Basic Concepts

- Query optimization: Process of producing an optimal (close to optimal) query execution plan which represents an execution strategy
  - The main task in query optimization is to consider different orderings of the operations
- Centralized query optimization:
  - Find (the best) query execution plan in 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

Search space: The set of alternative query execution plans (query trees)

- Typically very large
- The main issue is to optimize 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

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

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 trees)
Basic Concepts/4
- 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 (BFS): build all possible plans before choosing the “best” plan (dynamic programming approach)
  - Depth-first strategy (DFS): build only one plan (greedy approach)

Basic Concepts/5
- 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/1
- 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/2
- 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
    \[
    Total_{time} = T_{CPU} \times \#\text{instructions} + T_{I/O} \times \#I/Os + T_{MSG} \times \#\text{messages} + T_{TR} \times \#\text{bytes}
    \]
  - 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/3

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

\[
\text{Response time} = T_{CPU} \times \#\text{seq.instructions} + T_{I/O} \times \#\text{seq.I/Os} + T_{MSG} \times \#\text{seq.messages} + T_{TR} \times \#\text{seq.bytes}
\]

- 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

Database Statistics/1

- 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/2

- 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: \( \text{length}(A_i) \).
  - number of distinct values in each domain: \( \text{card}(\text{dom}(A_i)) \).
- **Fragment statistics**
  - cardinality of the fragment: \( \text{card}(R_i) \).
  - cardinality of each attribute of each fragment: \( \text{card}(\pi_A(R_i)), \text{card}(A_i) \).

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

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

\[
SF_{\sigma}(A = value) = \frac{1}{\text{card}(\pi_A(R))}
\]

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

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

**Cardinality** of intermediate results

- **Selection**
  \[
  \text{card}(\sigma_P(R)) = SF_{\sigma}(P) \times \text{card}(R)
  \]

- **Projection**
  - More difficult: correlations between projected attributes are unknown
  - Simple if the projected attribute is a key
  \[
  \text{card}(\pi_A(R)) = \text{card}(R)
  \]

- **Cartesian Product**
  \[
  \text{card}(R \times S) = \text{card}(R) \times \text{card}(S)
  \]

- **Union**
  - upper bound: \( \text{card}(R \cup S) \leq \text{card}(R) + \text{card}(S) \)
  - lower bound: \( \text{card}(R \cup S) \geq \max(\text{card}(R), \text{card}(S)) \)

- **Set Difference**
  - upper bound: \( \text{card}(R - S) = \text{card}(R) \)
  - lower bound: 0

**Selectivity factor for joins**

\[
SF_{\bowtie} = \frac{\text{card}(R \bowtie S)}{\text{card}(R) \times \text{card}(S)}
\]

**Cardinality of joins**

- Upper bound: cardinality of Cartesian Product
  \[
  \text{card}(R \bowtie S) \leq \text{card}(R) \times \text{card}(S)
  \]

- General case (if SF is given):
  \[
  \text{card}(R \bowtie S) = SF_{\bowtie} \times \text{card}(R) \times \text{card}(S)
  \]

- 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 \( R \)
  \[
  \text{card}(R \bowtie_{R.A=S.A} S) = \text{card}(S)
  \]
Selectivity factor for semijoins: fraction of R-tuples that join with S-tuples

- An approximation is the selectivity of A in S

\[ SF_{\bowtie}(R \bowtie_A S) = SF_{\bowtie}(S.A) = \frac{\text{card}(\pi_A(S))}{\text{card}(\text{dom}[A])} \]

Cardinality of semijoin (general case):

\[ \text{card}(R \bowtie_A S) = SF_{\bowtie}(S.A) \ast \text{card}(R) \]

- 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 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

Direct join ordering 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

Example: Consider the following query and the respective join graph, where we make also assumptions about the locations of the three relations/fragments

\[ \text{PROJ} \bowtie_{\text{PNO}} \text{ASG} \bowtie_{\text{ENO}} \text{EMP} \]
Join Ordering in Fragment Queries/4

Example (contd.): The query can be evaluated in at least 5 different ways.

- Plan 1:
  EMP → Site 2
  Site 2: EMP′ = EMP ⊙ ASG
  EMP′ → Site 3
  Site 3: EMP′ ⊙ PROJ

- Plan 2:
  ASG → Site 1
  Site 1: EMP′ = EMP ⊙ ASG
  EMP′ → Site 3
  Site 3: EMP′ ⊙ PROJ

- Plan 3:
  ASG → Site 3
  Site 3: ASG′ = ASG ⊙ PROJ
  ASG′ → Site 1
  Site 1: ASG′ ⊙ EMP

- Plan 4:
  PROJ → Site 2
  Site 2: PROJ′ = PROJ ⊙ ASG
  PROJ′ → Site 1
  Site 1: PROJ′ ⊙ EMP

- Plan 5:
  EMP → Site 2
  PROJ → Site 2
  Site 2: EMP ⊙ PROJ ⊙ ASG

To select a plan, a lot of information is needed, including

- size(EMP), size(ASG), size(PROJ)
- size(EMP ⊙ ASG), size(ASG ⊙ PROJ)
- Possibilities of parallel execution if response time is used

Semijoin Based Algorithms/1

- 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 at site 2
  - Solution with semijoins: Replace one or both operand relations/fragments by a semijoin, using the following rules:
    
    \[
    R \bowtie_A S \iff (R \bowtie_A S) \bowtie_A S
    \]

    \[
    \iff R \bowtie_A (S \bowtie_A R)
    \]

    \[
    \iff (R \bowtie_A S) \bowtie_A (S \bowtie_A R)
    \]

- 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 doing the actual join.

Join Ordering in Fragment Queries/5

- Cost analysis R ⊢ A S vs. (R ⊲ A S) ⊢ S, assuming that size(R) < size(S)
  - Perform the join R ⊢ S:
    - R → Site 2
    - Site 2 computes R ⊢ S
  - Perform the semijoins (R ⊲ S) ⊢ S:
    - S′ = π_A(S)
    - S′ → Site 1
    - Site 1 computes R′ = R ⊲ S′
    - R′ → Site 2
    - Site 2 computes R′ ⊢ S
  - Semijoin is better if: size(π_A(S)) + size(R ⊲ 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/1

- 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 is a processor that can efficiently process mono-relation queries
  - Optimizes each query independently for the access to a single relation

INGRES Algorithm/2

- 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 $R_1.A_1, \ldots, R_n.A_n$
    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 $R_i$ into $q'$: SELECT $R_i.A_i$
    INTO $R_i'$
    FROM $R_i$
    WHERE $P_1(R_1.A_1')$
    $q'':$ SELECT $R_2.A_2, \ldots, R_n.A_n$
    FROM $R_i', R_2, \ldots, R_n$
    WHERE $P_2(R_i', A_1, \ldots, R_n.A_n)$
  - Detachment reduces the size of the relation on which the query $q''$ is defined.

Example: Consider query $q_1$: “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

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

$q_{12}$: 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/5**

- **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.,
    $$q(R_1, R_2, \ldots, R_n) \text{ is replaced by } \{q'(t_1, R_2, \ldots, R_n), t_1 \in R_1\}$$

- **Example (contd.):** Assume GVAR consists only of the tuples $\{E1, E2\}$. Then $q_{13}$ is rewritten with tuple substitution in the following way
  
  $q_{13}$: 
  ```sql
  SELECT EMP.ENAME
  FROM EMP, GVAR
  WHERE EMP.ENO = GVAR.ENO
  ```

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

**System R Algorithm/1**

- 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)
    2. Consider using index, file scan, etc.
    3. For each relation $R$, estimate the best join ordering
    4. $R$ is first accessed using its best single-relation access method
    5. Efficient access to inner relation is crucial
  - Considers two different join strategies
    - (Indexed-) nested loop join
    - Sort-merge join

**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

**INGRES Algorithm/6**

- **Example (contd.):**
  
  $q_{132}$: 
  ```sql
  SELECT EMP.ENAME
  FROM EMP
  WHERE EMP.ENO = "E2"
  ```

  - $q_{131}$ and $q_{132}$ are mono-relation queries
System R Algorithm/2

▶ Example: Consider query q1: “Names of employees working on the CAD/CAM project”

\[ \text{PROJ} \bowtie_{\text{PNO}} \text{ASG} \bowtie_{\text{ENO}} \text{EMP} \]

▶ Join graph

![Join graph diagram]

▶ 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/3

▶ 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)

DDBS12, SL05

System R Algorithm/4

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

![Join ordering diagram]

▶ (EMP \times PROJ) and (PROJ \times EMP) are pruned because they are CPs
▶ (ASG \bowtie PROJ) pruned because (we assume) it has higher cost than (PROJ \bowtie ASG); similar for (ASG \bowtie 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/1

▶ 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

DDBS12, SL05
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 outer 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

Four main join strategies for \( R \bowtie S \):  
- **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

\[
\text{Total cost} = LT(\text{retrieve card}(R) \text{ tuples from } R) + \\
CT(\text{size}(R)) + \\
LT(\text{store card}(S) \text{ tuples in } T) + \\
LT(\text{retrieve card}(R) \text{ tuples from } R) + \\
LT(\text{retrieve s tuples from } T) \times \text{card}(R)
\]

- **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

\[
\text{Total cost} = LT(\text{retrieve card}(S) \text{ tuples from } S) + \\
CT(\text{length}(A)) \times \text{card}(R) + \\
LT(\text{store card}(S) \text{ tuples in } T) + \\
LT(\text{retrieve card}(R) \text{ tuples from } R) + \\
LT(\text{retrieve s tuples from } T) \times \text{card}(R)
\]

- **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 \( R \)

\[
\text{Total cost} = LT(\text{retrieve card}(R) \text{ tuples from } R) + \\
CT(\text{length}(A)) \times \text{card}(R) + \\
LT(\text{retrieve s tuples from } S) \times \text{card}(R) + \\
CT(s \times \text{length}(S)) \times \text{card}(R)
\]
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.

Total cost = LT(retrieve card($S$) tuples from $S$) + $CT(size(S))$ + $LT(store card(S) tuples in T)$ + $LT(retrieve card(R) tuples from R)$ + $CT(size(R))$ + $LT(retrieve s tuples from T) \times card(R)$

Hill-Climbing Algorithm/1
- 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/2
- 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
     \[ cost(ES1) + cost(local join) + cost(ES2) < cost(ES0) \]
  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/3
- Example: What are the salaries of engineers who work on the CAD/CAM project?
  \[ \pi_{SAL}(PAY \bowtie_{TITLE\bowtie\text{EN}} EMP \bowtie_{PNO\bowtie\text{RESP, DUR}} ASG \bowtie_{PNO, PNAME, BUDGET, LOC, PAY(TITLE, SAL)} PROJ(PNO, PNAME, BUDGET, LOC, PAY(TITLE, SAL)) ) \]
  - Schemas: EMP(ENO, ENAME, TITLE), ASG(ENO, PNO, RESP, DUR), PROJ(PNO, PNAME, BUDGET, LOC), PAY(TITLE, SAL)
  - Statistics

<table>
<thead>
<tr>
<th>Relation</th>
<th>Size</th>
<th>Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>PAY</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>PROJ</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ASG</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

- Assumptions:
  - Size of relations is defined as their cardinality
  - Minimize total cost
  - Transmission cost between two sites is 1
  - Ignore local processing cost
  - $size(EMP \bowtie PAY) = 8$, $size(PROJ \bowtie ASG) = 2$, $size(ASG \bowtie EMP) = 10$
Hill-Climbing Algorithm/4
Example (contd.): Determine initial feasible execution strategy
- Alternative 1: Resulting site is site 1
  \[ Total\_cost = cost(PAY \rightarrow Site1) + cost(ASG \rightarrow Site1) + cost(PROJ \rightarrow Site1) \]
  \[ = 4 + 10 + 1 = 15 \]
- Alternative 2: Resulting site is site 2
  Total cost = 8 + 10 + 1 = 19
- Alternative 3: Resulting site is site 3
  Total cost = 8 + 4 + 10 = 22
- Alternative 4: Resulting site is site 4
  Total cost = 8 + 4 + 1 = 13
- Therefore ES0 = EMP→Site4; PAY→Site4; PROJ→Site4

Hill-Climbing Algorithm/5
Example (contd.): Candidate split
- Alternative 1: ES1, ES2, ES3
  - ES1: EMP→Site 2
  - ES2: (EMP\(\bowtie\)PAY)→Site4
  - ES3: PROJ→Site 4
  Total cost = 8 + 8 + 1 = 17
- Alternative 2: ES1, ES2, ES3
  - ES1: PAY→Site1
  - ES2: (PAY\(\bowtie\)EMP)→Site4
  - ES3: PROJ→Site 4
  Total cost = 4 + 8 + 1 = 13
- Both alternatives are not better than ES0, so keep ES0 (or take alternative 2 which has the same cost)

Hill-Climbing Algorithm/6
Example: A better schedule is
- PROJ→Site 4
- ASG′ = (PROJ\(\bowtie\)ASG)→Site 1
- (ASG\(\bowtie\)EMP)→Site 2
- Total cost = 1 + 2 + 2 = 5

SDD-1
The SDD-1 algorithm extends the hill climbing algorithm with semijoins and has the following properties:
- Considers semijoins
  - cost(R\(\bowtie\)A\(S\)) = C\(MSG\) + size(\(\pi A(S)\)) * C\(TR\)
  - benefit(R\(\bowtie\)A\(S\)) = (1 - SF\(\bowtie\)\(S.A\)) * size(R) * C\(TR\)
- Does not consider replication and fragmentation
- Cost of transferring the result to the user site from the final result site is not considered
- Can minimize either total time or response time
The SDD-1 algorithm works with and updates a database profile:

<table>
<thead>
<tr>
<th>R</th>
<th>size(R)</th>
<th>A</th>
<th>SF(\bowtie)</th>
<th>size((\pi A))</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1500</td>
<td>R1.A</td>
<td>0.3</td>
<td>36</td>
</tr>
<tr>
<td>R2</td>
<td>3000</td>
<td>R2.A</td>
<td>0.8</td>
<td>320</td>
</tr>
<tr>
<td>R3</td>
<td>2000</td>
<td>R2.B</td>
<td>1.0</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R3.B</td>
<td>0.4</td>
<td>80</td>
</tr>
</tbody>
</table>
SDD-1 Algorithm

Step 1 Include all local processing in the execution strategy ES.
Step 2 Update database profile with effects of local processing.
Step 3 Determine beneficial $\kappa$, i.e., $cost(\kappa_i) < benefit(\kappa_j)$.
Step 4 Remove the most beneficial $\kappa$ and append it to ES.
Step 5 Update the database profile.
Step 6 Update the set of beneficial semijoins; possibly include new ones.
Step 7 If there are beneficial semijoins go back to Step 4.
Step 8 Find the site where the largest amount of data resides and select it as the result site.
Step 9 For each $R_i$ at the result site, remove semijoins of the form $R_i \kappa R_j$ where the total cost of ES without this semijoin is smaller than the cost with it.
Step 10 Permute the order of semijoins if doing so would improve the total cost of ES.

Conclusion

- Distributed query optimization is more complex than 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.
- Semi-joins can be used to implement a join. The semi-joins require more operations to perform, but the data transfer rate is reduced.
- INGRES, System R and Hill Climbing are distributed query optimization algorithms.

Course Project

- Hand in of project: December 23, 2012
- Report
  - problem definition
  - running example
  - description of solution
  - evaluation
  - strength, weaknesses, limitations
- Report (5 pages) and implementation (source code, data, steps to install and run) as zip/tar file
- Send by email to boehlen@ifi.uzh.ch and cafagna@ifi.uzh.ch

Course Exam

- Exam date: 16.01.2013
- Exam time: 12:15 - 12:45
- Exam location: BIN 2.E.13
- Exam form and procedure
  - oral, 20 minutes
  - 10 minutes about project (demo, code, algorithm)
  - 10 about a topic of the course
- During exam: present solutions on examples
- Prepare suitable examples beforehand