Distributed Data Processing

- Introduction
- Distributed DBMS Architecture
- Distributed DB Design
- Semantic Data Control
- **Query Processing**
- Transaction Management

Global Query Optimization

Input: Algebra query on fragments

- Find the ***best*** (not necessarily optimal) global schedule, that is to find the best ordering of operations in the fragment query, including communication operations which minimize a cost function
  - Minimize a cost function
    - Available statistics on fragments
  - Distributed join processing
    - Decide on the use of semijoins
    - Join methods

Output: optimized (best) algebraic query with communication operations included on fragments

Local Query Optimization

Input: Best global execution schedule

- Select the **best access path** by all the site
- Use the **centralized** optimization techniques

7. Optimization of Distributed Queries

- Query Optimization
- Centralized Query Optimization
- Join Ordering in Fragment Queries
- Distributed Query Optimization Algorithms
- Local Optimization
- Conclusion

Objective of optimizer

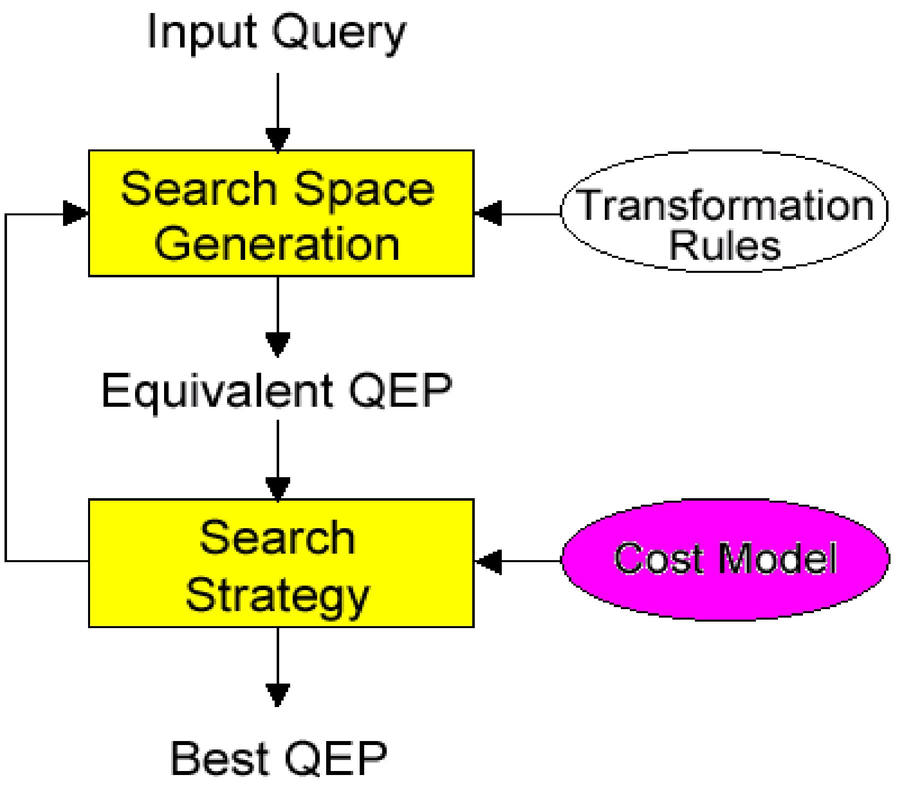
- Finding an “optimal” ordering of operations for a given query
  - Selecting the optimal execution strategy for a query is NP-hard in the number of relations
    - The actual objective is to find a strategy **close to optimal**
  - Input to optimizer
    - query on fragments
    - fragment statistics, formulas for estimating the cardinalities of results of relational operations
  - Focus mostly on the ordering of **join** operation
    - It is a well-understood problem, and queries involving joins, selections, and projection are usually considered to be the most frequent type
    - It is easier to generalize the basic algorithm for other binary operations, such as unions.

7.1 Query Optimization

Query Optimization

- Solution space
  - The set of equivalent algebra expressions (operator trees).
- Cost function (in terms of time)
  - Total time (Response time): I/O cost + CPU cost + communication cost
  - These might have different weights in different distributed environments (LAN vs WAN).
- Search algorithm
  - How do we move inside the solution space?
  - Exhaustive search, heuristic algorithms

Query Optimization Process

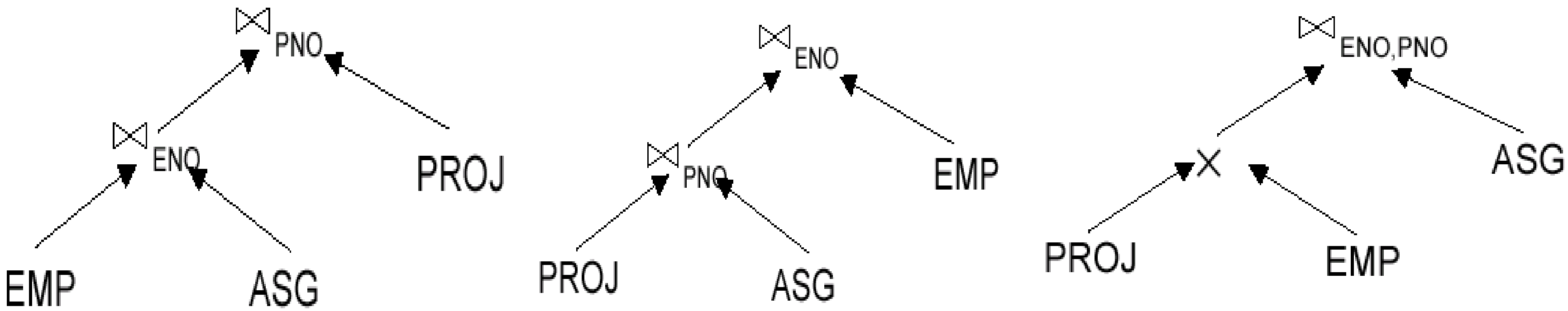


7.1.1 Search Space

- **Search space** characterized by alternative execution plans
  - could be expressed as operator trees
- Focus on **join trees**
  - Operator tree whose operators are join or Cartesian product
  - Because permutations of the join order have the most important effect on performance of queries

Example

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

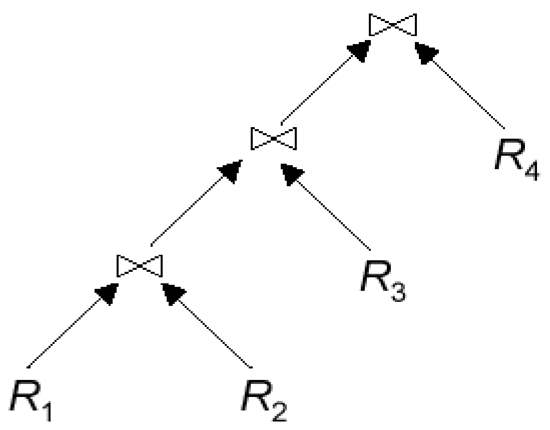


Restrict the size of the search space

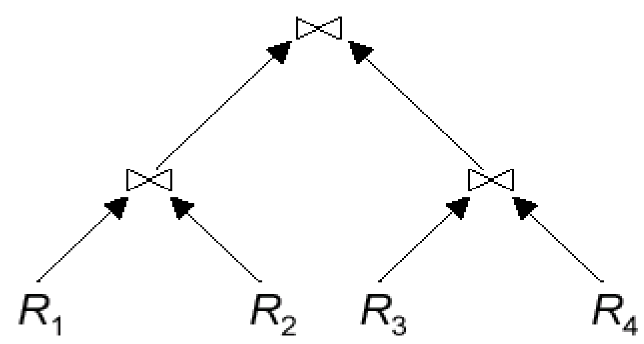
- Restrict by means of heuristics
  - Perform unary operations before binary operations
  - Avoid Cartesian product
- Restrict the shape of the join tree
  - Consider only **linear trees**, ignore **bushy trees**

Linear tree *vs.* Bushy tree

Linear Join Tree



Bushy Join Tree



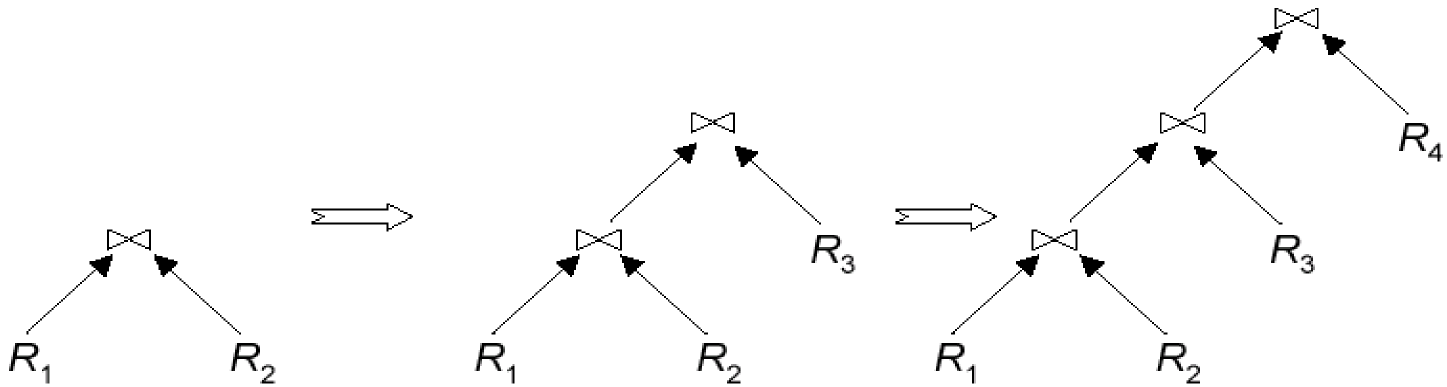
7.1.2 Search Strategy

How to “move” in the search space.

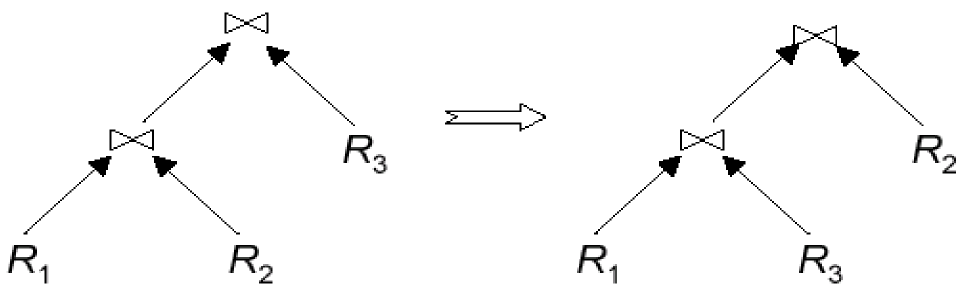
- Deterministic
  - Start from base relations and build plans by adding one relation at each step until complete plans are obtained
  - Dynamic programming: breadth-first
  - Greedy: depth-first
- Randomized
  - Search for optimalities around a particular starting point
  - Trade optimization time for execution time
  - Better when > 5-6 relations

Deterministic *vs.* Randomized

Deterministic



Randomized



7.1.3 Distributed Cost Model

- Cost function
  - To predict the cost of operators
- Database statistics
  - About the base relations and formulas to evaluate the sizes of intermediate results

Cost Function

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

    => Increase system throughput
- Response Time
  - Do as many things as possible in parallel
    - May increase total time because of increased total activity
  - Optimizes the degree of parallel execution

    => Improve user's response time

Total cost

Summation of all cost factors

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

- **CPU cost** =

  unit instruction cost *number of instructions

- **I/O cost** =

  unit disk I/O cost * number of disk I/Os

- **communication cost** =

  message initiation + transmission

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 * number of *sequential* instructions

- **I/O time** =

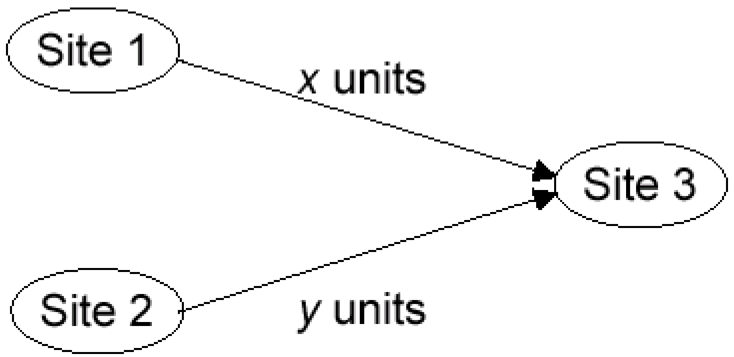
  unit I/O time * number of *sequential* I/Os

- **communication time** =

  unit msg initiation time * number of *sequential* msg +

  unit transmission time * number 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*

Database statistics

- Primary cost factor:

  size of intermediate relations

- The estimation is based on statistical information about the **base relations** and **formulas** to predict the cardinalities of the results of the relational operations
- Make them precise more costly to maintain

Database statistics – Base relation

- For each relation R[*A*1, *A*2, …, *An*] fragmented as *R*1, …, *Rr*
  - length of each attribute: *length(Ai)*
  - the number of distinct values for each attribute in each fragment: $card(\Pi_{Ai}\, Rj)$
  - maximum and minimum values in the domain of each attribute: *min(Ai), max(Ai)*
  - the cardinalities of each domain: *card(dom[Ai])*
  - the cardinalities of each fragment: *card(Rj)*
- **Selectivity factor** of each operation for relations
  - the proportion of tuples of an operand relation that participate in the result of that operation
  - For joins

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

Database statistics – Cardinalities of intermediate results

- Assumptions
  - The distribution of attribute values in a relation is supposed to be uniform
  - All attributes are independent

Cardinalities of intermediate results

Selection

$$size(R) = card(R) * length(R)$$
$$card(\sigma_F(R)) = SF_\sigma(F) * card(R)$$

where

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

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

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

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

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

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

Cardinalities of intermediate results

Projection

$card(\Pi_A(R)) = card(R)$ If one of the projected attributes is a key of R

Cartesian Product

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

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

Cardinalities of intermediate results

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:

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

Semijoin

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

where

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

7.2 Centralized Query Optimization

Centralized Query Optimization

- Centralized query optimization is a **simpler** problem
- Distributed query optimization techniques are often **extensions** of the techniques for centralized system
- A distributed query is translated into **local queries**, each of which is processed in a centralized way

Two popular relational database

- INGRES
  - dynamic
- System R
  - static
  - exhaustive search

7.2.1 INGRES Algorithm

Dynamic query optimization

- **Combine the two phase of decomposition and optimization**

  Step1: Decompose each multi-variable query into a sequence of mono-variable queries with a common variable

  Step2: 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

Decomposition

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

$q1 \rightarrow q2 \rightarrow \ldots \rightarrow qn$

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

    $q(V1, V2, \ldots Vn) \rightarrow (q'(t1, V2, V3, \ldots, Vn), t1 \in R)$

Detachment

```
q:    SELECT  R2.A2, R3.A3, …, Rn.An
      FROM    R1, R2, …, Rn
      WHERE   P1(R1.A1')
      AND     P2(R1.A1, R2.A2, …, Rn.An)


q':   SELECT  R1.A1 INTO R1'
      FROM    R1
      WHERE   P1(R1.A1')

q'':  SELECT  R2.A2, R3.A3, …, Rn.An
      FROM    R1', R2, …, Rn
      WHERE   P2(R1'.A1, R2.A2, …, Rn.An)
```

Example of detachment

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"
```

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
```

Example of detachment

q':

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

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
```

Example of 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"
```

INGRES Algorithm

- Applying the **selections and projections** as soon as possible by detachment
- Results of the monorelation queries are **stored** in data structures that are capable of optimizing the later queries (such as joins)
- The irreducible queries that remain after detachment must be processed by **tuple substitution**
- For the irreducible query, the **smallest relation** whose cardinality is known from the result of the preceding query is chosen for substitution
- Monorelation queries generated by the reduction algorithm are processed by the **OVQP** that chooses the best existing access path to the relation, according to the query qualification

7.2.2 System R Algorithm

Static query optimization based on the exhaustive search of the solution space

- **INPUT**: relational algebra tree resulting from the query decomposition
- **OUTPUT**: an execution plan that implements the "optimal" relational algebra tree

- Statistical information is maintained

System R Algorithm

- To limit the overhead of optimization, the number of alternative trees is reduced using **dynamic programming**
  - The set of alternative strategies is constructed dynamically so that, when two joins are equivalent by commutatively, **only the cheapest one is kept**
  - The strategies that include **Cartesian products are eliminated** whenever possible

System R Algorithm

- **Step1**: ***the best access path*** to each individual relation based on a select predicate is predicted
- **Step2**: ***the best join ordering*** is estimated for each relation R
  - Determine the possible ordering of joins
  - Determine the cost of each ordering
  - Choose the join ordering with minimal cost

Alternatives for Join

- Nested loops

  for each tuple of ***external relation*** *(cardinality n1)*

  for each tuple of ***internal relation*** *(cardinality n2)*

  join two tuples if the join predicate is true

  end

  end

  - Complexity: $n1 * n2$
- Merge join

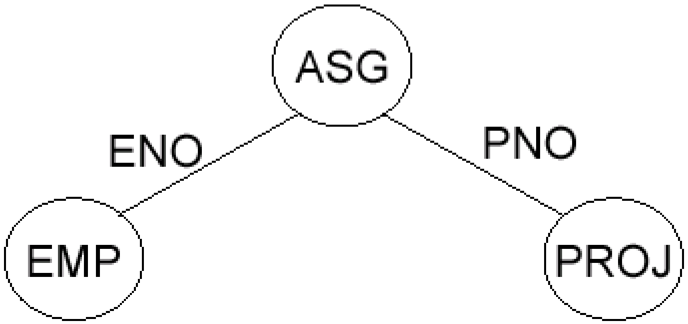
  sort relations

  merge relations

  - Complexity: $n1 + n2$ if relations are previously sorted and equijoin

Example of System R Algorithm

- **Query:** 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
  - (EMP ∞ASG) and (ASG ∞ PROJ) have a cost higher than (ASG ∞ EMP) and (PROJ∞ASG)



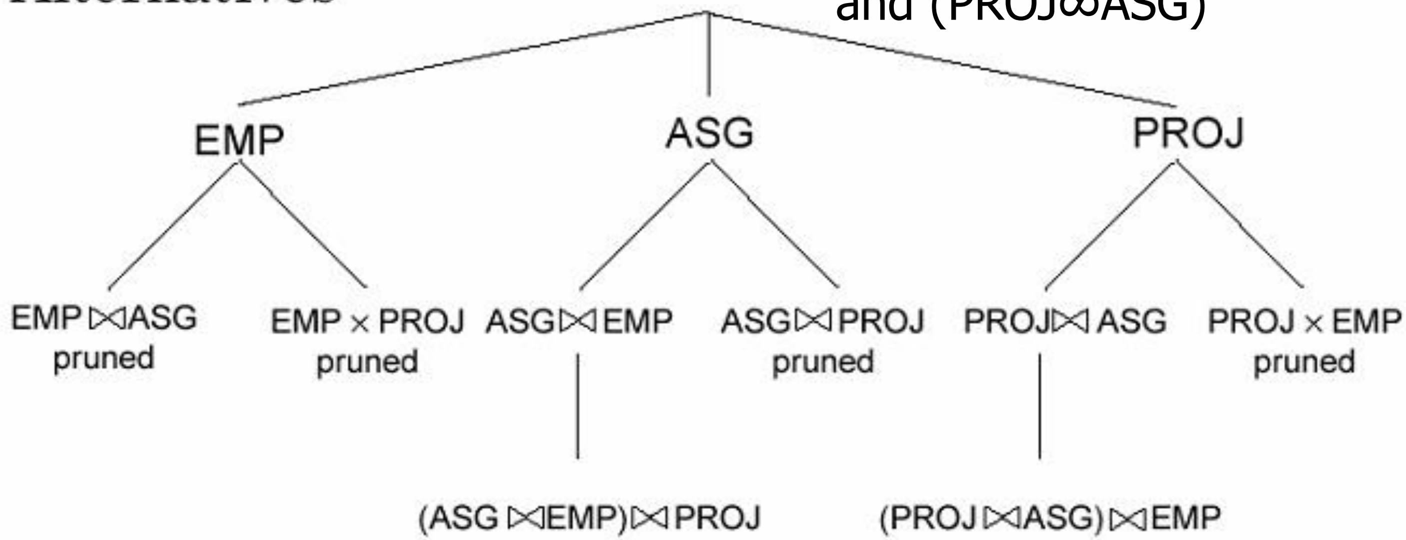
Example of System R Algorithm

- Choose the best access paths to each relation
  - EMP: sequential scan (no selection on EMP based on ENO)
  - ASG: sequential scan (no selection on ASG based on PNO)
  - PROJ: index on PNAME (there is a selection on PROJ based on PNAME)
- Determine the best join ordering
  - EMP ∞ ASG ∞ PROJ
  - ASG ∞ PROJ ∞ EMP
  - PROJ ∞ ASG ∞ EMP
  - ASG ∞ EMP ∞ PROJ
  - EMP x PROJ ∞ ASG
  - PROJ x EMP ∞ ASG
  - Select the best ordering based on the join costs evaluated according to the two methods

Example of System R Algorithm

- (EMP ∞ASG) and (ASG ∞ PROJ) have a cost higher than (ASG ∞ EMP) and (PROJ∞ASG)

Alternatives



Best total join order is one of

((ASG ⋈ EMP) ⋈ PROJ)

((PROJ ⋈ ASG) ⋈ EMP)

Example of System R Algorithm

- $((PROJ \infty ASG) \infty EMP)$ 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

System R Algorithm

- To select the best single-relation access method to each relation in the query
- To examine all possible permutations of join orders and select the best access strategy for the query
  - First, the join of each relation with every other relation is considered
  - Then, joins of three relations are optimized
  - The continues until joins of n relations are optimized

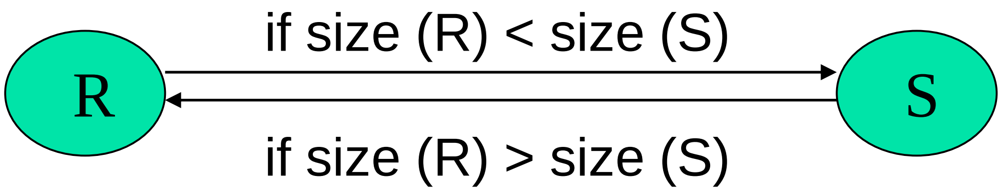
7.3 Join Ordering in Fragment Queries

Ordering Joins in Fragment Queries

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

7.3.1 Join Ordering

- Consider two relations only
  - Send the smaller relation to the site of the large one

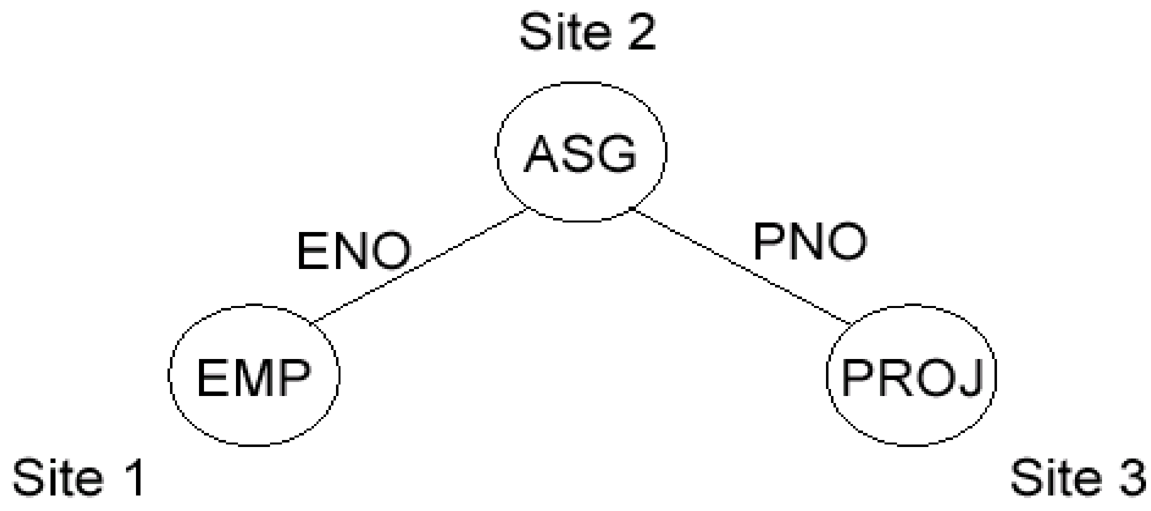


- Multiple relations more difficult because too many alternatives.
  - Compute the cost of all alternatives and select the best one.
    - Join may reduce or increase the size of the intermediate results
    - Necessary to compute the size of intermediate relations which is difficult
  - Use heuristics

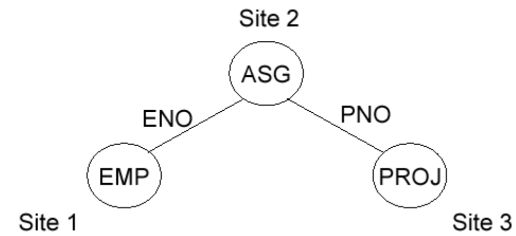
Example of Join Ordering

Consider

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



Example of Join Ordering



Execution alternatives:

1. EMP $\rightarrow$ Site 2
   Site 2 computes EMP'=EMP $\bowtie$ ASG
   EMP' $\rightarrow$ Site 3
   Site 3 computes EMP' $\bowtie$ PROJ

2. ASG $\rightarrow$ Site 1
   Site 1 computes EMP'=EMP $\bowtie$ ASG
   EMP' $\rightarrow$ Site 3
   Site 3 computes EMP' $\bowtie$ PROJ

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

4. PROJ $\rightarrow$ Site 2
   Site 2 computes PROJ'=PROJ $\bowtie$ ASG
   PROJ' $\rightarrow$ Site 1
   Site 1 computes PROJ' $\bowtie$ EMP

5. EMP $\rightarrow$ Site 2
   PROJ $\rightarrow$ Site 2
   Site 2 computes EMP $\bowtie$ PROJ $\bowtie$ ASG

7.3.2 Semijoin Based Algorithms

- Consider the join of two relations:
  - R[A] (located at site 1)
  - S[A](located at site 2)
- Alternatives:
  1. Do the join $R \infty_A S$
  2. Perform one of the semijoin equivalents

$$R \infty_A S \Leftrightarrow (R \propto_A \mathrm{S}) \infty_A S$$
$$\Leftrightarrow R \infty_A (S \propto_A \mathrm{R})$$
$$\Leftrightarrow (R \propto_A \mathrm{S}) \infty_A (S \propto_A \mathrm{R})$$

Join *vs.* Semijoin Algorithms

- Perform the join
  - send R to Site 2
  - Site 2 computes $R \bowtie_A S$
- Consider semijoin $(R \ltimes_A S) \bowtie_A S$
  - $S' \leftarrow \prod_A(S)$
  - $S' \rightarrow$ Site 1
  - Site 1 computes $R' = R \ltimes_A S'$
  - $R' \rightarrow$ Site 2
  - Site 2 computes $R' \bowtie_A S$

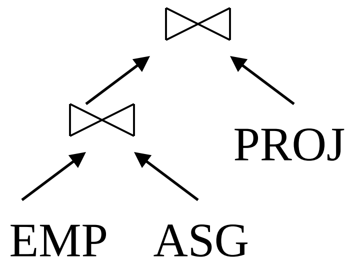
Semijoin is better if

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

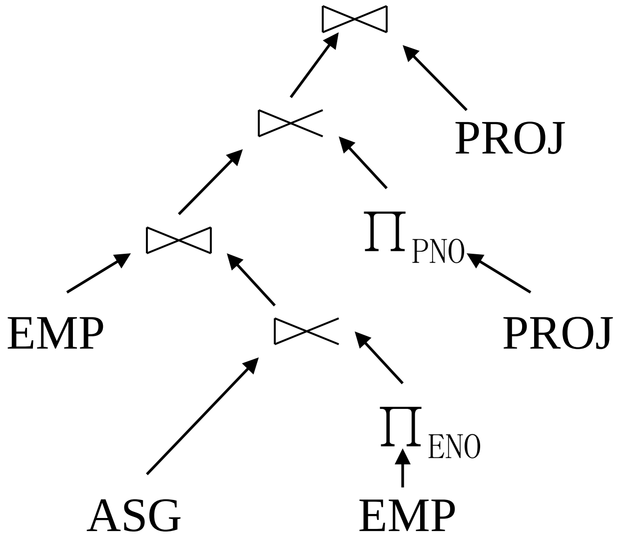
Join *vs.* Semijoin

- **Semijoin may increase the local processing time**
- **However, if**
  - The join attribute length is smaller than the length of an entire tuple
  - The semijoin has good selectivity

Then the semijoin approach can result in significant savings in communication time



Join approach



Semijoin approach

7.4 Distributed Query Optimization Algorithms

Three Basic Distributed Query Optimization algorithms

- Distributed INGRES
  - Ordering joins
- R*
  - Ordering joins
- SDD-1
  - Semijoin ordering

Distributed Query Optimization Algorithms

Algorithms	Opt. Timing	Objective Function	Opt. Factors	Network Topology	Semijoin	Stats	Fragments
Dist. INGRES	Dynamic	Resp. time or Total time	Msg. Size, Proc. Cost	General or Broadcast	No	1	Horizontal
R*	Static	Total time	No. Msg., Msg. Size, IO, CPU	General or Local	No	1, 2	No
SDD-1	Static	Total time	Msg. Size	General	Yes	1,3,4, 5	No

1: relation cardinality; 2: number of unique values per attribute; 3: join selectivity factor; 4: size of projection on each join attribute; 5: attribute size and tuple size

SDD-1

7.4.1 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

Distributed INGRES Algorithm

- Dynamic
- Considers only joins
- The algorithm also takes advantage of fragmentation, but only horizontal fragmentation is handled for simplicity
- Both general and broadcast networks are considered

Distributed INGRES Algorithm

Input: MRQ: multirelation query

Output: result of the last multirelation query

Begin

Run all detachable one-relation queries in MRQ;

Replace MRQ by a list of n irreducible queries MRQ'_list;

Repeat

***choose** next irreducible query MRQ' from MRQ'_list involving the smallest fragments ;*

***Determine** fragments to transfer and processing site for MRQ';*

Move the selected fragments to the selected sites;

Run MRQ';

Until MRQ'_list empty

End

Optimization in Distributed INGRES Algorithm

- **choose** next irreducible query
  - *Having no predecessor and involving the smaller fragments*
- **Determine** fragments to transfer and processing site
  - based on communication cost
  - *Assuming that one relation Rp is the remaining fragmented, and K sites participate in processing, then the fragments of Ri have to be moved as follows:*
    - *For a processing site j, Rij is moved to k-1 other sites*
    - *For a non-processing site j, Rij is moved to k other sites and any fragments of Rpj are moved to any one processing site*
  - *broadcast*
    - *CTk(#bytes) = CT1(#bytes)*
  - *point—to—point*
    - *CTk(#bytes) = k*CT1(#bytes)*

Example of Distributed INGRES Algorithm

- PROJ ∞ ASG

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

- Point-to-point
  - Send each PROJi to site 3 (cost:3000)
- Broadcast
  - Send ASG (in a single transfer) to 1,2,4 (cost:2000)

7.4.2 R* Algorithm

- **Exhaustive search**
- **Compilation**
- **Considers only joins**
- **Cost function includes local processing as well as transmission**
- **Published papers provide solutions to handling horizontal and vertical fragmentations but the implemented prototype does not**

R* Algorithm

Input: QT: Query tree

Output: minimum cost strategy

Begin

For each relation $R_i \in QT$ do

Get the best_AP$_i$ from all access path to R_i

For each order do

Get the best join order from all the orders

For each site k storing a relation involved in QT do

$LS_k \leftarrow$ local strategy

Send(LS_k, site k)

end

R* Algorithm

- Select the join ordering
- Join algorithm
  - Nested loop
  - Merge join
- Access path for each fragment
  - Index
  - Sequential scan

- Select the sites of join results
- Method of transferring data between sites

Performing joins – transfer method

- 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

Strategy 1

- Move entire external relation to the site of the internal relation
  - *Retrieve external relation R tuples*
  - *Send them to the internal relation S site*
  - *Join them as they arrive*

  Total Cost = LT(retrieve card(R) tuples from R)
  + CT(size(R))
  + LT(retrieve card($S \propto_A R$) tuples from S)

Strategy 2

- Ship the entire internal relation to the site of the external relation
  - *Cannot join as they arrive; they need to be stored*

Total Cost = LT(retrieve card(S) tuples from S)
+ CT(size(S))
+LT(store card(S) tuple in T)
+LT(retrieve card(R) tuples from R)
+ LT(retrieve card($S \propto_A R$) tuples from T)

Strategy 3

- Fetch internal tuples as needed
  - *Retrieve qualified tuples at external relation site*
  - *Send request containing join column values of external relation to internal relation site*
  - *Retrieve matching internal tuples at internal relation site*
  - *Send the matching internal tuples to external relation site*
  - *Join as they arrive*

Total Cost = LT(retrieve card(R) tuples from R)
+ CT(length(A)*card(R))
+ LT(retrieve card($S \propto_A R$) tuples from T)
+ CT(card($S \propto_A R$) *length(S))

Strategy 4

- Move both relations to another site

Total Cost = LT(retrieve card(S) tuples from S)
+ CT(size(S))
+LT(store card(S) tuple in T)
+LT(retrieve card(R) tuples from R)
+ CT(size(R))
+ LT(retrieve card($S \propto_A R$) tuples from T)

R* Algorithm

- Predict the total time of each strategy
- Selects the cheapest.

Example of R* Algorithm

PROJ ∞_{PNO} ASG

(Assume: two site; ASG: Index on PNO)

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*

7.4.3 SDD-1 Algorithm

- Based on the Hill Climbing Algorithm
  - Static
  - Semijoins
  - No replication
  - No fragmentation
  - Cost of transferring the result to the user site from the final result site is not considered

Idea – Greedy Algorithm

Refinements of an initial feasible solution are recursively computed until no more cost improvement can be made

Hill Climbing Algorithm

The first distributed query processing algorithm

Assume join is between three relations.

Step 1**: Do initial local processing**

Step 2**: Select** initial feasible solution ***(ES0)***

- Determine the candidate result sites - sites where a relation referenced in the query exist
- Compute the cost of transferring all the other referenced relations to each candidate site
- *E*S0 = candidate site with minimum cost

Hill Climbing Algorithm

Step 3**: Determine candidate** splits **of *E*S0 into {*E*S1, *E*S2 }**

- *E*S1 consists of sending one of the relations to the other relation's site
- *E*S2 consists of sending the join of the relations to the final result site

Step 4**: Replace *E*S0 with the split schedule which gives *cos*t(*E*S1) + *cos*t(local join) + *cos*t(*E*S2) < *cos*t(*E*S0)**

Step 5**: Recursively apply steps 3–4 on *E*S1 and *E*S2 until no such plans can be found**

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

Example of Hill Climbing Algorithm

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
- Minimize total cost
- Transmission cost between two sites is 1
- Ignore local processing cost

Based on join selectivities,

size(EMP∞PAY) = size(EMP),

size(PROJ∞ASG) = 2*size(PROJ),

size(ASG∞EMP) = size(ASG)

Example of Hill Climbing Algorithm

Step 1:

Selection on PROJ; result has cardinality 1

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

Example of Hill Climbing Algorithm

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

Step 2: Initial feasible solution

Alternative 1: Resulting site is Site 1

$$\begin{aligned}\text{Total cost} &= cost(\texttt{PAY} \rightarrow \text{Site 1}) + cost(\texttt{ASG} \rightarrow \text{Site 1}) + cost(\texttt{PROJ} \rightarrow \text{Site 1}) \\ &= 4 + 10 + 1 = 15\end{aligned}$$

Alternative 2: Resulting site is Site 2

$$\text{Total cost} = 8 + 10 + 1 = 19$$

Alternative 3: Resulting site is Site 3

$$\text{Total cost} = 8 + 4 + 10 = 22$$

Alternative 4: Resulting site is Site 4

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

Therefore $ES_0 = \{\texttt{EMP} \rightarrow \text{Site 4}; \texttt{PAY} \rightarrow \text{Site 4}; \texttt{PROJ} \rightarrow \text{Site 4}\}$

Example of Hill Climbing Algorithm

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 $\rightarrow$ Site 2

ES_2: (EMP $\bowtie$ PAY) $\rightarrow$ Site 4

ES_3: PROJ $\rightarrow$ Site 4

Alternative 2: $\{ES_1, ES_2, ES_3\}$ where

ES_1: PAY $\rightarrow$ Site 1

ES_2: (PAY $\bowtie$ EMP) $\rightarrow$ Site 4

ES_3: PROJ $\rightarrow$ Site 4

Example of Hill Climbing Algorithm

Step 4: Determine costs of each split alternative

$$
\begin{aligned}
cost(\text{Alternative 1}) &= cost(\texttt{EMP}\rightarrow\text{Site 2}) + cost((\texttt{EMP}\bowtie\texttt{PAY})\rightarrow\text{Site 4}) + \\
&\quad cost(\texttt{PROJ}\rightarrow\text{Site 4}) \\
&= 8 + 8 + 1 = 17 \\
cost(\text{Alternative 2}) &= cost(\texttt{PAY}\rightarrow\text{Site 1}) + cost((\texttt{PAY}\bowtie\texttt{EMP})\rightarrow\text{Site 4}) + \\
&\quad cost(\texttt{PROJ}\rightarrow\text{Site 4}) \\
&= 4 + 8 + 1 = 13
\end{aligned}
$$

Decision : DO NOT SPLIT

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

Step 5: ES_0 is the "best".

Step 6: No redundant transmissions.

Based on join selectivities,

- size(EMP∞PAY) = size(EMP),
- size(PROJ∞ASG) = 2*size(PROJ),
- size(ASG∞EMP) = size(ASG)

Hill Climbing Algorithm

Problems :

1. Greedy algorithm → determines an initial feasible solution and iteratively tries to improve it
2. If there are local minimas, it may not find global minima
3. If the optimal schedule has a high initial cost, it won't find it since it won't choose it as the initial feasible solution

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

Example : A better schedule is

PROJ → Site 4

ASG' = (PROJ ⋈ ASG) → Site 1

(ASG' ⋈ EMP) → Site 2

Total cost = 1 + 2 + 2 = 5

Based on join selectivities,

size(EMP∞PAY) = size(EMP),

size(PROJ∞ASG) = 2*size(PROJ),

size(ASG∞EMP) = size(ASG)

SDD-1

- Extensive use of semijoins
- Objective function is expressed in terms of total communication time
- The algorithm uses statistics on the database, called database profiles
- Like its predecessor hill-climbing algorithm, the SDD-1 algorithm selects locally optimal strategies

SDD-1 Algorithm

Initialization

Step 1: In the execution strategy (call it ES), include all the local processing

Step 2: Reflect the effects of local processing on the database profile

Step 3: Construct a set of beneficial semijoin operations (BS) as follows :

$$BS = \emptyset$$

For each semijoin SJ_i

$$BS \leftarrow BS \cup SJ_i \quad \text{if } cost(SJ_i) < benefit(SJ_i)$$

Cost *vs.* Benefit of Semi-Join

Cost$(R \propto_A S)$ =

$$\mathbf{T_{MSG} + T_{TR} * size(\Pi_A(S))}$$

Benefit$(R \propto_A S)$ =

$$\mathbf{(1 - SF_{\propto}(S.A)) * size(R) * T_{TR}}$$

SDD-1 Algorithm

Iterative Process

Step 4: Remove the most beneficial SJ_i from BS and append it to ES

Step 5: Modify the database profile accordingly

Step 6: Modify BS appropriately

- compute new benefit/cost values
- check if any new semijoin need to be included in BS

Step 7: If $BS \neq \emptyset$, go back to Step 4.

SDD-1 Algorithm

Assembly Site Selection

Step 8: Find the site where the largest amount of data resides and select it as the assembly site

SDD-1 Algorithm

Postprocessing

Step 9: For each R_i at the assembly site, find the semijoins of the type

$$R_i \ltimes R_j$$

where the total cost of ES without this semijoin is smaller than the cost with it and remove the semijoin from ES.

Note : There might be indirect benefits.

Step 10: Permute the order of semijoins if doing so would improve the total cost of ES.

Example of SDD-1 (SQL)

```
Select R3 . C
From R1, R2, R3
Where R1.A=R2.A
          and R2.B=R3.B
```

Example of SDD-1 (Beneficial Semi-Join)

relation	card	tuple size	relation size
R1	**30**	**50**	**1500**
R2	**100**	**30**	**3000**
R3	**50**	**40**	**2000**

	Semi-join	benefit	cost
SJ1	R2∝R1	(1-0.3)*3000=2100	36
SJ2	R2∝R3	(1-0.4)*3000=1800	80
SJ3	R1∝R2	(1-0.8)*1500=300	320
SJ4	R3∝R2	(1-1)*2000=0	400

Attribute	SFsj	Size (Пattribute)
R1.A	**0.3**	**36**
R2.A	**0.8**	**320**
R2.B	**1.0**	**400**
R3.B	**0.4**	**80**

Example of SDD-1 (DB Profile)

relation	card	tuple size	relation size
R1	**30**	**50**	**1500**
R2	**100**	**30**	**3000**
R3	**50**	**40**	**2000**

relation	card	tuple size	relation size
R1	**30**	**50**	**1500**
R2	**30**	**30**	**900**
R3	**50**	**40**	**2000**

Attribute	SFsj	Size (Πattribute)
R1.A	**0.3**	**36**
R2.A	**0.8**	**320**
R2.B	**1.0**	**400**
R3.B	**0.4**	**80**

Attribute	SFsj	Size (Πattribute)
R1.A	**0.3**	**36**
R2.A	**0.24**	**96**
R2.B	**0.3**	**120**
R3.B	**0.4**	**80**

SDD-1 Algorithm

- Initialization
- Selection of beneficial semijoins
- Assembly site selection
- Postoptimization

- Like its predecessor hill-climbing algorithm, the SDD-1 algorithm selects locally optimal strategies
  - It ignores the higher-cost semijoins which would result in increasing the benefits and decreasing the costs of other semijoins. Thus this algorithm may not be able to select the global minimum cost solution

7.5 Local Optimization

Local Optimization

Input: **Best global execution schedule**

- Select the best access path by all the site
- Use the centralized optimization techniques

7.6 Conclusion

Conclusion

- Objective function
  - Communication cost
  - Local processing cost
- Input
  - Database statistics
  - Formulas
- Join vs. Semijoin

Conclusion

- 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/reoptimization interval
  - extent of changes in database profile before reoptimization is necessary

References

- Y. Ioannidis, Query Optimization, In A. Tucker (ed.), The Computer Science and Engineering Handbook, CRC Press, 1996, pp. 1038-1054
- P. Valduriez, Semi-Join Algorithms for Distributed Database Machines, in J.-J. Schneider (ed.) Distributed Data Bases, Amsterdam: North-Holland, 1982, pp.23-37
- P. Mishra and M. H. Eich. “Join Processing in Relational Databases”, *ACM Computing Surveys*, 24(1): 63-113, March 1992.
- M. Jarke and J. Koch. “Query Optimization in Database Systems,” *Computing Surveys*, June 1984, 16(2): 227-269.
- E. Wong, Retrieving Dispersed Data from SDD-1. In Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, 1977, pp.217-235