Outline

• Introduction
• Background
• Distributed Database Design
• Database Integration
• Semantic Data Control
• Distributed Query Processing
  † Overview
  † Query decomposition and localization
  † Distributed query optimization
• Multidatabase Query Processing
• Distributed Transaction Management
• Data Replication
• Parallel Database Systems
• Distributed Object DBMS
• Peer-to-Peer Data Management
• Web Data Management
• Current Issues
Step 3 – 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

Input Query → Search Space Generation → Equivalent QEP → Search Strategy → Best QEP

- Transformation Rules
- Cost Model
Search Space

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

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

- Restrict by means of heuristics
  - Perform unary operations before binary operations
  - ...

- Restrict the shape of the join tree
  - Consider only linear trees, ignore bushy ones

Linear Join Tree

```
R_1 ─── R_2 ─── R_3 ─── R_4
```

Bushy Join Tree

```
R_1 ─── R_2 ─── R_3 ─── R_4
```
Search Strategy

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

- Deterministic

- Randomized
Cost Functions

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

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

Summation of all cost factors

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

CPU cost = unit instruction cost * no. of instructions

I/O cost = unit disk I/O cost * no. of disk I/Os

communication cost = message initiation + transmission
Total Cost Factors

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

- Local area networks
  - Communication and local processing costs are more or less equal
  - Ratio = 1:1.6
Response Time

Elapsed time between the initiation and the completion of a query

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

Assume that only the communication cost is considered.

Total time = 2 \times \text{message initialization time} + \text{unit transmission time} \times (x+y)

Response time = \max \{ \text{time to send } x \text{ from 1 to 3, time to send } y \text{ from 2 to 3} \}

time to send \ x \text{ from 1 to 3} = \text{message initialization time} + \text{unit transmission time} \times x

time to send \ y \text{ from 2 to 3} = \text{message initialization time} + \text{unit transmission time} \times y
Optimization Statistics

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

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

Selection

\[
\text{size}(R) = \text{card}(R) \times \text{length}(R)
\]
\[
\text{card}(\sigma_F(R)) = SF_\sigma(F) \times \text{card}(R)
\]

where

\[
SF_\sigma(A = \text{value}) = \frac{1}{\text{card}(\prod_A(R))}
\]
\[
SF_\sigma(A > \text{value}) = \frac{\max(A) - \text{value}}{\max(A) - \min(A)}
\]
\[
SF_\sigma(A < \text{value}) = \frac{\text{value} - \max(A)}{\max(A) - \min(A)}
\]
\[
SF_\sigma(p(A_i) \land p(A_j)) = SF_\sigma(p(A_i)) \times SF_\sigma(p(A_j))
\]
\[
SF_\sigma(p(A_i) \lor p(A_j)) = SF_\sigma(p(A_i)) + SF_\sigma(p(A_j)) - (SF_\sigma(p(A_i)) \times SF_\sigma(p(A_j)))
\]
\[
SF_\sigma(A \in \{\text{values}\}) = SF_\sigma(A = \text{value}) \times \text{card(\{values\})}
\]
Intermediate Relation Sizes

**Projection**
\[ \text{card}(\Pi_A(R)) = \text{card}(R) \]

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

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

**Set Difference**
- upper bound: \[ \text{card}(R - S) = \text{card}(R) \]
- lower bound: \( 0 \)
Intermediate Relation Size

Join

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

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

More general:

$$\text{card}(R \bowtie S) = SF \bowtie * \text{card}(R) \times \text{card}(S)$$

Semijoin

$$\text{card}(R \bowtie_A S) = SF \bowtie (S.A) * \text{card}(R)$$

where

$$SF \bowtie (R \bowtie_A S) = SF \bowtie (S.A) = \frac{\text{card}(\Pi_A(S))}{\text{card(dom}[A])]$$
Histograms for Selectivity Estimation

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

For ASG.DUR=18: we have SF=1/12 so the card of selection is 300/12 = 25 tuples

For ASG.DUR≤18: we have min(range_{3})=12 and max(range_{3})=24 so the card. of selection is 100+75+(((18−12)/(24 − 12))*50) = 200 tuples
Centralized Query Optimization

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

1. Decompose each multi-variable query into a sequence of mono-variable queries with a common variable
2. Process each by a one variable query processor
   † Choose an initial execution plan (heuristics)
   † Order the rest by considering intermediate relation sizes

No statistical information is maintained
Dynamic Algorithm–Decomposition

- Replace an $n$ variable query $q$ by a series of queries
  
  $q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n$

  where $q_i$ uses the result of $q_{i-1}$.

- Detachment
  
  Query $q$ decomposed into $q' \rightarrow q''$ where $q'$ and $q''$ have a common variable which is the result of $q'$

- Tuple substitution
  
  Replace the value of each tuple with actual values and simplify the query

  \[ q(V_1, V_2, \ldots, V_n) \rightarrow (q'(t_1, V_2, V_2, \ldots, V_n), t_1 \in R) \]
Detachment

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

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

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

Names of employees working on CAD/CAM project

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

$q_{11}$:  
```
SELECT PROJ.PNO INTO JVAR  
FROM PROJ  
WHERE PROJ.PNAME="CAD/CAM"
```

$q'$:  
```
SELECT EMP.ENAME  
FROM EMP, ASG, JVAR  
WHERE EMP.ENO=ASG.ENO  
AND ASG.PNO=JVAR.PNO
```
Detachment Example (cont’d)

$q'$:  
\[
\begin{align*}
\text{SELECT} & \quad \text{EMP.ENAME} \\
\text{FROM} & \quad \text{EMP, ASG, JVAR} \\
\text{WHERE} & \quad \text{EMP.ENO=ASG.ENO} \\
\text{AND} & \quad \text{ASG.PNO=JVAR.PNO}
\end{align*}
\]

$q_{12}$:  
\[
\begin{align*}
\text{SELECT} & \quad \text{ASG.ENO INTO GVAR} \\
\text{FROM} & \quad \text{ASG, JVAR} \\
\text{WHERE} & \quad \text{ASG.PNO=JVAR.PNO}
\end{align*}
\]

$q_{13}$:  
\[
\begin{align*}
\text{SELECT} & \quad \text{EMP.ENAME} \\
\text{FROM} & \quad \text{EMP, GVAR} \\
\text{WHERE} & \quad \text{EMP.ENO=GVAR.ENO}
\end{align*}
\]
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

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

$q_{132}$: \textbf{SELECT} EMP.ENAME
\textbf{FROM} EMP
\textbf{WHERE} EMP.ENO="E2"
Static Algorithm

1. Simple (i.e., mono-relation) queries are executed according to the best access path
2. Execute joins
   † Determine the possible ordering of joins
   † Determine the cost of each ordering
   † Choose the join ordering with minimal cost
Static Algorithm

For joins, two alternative algorithms:

- Nested loops

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

end

† Complexity: $n_1 \times n_2$

- Merge join

  sort relations
  merge relations

  † Complexity: $n_1 + n_2$ if relations are previously sorted and equijoin
Static Algorithm – Example

Names of employees working on the CAD/CAM project
Assume

- EMP has an index on ENO,
- ASG has an index on PNO,
- PROJ has an index on PNO and an index on PNAME
Example (cont’d)

1. 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)

2. Determine the best join ordering
   - EMP ⟿ ASG ⟿ PROJ
   - ASG ⟿ PROJ ⟿ EMP
   - PROJ ⟿ ASG ⟿ EMP
   - ASG ⟿ EMP ⟿ PROJ
   - EMP × PROJ ⟿ ASG
   - PRO × JEMP ⟿ ASG

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

Alternatives

Best total join order is one of

\(((\text{ASG} \bowtie \text{EMP}) \bowtie \text{PROJ})\)

\(((\text{PROJ} \bowtie \text{ASG}) \bowtie \text{EMP})\)
Static Algorithm

- \(((\text{PROJ} \bowtie \text{ASG}) \bowtie \text{EMP})\) has a useful index on the select attribute and direct access to the join attributes of ASG and EMP
- Therefore, chose it with the following access methods:
  - select PROJ using index on PNAME
  - then join with ASG using index on PNO
  - then join with EMP using index on ENO
Hybrid optimization

- In general, static optimization is more efficient than dynamic optimization
  - Adopted by all commercial DBMS
- But even with a sophisticated cost model (with histograms), accurate cost prediction is difficult
- Example
  - Consider a parametric query with predicate
    `WHERE R.A = $a        /* $a is a parameter`
  - The only possible assumption at compile time is uniform distribution of values
- Solution: Hybrid optimization
  - Choose-plan done at runtime, based on the actual parameter binding
Hybrid Optimization Example

$\sigma a = A$

$\sigma a = A$

Choose-plan

Choose-plan

$R_1$

$R_2$

$R_3$

$R_2$

$R_1$
Join Ordering in Fragment Queries

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

- Consider two relations only

\[
\begin{align*}
\text{if } \text{size}(R) &< \text{size}(S) \\
\text{if } \text{size}(R) &> \text{size}(S)
\end{align*}
\]

- 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

\[
\text{PROJ} \bowtie_{\text{PNO}} \text{ASG} \bowtie_{\text{ENO}} \text{EMP}
\]
Join Ordering – Example

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

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

5. EMP → Site 2
   PROJ → Site 2
   Site 2 computes EMP ⋈ PROJ ⋈ ASG
Semijoin Algorithms

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

• Perform the join
  † send $R$ to Site 2
  † Site 2 computes $R \bowtie_A S$

• Consider semijoin $(R \bowtie_A S) \bowtie_A S$
  † $S' = \Pi_A(S)$
  † $S' \rightarrow$ Site 1
  † Site 1 computes $R' = R \bowtie_A S'$
  † $R' \rightarrow$ Site 2
  † Site 2 computes $R' \bowtie_A S$

Semijoin is better if

$$\text{size}(\Pi_A(S)) + \text{size}(R \bowtie_A S)) < \text{size}(R)$$
Semijoin Algorithms

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

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

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

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

(a) Cyclic query
(b) Equivalent acyclic query
Distributed Dynamic Algorithm

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

**Algorithm 8.4: Dynamic*-QOA**

**Input:** $MRQ$: multirelation query  
**Output:** result of the last multirelation query  
**begin**

- for each detachable $ORQ_i$ in $MRQ$ do  
  - $run(ORQ_i)$  
    - \( MRQ'_\text{list} \leftarrow \text{REDUCE}(MRQ) \) \{ MRQ repl. by \( n \) irreducible queries \} (2)

- while \( n \neq 0 \) do  
  - \{ choose next irreducible query involving the smallest fragments \}
    - $MRQ' \leftarrow \text{SELECT\_QUERY}(MRQ'_\text{list})$;  
      - \{ determine fragments to transfer and processing site for $MRQ'$ \}
    - Fragment-site-list $\leftarrow \text{SELECT\_STRATEGY}(MRQ')$;  
    - \{ move the selected fragments to the selected sites \}
    - for each pair $(F, S)$ in Fragment-site-list do
      - move fragment $F$ to site $S$  
    - execute $MRQ'$;  
    - \( n \leftarrow n - 1 \)  
    - \{ output is the result of the last $MRQ'$ \}

**end**
Distributed Dynamic Algorithm - Example

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

<table>
<thead>
<tr>
<th></th>
<th>Site 1</th>
<th>Site 2</th>
<th>Site 3</th>
<th>Site 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJ</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>ASG</td>
<td></td>
<td></td>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>
Distributed Static Algorithm

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

**Algorithm 8.5: Static*-QOA**

**Input:** $QT$: query tree  
**Output:** $strat$: minimum cost strategy

begin

for each relation $R_i \in QT$ do

for each access path $AP_{ij}$ to $R_i$ do

compute $cost(AP_{ij})$

$best \cdot AP_i \leftarrow AP_{ij}$ with minimum cost

end

for each order $(R_{i1}, R_{i2}, \cdots, R_{in})$ with $i = 1, \cdots, n!$ do

build strategy $((best \cdot AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie \cdots \bowtie R_{in})$;

compute the cost of strategy

$strat \leftarrow$ strategy with minimum cost;

end

for each site $k$ storing a relation involved in $QT$ do

$L_{S_k} \leftarrow$ local strategy $(strategy, k)$;

send $(L_{S_k}, site k)$ \{each local strategy is optimized at site $k$\}

end
Static Approach – Performing Joins

- 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
Static Approach – Vertical Partitioning & Joins

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

Total Cost = \[ \text{cost(retrieving qualified outer tuples)} + \text{no. of outer tuples fetched} \times \text{cost(retrieving qualified inner tuples)} + \text{msg. cost} \times \left( \frac{\text{no. outer tuples fetched} \times \text{avg. outer tuple size}}{\text{msg. size}} \right) \]
Static Approach – Vertical Partitioning & Joins

2. Move inner relation to the site of outer relation
   Cannot join as they arrive; they need to be stored
   Total cost = cost(retrieving qualified outer tuples)
   + no. of outer tuples fetched * cost(retrieving matching inner tuples from temporary storage)
   + cost(retrieving qualified inner tuples)
   + cost(storing all qualified inner tuples in temporary storage)
   + msg. cost * no. of inner tuples fetched * avg. inner tuple size/msg. size
Static Approach – Vertical Partitioning & Joins

3. Fetch inner tuples as needed
   (a) Retrieve qualified tuples at outer relation site
   (b) Send request containing join column value(s) for outer tuples to inner relation site
   (c) Retrieve matching inner tuples at inner relation site
   (d) Send the matching inner tuples to outer relation site
   (e) Join as they arrive

Total Cost = cost(retrieving qualified outer tuples) + msg. cost * (no. of outer tuples fetched)
            + no. of outer tuples fetched * no. of inner tuples fetched * avg. inner tuple size * msg. cost / msg. size)
            + no. of outer tuples fetched * cost(retrieving matching inner tuples for one outer value)
4. Move both inner and outer relations to another site

Total cost \( = \) cost(retrieving qualified outer tuples) 
+ cost(retrieving qualified inner tuples) 
+ cost(storing inner tuples in storage) 
+ msg. cost \( \times \) (no. of outer tuples fetched \( \times \) avg. outer tuple size)/msg. size 
+ msg. cost \( \times \) (no. of inner tuples fetched \( \times \) avg. inner tuple size)/msg. size 
+ no. of outer tuples fetched \( \times \) cost(retrieving inner tuples from temporary storage)
Static Approach – Example

- Join of relations PROJ, the external relation, and ASG, the internal relation, on attribute PNO
  - \( \text{PROJ} \bowtie \text{ASG} \)
- We assume that
  - PROJ and ASG are stored at two different sites
  - there is an index on attribute PNO for relation ASG
- The possible execution strategies for the query are as follows:
  1. Ship whole PROJ to site of ASG.
  2. Ship whole ASG to site of PROJ.
  3. Fetch ASG tuples as needed for each tuple of PROJ.
  4. Move ASG and PROJ to a third site.
- Discussion
  - Strategy 4: the highest cost since both relations must be transferred
  - Strategy 2: size(PROJ) >> size(ASG)
    - minimizes the communication time
    - likely to be the best (if local processing time is not too high compared to
      - strategies 1 and 3)
Static Approach – Example

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

- Semijoin
  - SDD1 selects only locally optimal schedules
- Dynamic and static approaches have the same advantages and drawbacks as in centralized case
  - But the problems of accurate cost estimation at compile-time are more severe
    - More variations at runtime
    - Relations may be replicated, making site and copy selection important
- Hybrid optimization
  - Choose-plan approach can be used
  - 2-step approach simpler
2-Step Optimization

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

(a) Static plan

(b) Run-time plan
2-Step – Problem Definition

• Given
  † A set of sites $S = \{s_1, s_2, ..., s_n\}$ with the load of each site
  † A query $Q = \{q_1, q_2, q_3, q_4\}$ such that each subquery $q_i$ is the maximum processing unit that accesses one relation and communicates with its neighboring queries
  † For each $q_i$ in $Q$, a feasible allocation set of sites $S_q = \{s_1, s_2, ..., s_k\}$ where each site stores a copy of the relation in $q_i$

• The objective is to find an optimal allocation of $Q$ to $S$ such that
  † the load unbalance of $S$ is minimized
  † The total communication cost is minimized
2-Step – Problem Definition

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

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

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

$$\text{UF}(S) = \frac{1}{n} \sum_{i=1}^{n} \left( \text{load}(s_i) - \text{Avg}_{\text{load}}(S) \right)^2$$
2-Step – Problem Definition

• The problem addressed by the second step of two-step query optimization can be formalized as the following subquery allocation problem. Given
  1. a set of sites \( S = \{s_1, ..., s_n\} \) with the load of each site;
  2. a query \( Q = \{q_1, ..., q_m\} \); and
  3. for each subquery \( q_i \) in \( Q \), a feasible allocation set of sites \( S_{q_i} = \{s_1, ..., s_k\} \)

where each site stores a copy of the relation involved in \( q_i \);

• the objective is to find an optimal allocation on \( Q \) to \( S \) such that
  1. \( UF(S) \) is minimized, and
  2. the total communication cost is minimized.
2-Step – Algorithm

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

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

Algorithm 8.7: SQAllocation

Input: $Q: q_1, \ldots, q_m$;
Feasible allocation sets: $S_{q_1}, \ldots, S_{q_m}$;
Loads: $load(S_1), \ldots, load(S_m)$;
Output: an allocation of $Q$ to $S$

begin

for each $q$ in $Q$ do
\hspace{1em} compute($load(S_q)$)

while $Q$ not empty do
\hspace{1em} $a \leftarrow q \in Q$ with least allocation flexibility; \{select subquery $a$ for allocation\} \hspace{1em} (1)
\hspace{1em} $b \leftarrow s \in S_a$ with least load and best benefit; \{select best site $b$ for $a$\} \hspace{1em} (2)
\hspace{1em} $Q \leftarrow Q - a$;
\hspace{1em} \{recompute loads of remaining feasible allocation sets if necessary\} \hspace{1em} (3)

for each $q \in Q$ where $b \in S_q$ do
\hspace{2em} compute($load(S_q)$)

end
2-Step Algorithm Example

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

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