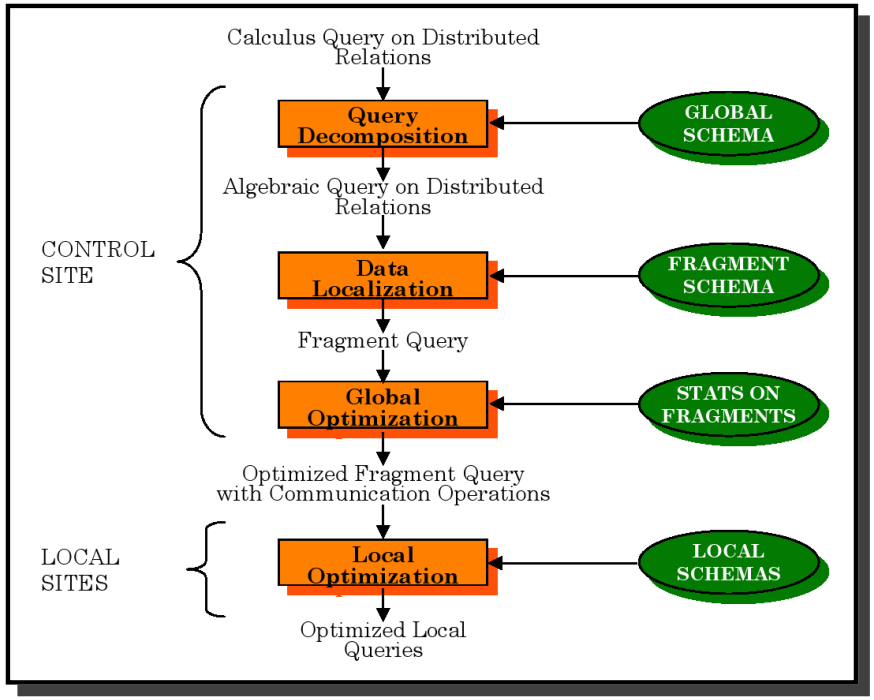
Chapter 7: Optimization of Distributed Queries

- Basic Concepts
- Distributed Cost Model
- Database Statistics
- Joins and Semijoins
- Query Optimization Algorithms

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Basic Concepts

- **Query optimization:** Process of producing an optimal (close to optimal) query execution plan which represents an execution strategy for the query
  - The main task in query optimization is to consider different orderings of the operations
- Centralized query optimization:
  - Find (the best) query execution plan in the space of equivalent query trees
  - Minimize an objective cost function
  - Gather statistics about relations

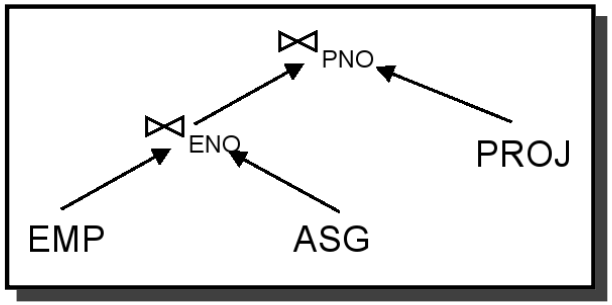


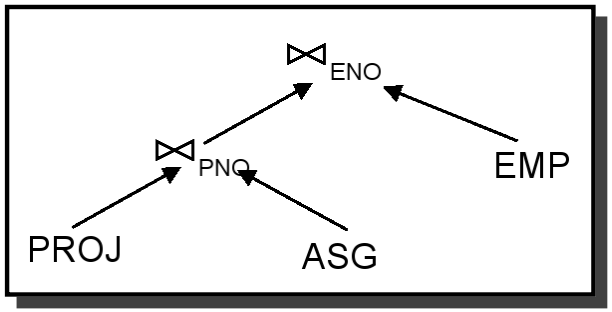
- Distributed query optimization brings additional issues
  - Linear query trees are not necessarily a good choice
  - Bushy query trees are not necessarily a bad choice
  - What and where to ship the relations
  - How to ship relations (ship as a whole, ship as needed)
  - When to use semi-joins instead of joins

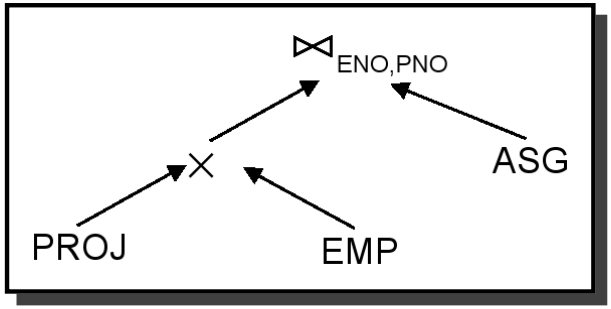
Basic Concepts ...

- **Search space:** The set of alternative query execution plans (query trees)
  - Typically very large
  - The main issue is to optimize the joins
  - For N relations, there are $O(N!)$ equivalent join trees that can be obtained by applying commutativity and associativity rules
- **Example**: 3 equivalent query trees (join trees) of the joins in the following query

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

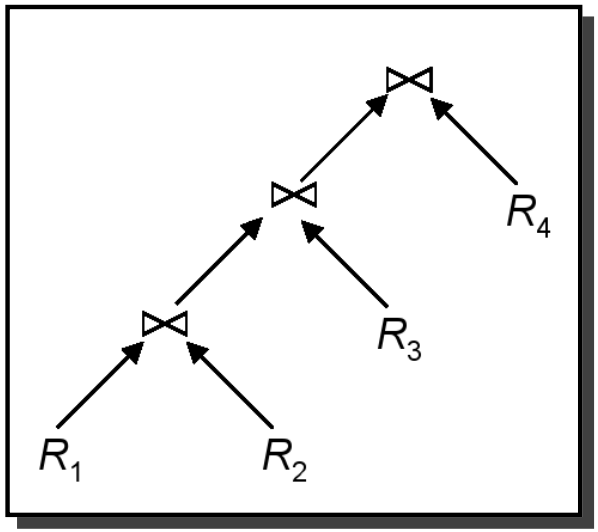




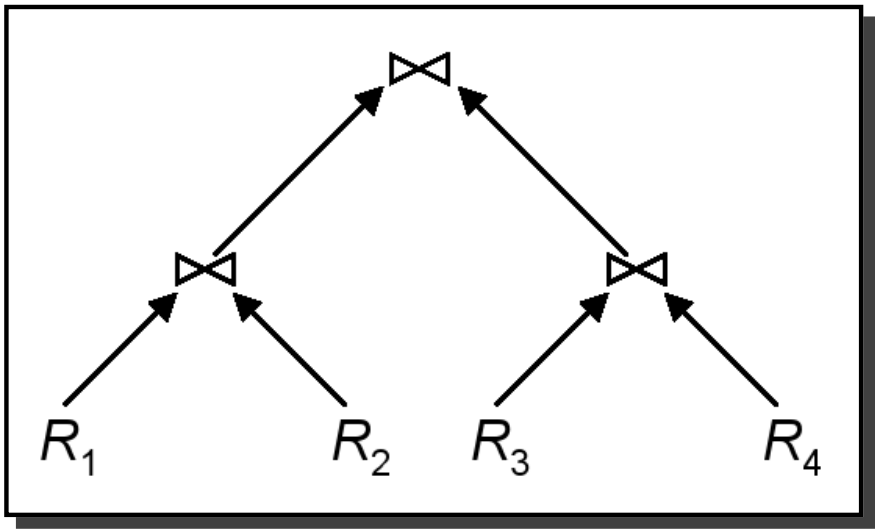


Basic Concepts . . .

- **Reduction** of the search space
  - Restrict by means of heuristics
    - Perform unary operations before binary operations, etc
  - Restrict the shape of the join tree
    - Consider the type of trees (linear trees, vs. bushy ones)



Linear Join Tree

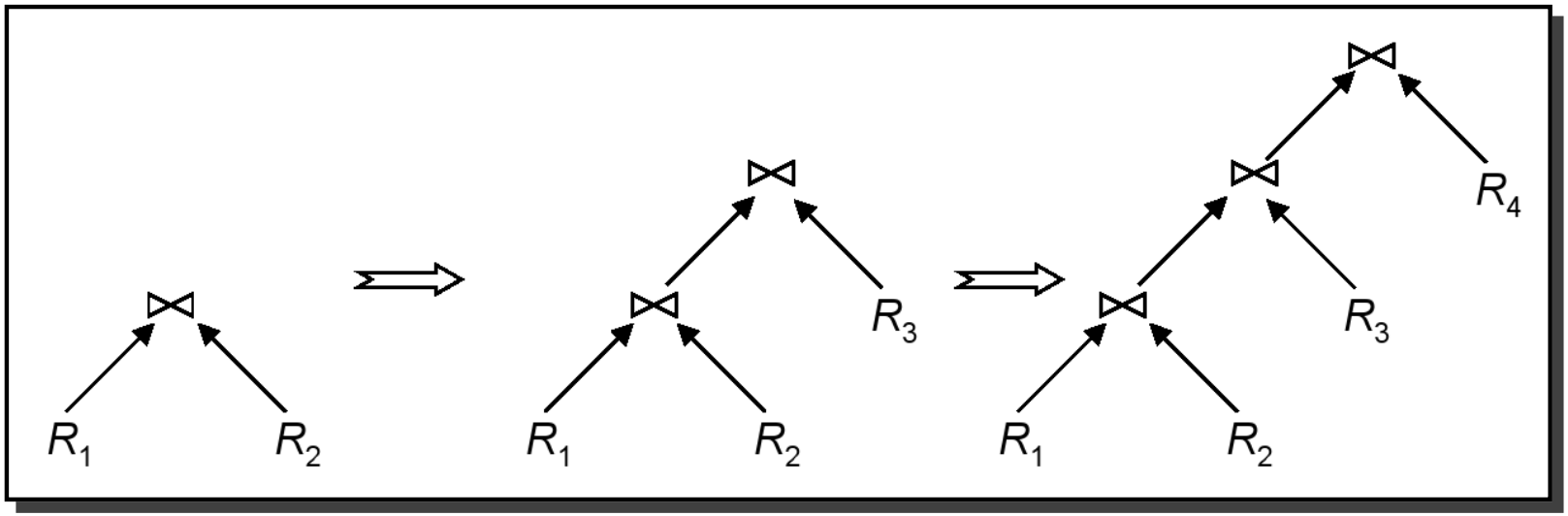


Bushy Join Tree

Basic Concepts . . .

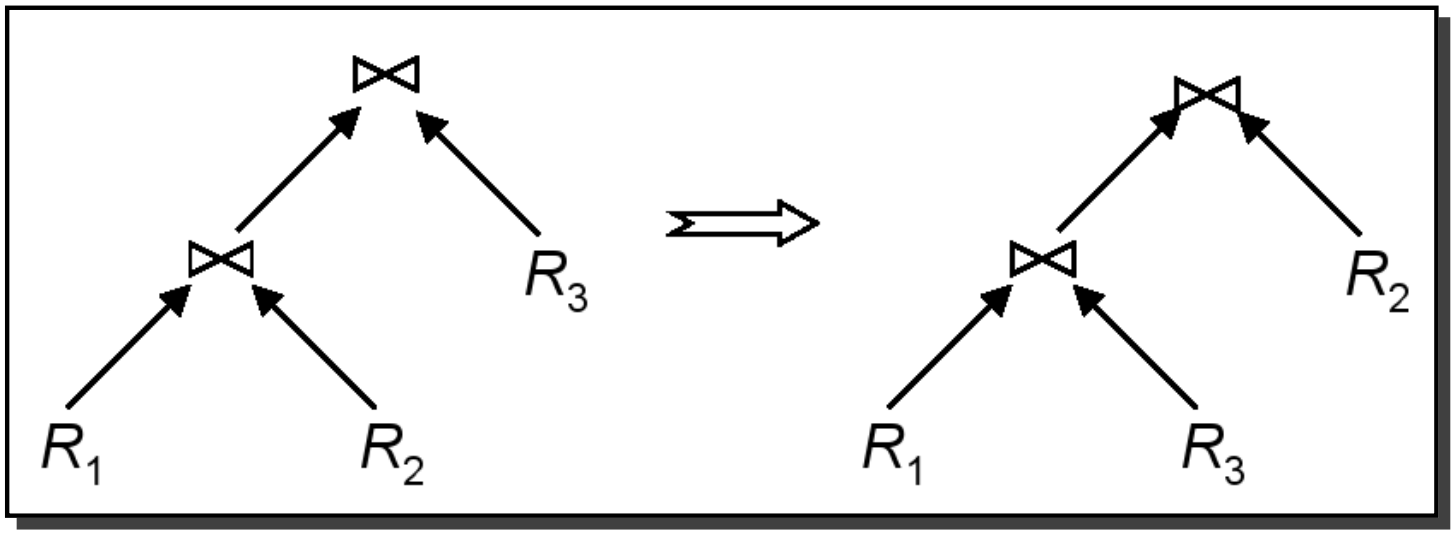
- There are two main strategies to **scan the search space**
  - Deterministic
  - Randomized

- **Deterministic scan** of the search space
  - Start from base relations and build plans by adding one relation at each step
  - Breadth-first strategy: build all possible plans before choosing the “best” plan (dynamic programming approach)
  - Depth-first strategy: build only one plan (greedy approach)



Basic Concepts . . .

- **Randomized scan** of the search space
  - Search for optimal solutions around a particular starting point
  - e.g., iterative improvement or simulated annealing techniques
  - Trades optimization time for execution time
    - Does not guarantee that the best solution is obtained, but avoid the high cost of optimization
  - The strategy is better when more than 5-6 relations are involved



Distributed Cost Model

- Two different types of **cost functions** can be used
  - Reduce **total time**
    - Reduce each cost component (in terms of time) individually, i.e., do as little for each cost component as possible
    - Optimize the utilization of the resources (i.e., increase system throughput)
  - Reduce **response time**
    - Do as many things in parallel as possible
    - May increase total time because of increased total activity

Distributed Cost Model . . .

- **Total time**: Sum of the time of all individual components
  - Local processing time: CPU time + I/O time
  - Communication time: fixed time to initiate a message + time to transmit the data

$$Total\_time = T_{CPU} * \#instructions + T_{I/O} * \#I/Os + T_{MSG} * \#messages + T_{TR} * \#bytes$$

- The individual components of the total cost have different weights:
  - Wide area network
    - Message initiation and transmission costs are high
    - Local processing cost is low (fast mainframes or minicomputers)
    - Ratio of communication to I/O costs is 20:1
  - Local area networks
    - Communication and local processing costs are more or less equal
    - Ratio of communication to I/O costs is 1:1.6 (10MB/s network)

Distributed Cost Model . . .

- **Response time**: Elapsed time between the initiation and the completion of a query

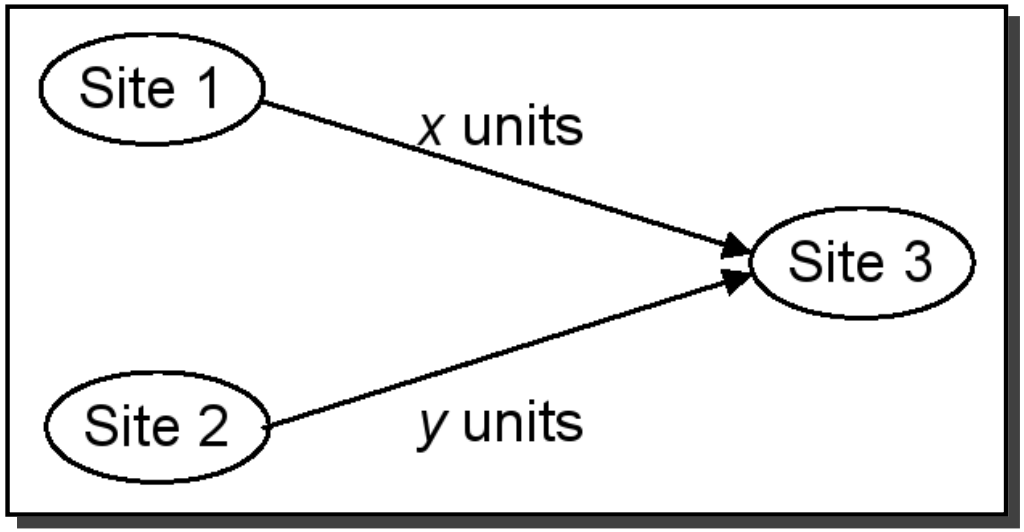
$$\begin{aligned} Response\_time = & T_{CPU} * \#seq\_instructions + T_{I/O} * \#seq\_I/Os + \\ & T_{MSG} * \#seq\_messages + T_{TR} * \#seq\_bytes \end{aligned}$$

  - where $\#seq\_x$ (x in instructions, I/O, messages, bytes) is the **maximum number** of x which must be done sequentially.

- Any processing and communication done in parallel is ignored

Distributed Cost Model . . .

- **Example:** Query at site 3 with data from sites 1 and 2.



  - Assume that only the communication cost is considered
  - $Total\_time = T_{MSG} * 2 + T_{TR} * (x + y)$
  - $Response\_time = \max\{T_{MSG} + T_{TR} * x, T_{MSG} + T_{TR} * y\}$

Database Statistics

- The **primary cost factor** is the **size of intermediate relations**
  - that are produced during the execution and
  - must be transmitted over the network, if a subsequent operation is located on a different site
- It is costly to compute the size of the intermediate relations precisely.
- Instead **global statistics of relations and fragments** are computed and used to provide approximations

Database Statistics . . .

- Let $R(A_1, A_2, \ldots, A_k)$ be a relation fragmented into $R_1, R_2, \ldots, R_r$.
- **Relation statistics**
  - min and max values of each attribute: $\min\{A_i\}$, $\max\{A_i\}$.
  - length of each attribute: $length(A_i)$
  - number of distinct values in each fragment (cardinality): $card(A_i)$, ($card(dom(A_i))$)
- **Fragment statistics**
  - cardinality of the fragment: $card(R_i)$
  - cardinality of each attribute of each fragment: $card(\Pi_{A_i}(R_j))$

Database Statistics . . .

- **Selectivity factor** of an operation: the proportion of tuples of an operand relation that participate in the result of that operation
- Assumption: independent attributes and uniform distribution of attribute values
- **Selectivity factor of selection**

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

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

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

- Properties of the selectivity factor of the selection

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

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

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

Database Statistics . . .

- **Cardinality** of intermediate results
  - Selection
    $$card(\sigma_P(R)) = SF_\sigma(P) * card(R)$$
  - Projection
    - More difficult: duplicates, correlations between projected attributes are unknown
    - Simple if the projected attribute is a key
      $$card(\Pi_A(R)) = card(R)$$
  - Cartesian Product
    $$card(R \times S) = card(R) * card(S)$$
  - Union
    - upper bound: $card(R \cup S) \leq card(R) + card(S)$
    - lower bound: $card(R \cup S) \geq \max\{card(R), card(S)\}$
  - Set Difference
    - upper bound: $card(R - S) = card(R)$
    - lower bound: 0

Database Statistics . . .

- **Selectivity factor** for joins

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

- **Cardinality** of joins
  - Upper bound: cardinality of Cartesian Product
    $card(R \bowtie S) \leq card(R) * card(S)$
  - General case (if SF is given):

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

  - Special case: $R.A$ is a key of R and $S.A$ is a foreign key of S;
    $*$ each S-tuple matches with at most one tuple of R

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

Database Statistics . . .

- **Selectivity factor** for semijoins: fraction of R-tuples that join with S-tuples
  - An approximation is the selectivity of A in S

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

- **Cardinality** of semijoin (general case):

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

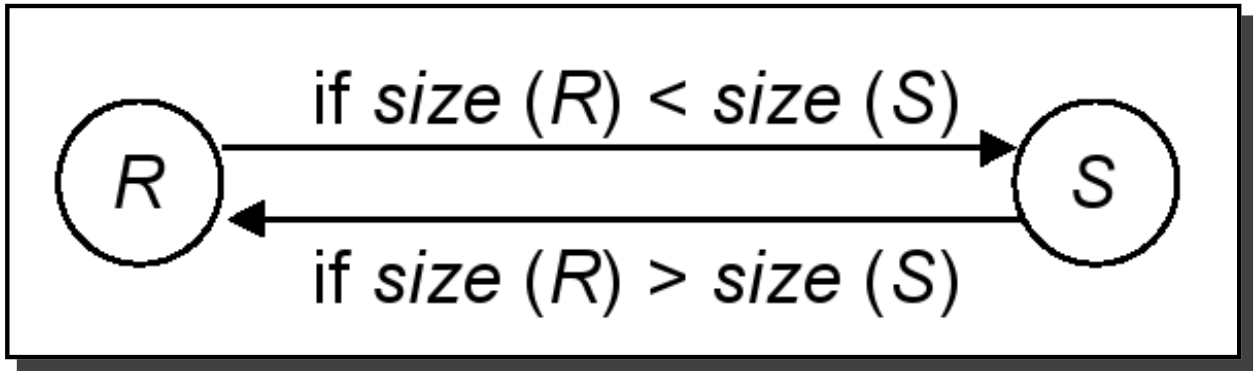
- Example: $R.A$ is a foreign key in S ($S.A$ is a primary key)
  Then $SF = 1$ and the result size corresponds to the size of R

Join Ordering in Fragment Queries

- **Join ordering** is an important aspect in centralized DBMS, and it **is even more important in a DDBMS** since joins between fragments that are stored at different sites may increase the communication time.
- Two approaches exist:
  - Optimize the ordering of joins directly
    - INGRES and distributed INGRES
    - System R and System R^*
  - Replace joins by combinations of semijoins in order to minimize the communication costs
    - Hill Climbing and SDD-1

Join Ordering in Fragment Queries . . .

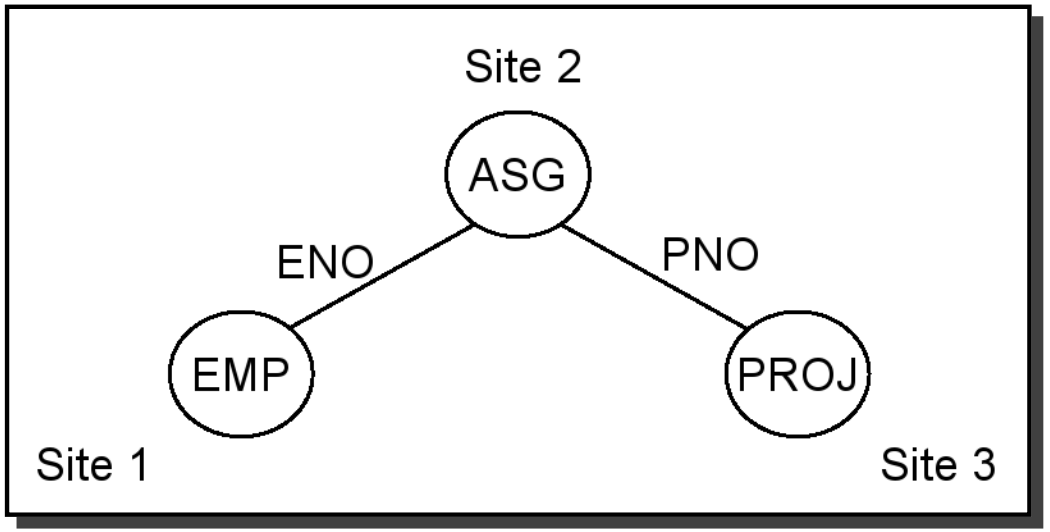
- **Direct join odering** of two relation/fragments located at different sites
  - Move the smaller relation to the other site
  - We have to estimate the size of R and S



Join Ordering in Fragment Queries . . .

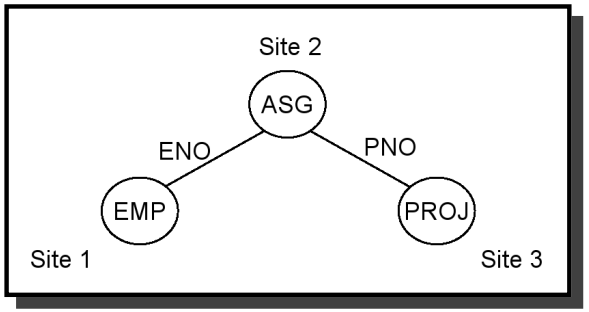
- **Direct join ordering** of queries involving more than two relations is substantially more complex
- **Example:** Consider the following query and the respective join graph, where we make also assumptions about the locations of the three relations/fragments

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



Join Ordering in Fragment Queries . . .

- **Example (contd.):** The query can be evaluated in at least 5 different ways.
  - Plan 1: EMP$\longrightarrow$Site 2
    Site 2: EMP'=EMP$\bowtie$ASG
    EMP'$\longrightarrow$Site 3
    Site 3: EMP'$\bowtie$PROJ
  - Plan 2: ASG$\longrightarrow$Site 1
    Site 1: EMP'=EMP$\bowtie$ASG
    EMP'$\longrightarrow$Site 3
    Site 3: EMP'$\bowtie$PROJ
  - Plan 3: ASG$\longrightarrow$Site 3
    Site 3: ASG'=ASG$\bowtie$PROJ
    ASG'$\longrightarrow$Site 1
    Site 1: ASG'$\bowtie$EMP



  - Plan 4: PROJ$\longrightarrow$Site 2
    Site 2: PROJ'=PROJ$\bowtie$ASG
    PROJ'$\longrightarrow$Site 1
    Site 1: PROJ'$\bowtie$EMP
  - Plan 5: EMP$\longrightarrow$Site 2
    PROJ$\longrightarrow$Site 2
    Site 2: EMP$\bowtie$PROJ$\bowtie$ASG

- To select a plan, a lot of information is needed, including
  - $size(EMP)$, $size(ASG)$, $size(PROJ)$, $size(EMP \bowtie ASG)$, $size(ASG \bowtie PROJ)$
  - Possibilities of parallel execution if response time is used

Semijoin Based Algorithms

- **Semijoins** can be used to efficiently implement joins
  - The semijoin acts as a size reducer (similar as to a selection) such that smaller relations need to be transferred
- Consider two relations: R located at site 1 and S located and site 2
  - Solution with semijoins: Replace one or both operand relations/fragments by a semijoin, using the following rules:

$$\begin{aligned} R \ltimes_A S &\iff (R \rtimes S) \ltimes_A S \\ &\iff R \ltimes_A (S \rtimes R) \\ &\iff (R \rtimes S) \ltimes_A (S \rtimes R) \end{aligned}$$

- The semijoin is beneficial if the cost to produce and send it to the other site is less than the cost of sending the whole operand relation and of doing the actual join.

Semijoin Based Algorithms

- **Cost analysis** $R \bowtie_A S$ vs. $(R \rtimes_A S) \bowtie S$, assuming that $size(R) < size(S)$
  - Perform the join $R \bowtie S$:
    - $R \rightarrow$ Site 2
    - Site 2 computes $R \bowtie S$
  - Perform the semijoins $(R \rtimes S) \bowtie S$:
    - $S' = \Pi_A(S)$
    - $S' \rightarrow$Site 1
    - Site 1 computes $R' = R \rtimes S'$
    - $R' \rightarrow$Site 2
    - Site 2 computes $R' \bowtie S$
  - Semijoin is better if: $size(\Pi_A(S)) + size(R \rtimes S) < size(R)$
- The **semijoin** approach is better if the semijoin acts as a **sufficient reducer** (i.e., a few tuples of R participate in the join)
- The **join** approach is better if **almost all tuples of** R **participate** in the join

INGRES Algorithm

- **INGRES** uses a dynamic query optimization algorithm that recursively breaks a query into smaller pieces. It is based on the following ideas:
  - An n-relation query q is **decomposed** into n subqueries $q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n$
    - Each q_i is a mono-relation (mono-variable) query
    - The output of q_i is consumed by q_{i+1}
  - For the decomposition two basic techniques are used: **detachment** and **substitution**
  - There's a processor that can **efficiently** process mono-relation queries
    - Optimizes each query independently for the access to a single relation

INGRES Algorithm . . .

- **Detachment:** Break a query q into $q' \rightarrow q''$, based on a common relation that is the result of q', i.e.
  - The query

    q: **SELECT** $R_2.A_2, \ldots, R_n.A_n$
    FROM $R_1, R_2, \ldots, R_n$
    WHERE $P_1(R_1.A'_1)$
    AND $P_2(R_1.A_1, \ldots, R_n.A_n)$

  - is decomposed by detachment of the common relation R_1 into

    q': **SELECT** $R_1.A_1$ **INTO** R'_1
    FROM R_1
    WHERE $P_1(R_1.A'_1)$

    q'': **SELECT** $R_2.A_2, \ldots, R_n.A_n$
    FROM $R'_1, R_2, \ldots, R_n$
    WHERE $P_2(R'_1.A_1, \ldots, R_n.A_n)$

- Detachment **reduces the size** of the relation on which the query q'' is defined.

INGRES Algorithm . . .

- **Example:** Consider query $q1$: “Names of employees working on the 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”

- Decompose q_1 into $q_{11} \rightarrow q'$:

  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

INGRES Algorithm . . .

- **Example (contd.):** The successive detachments may transform q' into $q_{12} \rightarrow q_{13}$:

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
```

- q_1 is now decomposed by detachment into $q_{11} \rightarrow q_{12} \rightarrow q_{13}$
- q_{11} is a mono-relation query
- q_{12} and q_{13} are multi-relation queries, which cannot be further detached.
  - also called **irreducible**

INGRES Algorithm . . .

- **Tuple substitution** allows to convert an irreducible query q into mono-relation queries.
  - Choose a relation R_1 in q for tuple substitution
  - For each tuple in R_1, replace the R_1-attributes referred in q by their actual values, thereby generating a set of subqueries q' with $n-1$ relations, i.e.,

$$q(R_1, R_2, \ldots, R_n) \text{ is replaced by } \{q'(t_{1_i}, R_2, \ldots, R_n), t_{1_i} \in R_1\}$$

- **Example (contd.):** Assume $GVAR$ consists only of the tuples $\{E1, E2\}$. Then q_{13} is rewritten with tuple substitution in the following way

q_{13}:
SELECT EMP.ENAME
FROM EMP, GVAR
WHERE EMP.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”

  - q_{131} and q_{132} are mono-relation queries

Distributed INGRES Algorithm

- The **distributed INGRES query optimization algorithm** is very similar to the centralized INGRES algorithm.
  - In addition to the centralized INGRES, the distributed one should break up each query q_i into sub-queries that operate on fragments; only horizontal fragmentation is handled.
  - Optimization with respect to a combination of communication cost and response time

System R Algorithm

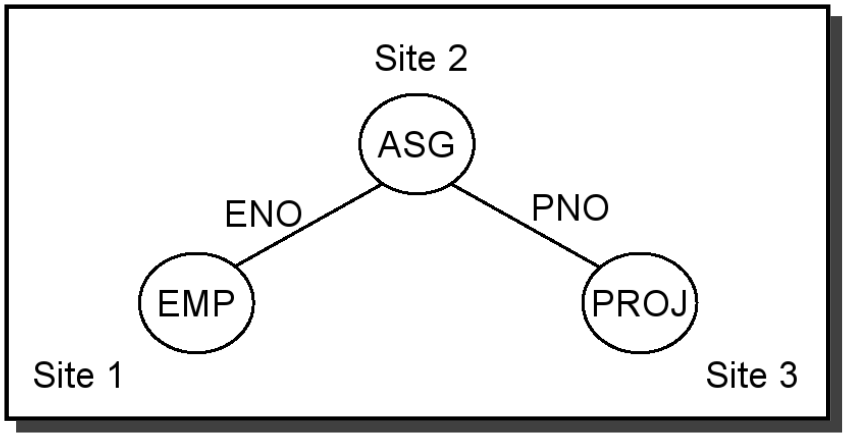
- The **System R** (centralized) query optimization algorithm
  - Performs static query optimization based on “exhaustive search” of the solution space and a cost function (IO cost + CPU cost)
    - Input: relational algebra tree
    - Output: optimal relational algebra tree
    - Dynamic programming technique is applied to reduce the number of alternative plans
  - The **optimization algorithm** consists of two steps
    1. Predict the best access method to each individual relation (mono-relation query)
       - Consider using index, file scan, etc.
    2. For each relation R, estimate the best join ordering
       - R is first accessed using its best single-relation access method
       - Efficient access to inner relation is crucial
  - Considers two different join strategies
    - (Indexed-) nested loop join
    - Sort-merge join

System R Algorithm . . .

- **Example:** Consider query $q1$: "*Names of employees working on the CAD/CAM project*"

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

  - Join graph



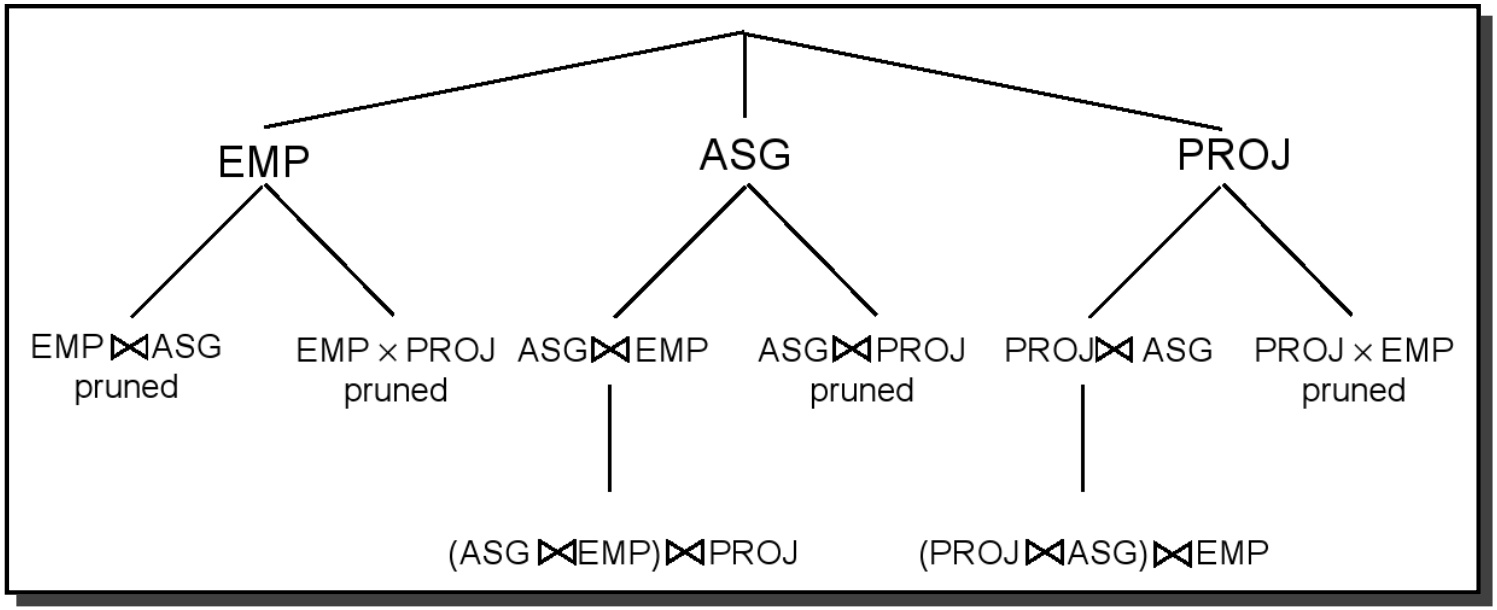
  - Indexes
    - EMP has an index on ENO
    - ASG has an index on PNO
    - PROJ has an index on PNO and an index on PNAME

System R Algorithm . . .

- **Example (contd.):** Step 1 – Select the best single-relation access paths
  - EMP: sequential scan (because there is no selection on EMP)
  - ASG: sequential scan (because there is no selection on ASG)
  - PROJ: index on PNAME (because there is a selection on PROJ based on PNAME)

System R Algorithm . . .

- **Example (contd.):** Step 2 – Select the best join ordering for each relation



  - (EMP $\times$ PROJ) and (PROJ $\times$ EMP) are pruned because they are CPs
  - (ASG $\times$ PROJ) pruned because we assume it has higher cost than (PROJ $\times$ ASG); similar for (PROJ $\times$ EMP)
  - Best total join order ((PROJ $\bowtie$ ASG) $\bowtie$ EMP), since it uses the indexes best
    - Select PROJ using index on PNAME
    - Join with ASG using index on PNO
    - Join with EMP using index on ENO

Distributed System R^* Algorithm

- The **System R^* query optimization** algorithm is an extension of the System R query optimization algorithm with the following main characteristics:
  - Only the whole relations can be distributed, i.e., fragmentation and replication is not considered
  - Query compilation is a distributed task, coordinated by a **master site**, where the query is initiated
  - Master site makes all inter-site decisions, e.g., selection of the execution sites, join ordering, method of data transfer, ...
  - The **local sites** do the intra-site (local) optimizations, e.g., local joins, access paths
- Join ordering and data transfer between different sites are the most critical issues to be considered by the master site

Distributed System R^* Algorithm ...

- Two methods for **inter-site data transfer**
  - **Ship whole:** The entire relation is shipped to the join site and stored in a temporary relation
    - Larger data transfer
    - Smaller number of messages
    - Better if relations are small
  - **Fetch as needed:** The external relation is sequentially scanned, and for each tuple the join value is sent to the site of the inner relation and the matching inner tuples are sent back (i.e., semijoin)
    - Number of messages = O(cardinality of outer relation)
    - Data transfer per message is minimal
    - Better if relations are large and the selectivity is good

Distributed System R^* Algorithm ...

- Four main **join strategies** for $R \bowtie S$:
  - R is outer relation
  - S is inner relation
- Notation:
  - LT denotes local processing time
  - CT denotes communication time
  - s denotes the average number of S-tuples that match an R-tuple
- **Strategy 1:** Ship the entire outer relation to the site of the inner relation, i.e.,
  - Retrieve outer tuples
  - Send them to the inner relation site
  - Join them as they arrive

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(R) \text{ tuples from } R) + \\ & CT(size(R)) + \\ & LT(\text{retrieve } s \text{ tuples from } S) * card(R) \end{aligned}$$

Distributed System R^* Algorithm ...

- **Strategy 2:** Ship the entire inner relation to the site of the outer relation. We cannot join as they arrive; they need to be stored.
  - The inner relation S need to be stored in a temporary relation

$$\begin{aligned} Total\_cost = & LT(\text{retrieve } card(S) \text{ tuples from } S) + \\ & CT(size(S)) + \\ & LT(\text{store } card(S) \text{ tuples in } T) + \\ & LT(\text{retrieve } card(R) \text{ tuples from } R) + \\ & LT(\text{retrieve } s \text{ tuples from } T) * card(R) \end{aligned}$$

Distributed System R^* Algorithm ...

- **Strategy 3:** Fetch tuples of the inner relation as needed for each tuple of the outer relation.
  - For each R-tuple, the join attribute A is sent to the site of S
  - The s matching S-tuples are retrieved and sent to the site of R

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(R) \text{ tuples from } R) + \\ & CT(length(A)) * card(R) + \\ & LT(\text{retrieve } s \text{ tuples from } S) * card(R) + \\ & CT(s * length(S)) * card(R) \end{aligned}$$

Distributed System R^* Algorithm ...

- **Strategy 4:** Move both relations to a third site and compute the join there.
  - The inner relation S is first moved to a third site and stored in a temporary relation.
  - Then the outer relation is moved to the third site and its tuples are joined as they arrive.

$$
\begin{aligned}
Total\_cost = {} & LT(\text{retrieve } card(S) \text{ tuples from } S) + \\
& CT(size(S)) + \\
& LT(\text{store } card(S) \text{ tuples in } T) + \\
& LT(\text{retrieve } card(R) \text{ tuples from } R) + \\
& CT(size(R)) + \\
& LT(\text{retrieve } s \text{ tuples from } T) * card(R)
\end{aligned}
$$

Hill-Climbing Algorithm

- **Hill-Climbing query optimization** algorithm
  - Refinements of an initial feasible solution are recursively computed until no more cost improvements can be made
  - Semijoins, data replication, and fragmentation are not used
  - Devised for wide area point-to-point networks
  - The first distributed query processing algorithm

Hill-Climbing Algorithm . . .

- The hill-climbing algorithm proceeds as follows
  1. Select initial feasible execution strategy ES0
     - i.e., a global execution schedule that includes all intersite communication
     - Determine the candidate result sites, where a relation referenced in the query exist
     - Compute the cost of transferring all the other referenced relations to each candidate site
     - ES0 = candidate site with minimum cost
  2. Split ES0 into two strategies: ES1 followed by ES2
     - ES1: send one of the relations involved in the join to the other relation's site
     - ES2: send the join result to the final result site
  3. Replace ES0 with the split schedule which gives

$$cost(ES1) + cost(\text{local join}) + cost(ES2) < cost(ES0)$$

  4. Recursively apply steps 2 and 3 on ES1 and ES2 until no more benefit can be gained
  5. 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?*

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

  - Schemas: EMP(ENO, ENBAME, TITLE), ASG(ENO, PNO, RESP, DUR), PROJ(PNO, PNAME, BUDGET, LOC), PAY(TITLE, SAL)
  - Statistics

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

  - Assumptions:
    - Size of relations is defined as their cardinality
    - Minimize total cost
    - Transmission cost between two sites is 1
    - Ignore local processing cost
    - size(EMP ⋈ PAY) = 8, size(PROJ ⋈ ASG) = 2, size(ASG ⋈ EMP) = 10

Hill-Climbing Algorithm . . .

- **Example (contd.):** Determine initial feasible execution strategy
  - Alternative 1: Resulting site is site 1

$$Total\_cost = cost(\text{PAY} \rightarrow \text{Site}1) + cost(\text{ASG} \rightarrow \text{Site}1) + cost(\text{PROJ} \rightarrow \text{Site}1)$$
$$= 4 + 10 + 1 = 15$$

  - 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 ES0 = EMP$\rightarrow$Site4; PAY $\rightarrow$ Site4; PROJ $\rightarrow$ Site4

Hill-Climbing Algorithm . . .

- **Example (contd.):** Candidate split
  - Alternative 1: ES1, ES2, ES3
    - ES1: EMP$\longrightarrow$Site 2
    - ES2: (EMP$\bowtie$PAY) $\longrightarrow$ Site4
    - ES3: PROJ$\longrightarrow$Site 4

$$\begin{aligned} Total\_cost &= cost(\text{EMP} \longrightarrow \text{Site2}) + \\ &\quad cost((\text{EMP} \bowtie \text{PAY}) \longrightarrow \text{Site4}) + \\ &\quad cost(\text{PROJ} \longrightarrow \text{Site4}) \\ &= 8 + 8 + 1 = 17 \end{aligned}$$

  - Alternative 2: ES1, ES2, ES3
    - ES1: PAY $\longrightarrow$ Site1
    - ES2: (PAY $\bowtie$ EMP) $\longrightarrow$ Site4
    - ES3: PROJ $\longrightarrow$ Site 4

$$\begin{aligned} Total\_cost &= cost(\text{PAYSite} \longrightarrow 1) + \\ &\quad cost((\text{PAY} \bowtie \text{EMP}) \longrightarrow \text{Site4}) + \\ &\quad cost(\text{PROJ} \longrightarrow \text{Site4}) \\ &= 4 + 8 + 1 = 13 \end{aligned}$$

- Both alternatives are not better than ES0, so keep it (or take alternative 2 which has the same cost)

Hill-Climbing Algorithm . . .

- **Problems**
  - Greedy algorithm determines an initial feasible solution and iteratively improves it
  - If there are local minima, it may not find the global minimum
  - An optimal schedule with a high initial cost would not be found, since it won't be chosen as the initial feasible solution
- **Example:** A better schedule is
  - PROJ$\longrightarrow$Site 4
  - ASG' = (PROJ$\bowtie$ASG)$\longrightarrow$Site 1
  - (ASG'$\bowtie$EMP)$\longrightarrow$Site 2
  - Total cost$= 1 + 2 + 2 = 5$

SDD-1

- The **SDD-1 query optimization** algorithm improves the Hill-Climbing algorithm in a number of directions:
  - Semijoins are considered
  - More elaborate statistics
  - Initial plan is selected better
  - Post-optimization step is introduced

Conclusion

- Distributed query optimization is more complex that centralized query processing, since
  - bushy query trees are not necessarily a bad choice
  - one needs to decide what, where, and how to ship the relations between the sites
- Query optimization searches the optimal query plan (tree)
- For N relations, there are $O(N!)$ equivalent join trees. To cope with the complexity heuristics and/or restricted types of trees are considered
- There are two main strategies in query optimization: randomized and deterministic
- (Few) semi-joins can be used to implement a join. The semi-joins require more operations to perform, however the data transfer rate is reduced
- INGRES, System R, Hill Climbing, and SDD-1 are distributed query optimization algorithms