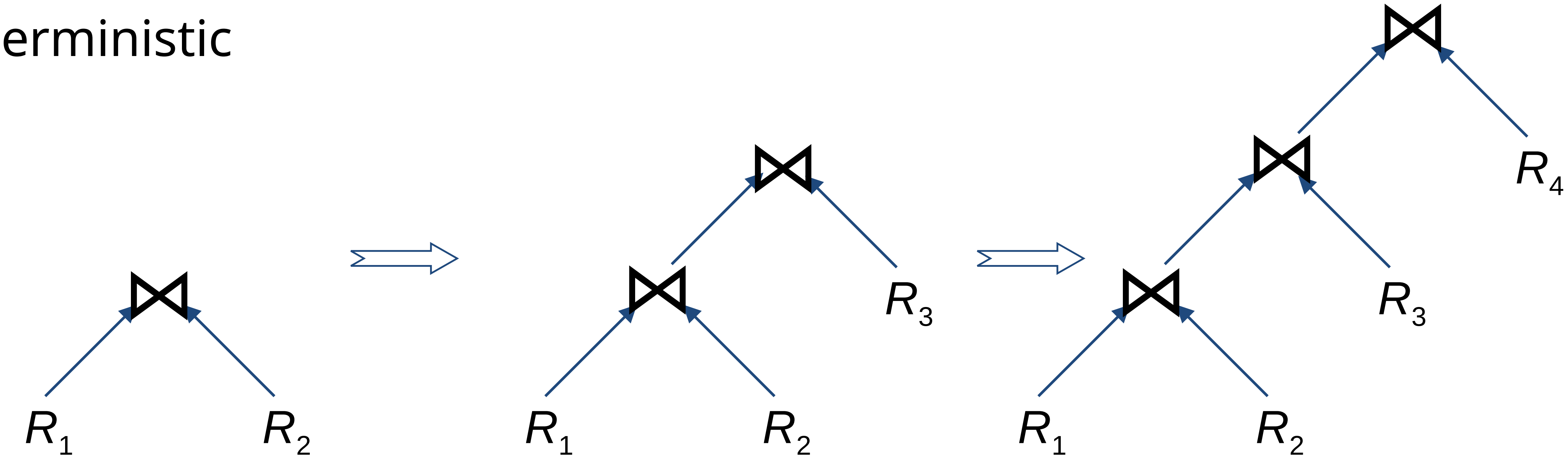


Search Strategy

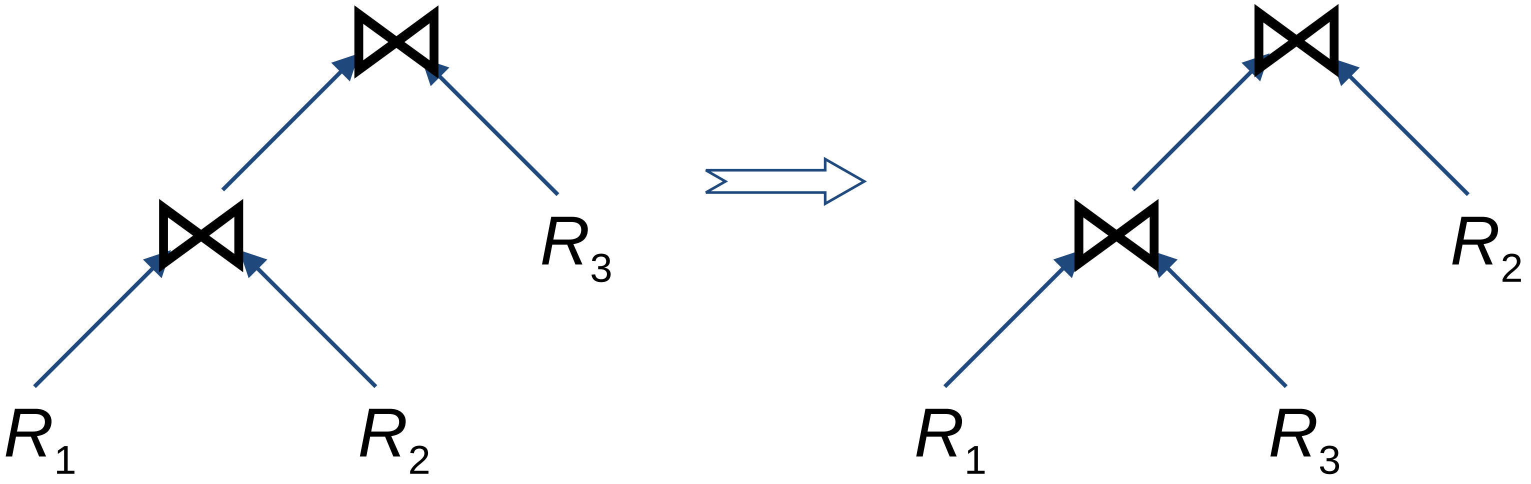
- How to “move” in the search space.
- Deterministic
  - Start from base relations and build plans by adding one relation at each step
  - Dynamic programming: breadth-first
  - Greedy: depth-first
- Randomized
  - Search for optimalities around a particular starting point
  - Trade optimization time for execution time
  - Better when > 10 relations
  - Simulated annealing
  - Iterative improvement

Search Strategies

- Deterministic



- Randomized



Cost Functions

- Total Time (or Total Cost)
  - Reduce each cost (in terms of time) component individually
  - Do as little of each cost component as possible
  - Optimizes the utilization of the resources

Increases system throughput

- Response Time
  - Do as many things as possible in parallel
  - May increase total time because of increased total activity

Total Cost

Summation of all cost factors

Total cost = CPU cost + I/O cost + communication cost

CPU cost = unit instruction cost ✶ no.of instructions
I/O cost = unit disk I/O cost ✶ no. of disk I/Os
communication cost = message initiation + transmission

Total Cost Factors

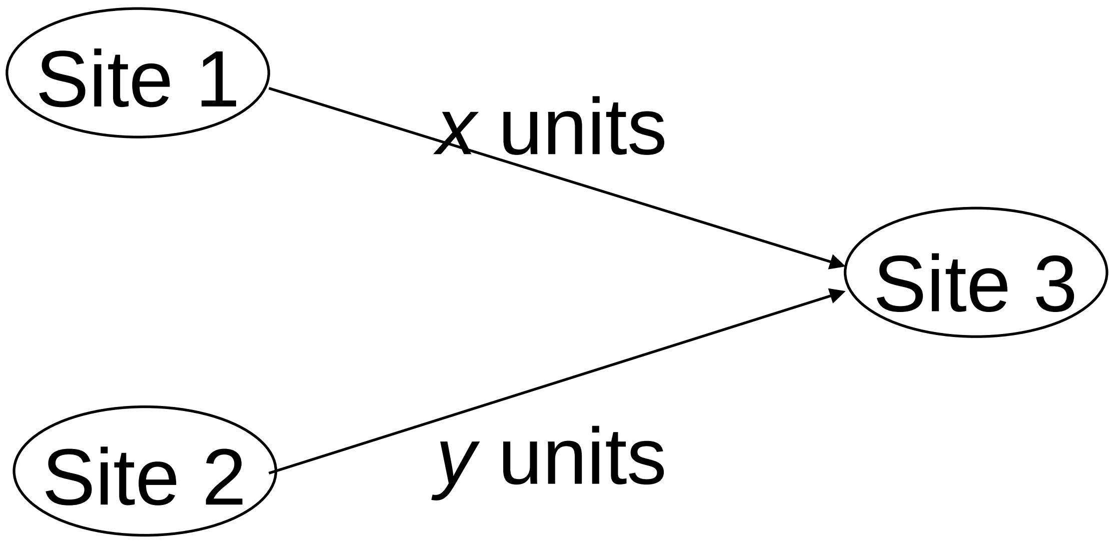
- Wide area network
  - Message initiation and transmission costs high
  - Local processing cost is low (fast mainframes or minicomputers)
  - Ratio of communication to I/O costs = 20:1
- Local area networks
  - Communication and local processing costs are more or less equal
  - Ratio = 1:1.6

Response Time

Elapsed time between the initiation and the completion of a query

Response time = CPU time + I/O time + communication time
CPU time = unit instruction time * no. of sequential instructions
I/O time = unit I/O time * no. of sequential I/Os
communication time = unit msg initiation time * no. of sequential msg
+ unit transmission time * no. of sequential bytes

Example



Assume that only the communication cost is considered
Total time = 2 · message initialization time + unit transmission time * $(x+y)$
Response time = max {time to send x from 1 to 3, time to send y from 2 to 3}
time to send x from 1 to 3 = message initialization time
+ unit transmission time * x
time to send y from 2 to 3 = message initialization time
+ unit transmission time * y

Optimization Statistics

- Primary cost factor: size of intermediate relations
  - Need to estimate their sizes
- Make them precise ⇒ more costly to maintain
- Simplifying assumption: uniform distribution of attribute values in a relation

Statistics

- For each relation $R[A_1, A_2, \ldots, A_n]$ fragmented as $R_1, \ldots, R_r$
    - length of each attribute: $length(A_i)$
    - the number of distinct values for each attribute in each fragment: $card(\Pi_{A_i} R_j)$
    - maximum and minimum values in the domain of each attribute: $min(A_i)$, $max(A_i)$
    - the cardinalities of each domain: $card(dom[A_i])$
- The cardinalities of each fragment: $card(R_j)$
- Selectivity factor of each operation for relations
    - For joins

$$SF_{\bowtie A=B}(R,S) = \frac{card(R \bowtie_{A=B} S)}{card(R) * card(S)}$$

$$= \frac{1}{max(card(\Pi_A R), card(\Pi_B S))}$$

Intermediate Relation Sizes

Selection

$size(R) = card(R) \cdot length(R)$

$card(\sigma_F(R)) = SF_\sigma(F) \cdot card(R)$

where

$$S\,F_\sigma(A = value) = \frac{1}{card(\prod_A(R))}$$

$$S\,F_\sigma(A > value) = \frac{max(A) - value}{max(A) - min(A)}$$

$$S\,F_\sigma(A < value) = \frac{value - max(A)}{max(A) - min(A)}$$

$SF_\sigma(p(A_i) \wedge p(A_j)) = SF_\sigma(p(A_i)) * SF_\sigma(p(A_j))$

$SF_\sigma(p(A_i) \vee p(A_j)) = SF_\sigma(p(A_i)) + SF_\sigma(p(A_j)) - (SF_\sigma(p(A_i)) * SF_\sigma(p(A_j)))$

$SF_\sigma(A \in \{value\}) = SF_\sigma(A = value) * card(\{values\})$

Intermediate Relation Sizes

Projection

$card(\Pi_A(R))=card(R)$

Cartesian Product

$card(R \cdot S) = card(R) * card(S)$

Union

upper bound: $card(R \cup S) = card(R) + card(S)$

lower bound: $card(R \cup S) = max\{card(R), card(S)\}$

Set Difference

upper bound: $card(R–S) = card(R)$

lower bound: 0

Intermediate Relation Size

Join

Special case: A is a key of R and B is a foreign key of S

$$card(R \bowtie_{A=B} S) = card(S)$$

More general:

$$card(R \bowtie S) = SF_{\bowtie} * card(R) \cdot card(S)$$

Semijoin

$$card(R \ltimes_{A} S) = SF_{\ltimes}(S.A) * card(R)$$

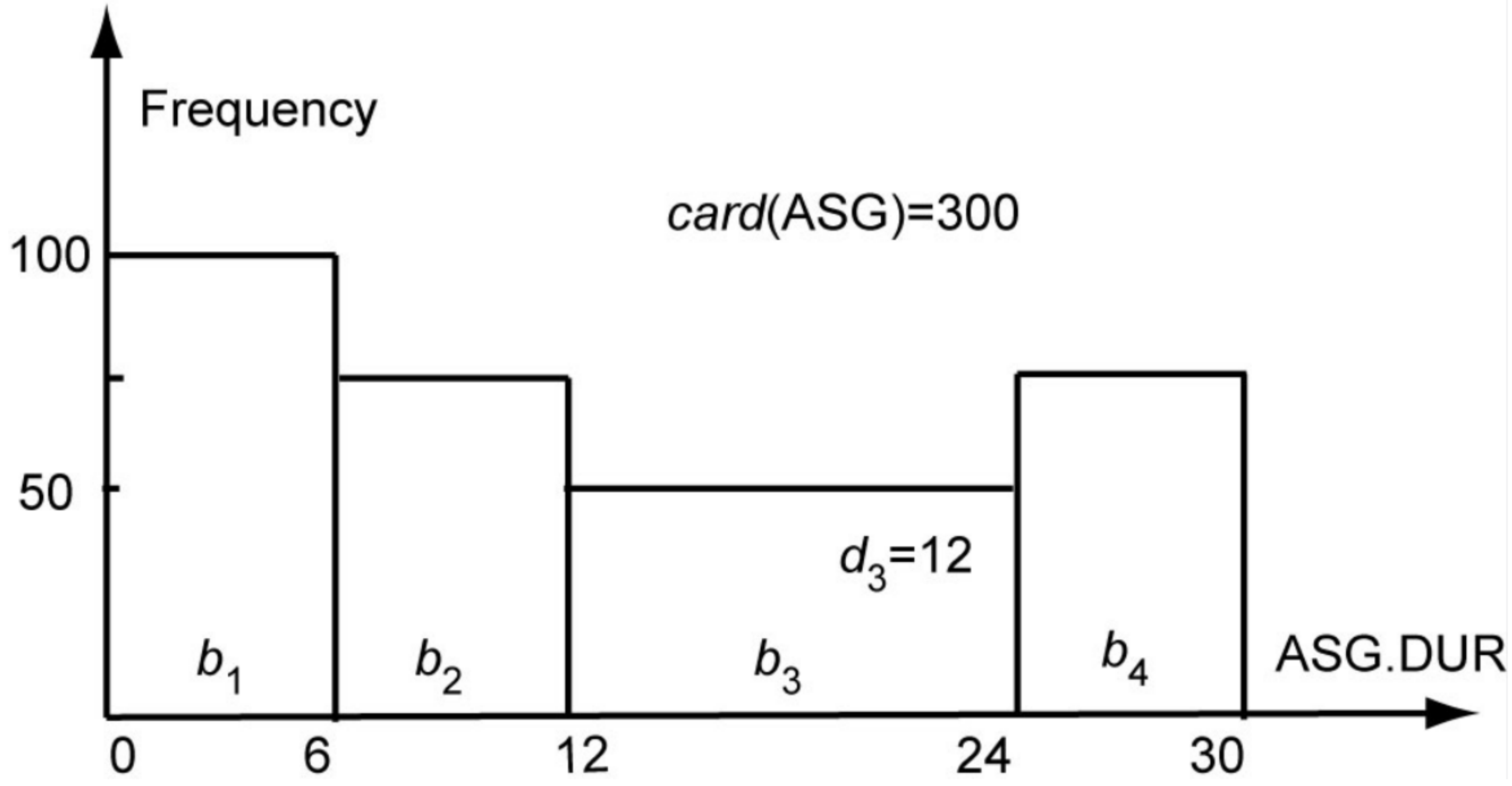
where

$$SF_{\ltimes}(R \ltimes_{A} S) = SF_{\ltimes}(S.A) = \frac{card(\prod_{A}(S))}{card(dom[A])}$$

Histograms for Selectivity Estimation

- For skewed data, the uniform distribution assumption of attribute values yields inaccurate estimations
- Use an histogram for each skewed attribute A
  - Histogram = set of buckets
    - Each bucket describes a range of values of A, with its average frequency *f* (number of tuples with A in that range) and number of distinct values *d*
    - Buckets can be adjusted to different ranges
- Examples
  - Equality predicate
    - With (value in Range$_i$), we have: $SF_\sigma(A = value) = 1/d_i$
  - Range predicate
    - Requires identifying relevant buckets and summing up their frequencies

Histogram Example



For ASG.DUR=18: we have SF=1/12 so the card of selection is 50/12 = 5 tuples

For ASG.DUR≤18: we have min($range_3$)=12 and max($range_3$)=24 so the card. of selection is 100+75+(((18−12)/(24 − 12))*50) = 200 tup.

Outline

- Distributed Query Processing
  - ❑ Introduction
  - ❑ Query Decomposition and Localization
  - ❑ Introduction to QO
  - ❑ Centralized query optimization
  - ❑ Join Ordering
  - ❑ Distributed Query Optimization
  - ❑ Adaptive Query Processing

Centralized Query Optimization

- Dynamic (Ingres project at UCB)
  - Interpretive
- Static (System R project at IBM)
  - Exhaustive search
- Hybrid (Volcano project at OGI)
  - Choose node within plan

Dynamic Algorithm

❶ Decompose each multi-variable query into a sequence of mono-variable queries with a common variable

❷ Process each by a one variable query processor
  - Choose an initial execution plan (heuristics)
  - Order the rest by considering intermediate relation sizes

⬇

No statistical information is maintained

Dynamic Algorithm–Decomposition

- Replace an *n* variable query *q* by a series of queries

  $$q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n$$

  where q_i uses the result of q_{i-1}.
- Detachment
  - Query *q* decomposed into $q' \rightarrow q''$ where q' and q'' have a common variable which is the result of q'
- Tuple substitution
  - Replace the value of each tuple with actual values and simplify the query

$$q(V_1, V_2, \ldots V_n) \rightarrow (q'(t_1, V_2, V_2, \ldots, V_n), t_1 \in R)$$

Detachment

q: **SELECT** $V_2.A_2, V_3.A_3, \ldots, V_n.A_n$
FROM $R_1\ V_1, \ldots, R_n\ V_n$
WHERE $P_1(V_1.A_1')$ **AND** $P_2(V_1.A_1, V_2.A_2, \ldots, V_n.A_n)$

$\Downarrow$

q': **SELECT** $V_1.A_1$ **INTO** R_1'
FROM $R_1\ V_1$
WHERE $P_1(V_1.A_1)$

q'': **SELECT** $V_2.A_2, \ldots, V_n.A_n$
FROM $R_1'\ V_1, R_2\ V_2, \ldots, R_n\ V_n$
WHERE $P_2(V_1.A_1, V_2.A_2, \ldots, V_n.A_n)$

Detachment Example

Names of employees working on CAD/CAM project

q_1:

```
SELECT EMP.ENAME
FROM EMP, ASG, PROJ
WHERE  EMP.ENO=ASG.ENO
AND  ASG.PNO=PROJ.PNO
AND  PROJ.PNAME="CAD/CAM"
```

$\Downarrow$

q_{11}:

```
SELECT PROJ.PNO INTO JVAR
FROM PROJ
WHERE  PROJ.PNAME="CAD/CAM"
```

q':

```
SELECT EMP.ENAME
FROM EMP,ASG,JVAR
WHERE  EMP.ENO=ASG.ENO
AND  ASG.PNO=JVAR.PNO
```

Detachment Example (cont’d)

q':

```
SELECT EMP.ENAME
FROM EMP,ASG,JVAR
WHERE  EMP.ENO=ASG.ENO
AND  ASG.PNO=JVAR.PNO
```

$\Downarrow$

q_{12}:

```
SELECT ASG.ENO INTO GVAR
FROM ASG,JVAR
WHERE  ASG.PNO=JVAR.PNO
```

q_{13}:

```
SELECT EMP.ENAME
FROM EMP,GVAR
WHERE  EMP.ENO=GVAR.ENO
```

Tuple Substitution

q_{11} is a mono-variable query

q_{12} and q_{13} is subject to tuple substitution

Assume GVAR has two tuples only: E1 and E2

Then q_{13} becomes

q_{131}:
```
SELECT EMP.ENAME
FROM EMP
WHERE  EMP.ENO="E1"
```

q_{132}:
```
SELECT EMP.ENAME
FROM EMP
WHERE  EMP.ENO="E2"
```

Static Algorithm

➊ Simple (i.e., mono-relation) queries are executed according to the best access path

➋ Execute joins

  - Determine the possible ordering of joins
  - Determine the cost of each ordering
  - Choose the join ordering with minimal cost

Static Algorithm

For joins, two alternative algorithms :

- Nested loops

```
for each tuple of *external* relation (cardinality $n_1$)
    for each tuple of *internal* relation (cardinality $n_2$)
        join two tuples if the join predicate is true
    end
End
```

  - Complexity: $n_1 * n_2$

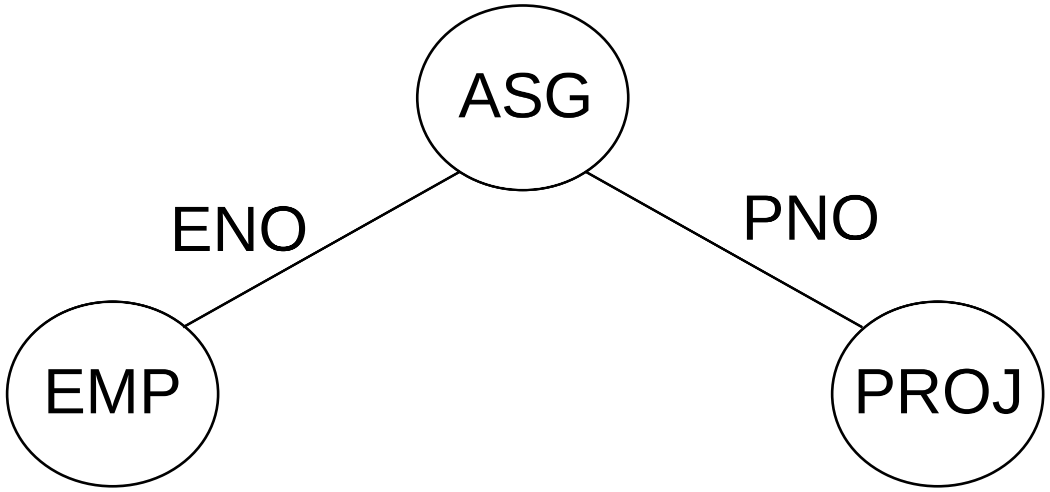
- Merge join
  - sort relations
  - merge relations
  - Complexity: $n_1 + n_2$ if relations are previously sorted and equijoin

Static Algorithm – Example

Names of employees working on the CAD/CAM project

Assume

EMP has an index on ENO,
ASG has an index on PNO,
PROJ has an index on PNO and an index on PNAME



q_1:
```
SELECT EMP.ENAME
FROM   EMP, ASG, PROJ
WHERE  EMP.ENO=ASG.ENO
AND    ASG.PNO=PROJ.PNO
AND    PNAME="CAD/CAM"
```

Example (cont'd)

❶ Choose the best access paths to each relation

EMP: sequential scan (no selection on EMP)
ASG: sequential scan (no selection on ASG)
PROJ: index on PNAME (there is a selection on PROJ based on PNAME)

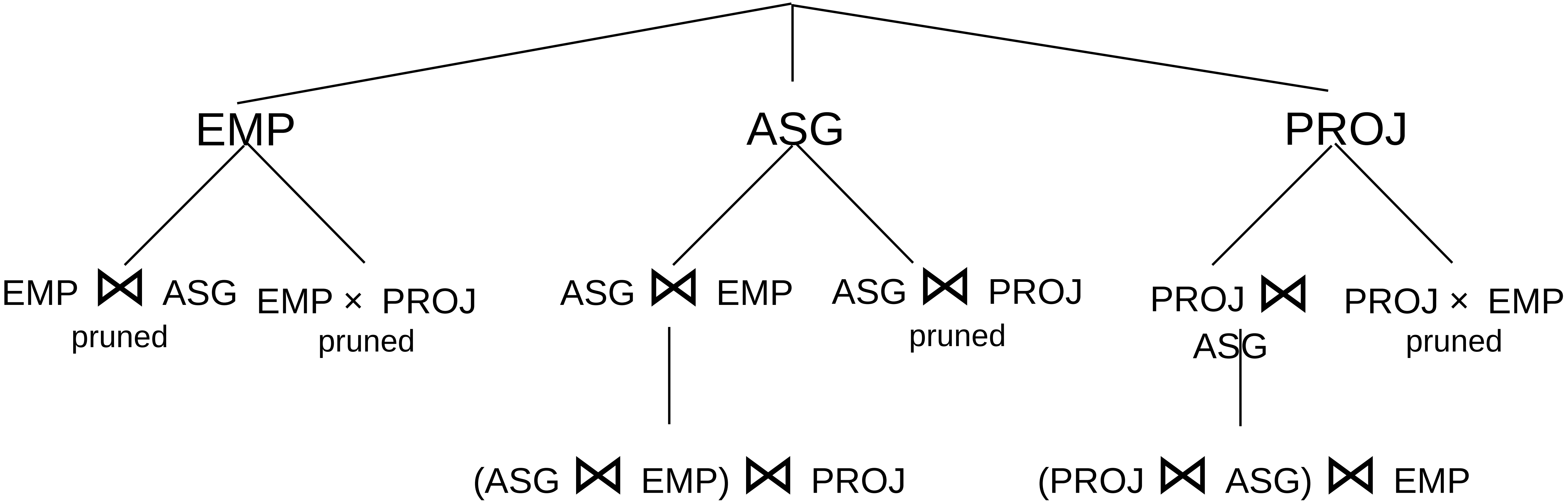
❷ Determine the best join ordering

EMP ⋈ ASG ⋈ PROJ
ASG ⋈PROJ ⋈ EMP
PROJ ⋈ASG ⋈ EMP
ASG ⋈EMP ⋈ PROJ
EMP × PROJ ⋈ ASG
PRO × JEMP ⋈ASG

Select the best ordering based on the join costs evaluated according to the two methods

Static Algorithm

Alternatives



EMP

ASG

PROJ

EMP $\bowtie$ ASG
pruned

EMP × PROJ
pruned

ASG $\bowtie$ EMP

ASG $\bowtie$ PROJ
pruned

PROJ $\bowtie$ ASG

PROJ × EMP
pruned

(ASG $\bowtie$ EMP) $\bowtie$ PROJ

(PROJ $\bowtie$ ASG) $\bowtie$ EMP

Best total join order is one of

((ASG $\bowtie$ EMP) $\bowtie$ PROJ)

((PROJ $\bowtie$ ASG) $\bowtie$ EMP)

Static Algorithm

- ((PROJ ⋈ ASG) ⋈ EMP) has a useful index on the select attribute and direct access to the join attributes of ASG and EMP
- Therefore, chose it with the following access methods:

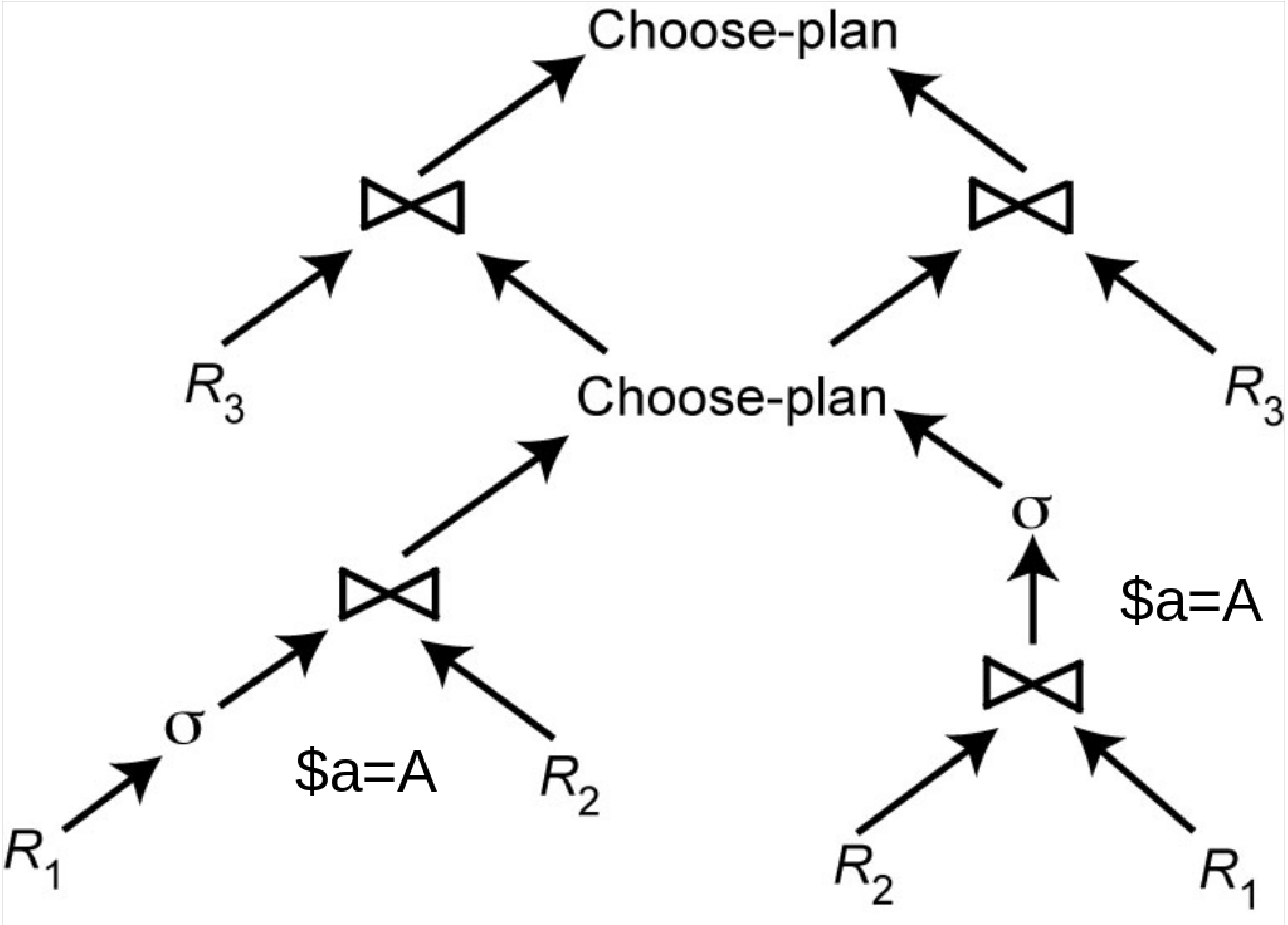
  select PROJ using index on PNAME
  then join with ASG using index on PNO
  then join with EMP using index on ENO

Hybrid optimization

- In general, static optimization is more efficient than dynamic optimization
  - Adopted by all commercial DBMS
- But even with a sophisticated cost model (with histograms), accurate cost prediction is difficult
- Example
  - Consider a parametric query with predicate

    WHERE R.A = \$a   /* \$a is a parameter
  - The only possible assumption at compile time is uniform distribution of values
- Solution: Hybrid optimization
  - Choose-plan done at runtime, based on the actual parameter binding

Hybrid Optimization Example



Outline

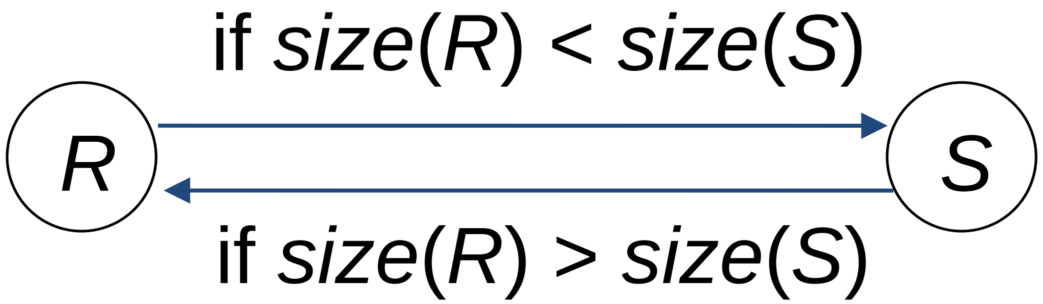
- Distributed Query Processing
  - Introduction
  - Query Decomposition and Localization
  - Centralized query optimization
  - Join Ordering
  - Distributed Query Optimization
  - Adaptive Query Processing

Join Ordering in Fragment Queries

- Ordering joins
  - Distributed INGRES
  - System R*
  - Two-step
- Semijoin ordering
  - SDD-1

Join Ordering

- Assumptions: query expressed on fragments (relations); ignore local processing (reducers); consider only join operands at diff. sites; set-at-a-time not tuple-at-a-time; does not count transfer of result.
- Consider two relations only: PROJ ⋈ ASG

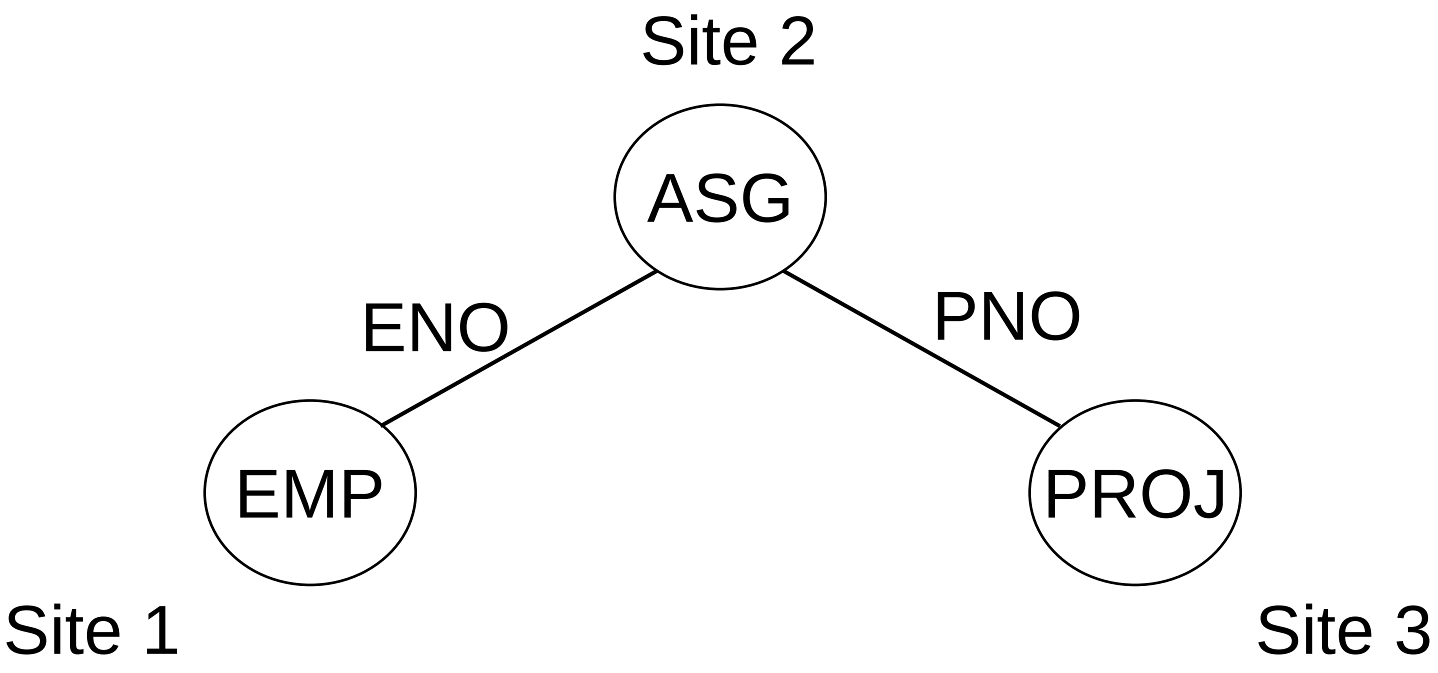


- Obvious choice: copy smaller relation to the site of larger
- Multiple relations more difficult because of too many alternatives.
  - As in the case of two; selecting smaller input argument to obtain small intermediate size of result; estimation of the join result necessary (difficult); join can increase the size of result
- Solution: use heuristics, i.e., only the communication cost

Join Ordering – Example

Consider

PROJ $\bowtie_{\text{PNO}}$ ASG $\bowtie_{\text{ENO}}$ EMP



Join Ordering – Example

Execution alternatives:

1. EMP → Site 2

   Site 2 computes EMP'=EMP $\bowtie$ ASG
   EMP' → Site 3
   Site 3 computes EMP' $\bowtie$ PROJ

2. ASG → Site 1

   Site 1 computes EMP'=EMP $\bowtie$ ASG
   EMP' → Site 3
   Site 3 computes EMP’ $\bowtie$ PROJ

3. ASG → Site 3

   Site 3 computes ASG'=ASG $\bowtie$ PROJ
   ASG' → Site 1
   Site 1 computes ASG' $\bowtie$ EMP

4. PROJ → Site 2

   Site 2 computes PROJ'=PROJ $\bowtie$ ASG
   PROJ' → Site 1
   Site 1 computes PROJ' $\bowtie$ EMP

5. EMP → Site 2

   PROJ → Site 2
   Site 2 computes EMP $\bowtie$ PROJ $\bowtie$ ASG

Examples: The order (EMP, ASG, PROJ) could use strategy 1.
The order (PROJ, ASG, EMP) could use strategy 4.

Semijoin Algorithms

- General form of semijoin (derivation):

  $R \ltimes_F S = \Pi_A(R \bowtie_F S) = \Pi_A(R) \bowtie \Pi_{A \cap B}(S) = R \bowtie_F \Pi_{A \cap B}(S)$

  where

  R[A], *S[B]* are relations

- Consider the join of two relations:

  $R[A]$ (located at site 1)

  $S[A]$ (located at site 2)

- Alternatives:

  1. Do the join $R \bowtie_A S$
  2. Perform one of the semijoin equivalents

$$R \bowtie_A S \Leftrightarrow (R \ltimes_A S) \bowtie_A S$$

$$\Leftrightarrow R \bowtie_A (S \ltimes_A R)$$

$$\Leftrightarrow (R \ltimes_A S) \bowtie_A (S \ltimes_A R)$$

Semijoin Algorithms

- Perform the join

  send R to Site 2

  Site 2 computes $R \bowtie_A S$

- Consider semijoin $(R \ltimes_A S) \bowtie_A S$

  $S' = \Pi_A(S)$

  $S' \rightarrow$ Site 1

  Site 1 computes $R' = R \ltimes_A S'$

  $R' \rightarrow$ Site 2

  Site 2 computes $R' \bowtie_A S$

Semijoin is better if

$$size(\Pi_A(S)) + size(R \ltimes_A S)) < size(R)$$

Semijoin Algorithms

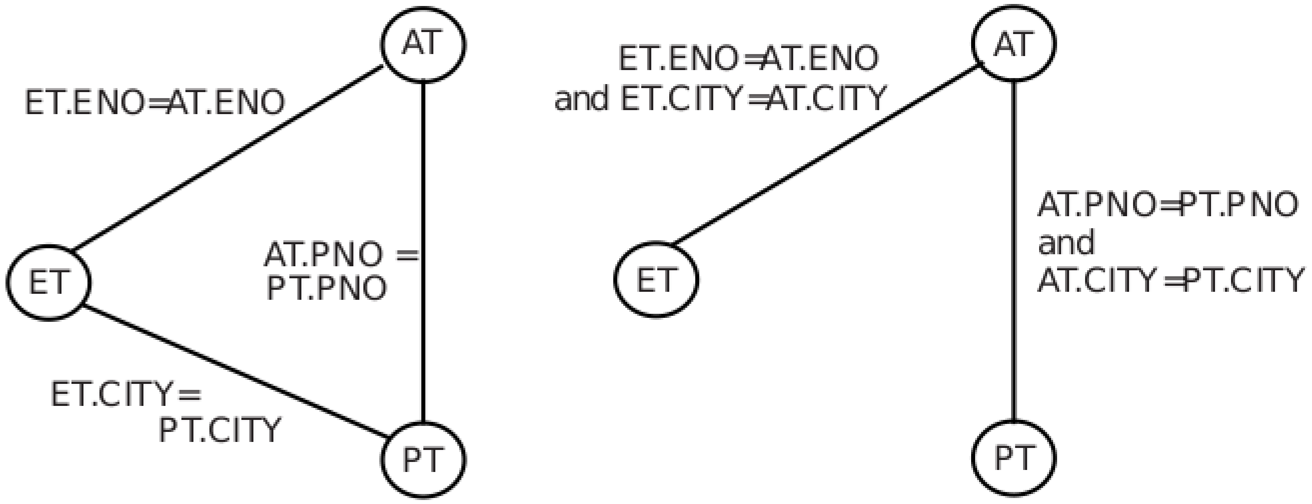
- Semijoins are useful for multi-join queries
  - Reducing the size of the operand relations involved in multiple join queries
  - Optimization becomes more complex
  - Example: program to compute EMP ⋈ ASG ⋈ PROJ is
  - EMP' ⋈ ASG' ⋈ PROJ,
  - where EMP' = EMP ⋉ ASG and ASG' = ASG ⋉ PROJ.
  - We may further reduce the size of an operand relation
  - EMP'' = EMP ⋉ (ASG ⋉ PROJ)
    - size(ASG ⋉ PROJ) ≤ size(ASG), we have size(EMP'') ≤ size(EMP')
    - EMP ⋉ (ASG ⋉ PROJ) is *semijoin program* for EMP
    - there exist several potential semijoin programs
    - there is one optimal semijoin program, called the *full reducer*

Semijoin Algorithms

- The problem is to find the full reducer
    - Evaluate the size reduction of all possible semijoin programs
    - Problems with the enumerative method
        - Cyclic queries, that have cycles in their join graph and for which full reducers cannot be found
        - Tree queries: full reducers exist, but the number of candidate semijoin programs is exponential in the number of relations, which makes the enumerative approach NP-hard
- Full reducers for tree queries exist
    - The problem of finding them is NP-hard
    - Important class of queries, called chained queries
        - A chained query has a join graph where relations can be ordered, and each relation joins only with the next relation in the order
    - Polynomial algorithm exists

Semijoin:Example

ET(ENO, ENAME, TITLE, CITY)
AT(ENO, PNO, RESP, DUR)
PT(PNO, PNAME, BUDGET, CITY)



(a) Cyclic query

(b) Equivalent acyclic query

```
SELECT ENAME, PNAME
FROM   ET, AT, PT
WHERE  ET.ENO = AT.ENO
AND    AT.ENO = PT.ENO
AND    ET.CITY = PT.CITY
```

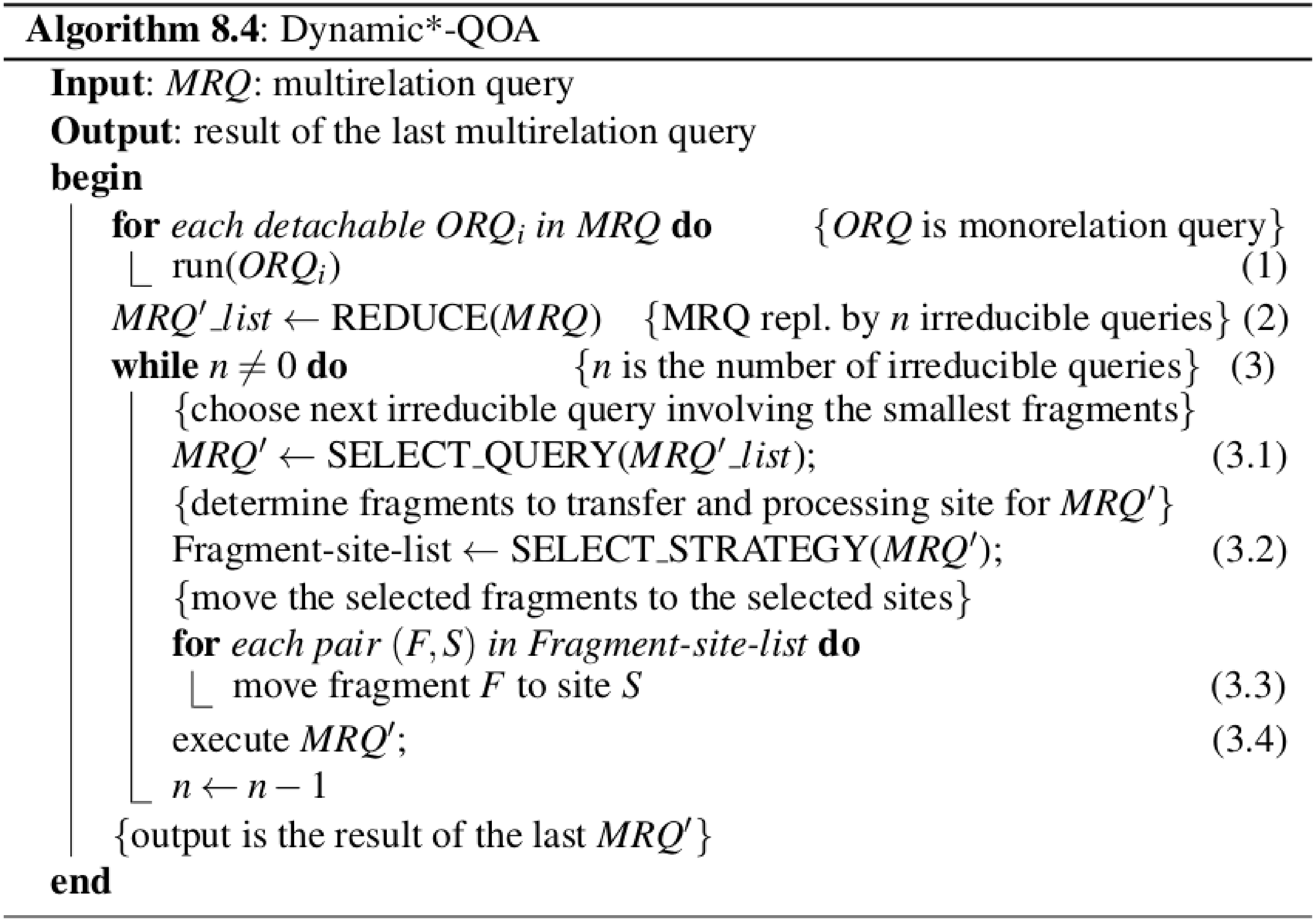
Outline

- Distributed Query Processing
  - Introduction
  - Query Decomposition and Localization
  - Centralized query optimization
  - Join Ordering
  - Distributed Query Optimization
  - Adaptive Query Processing

Distributed Dynamic Algorithm

1. Execute all monorelation queries (e.g., selection, projection)
2. Reduce the multirelation query to produce irreducible subqueries $q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n$ such that there is only one relation between q_i and q_{i+1}
3. Choose q_i involving the smallest fragments to execute (call MRQ')
4. Find the best execution strategy for MRQ'
   a) Determine processing site
   b) Determine fragments to move
5. Repeat 3 and 4

Distributed Dynamic Algorithm



Algorithm 8.4: Dynamic*-QOA

```
Input: $MRQ$: multirelation query
Output: result of the last multirelation query
begin
    for each detachable $ORQ_i$ in $MRQ$ do        {$ORQ$ is monorelation query}
        run($ORQ_i$)                                                        (1)
    $MRQ'\_list \leftarrow$ REDUCE($MRQ$)   {MRQ repl. by $n$ irreducible queries} (2)
    while $n \neq 0$ do              {$n$ is the number of irreducible queries}   (3)
        {choose next irreducible query involving the smallest fragments}
        $MRQ' \leftarrow$ SELECT_QUERY($MRQ'\_list$);                         (3.1)
        {determine fragments to transfer and processing site for $MRQ'$}
        Fragment-site-list $\leftarrow$ SELECT_STRATEGY($MRQ'$);               (3.2)
        {move the selected fragments to the selected sites}
        for each pair $(F, S)$ in Fragment-site-list do
            move fragment $F$ to site $S$                                      (3.3)
        execute $MRQ'$;                                                       (3.4)
        $n \leftarrow n - 1$
    {output is the result of the last $MRQ'$}
end
```

Distributed Dynamic Algorithm

- Query is expressed in tuple rel. calculus (CNF); schema info available (network type, as well as the location and size of each fragment)
  - Query optimization executed at *master site*
- Algorithm
  - (step 1) Detached mono-relational queries are run locally
  - (step 2) Query reduced to irreducible (part.ordered) and mono-relational queries
  - (step 3.1) Select next subquery with smallest fragments involved (=>smallest result)
  - (step 3.2) Selects the best strategy to process the subquery.
    Which fragm. to move and where join is executed? (->set of pairs (F,S))
    Intermed. results are always moved to the remaining table.
    Remaining rel. may be further partitioned into k fragments (parallel exec.)
  - (step 3.3) Transfers all the fragments to their processing sites
  - (step 3.4) Executes the selected subquery

Distributed Dynamic Algorithm

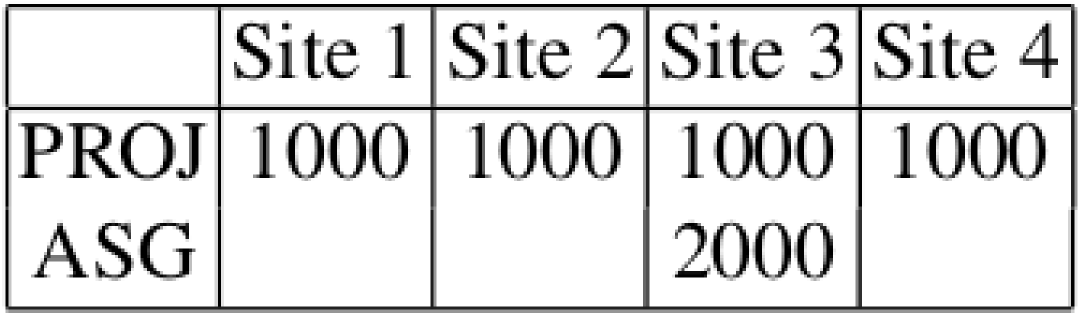
- The reduction algorithm is applied to the original query in step 2.
  - The algorithm has produced subqueries and their dependency order (poset).
- The optimization occurs in 3.1 and 3.2.
- At step 3.1: a simple choice for the next subquery is to take the next one having no predecessor and involving the smaller fragments.
  - Example: a query q has the subqueries q1 , q2 , and q3 , with dependencies q1 → q3 , q2 → q3. If fragments of q1 are smaller than those of q2 then q1 executes first.
  - This choice also depends on the number of sites having relevant fragments.
- At step 3.2: determines how to execute the subquery by selecting the fragms that will be moved and sites where processing will take place.
  - Fragms resulting from n-1 subqueries moved to fragms of n-th subquery.
  - *fragment-and-replicate*: remaining relation may be further partitioned into k "equalized" fragments in order to increase parallelism.
    - Replication is cheaper in broadcast networks than in point-to-point networks.
    - Decreases response time (parallel proc) but increases communication costs (total time)

Distributed Dynamic Algorithm

- Dynamic query optimization algorithm is characterized by a limited search of the solution space
- Optimization decision is taken for each step without concerning itself with the consequences of that decision on global optimization.

Distributed Dynamic Algorithm - Example

- Let us consider the query PROJ ⋈ ASG, where PROJ and ASG are fragmented
- Assume that the allocation of fragments and their sizes are as follows (in kilobytes)
- Discussion:
  - Point–to–point network, the best
  - strategy is to send each PROJ_i to site 3,
  - 3000 kbytes, versus 6000 kbytes
  - if ASG is sent to sites 1,2, and 4.
  - Broadcast network, the best strategy is to send ASG (in
  - a single transfer) to sites 1, 2, and 4, which incurs a transfer of 2000 kbytes.
  - The latter strategy is faster and maximizes response time because the joins can be done in parallel.

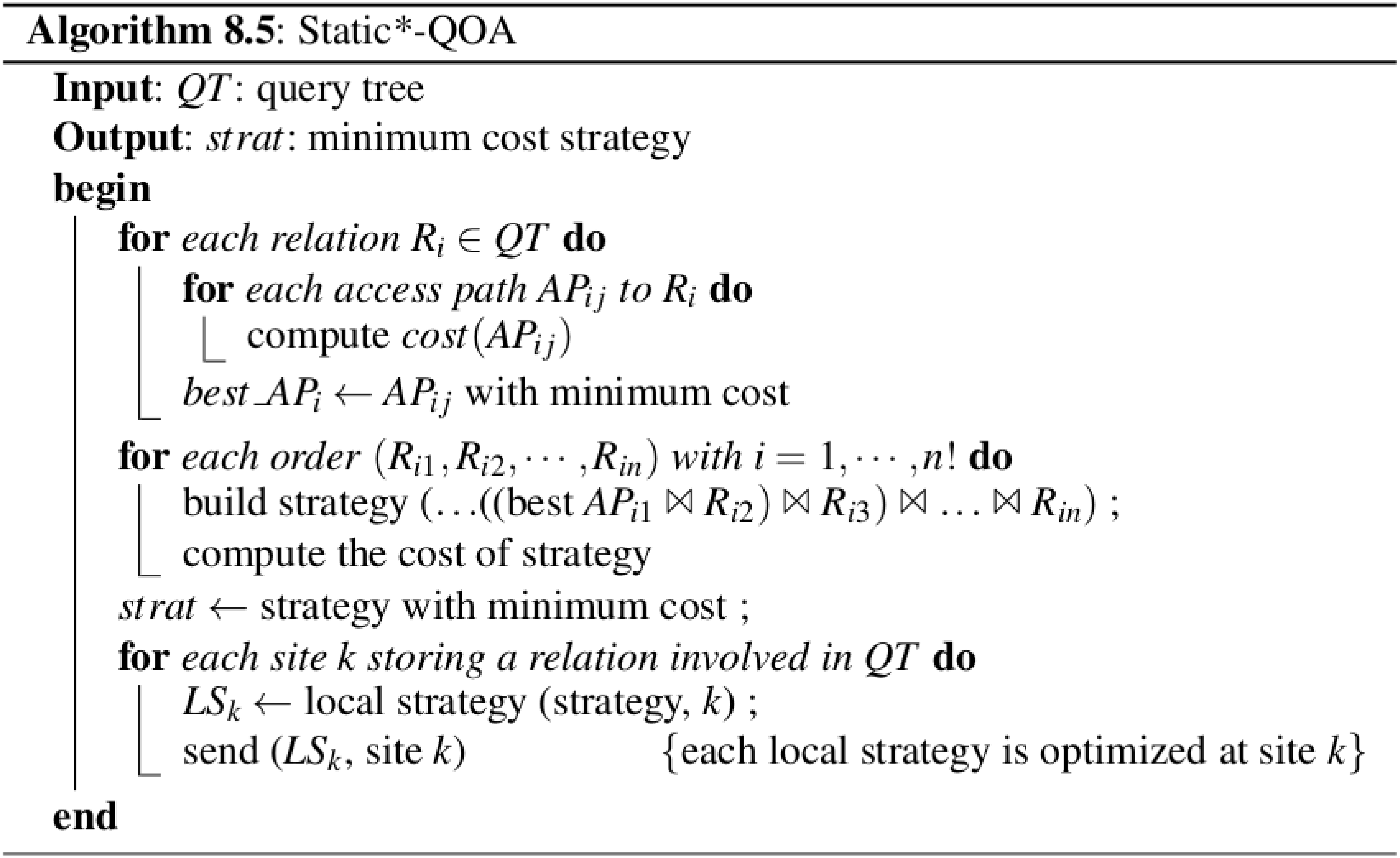


	Site 1	Site 2	Site 3	Site 4
PROJ	1000	1000	1000	1000
ASG			2000	

Distributed Static Algorithm

- Based on System R*, IBM
  - Cost function includes local processing as well as transmission
  - Considers only joins; (left) deep plans
  - “Exhaustive” search
  - Compilation
- Query compilation coordinated by a <u>master</u> site
  - Site where query was initiated.
  - Master handles query optimization
  - Master handles all intersite decisions
    - Selection of the execution sites and the fragments as well as
    - the method for transferring data
  - Apprentice sites involved in the query, make the remaining local decisions
    - Ordering of joins at a site

Distributed Static Algorithm



```
Algorithm 8.5: Static*-QOA
Input: $QT$: query tree
Output: $strat$: minimum cost strategy
begin
    for each relation $R_i \in QT$ do
        for each access path $AP_{ij}$ to $R_i$ do
            compute $cost(AP_{ij})$
        $best\_AP_i \leftarrow AP_{ij}$ with minimum cost
    for each order $(R_{i1}, R_{i2}, \cdots, R_{in})$ with $i = 1, \cdots, n!$ do
        build strategy $(\ldots((\text{best } AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie \ldots \bowtie R_{in})$ ;
        compute the cost of strategy
    $strat \leftarrow$ strategy with minimum cost ;
    for each site $k$ storing a relation involved in $QT$ do
        $LS_k \leftarrow$ local strategy (strategy, $k$) ;
        send ($LS_k$, site $k$)        {each local strategy is optimized at site $k$}
end
```

Distributed Static Algorithm

- Input to algorithm is a fragment query expressed as a query tree.
- Optimizer must select:
  - Join ordering, join algorithm, and the access path for each fragments
    - Statistics, estimated size of intermediate results and access path information
  - Sites of join results and the method of transferring data between sites
- Apprentice sites select the local join ordering
  - To join two relations, there are three candidate sites: the site of the first relation, the site of the second relation, or a third site (e.g., the site of a next relation to be joined with).
- Two methods are supported for intersite data transfers.
  - Ship whole
  - Fetch as needed

Static Approach – Performing Joins

- Ship whole
  - Whole outer relation is shipped to site a of inner relation, stored in a temp. table and joined with the inner relation.
  - Larger data transfer. Fast if relations are small.
  - Small number of messages.
  - Received relation can be directly pipelined into merge-join.
- Fetch as needed
  - Number of messages = *O*(cardinality of external relation)
  - The outer relation is sequentially scanned, and each tuple is sent to the site of the inner relation which accesses a local table and returns selected tuples back to the site of outer. relation.
  - Appropriate for small number of outer tuples.
  - Data transfer per message is minimal (but latency)

Static Approach – Join ordering

1. Move outer relation tuples to the site of the inner relation

(a) Retrieve outer tuples
(b) Send them to the inner relation site
(c) Join them as they arrive

Total Cost = cost(retrieving qualified outer tuples)
\+ no. of outer tuples fetched * cost(retrieving qualified inner tuples)
\+ msg. cost * (no. outer tuples fetched * avg. outer tuple size)/msg. size

Static Approach – Join ordering

2. Move inner relation to the site of outer relation

Cannot join as they arrive; they need to be stored

Total cost = cost(retrieving qualified outer tuples)

\+ no. of outer tuples fetched * cost(retrieving matching inner tuples from temporary storage)

\+ cost(retrieving qualified inner tuples)

\+ cost(storing all qualified inner tuples in temporary storage)

\+ msg. cost * no. of inner tuples fetched * avg. inner tuple size/msg. size

Static Approach – Join ordering

3.Fetch inner tuples as needed

(a) Retrieve qualified tuples at outer relation site

(b) Send request containing join column value(s) for outer tuples to inner relation site

(c) Retrieve matching inner tuples at inner relation site

(d) Send the matching inner tuples to outer relation site

(e) Join as they arrive

Total Cost = cost(retrieving qualified outer tuples)

\+ msg. cost * (no. of outer tuples fetched)

\+ no. of outer tuples fetched * no. of inner tuples fetched * avg. inner tuple size * msg. cost / msg. size)

\+ no. of outer tuples fetched * cost(retrieving matching inner tuples for one outer value)

© M. T. Özsu & P. Valduriez

Static Approach – Join ordering

4. Move both inner and outer relations to another site

Total cost = cost(retrieving qualified outer tuples)
+ cost(retrieving qualified inner tuples)
+ cost(storing inner tuples in storage)
+ msg. cost · (no. of outer tuples fetched * avg. outer tuple size)/msg. size
+ msg. cost * (no. of inner tuples fetched * avg. inner tuple size)/msg. size
+ no. of outer tuples fetched * cost(retrieving inner tuples from temporary storage)

Static Approach – Example

- Join of relations PROJ, the external relation, and ASG, the internal relation, on attribute PNO
  - PROJ ⋈ ASG
- We assume that
  - PROJ and ASG are stored at two different sites
  - there is an index on attribute PNO for relation ASG
- The possible execution strategies for the query are as follows:

  1. Ship whole PROJ to site of ASG.
  2. Ship whole ASG to site of PROJ.
  3. Fetch ASG tuples as needed for each tuple of PROJ.
  4. Move ASG and PROJ to a third site.
- Discussion
  - Strategy 4: the highest cost since both relations must be transferred
  - Strategy 2: size(PROJ) >> size(ASG)
  - minimizes the communication time
  - likely to be the best (if local processing time is not too high compared to strategies 1 and 3)

Static Approach – Example

- Discussion
  - local processing time of strategies 1 and 3 is probably much better than that of strategy 2 since they exploit the index
  - If strategy 2 is not the best, the choice is between strategies 1 and 3
  - If PROJ is large and only a few tuples of ASG match, strategy 3 wins
  - if PROJ is small or many tuples of ASG match, strategy 1 should be the best.

Dynamic Programming

- Used in almost all commercial database products
  - Pioneered in IBM’s System R project [Selinger et al. 1979]
  - Poduces the best possible plans if the cost model is sufficiently accurate
  - Exponential time and space complexity O(2^n); not viable for complex queries
- Iterative dynamic programming.
  - Good plans for simple queries and “as good as possible plans” for complex queries
- Basic dynamic programming algorithm for QO
  - It works in a bottom-up way:
    - building more complex (sub-) plans from simpler (sub-) plans

1) Builds an access plan for every table involved in the query (lines 1-4)

2) Builds a plan for n relations from the plan for n-1 relations + one more join (lines 5-12)

Dynamic Programming

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

```
for $i = 1$ to $n$ do {
    $optPlan(\{R_i\}) = accessPlans(R_i)$
    $prunePlans(optPlan(\{R_i\}))$
}
for $i = 2$ to $n$ do {
    for all $S \subseteq \{R_1, \ldots, R_n\}$ such that $|S| = i$ do {
        $optPlan(S) = \emptyset$
        for all $O \subset S$ do {
            $optPlan(S) = optPlan(S) \cup joinPlans(optPlan(O), optPlan(S - O))$
            $prunePlans(optPlan(S))$
        }
    }
}
return optPlan($\{R_1, \ldots, R_n\}$)
```

Dynamic Programming

1) Access plans for every single table involved in the query
    - If Table A is replicated at sites S1 and S2, the algorithm would enumerate scan(A, S1) and scan(A, S2).
2) Plan for n rels = best plans for o rels + best plans for n-o rels
    - First, two-way join plans using the access plans as building blocks (Lines 5 to 13)
        - The algorithm would enumerate alternative join plans for all relevant sites
    - Next, the algorithm builds three-way join plans, using access-plans and two-way join plans as building blocks.
    - Algorithm continues in this way until it has enumerated all n-way join plans which are complete plans for the query, if the query involves n tables.

- Some comments on dynamic programming algorithm
    - Inferior plans are discarded (i.e., pruned) as early as possible (Lines 3 and 10).
    - Prunning if alternative plan exists that does the same or more at a lower cost.
        - Pruning significantly reduces the complexity of query optimization (comparing to O(n!))
        - Neither scan(A,S1) nor scan(A, S2) may be immediately pruned in order to guarantee that the optimizer finds a good plan
        - scan(A,S2) is pruned: cost(scan(A,S1)) + cost(ship(A,S1,S2) < cost(scan(A,S2)

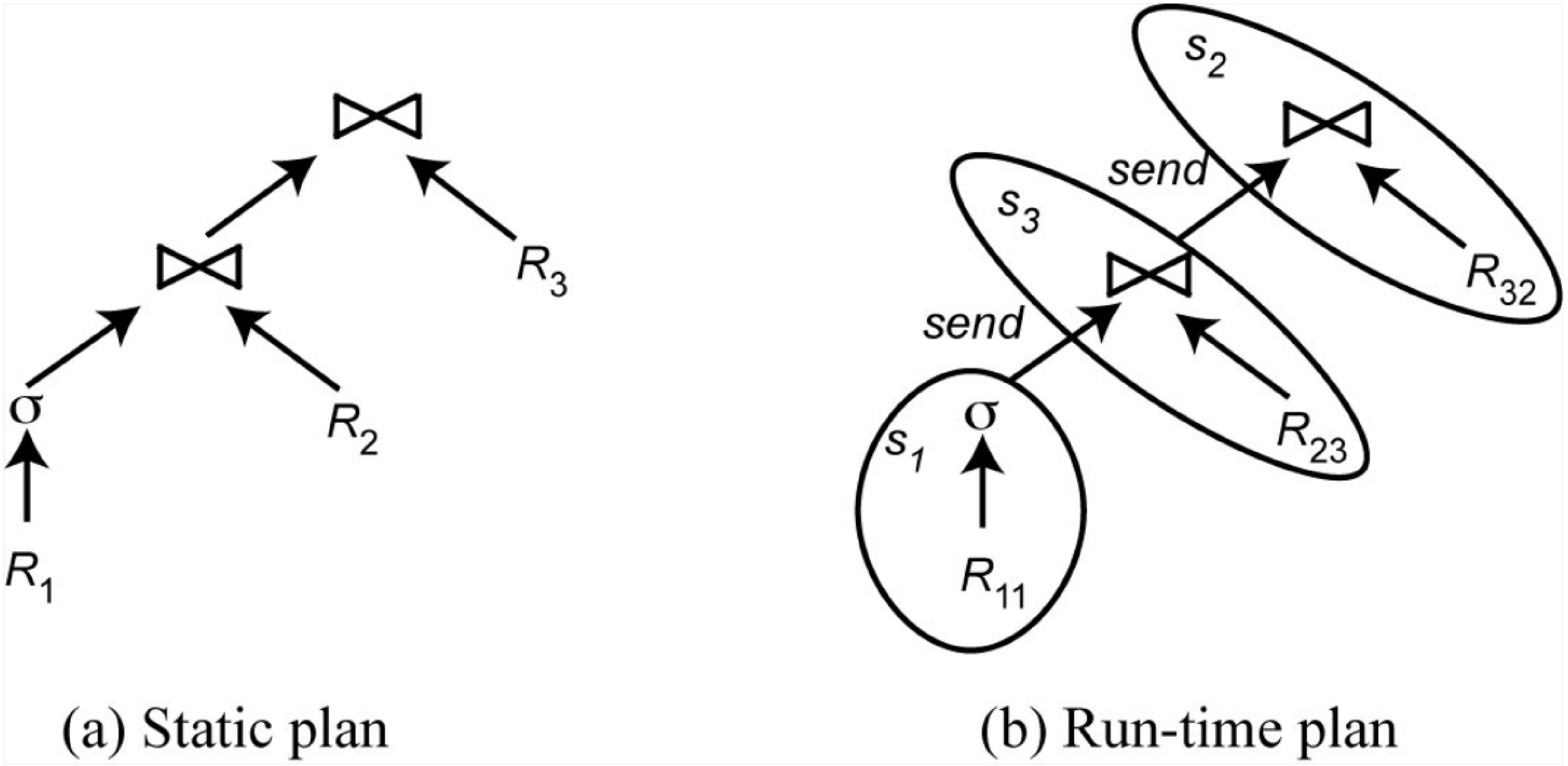
Dynamic Programming

- Some comments on dynamic programming algorithm (cont.)
  - In general, a plan P1 may be pruned if there exists a plan P2 that does the same or more work and the following criterion holds:

    $\forall i \in$ interesting sites(P1) : cost (ship(P1,i)) $\geq$ cost (ship(P2, i))
  - *Intresting_sites*: set of sites that are potentially involved in processing the query

2-Step Optimization

1. At compile time, generate a static plan with operation ordering and access methods only
2. At startup time, carry out site and copy selection and allocate operations to sites



(a) Static plan (b) Run-time plan

2-Step – Problem Definition

- Given
  - A set of sites $S = \{s_1, s_2, \ldots, s_n\}$ with the load of each site
  - A query $Q = \{q_1, q_2, q_3, q_4\}$ such that each subquery q_i is the maximum processing unit that accesses one relation and communicates with its neighboring queries
  - For each q_i in Q, a feasible allocation set of sites $S_q = \{s_1, s_2, \ldots, s_k\}$ where each site stores a copy of the relation in q_i
- The objective is to find an optimal allocation of Q to S such that
  - the load unbalance of S is minimized
  - The total communication cost is minimized

2-Step – Problem Definition

- Each site s_i has a load, denoted by load(s_i), which reflects the number of queries currently submitted
- The load can be expressed in different ways, e.g. as the number of I/O bound and CPU bound queries at the site
- The average load of the system is defined as:

$$Avg\_load(S) = \frac{\sum_{i=1}^{n} load(s_i)}{n}$$

- The balance of the system for a given allocation of subqueries to sites can be measured using the following unbalance factor

$$UF(S) = \frac{1}{n} \sum_{i=1}^{n} (load(s_i) - Avg\_load(S))^2$$

2-Step – Problem Definition

- The problem addressed by the second step of two-step query optimization can be formalized as the following subquery allocation problem. Given
- 1. a set of sites $S = \{s_1, .., s_n\}$ with the load of each site;
- 2. a query $Q = \{q_1, .., q_m\}$; and
- 3. for each subquery q_i in Q, a feasible allocation set of sites
-     $S_q = \{s_1, ..., s_k\}$
-     where each site stores a copy of the relation involved in q_i ;
- the objective is to find an optimal allocation on Q to S such that
- 1. UF(S) is minimized, and
- 2. the total communication cost is minimized.

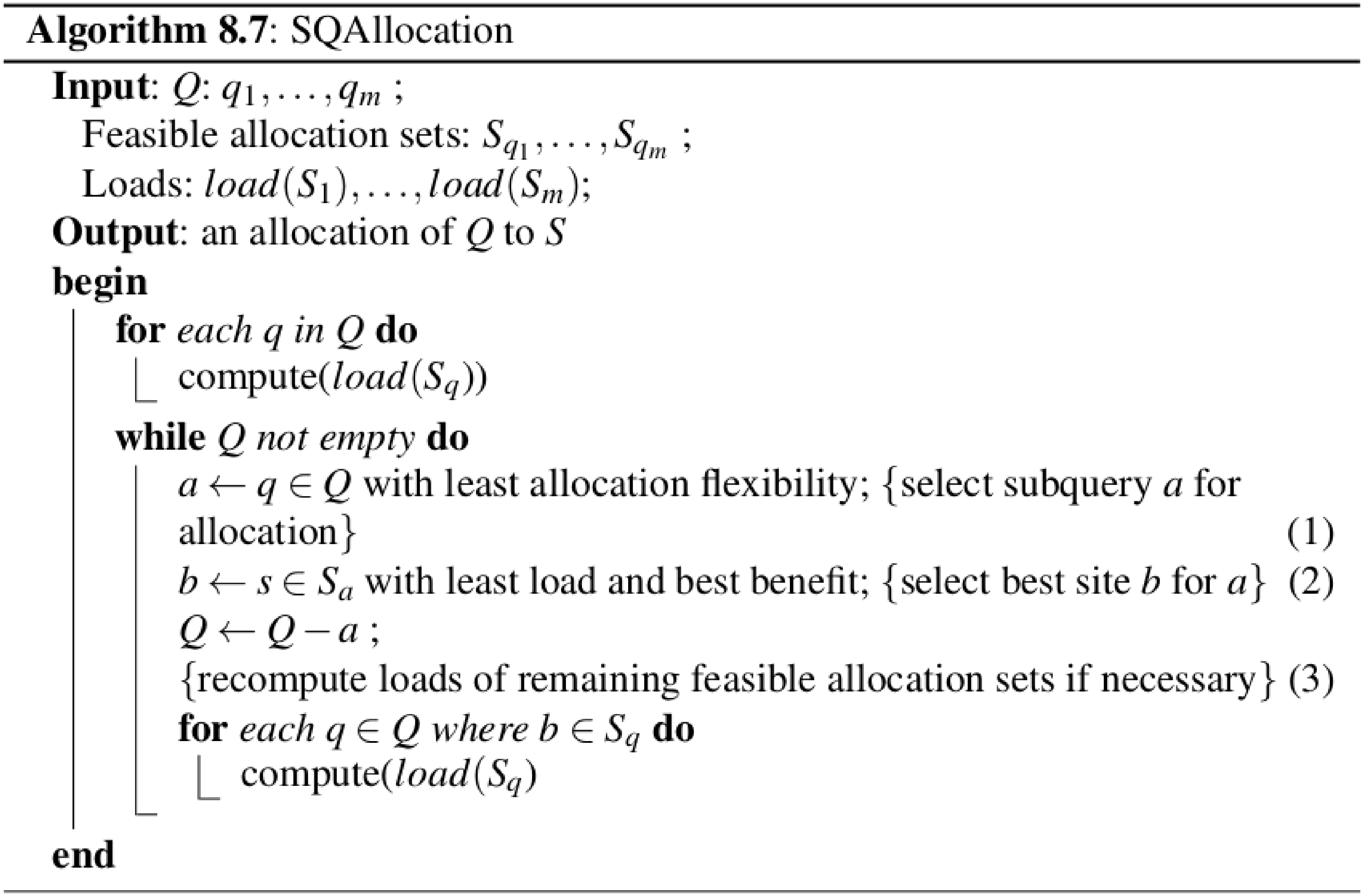
2-Step – Algorithm

- The algorithm, which we describe for linear join trees, uses several heuristics.
  1. Start by allocating subqueries with least allocation flexibility, i.e. with the smaller feasible allocation sets of sites.
  2. Consider the sites with least load and best benefit.
- The benefit of a site is defined as
  1. the number of subqueries already allocated to the site and
  2. measures the communication cost savings from allocating the subquery and
  3. the load information of any unallocated subquery that has a selected site in its feasible allocation set is recomputed

2-Step Algorithm

- For each *q* in *Q* compute load (S_q)
- While *Q* not empty do
    1. Select subquery *a* with least allocation flexibility
    2. Select best site *b* for *a* (with least load and best benefit)
    3. Remove *a* from *Q* and recompute loads if needed

2-Step – Algorithm

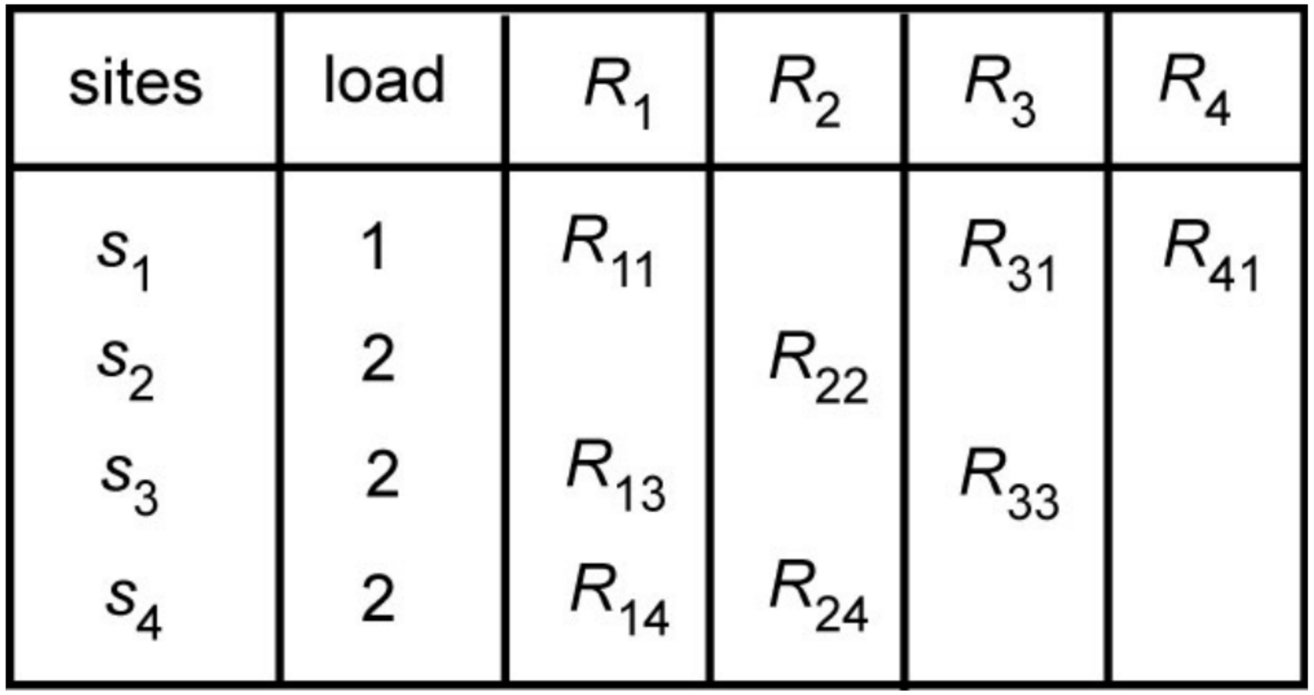


Algorithm 8.7: SQAllocation

```
Input: $Q$: $q_1, \ldots, q_m$ ;
  Feasible allocation sets: $S_{q_1}, \ldots, S_{q_m}$ ;
  Loads: $load(S_1), \ldots, load(S_m)$;
Output: an allocation of $Q$ to $S$
begin
    for each $q$ in $Q$ do
        compute($load(S_q)$)
    while $Q$ not empty do
        $a \leftarrow q \in Q$ with least allocation flexibility; {select subquery $a$ for allocation}   (1)
        $b \leftarrow s \in S_a$ with least load and best benefit; {select best site $b$ for $a$}   (2)
        $Q \leftarrow Q - a$ ;
        {recompute loads of remaining feasible allocation sets if necessary}   (3)
        for each $q \in Q$ where $b \in S_q$ do
            compute($load(S_q)$)
end
```

2-Step Algorithm Example

- Let $Q = \{q_1, q_2, q_3, q_4\}$ where q_1 is associated with R_1, q_2 is associated with R_2 joined with the result of q_1, etc.
- Iteration 1: select q_4, allocate to s_1, set load(s_1)=2
- Iteration 2: select q_2, allocate to s_2, set load(s_2)=3
- Iteration 3: select q_3, allocate to s_1, set load(s_1) =3
- Iteration 4: select q_1, allocate to s_3 or s_4



sites	load	R_1	R_2	R_3	R_4
s_1	1	R_{11}		R_{31}	R_{41}
s_2	2		R_{22}		
s_3	2	R_{13}		R_{33}	
s_4	2	R_{14}	R_{24}		

Note: if in iteration 2, q_2, were allocated to s_4, this would have produced a better plan. So hybrid optimization can still miss optimal plans

Outline

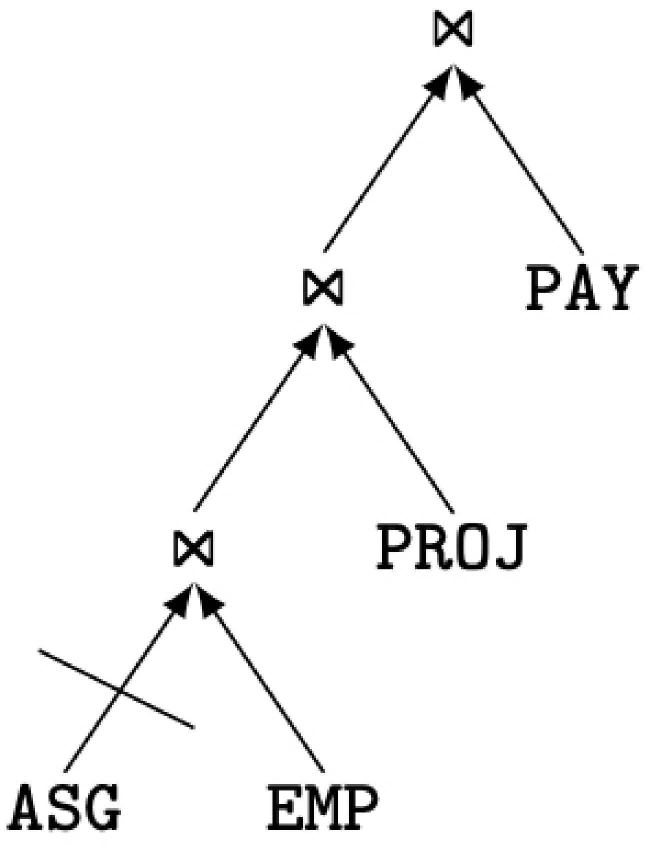
- Distributed Query Processing
  - Introduction
  - Query Decomposition and Localization
  - Centralized query optimization
  - Join Ordering
  - Distributed Query Optimization
  - Adaptive Query Processing

Adaptive Query Processing - Motivations

- Assumptions underlying query optimization
  - The optimizer has sufficient knowledge about runtime
    - Cost information
  - Runtime conditions remain stable during query execution
- Appropriate for systems with few data sources in a controlled environment
- Inappropriate for changing environments with large numbers of data sources and unpredictable runtime conditions

Example: QEP with Blocked Operator

- Assume ASG, EMP, PROJ and PAY each at a different site
- If ASG site is down, the entire pipeline is blocked
- However, with some reorganization, the join of EMP and PAY could be done while waiting for ASG



Adaptive Query Processing – Definition

- A query processing is adaptive if it receives information from the execution environment and determines its behavior accordingly
  - Feed-back loop between optimizer and runtime environment
  - Communication of runtime information between DDBMS components
- Additional components
  - Monitoring, assessment, reaction
  - Embedded in control operators of QEP
- Tradeoff between reactiveness and overhead of adaptation

Adaptive Components

- Monitoring parameters (collected by sensors in QEP)
  - Memory size
  - Data arrival rates
  - Actual statistics
  - Operator execution cost
  - Network throughput
- Adaptive reactions
  - Change schedule
  - Replace an operator by an equivalent one
  - Modify the behavior of an operator
  - Data repartitioning