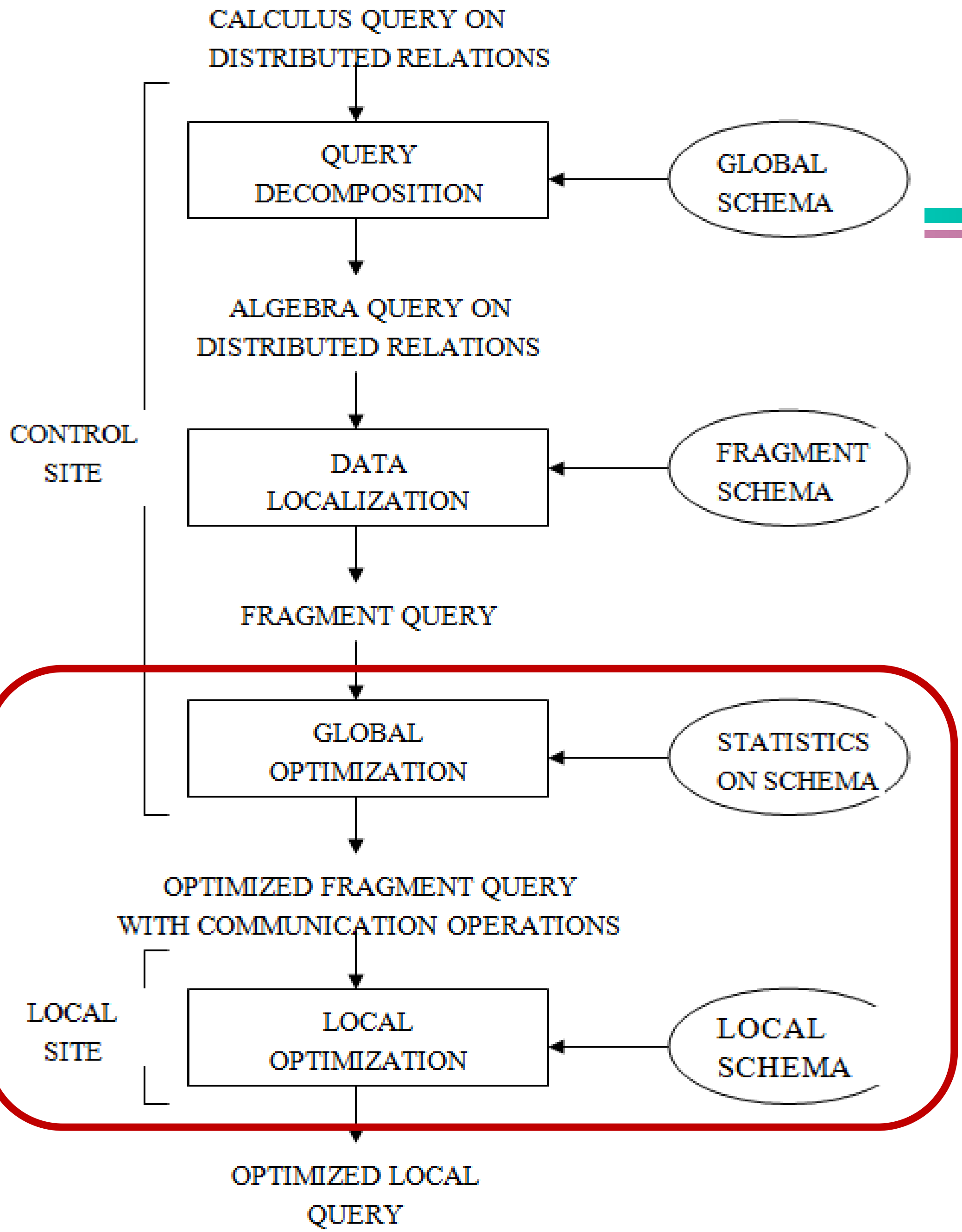
6. Distributed Query Optimization

Chapter 9

Optimization of Distributed Queries

Outline

- Overview of Query Optimization
- Centralized Query Optimization
    - Ingres
    - System R
- Distributed Query Optimization
    - Distributed Ingres
    - System R*



CALCULUS QUERY ON DISTRIBUTED RELATIONS

QUERY DECOMPOSITION

GLOBAL SCHEMA

ALGEBRA QUERY ON DISTRIBUTED RELATIONS

CONTROL SITE

DATA LOCALIZATION

FRAGMENT SCHEMA

FRAGMENT QUERY

GLOBAL OPTIMIZATION

STATISTICS ON SCHEMA

OPTIMIZED FRAGMENT QUERY WITH COMMUNICATION OPERATIONS

LOCAL SITE

LOCAL OPTIMIZATION

LOCAL SCHEMA

OPTIMIZED LOCAL QUERY

Step 3: Global Query Optimization

- The query resulting from decomposition and localization can be executed in many ways by choosing different data transfer paths.

- We need an optimizer to choose a strategy close to the optimal one.

Problem of Global Query Optimization

Input: Fragment query

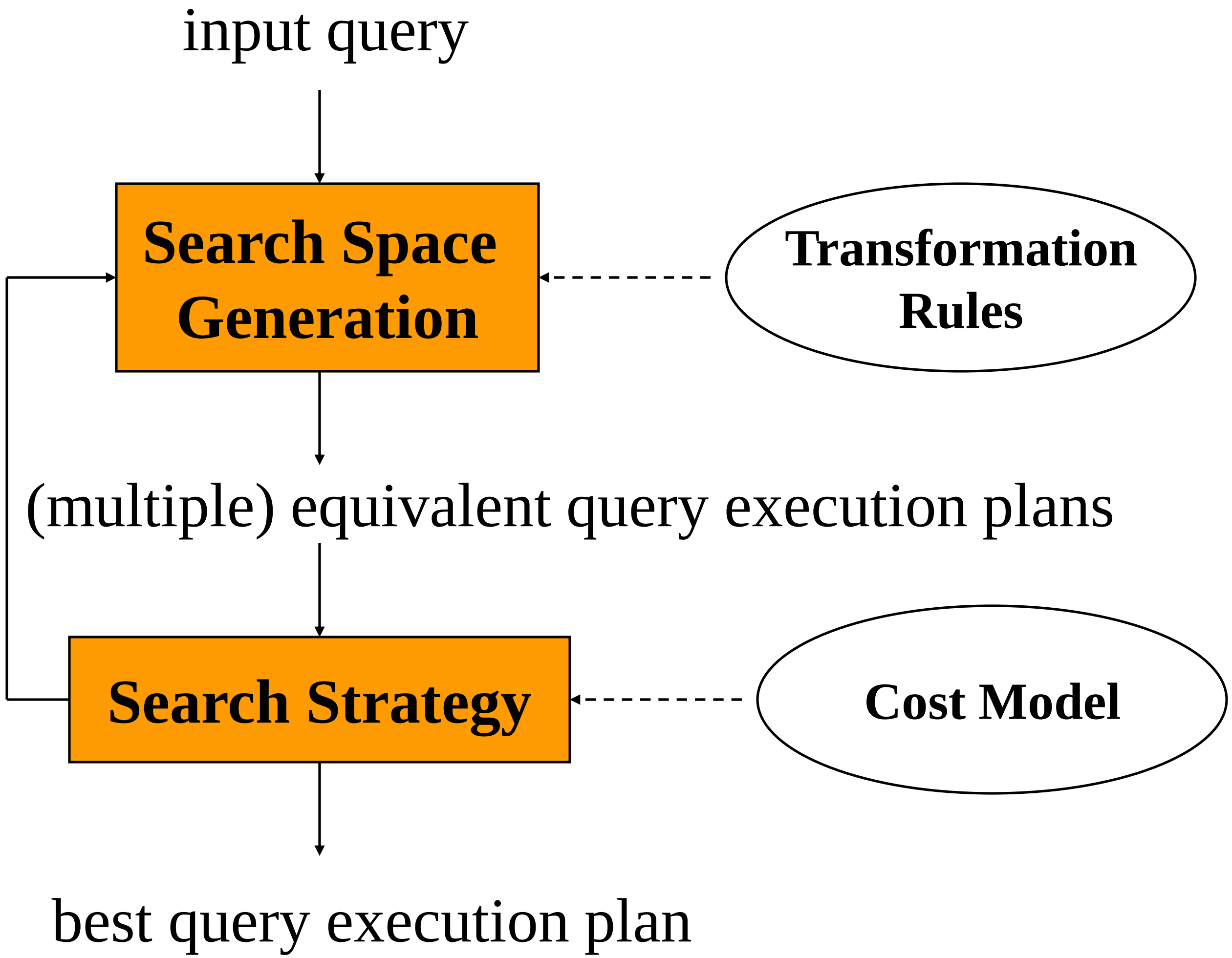
Find the *best* (not necessarily optimal) global schedule

- 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)

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

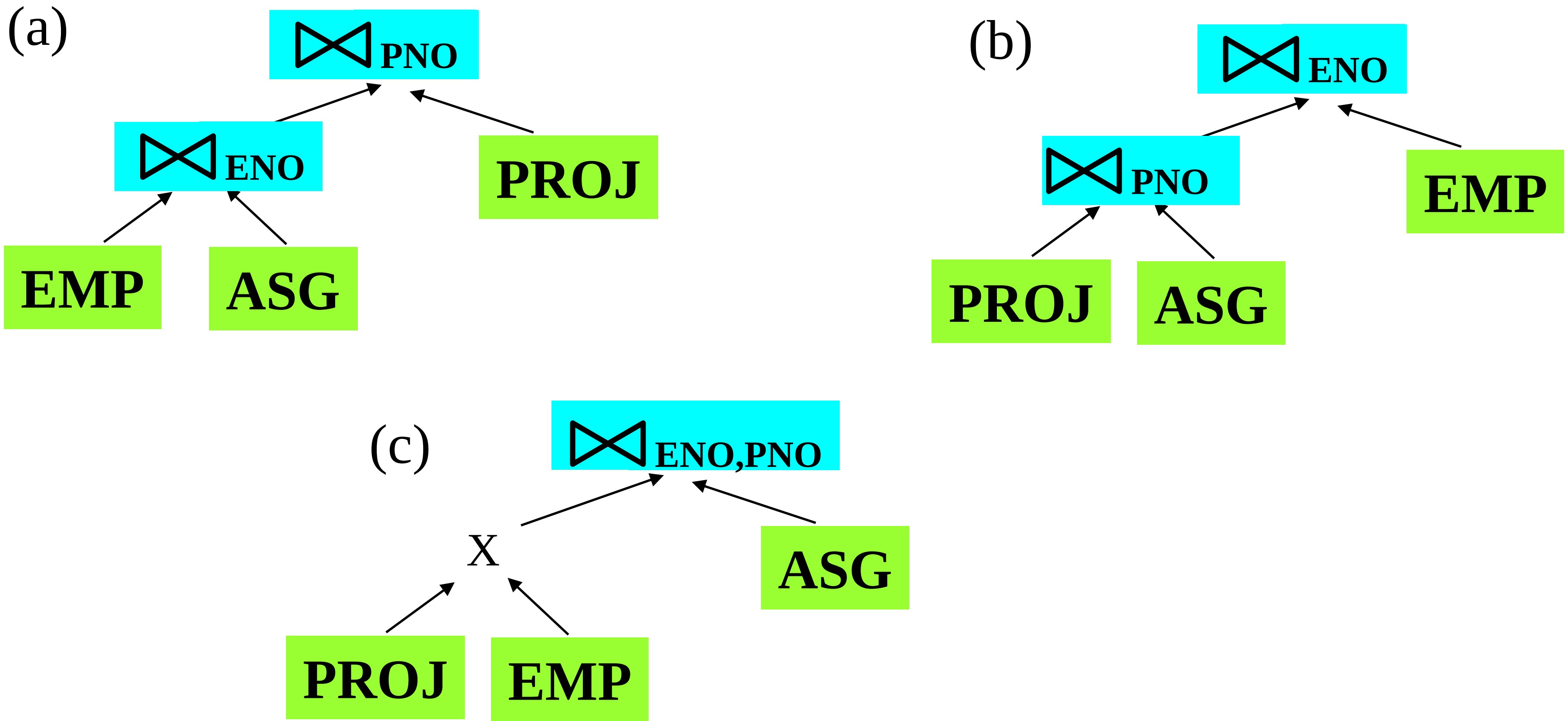


Search Space

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

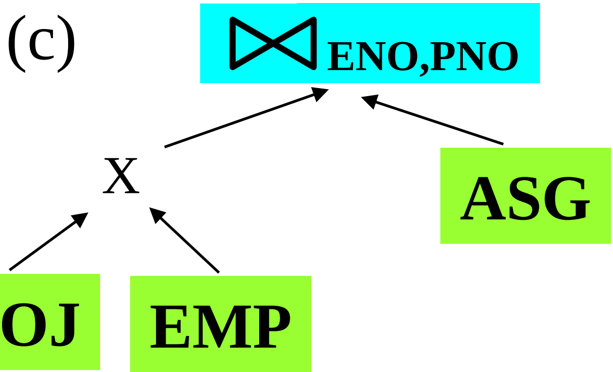
Three Join Tree Examples

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



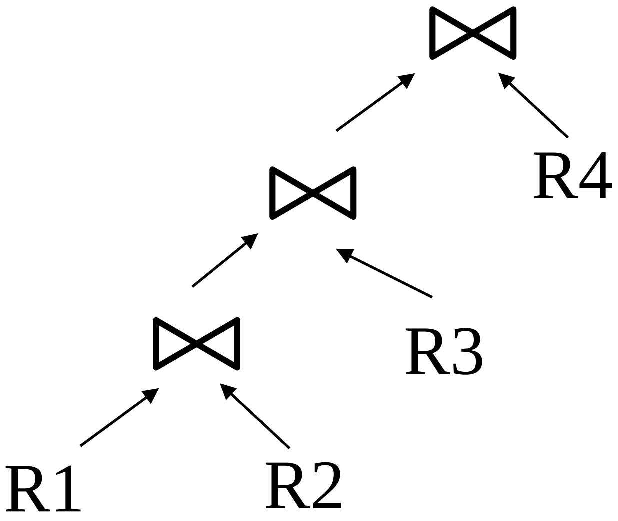
Restricting the Size of Search Space

- A large search space →
  - Optimization time much more than the actual execution time
- Restricting by means of heuristics
  - Perform unary operations (selection, projection) when accessing base relations
  - Avoid Cartesian products that are not required by the query
    - – E.g., previous (c) query plan is removed from the search space

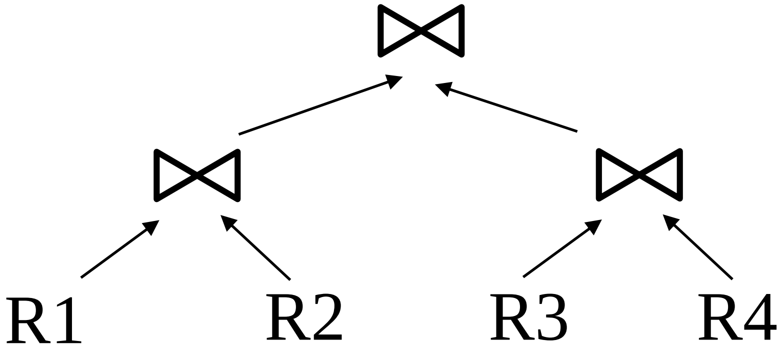


Restricting the Size of Search Space (*cont.*)

- Restricting the shape of the join tree
  - Consider only linear trees, ignore bushy ones
    - Linear tree –at least one operand of each operator node is a base relation
    - Bushy tree – more general and may have operators with no base relations as operands (i.e., both operands are intermediate relations)



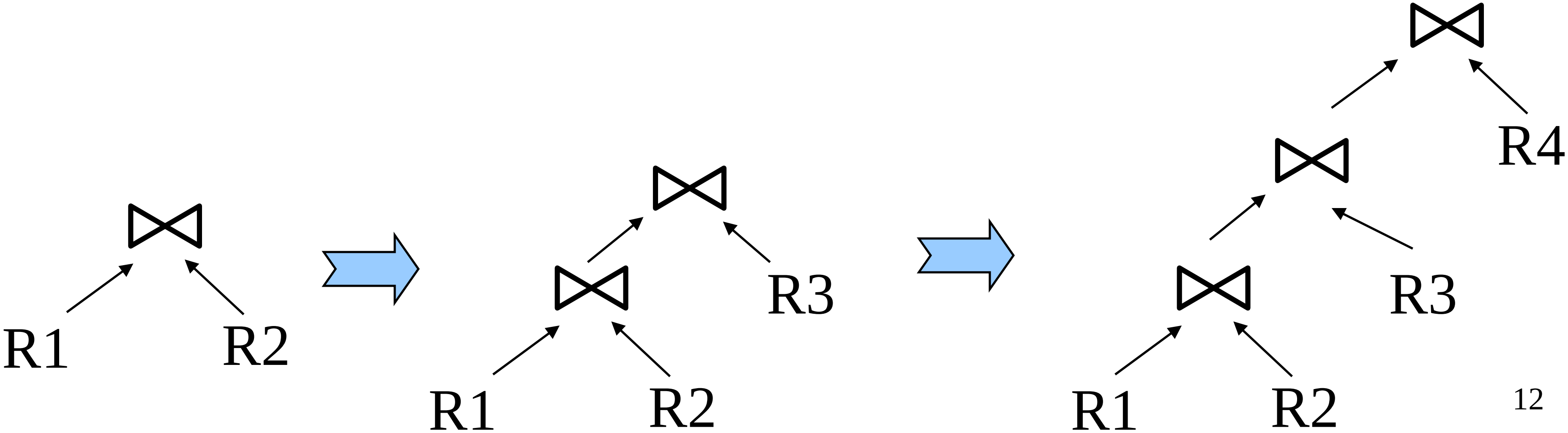
Linear Join Tree



Bushy Join Tree

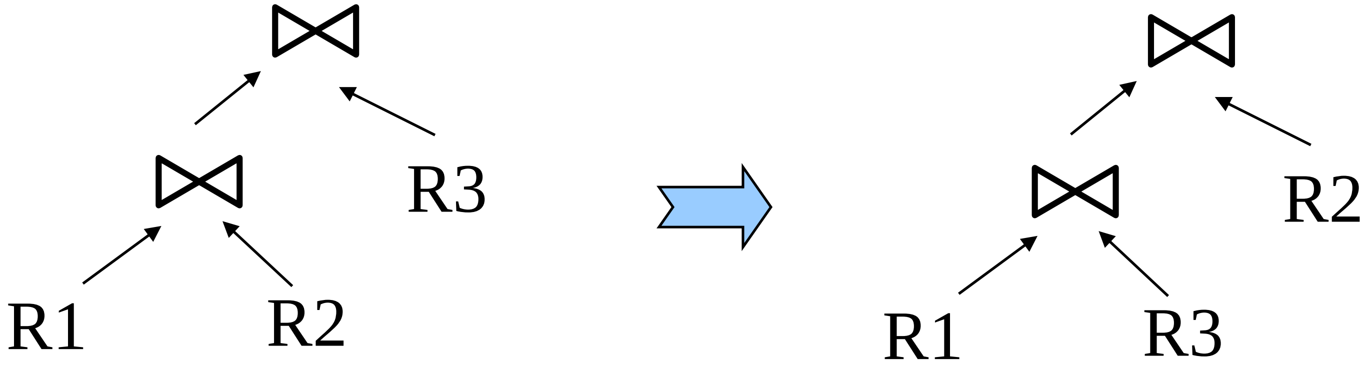
Search Strategy

- How to move in the search space?
  - Deterministic and randomized
- Deterministic
  - Starting from base relations, joining one more relation at each step until complete plans are obtained
  - Breadth-first and Depth-first



Search Strategy (*cont.*)

- Randomized
  - Trade optimization time for execution time
  - Better when > 5-6 relations
  - Do not guarantee the best solution is obtained, but avoid the high cost of optimization in terms of memory and time
  - Search for optimalities around a particular starting point
  - By iterative improvement and simulated annealing



Search Strategy (*cont*.)

- First, one or more start plans are built by a greedy strategy
- Then, the algorithm tries to improve the start plan by visiting its neighbors. A neighbor is obtained by applying a random transformation to a plan.
  - e.g., exchanging two randomly chosen operand relations of the plan.

Cost Functions

- Total time
  - the sum of all time (also referred to as cost) components
- Response Time
  - the elapsed time from the initiation to the completion of the query

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 I/O's

communication cost =

unit message initiation cost * no. of sequential messages +

unit transmission cost * no. of sequential bytes

Total Cost Factors

- Wide area network
  - Message initiation and transmission costs high
  - Local processing cost is low (fast mainframes or minicomputers)
- Local area network
  - 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. I/Os

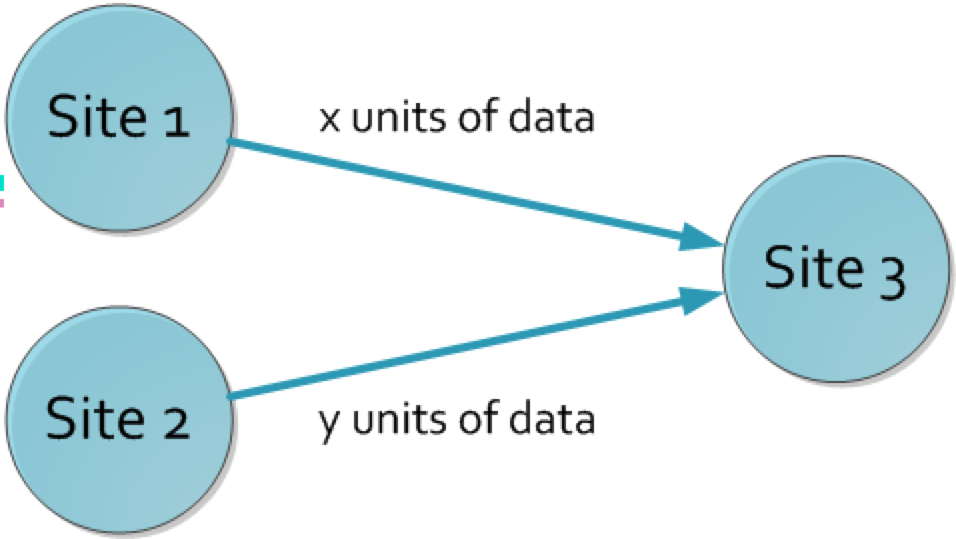
Communication time =

unit message initiation time * no. of sequential messages +

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
- The size of the intermediate relations produced during the execution facilitates the selection of the execution strategy
- This is useful in selecting an execution strategy that reduces data transfer
- The sizes of intermediate relations need to be estimated based on cardinalities of relations and lengths of attributes
  - More precise → more costly to maintain

Optimization Statistics (*cont.*)

- *R* [A_1, A_2, ..., A_n] fragmented as R_1,R_2, …, R_n
- The statistical data collected typically are
  - *len*(A_i), length of attribute A_i in bytes
  - *min*(A_i) and *max*(A_i) value for ordered domains
  - *card*(dom(A_i)), unique values in *dom*(A_i)
  - Number of tuples in each fragment *card*(R_j)
  - $card(\pi_{A_i}(R_j))$, the number of distinct values of A_i in fragment R_j
  - *size*(*R*) = *card*(*R*)**length*(*R*)

Optimization Statistics (*cont*.)

- Selectivity factor of each operation for relations
  - The join selectivity factor for *R* and *S*
    - – a real value between 0 and 1

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

Intermediate Relation Size

- Selection

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

$$SF_\sigma(A = value) = \frac{1}{card(\pi_A(R))}$$

$$SF_\sigma(A > value) = \frac{\max(A) - value}{\max(A) - \min(A)}$$

$$SF_\sigma(A < value) = \frac{value - \min(A)}{\max(A) - \min(A)}$$

$$SF_\sigma(P(Ai) \wedge P(Aj)) = SF_\sigma(P(Ai)) \times SF_\sigma(P(Aj))$$

$$SF_\sigma(P(Ai) \vee P(Aj)) =$$

$$SF_\sigma(P(Ai)) + SF_\sigma(P(Aj)) - SF_\sigma(P(Ai)) \times SF_\sigma(P(Aj))$$

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

Intermediate Relation Size (*cont.*)

- Projection

$card(\pi_A(R)) =$ the number of distinct values of A if A is a single attribute, or *card*(R) if A contains the key of R.

Otherwise, it's difficult.

Intermediate Relation Size (*cont.*)

- Cartesian product

$$card(R\times S)=card(R)\times card(S)$$

- 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 (*cont.*)

- Join
  - No general way for its calculation. Some systems use the upper bound of card (R x S) instead. Some estimations can be used for simple cases.
  - Special case: A is a key of R and B is a foreign key of S

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

  - More general:

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

Intermediate Relation Sizes (*cont.*)

- Semijoin

$$card(R \ltimes_A S) = SF_{\ltimes}(R \ltimes_A S) * card(R)$$

where

$$SF_{\ltimes}(R \ltimes_A S) = card(\prod_A(S)) / card(dom[A])$$

Centralized Query Optimization

- Two examples showing the techniques

INGRES – dynamic optimization, interpretive

System R – static optimization based on exhaustive search

SQL RDBMS - INGRES

- First created as a research project at the University of California, Berkeley, starting in the early 1970s and ending in the early 1980s.
- Since the mid-1980s, Ingres has spawned a number of commercial database applications, including Sybase, Microsoft SQL Server, NonStop SQL, etc.
- Postgres (Post Ingres), a project which started in the mid-1980s, later evolved into PostgreSQL.

Michael Stonebraker
(2014 ACM Turing Award Winner)

- Michael Stonebraker developed Ingres, proving the viability of the relational database theory
  - Ingres was one of the first two relational database systems (the other was IBM System R)

- Won 2014 ACM Turing Award for
  - bringing relational database systems from concept to commercial success.

Michael Stonebraker’s Contributions

- With Ingres:
  - query language design, query processing techniques, access methods, and concurrency control
  - showing that query rewrite techniques (query modification) could be used to implement relational views and access control

- With Postgres,
  - extending the relational database model, enabling users to define, store and manipulate rich objects with complex state and behavior
  - integrating important ideas from object-oriented programming into the relational database context
  - introducing the object-relational model of database architecture

INGRES Language: QUEL

- QUEL Language - a tuple calculus language

Example:

```
range of e is EMP
range of g is ASG
range of j is PROJ
retrieve e.ENAME
where  e.ENO=g.ENO and j.PNO=g.PNO
       and j.PNAME=”CAD/CAM”
```

Note: e, g, and j are called variables

INGRES Language: QUEL (*cont.*)

- One-variable query

  Queries containing a single variable.

- Multivariable query

  Queries containing more than one variable.

- QUEL can be equally translated into SQL. So we just use SQL for convenience.

INGRES – General Strategy

- Decompose a multivariable query into a sequence of mono-variable queries with a common variable
- Process each by an one variable query processor
  - Choose an initial execution plan (heuristics)
  - Order the rest by considering intermediate relation sizes
- No statistical information is maintained.

INGRES - Decomposition

- Replace an *n* variable query *q* by a series of queries $q_1 \to q_2 \to \ldots \to q_n$, where q_i uses the result of q_{i-1}.
- Detachment
  - Query *q* decomposed into *q′*→*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

$$q(V_{1,}V_{2,}\ldots,V_n) \to (q'(t_{1,}V_{2,}\ldots,V_n), t_1 \in R)$$

INGRES – Detachment

q:

SELECT $V_2.A_2$, $V_3.A_3$, …, $V_n.A_n$

FROM R_1 V_1, R_2 V_2, …, R_n V_n

WHERE $P_1(V_1.A_1)$ **AND**

$P_2(V_1.A_1, V_2.A_2, …, V_n.A_n)$

Note: $P_1(V_1.A_1)$ is an one-variable predicate, indicating a chance for optimization, i.e. to execute σ first expressed in following query.

INGRES – Detachment (*cont.*)

q: SELECT $V_2.A_2, V_3.A_3, \ldots, V_n.A_n$

FROM $R_1\ V_1, R_2\ V_2, \ldots, R_n\ V_n$

WHERE $P_1(V_1.A_1)$ AND $P_2(V_1.A_1, V_2.A_2, \ldots, V_n.A_n)$

q'- one variable query generated by the single variable predicate P_1:

SELECT $V_1.A_1$ INTO R_1'

FROM $R_1\ V_1$

WHERE $P_1(V_1.A_1)$

q"- in *q*, use R_1' to replace R_1 and eliminate P_1:

SELECT $V_2.A_2, V_3.A_3, \ldots, V_n.A_n$

FROM $R_1'\ V_1, R_2\ V_2, \ldots, R_n\ V_n$

WHERE $P_2(V_1.A_1, \ldots, V_n.A_n)$

INGRES – Detachment (*cont*.)

Note

- Query *q* is decomposed into $q' \rightarrow q''$
- It is an optimized sequence of query execution

INGRES – Detachment Example

Original query q_1

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

q_1 can be decomposed into $q_{11} \rightarrow q_{12} \rightarrow q_{13}$

INGRES – Detachment Example (*cont.*)

- First use the one variable predicate to get q_{11} and q' such that $q_1 = q_{11} \rightarrow q'$

q_1:

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

q_{11}:

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

q':

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

INGRES – Detachment Example (*cont.*)

- Then q' is further decomposed into $q_{12} \rightarrow q_{13}$

q':

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

q_{12}

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

q_{13}

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

q_{11} is a mono-variable query;

q_{12} and q_{13} are subject to tuple substitution.

Tuple Substitution

- Assume GVAR has two tuples only: <E1> and <E2> , then q_{13} becomes:

q_{13}:

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

↓

q_{131}

```
SELECT     EMP.ENAME
FROM       EMP
WHERE      EMP.ENO = "E1"
```

q_{132}

```
SELECT     EMP.ENAME
FROM       EMP
WHERE      EMP.ENO = "E2"
```

System R

- A research project at IBM's San Jose Research Laboratory beginning in 1974.
  - the first implementation of SQL
  - the first system to demonstrate that a RDBMS could provide good transaction processing performance.
- Design decisions in System R, as well as some fundamental algorithm choices (e.g., query optimization), influenced many later relational systems.

System R

- Static query optimization based on exhaustive search of the solution space
- Simple (i.e., mono-relation) queries are executed according to the best access path
- Execute joins
  - Determine the possible ordering of joins
  - Determine the cost of each ordering of joins
  - Choose the join ordering with minimal cost

System R's Join Methods

- For joins, two join methods are considered:
  - 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

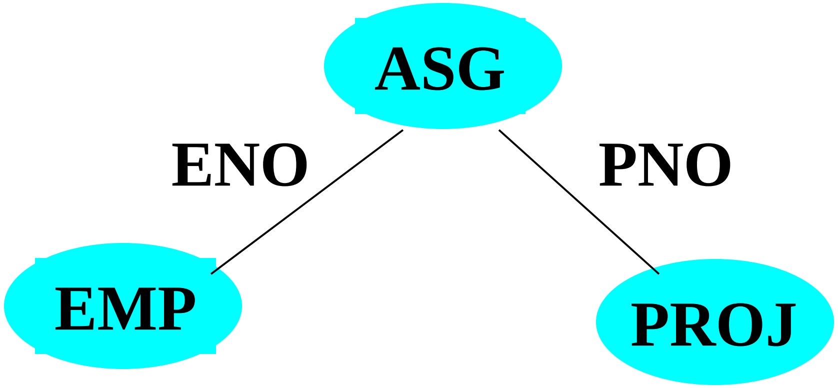
System R’s Join Methods (*cont.*)

- Hash join
  - Assume **hc** is the complexity of the hash table creation, and **hm** is the complexity of the hash match function.
  - The complexity of the Hash join is **O(N*hc + M*hm + J)**, where **N** is the smaller data set, **M** is the larger data set, and **J** is a complexity addition for the dynamic calculation and creation of the hash function.

System R Algorithm - Example

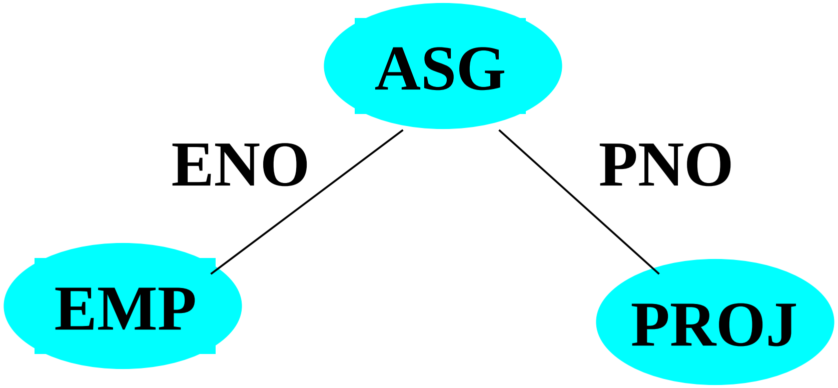
Find names of employees working on the CAD/CAM project.

- Assume
  - EMP has an index on ENO
  - ASG has an index on PNO
  - PROJ has an index on PNO and an index on PNAME



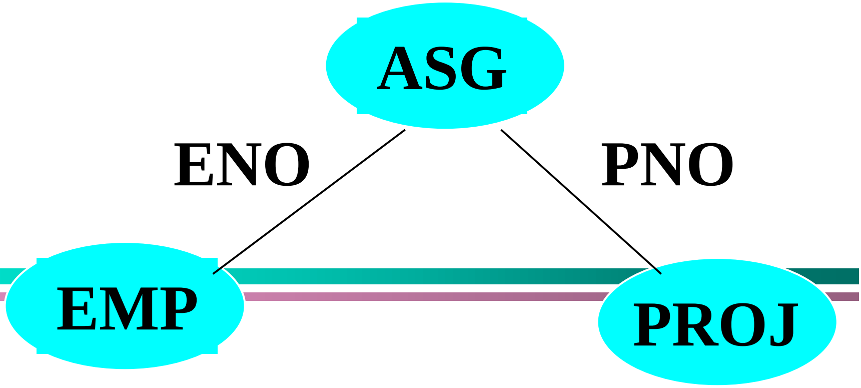
System R Example (*cont.*)

- 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)
- Determine the best join ordering
  - EMP $\bowtie$ ASG $\bowtie$ PROJ
  - ASG $\bowtie$ PROJ $\bowtie$ EMP
  - PROJ $\bowtie$ ASG $\bowtie$ EMP
  - ASG $\bowtie$ EMP $\bowtie$ PROJ
  - EMP × PROJ $\bowtie$ ASG
  - PROJ × EMP $\bowtie$ ASG

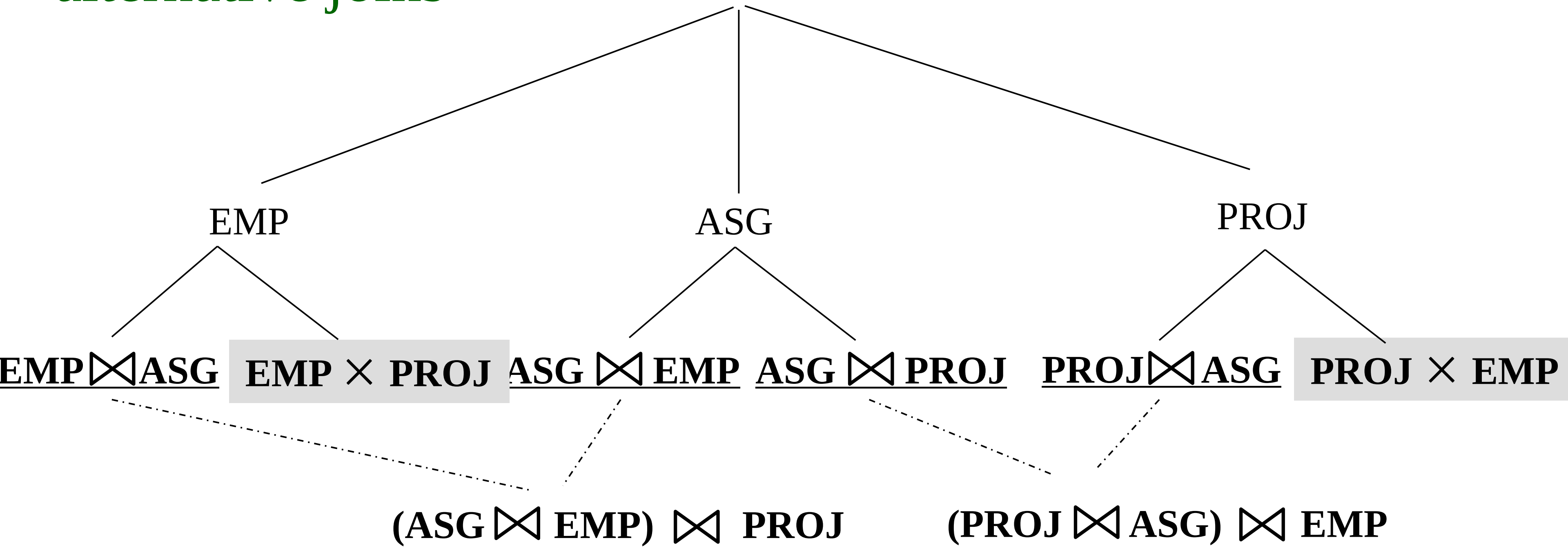


Select the best ordering based on the join costs evaluated according to the two join methods

System R Example



alternative joins



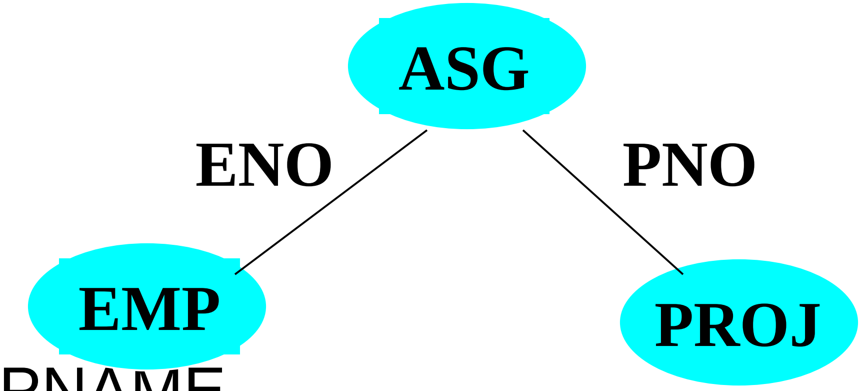
- Best total join order is one of

(ASG ⋈ EMP) ⋈ PROJ

(PROJ ⋈ ASG) ⋈ EMP

System R Example (*cont.*)

- (PROJ ⋈ ASG) ⋈ EMP has a useful index on the select attribute and direct access to the join attributes of ASG and EMP.



- Final plan:
  - select PROJ using index on PNAME
  - then join with ASG using index on PNO
  - then join with EMP using index on ENO

Join Ordering in Fragment Queries

- Join ordering is important in centralized DB, and is more important in distributed DB.
- Assumptions necessary to state the main issues
  - Fragments and relations are indistinguishable;
  - Local processing cost is omitted;
  - Relations are transferred in one-set-at-a-time mode;
  - Cost to transfer data to produce the final result at the result site is omitted

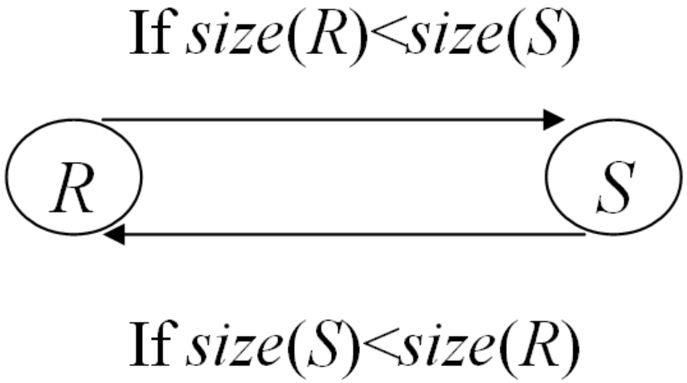
Join Ordering in Fragment Queries (*cont*.)

- Join ordering
  - Distributed INGRES
  - System R*

- Semijoin ordering
  - SDD-1

Join Ordering

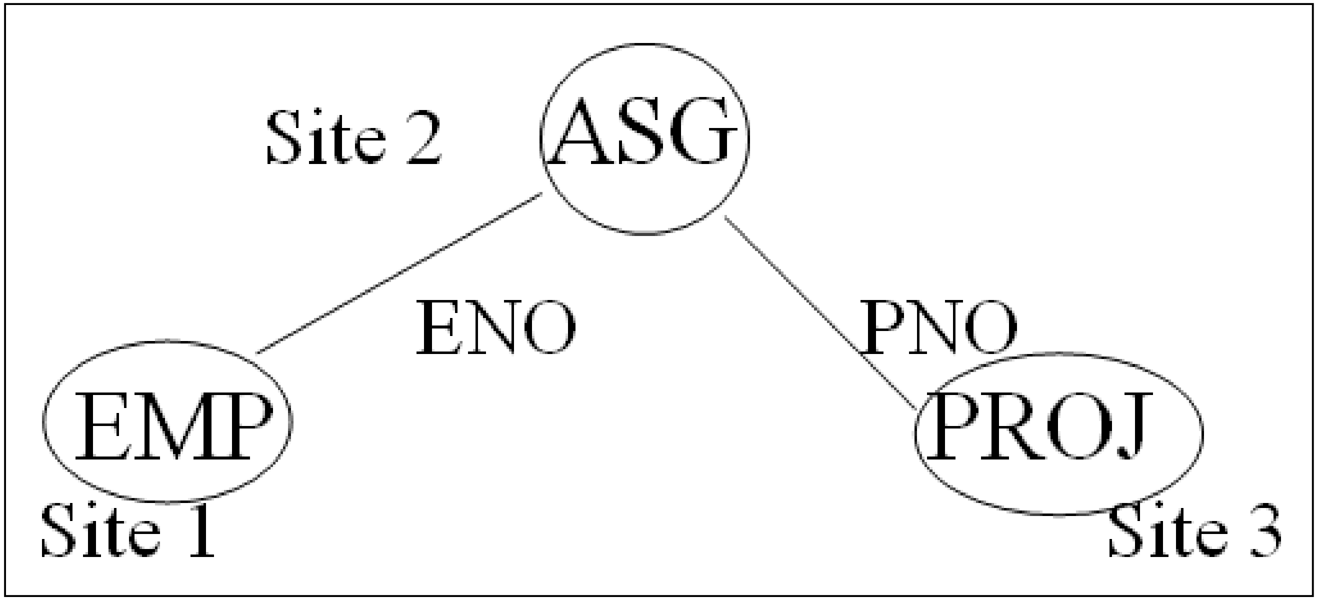
- Consider two relations only
  - $R \bowtie S$
  - Transfer the smaller size



- Multiple relations more difficult because too many alternatives
  - Compute the cost of all alternatives and select the best one
    - Necessary to compute the size of intermediate relations which is difficult.
    - Use heuristics

Join Ordering - Example

Consider: $\mathbf{PROJ} \bowtie_{\mathbf{PNO}} \mathbf{ASG} \bowtie_{\mathbf{ENO}} \mathbf{EMP}$



Join Ordering – Example (*cont.*)

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

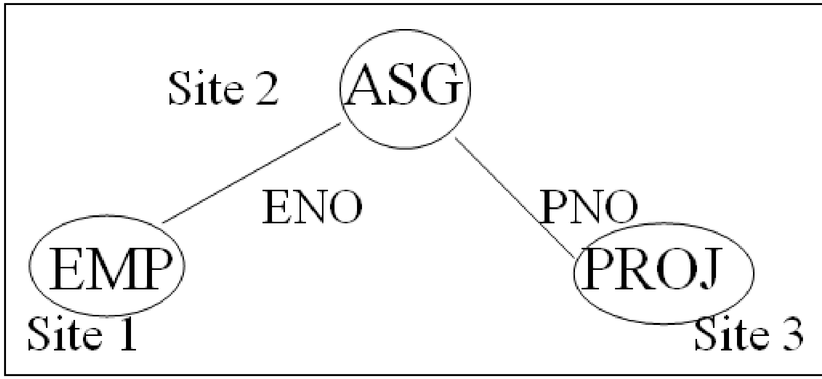
- Execution alternatives:

1. EMP → Site 2

   Site 2 computes EMP′=EMP⋈ASG

   EMP′ → Site 3

   Site 3 computes EMP′⋈PROJ



2. ASG → Site 1

   Site 1 computes EMP′=EMP⋈ASG

   EMP′ → Site 3

   Site 3 computes EMP′⋈PROJ

Join Ordering – Example (*cont.*)

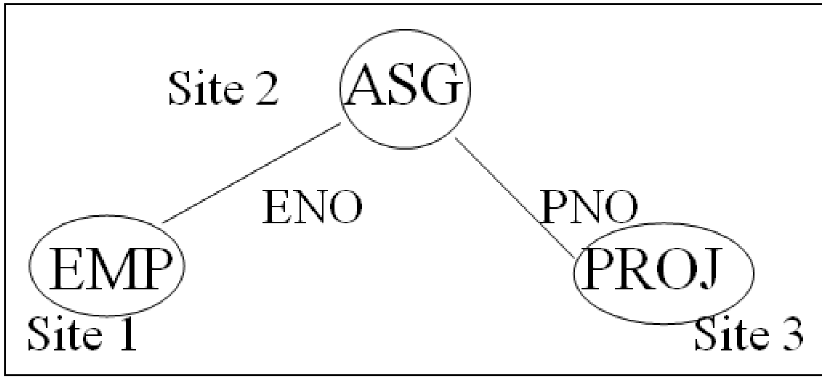
PROJ $\bowtie_{PNO}$ ASG $\bowtie_{ENO}$ EMP

3. ASG → Site 3

   Site 3 computes ASG′=ASG⋈PROJ

   ASG′ → Site 1

   Site 1 computes ASG′⋈EMP



4. PROJ → Site 2

   Site 2 computes PROJ′=PROJ⋈ASG

   PROJ′ → Site 1

   Site 1 computes PROJ′ ⋈ EMP

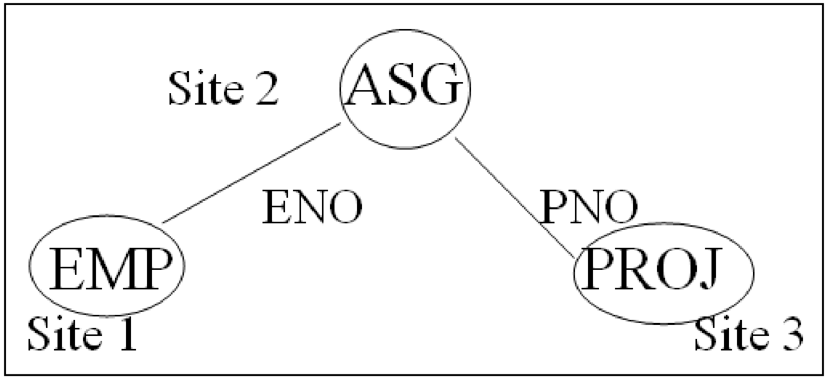
Join Ordering – Example (*cont.*)

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

5. EMP → Site 2

   PROJ → Site 2

   Site 2 computes EMP⋈ ROJ ⋈ASG



Semijoin Strategy

- Shortcoming of the joining method
  - Transfer the entire relation which may contain some useless tuples
  - Semi-join reduces the size of operand relation to be transferred.

- Semi-join is beneficial if the cost to produce and send to the other site is less than sending the whole relation.

Semijoin Strategy (*cont.*)

- Consider the join of two relations
  - R[A] (located at site 1)
  - S[A] (located at site 2)

- Alternatives

1. Do the join $R \bowtie_A S$

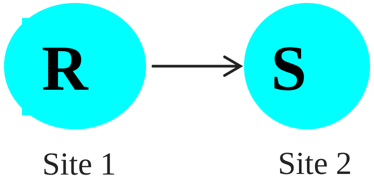
2. Perform one of the semijoin equivalents

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

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

Semijoin Strategy (*cont.*)

- 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′ → Site 1
  - Site 1 computes $R' = (R \ltimes_A S')$
  - R′ → Site 2
  - Site 2 computes $R' \bowtie_A S$

Semijoin is better if $(size(\Pi_A(S)) + size(R \ltimes_A S)) < size(R)$

Distributed INGRES Algorithm

- Same as the centralized version except
  - Movement of relations (and fragments) need to be considered
  - Optimization with respect to communication cost or response time possible

R* Algorithm

- Cost function includes local processing as well as transmission
- Consider only joins
- Exhaustive search
- At compilation time

R* Performing Join

$R \bowtie_A S$

- Ship whole
  - larger data transfer
  - smaller number of messages
  - better if relations are small
- Fetch as needed
  - number of messages = *O*(cardinality of external relation)
  - data transfer per message is minimal
  - better if relations are large and the selectivity is good

R* Strategy 1 - Vertical Partitioning & Joins

Move the entire outer relation to the site of the inner relation.

The outer tuples can be joined with inner ones as they arrive

$R \bowtie_A S$

(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. of outer tuples fetched $*$avg.

outer tuple size) / msg. size

R* Strategy 2 - Vertical Partitioning & Joins

Move inner relation to the site of outer relation.

The inner tuples cannot be joined as they arrive, and they need to be stored in a temporary relation.

$R \bowtie_A S$

Total Cost = cost(retrieving qualified outer tuples)

+ cost(retrieving qualified inner tuples)

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

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

+ msg. cost $*$ (no. of inner tuples fetched $*$ avg. inner tuple size) / msg. size

R* Strategy 3 - Vertical Partitioning & Joins

Fetch inner tuples as needed for each tuple of the outer relation. For each outer tuple in R, the join attribute value is sent to the site of S. Then the inner tuples of S which match that value are retrieved and sent to the site of R to be joined as they arrive.

$R \bowtie_A S$

(a) Retrieve qualified tuples at outer relation site

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

(c) Retrieve matching inner tuples at inner relation site

$S' = S \ltimes_A (\prod_A(R))$

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

(e) Join as they arrive

R* Strategy 3 - Vertical Partitioning & Joins (*cont*.)

$R \bowtie_A S$

Total Cost = cost(retrieving qualified outer tuples)

+ msg. cost $*$ (no. of outer tuples fetched $*$ avg. outer tuple size) / msg. size

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

+ msg. cost $*$ (no. of inner tuples fetched $*$ avg. inner tuple size) / msg. size

R* Strategy 4 - Vertical Partitioning & Joins (*cont.*)

Move both inner and outer relations to another site. $R \bowtie_A S$

The inner tuples are stored in a temporary relation.

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)

Hill Climbing Algorithm

Assume join is between three relations.

Step 1: Do initial processing

Step 2: Select initial feasible solution (ES_0)

2.1 Determine the candidate result sites – sites where a relation referenced in the query exists

2.2 Compute the cost of transferring all the other referenced relations to each candidate site

2.3 ES_0 = candidate site with minimum cost

Hill Climbing Algorithm (*cont*.)

Step 3: Determine candidate splits of ES_0 into $\{ES_1, ES_2\}$

3.1 ES_1 consists of sending one of the relations to the other relation's site

3.2 ES_2 consists of sending the join of the relations to the final result site

Step 4: Replace ES_0 with the split schedule which gives

$cost(ES_1) + cost(\text{local join}) + cost(ES_2) < cost(ES_0)$

Hill Climbing Algorithm (*cont*.)

Step 5: Recursively apply steps 3–4 on ES_1 and ES_2 until no such plans can be found

Step 6: Check for redundant transmissions in the final plan and eliminate them.

Hill Climbing Algorithm - Example

What are the salaries of engineers who work on the CAD/CAM project?

$\Pi_{SAL}(PAY \bowtie_{TITLE}(EMP \bowtie_{ENO} (ASG \bowtie_{PNO}(\sigma_{PNAME=\text{“CAD/CAM”}} PROJ))))$

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

Assume:

- Size of relations is defined as their cardinality
- Transmission cost between two sites is 1
- Ignore local processing cost

Hill Climbing – Example (*cont.*)

Step 1: Do initial processing

Selection on PROJ; result has cardinality 1

$$\Pi_{SAL}(PAY \bowtie_{TITLE}(EMP \bowtie_{ENO} (ASG \bowtie_{PNO}(\sigma_{PNAME=\text{“CAD/CAM”}} PROJ))))$$

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

Hill Climbing – Example

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

Step 2: Initial feasible solution

Alternative 1: Resulting candidate site is Site 1

Total cost = *cost*(PAY→Site 1) + *cost*(ASG→Site 1) + *cost*(PROJ→Site 1) = 4 + 10 + 1 = 15

Alternative 2: Resulting candidate site is Site 2

Total cost = 8 + 10 + 1 = 19

Alternative 3: Resulting candidate site is Site 3

Total cost = 8 + 4 + 10 = 22

Alternative 4: Resulting candidate site is Site 4

Total cost = 8 + 4 + 1 = $\boxed{13}$

Therefore ES_0 = {EMP → Site 4; PAY → Site 4; PROJ → Site 4}

Hill Climbing – Example

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

Step 3: Determine candidate splits

- Alternative 1: $\{ES_1, ES_2, ES_3\}$ where
  - ES_1: EMP → Site 2
  - ES_2: (EMP ⋈ PAY) → Site 4
  - ES_3: PROJ → Site 4
- Alternative 2: $\{ES_1, ES_2, ES_3\}$ where
  - ES_1: PAY → Site 1
  - ES_2: (PAY ⋈ EMP) → Site 4
  - ES_3: PROJ → Site 4

$$\Pi_{SAL}(\text{PAY} \bowtie_{\text{TITLE}}(\text{EMP} \bowtie_{\text{ENO}} (\text{ASG} \bowtie_{\text{PNO}}(\sigma_{\text{PNAME=“CAD/CAM”}} \text{PROJ}))))$$

Hill Climbing – Example

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

Step 4: Determine costs of each split alternative

cost(Alternative 1) = *cost*(EMP→Site 2) +

cost((EMP ⋈ PAY)→Site 4) + *cost*(PROJ → Site 4)

= 8 + 8 + 1 = 17

cost(Alternative 2) = *cost*(PAY→Site 1) +

cost((PAY ⋈ EMP)→Site 4) + *cost*(PROJ → Site 4)

= 4 + 8 + 1 = 13

Decision : DO NOT SPLIT

Step 5: ES_0 is the “best”.

Step 6: No redundant transmissions.

Comments on Hill Climbing Algorithm

- Greedy algorithm → determines an initial feasible solution and iteratively tries to improve it

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

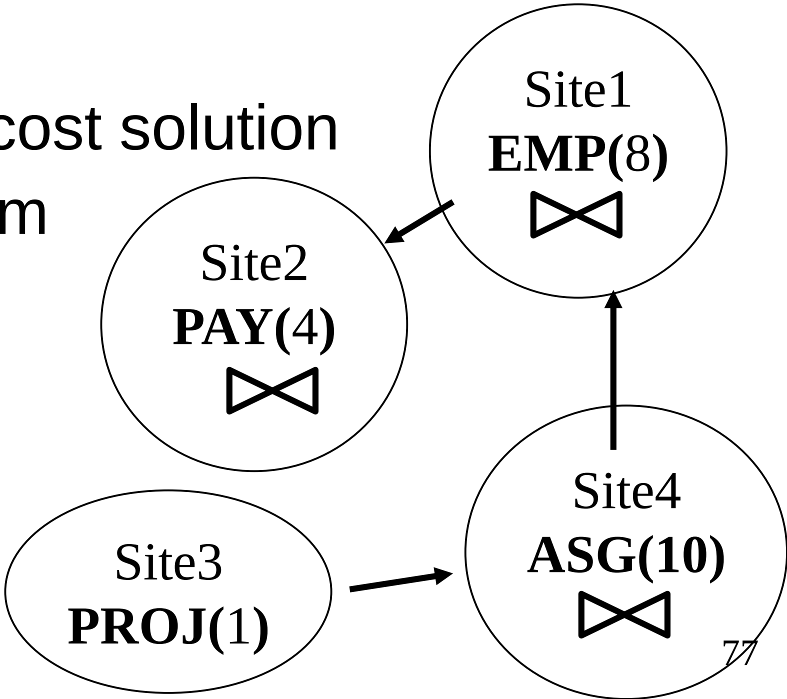
- Problem
  - Strategies with higher initial cost, which could produce better overall benefits, are ignored
  - May get stuck at a local minimum cost solution and fail to reach the global minimum
  - E.g., a better solution (ignored)

    PROJ → Site 4

    ASG′= (PROJ ⋈ ASG) → Site 1

    (ASG′ ⋈ EMP) → Site 2

    Total cost = 1 + 2 + 2 = 5



SDD-1 Algorithm

- SDD-1 algorithm improves the hill-climbing algorithm by making extensive use of semijoins
  - The objective function is expressed in terms of total communication time
    - Local time and response time are not considered
  - Using statistics on the database
    - Where a profile is associated with a relation
- The improved version also selects an initial feasible solution that is iteratively refined.

Distributed Query Optimization Problems

- Cost model
  - multiple query optimization
  - heuristics to cut down on alternatives
- Larger set of queries
  - optimization only on select-project-join queries
  - also need to handle complex queries (e.g., unions, disjunctions, aggregations and sorting)
- Optimization cost vs. execution cost tradeoff
  - heuristics to cut down on alternatives
  - controllable search strategies
- Optimization/re-optimization interval
  - extent of changes in database profile before re-optimization is necessary

Question & Answer