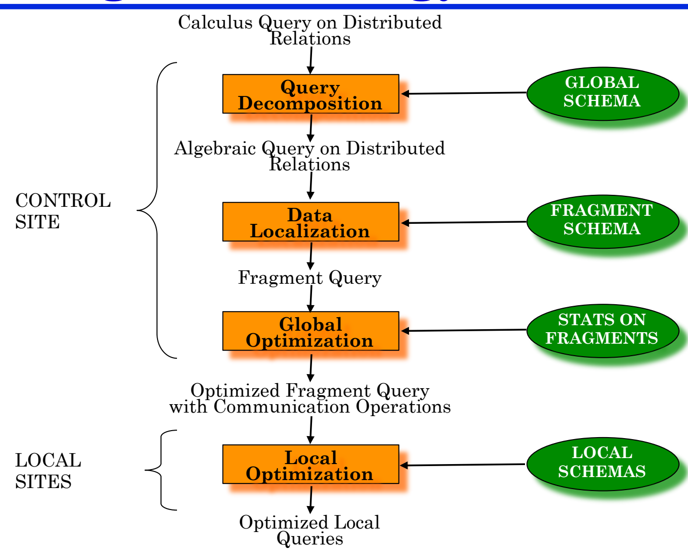
Outline

- Introduction & architectural issues
- Data distribution
- Distributed query processing
- ❑Distributed query optimization
- ❑Distributed transactions & concurrency control
- ❑Distributed reliability
- ❑Data replication
- ❑Parallel database systems
- ❑Database integration & querying
- ❑Peer-to-Peer data management
- ❑Stream data management
- ❑MapReduce-based distributed data management

Distributed Query Processing Methodology



Step 3 – Global Query Optimization

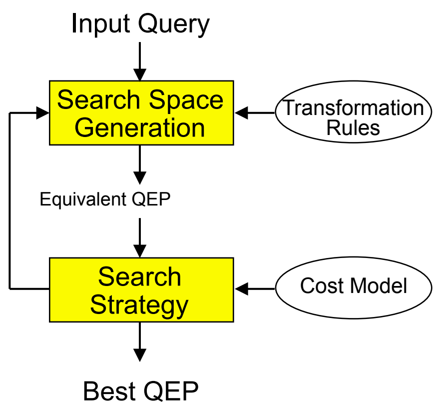
Input: Fragment query

- 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,…)

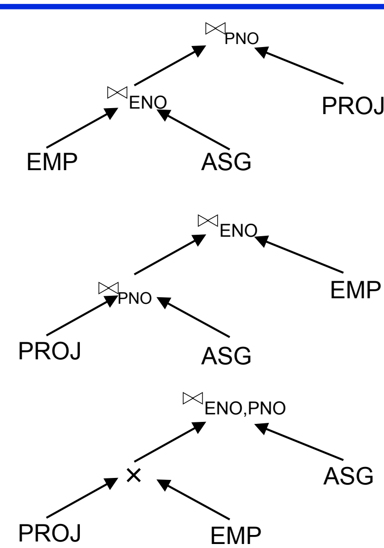
Query Optimization Process



Search Space

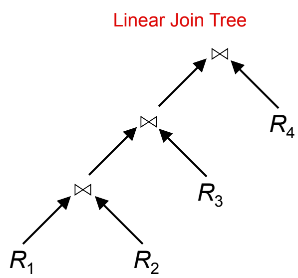
- Search space characterized by alternative execution
- Focus on join trees
- For N relations, there are $O(N!)$ equivalent join trees that can be obtained by applying commutativity and associativity rules

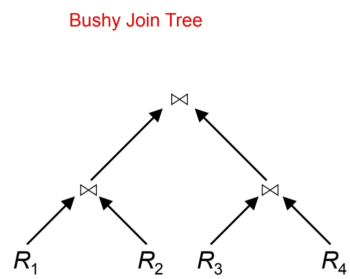
```
SELECT  ENAME,RESP
FROM    EMP, ASG,PROJ
WHERE   EMP.ENO=ASG.ENO
AND     ASG.PNO=PROJ.PNO
```



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



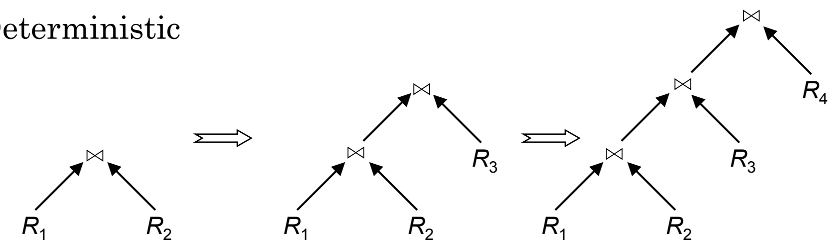


Search Strategy

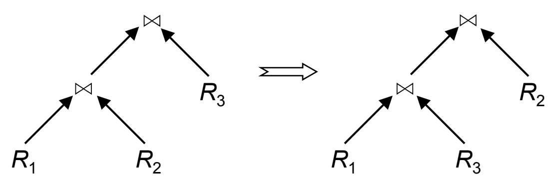
- How to “move” in the search space.
- Deterministic
  - Start from base relations and build plans by adding one relation at each step
  - Dynamic programming: breadth-first
  - Greedy: depth-first
- Randomized
  - Search for optimalities around a particular starting point
  - Trade optimization time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

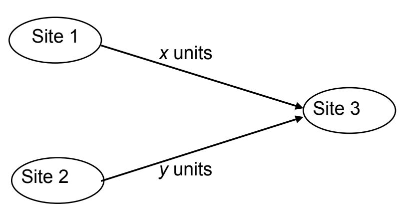
CPU time = unit instruction time * no. of sequential instructions

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 · message initialization time + unit transmission time * $(x+y)$

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

time to send x from 1 to 3 = message initialization time + unit transmission time * x

time to send y from 2 to 3 = message initialization time + unit transmission time * y

Optimization Statistics

- Primary cost factor: size of intermediate relations
  - 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: $length(A_i)$
  - the number of distinct values for each attribute in each fragment: $card(\Pi_{A_i} R_j)$
  - maximum and minimum values in the domain of each attribute: $min(A_i)$, $max(A_i)$
  - the cardinalities of each domain: $card(dom[A_i])$
- The cardinalities of each fragment: $card(R_j)$ Selectivity factor of each operation for relations
  - For joins

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

Intermediate Relation Sizes

Selection

$size(R) = card(R) \cdot length(R)$

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

where

$$S\,F_\sigma(A = value) = \frac{1}{card(\prod_A(R))}$$

$$S\,F_\sigma(A > value) = \frac{max(A) - value}{max(A) - min(A)}$$

$$S\,F_\sigma(A < value) = \frac{value - max(A)}{max(A) - min(A)}$$

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

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

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

Intermediate Relation Sizes

Projection

$card(\Pi_A(R)) = card(R)$

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

Intermediate Relation Size

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:

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

Semijoin

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

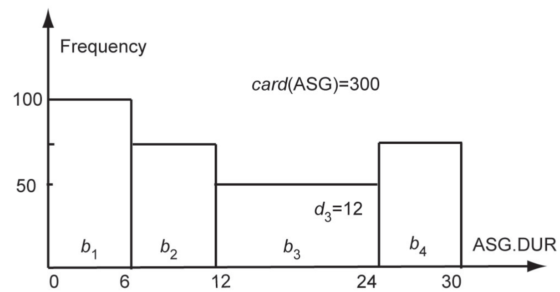
where

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

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: $SF_\sigma(A = value) = 1/d_i$
  - 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

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

❷ 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

$$q_1 \rightarrow q_2 \rightarrow \dots \rightarrow q_n$$

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

    $q(V_1, V_2, \dots V_n) \rightarrow (q'(t_1, V_2, V_2, \dots, V_n), t_1 \in R)$

Detachment

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

$\Downarrow$

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

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

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"
```

$\Downarrow$

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

$\Downarrow$

```
q12:  SELECT   ASG.ENO INTO GVAR
      FROM     ASG,JVAR
      WHERE    ASG.PNO=JVAR.PNO

q13:  SELECT   EMP.ENAME
      FROM     EMP,GVAR
      WHERE    EMP.ENO=GVAR.ENO
```


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: $\langle$E1$\rangle$ and $\langle$E2$\rangle$

Then q_{13} becomes

```
q131: SELECT   EMP.ENAME
      FROM     EMP
      WHERE    EMP.ENO="E1"

q132: SELECT   EMP.ENAME
      FROM     EMP
      WHERE    EMP.ENO="E2"
```

Static Algorithm

❶ Simple (i.e., mono-relation) queries are executed according to the best access path

❷ 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: $n_1 * n_2$

- Merge join

  sort relations
  merge relations

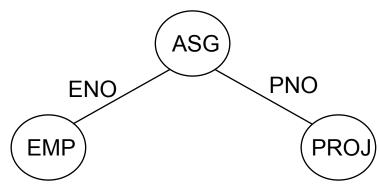
  - Complexity: $n_1 + n_2$ if relations are previously sorted and equijoin

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)

❶ Choose the best access paths to each relation

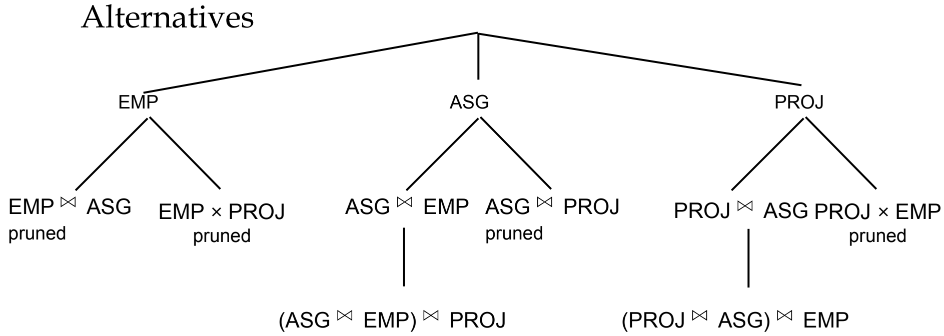
- EMP: sequential scan (no selection on EMP)
- ASG: sequential scan (no selection on ASG)
- PROJ: index on PNAME (there is a selection on PROJ based on PNAME)

❷ 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

((ASG $\bowtie$ EMP) $\bowtie$ PROJ)

((PROJ $\bowtie$ ASG) $\bowtie$ EMP)

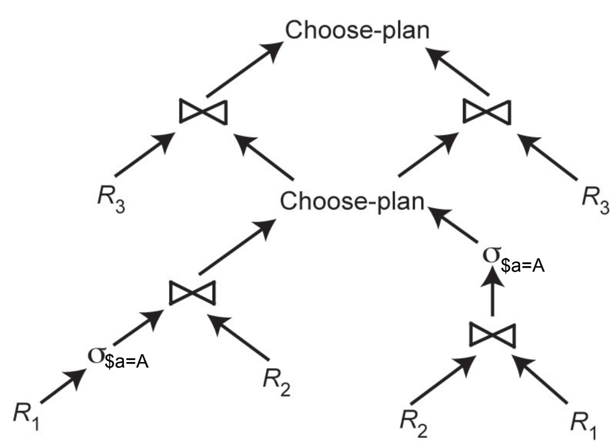
Static Algorithm

- ((PROJ $\bowtie$ ASG) $\bowtie$ 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

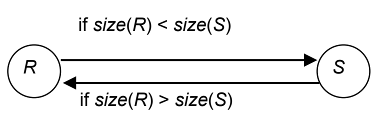


Join Ordering in Fragment Queries

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

Join Ordering

- Consider two relations only

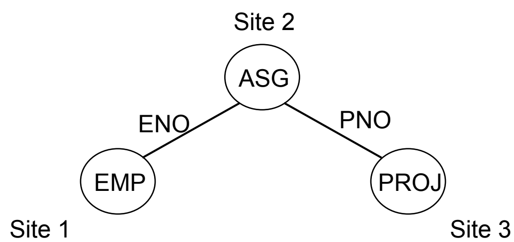


- Multiple relations more difficult because too many alternatives.
  - Compute the cost of all alternatives and select the best one.
    - Necessary to compute the size of intermediate relations which is difficult.
  - Use heuristics

Join Ordering – Example

Consider

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



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

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

Semijoin Algorithms

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

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

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}
1. Choose q_i involving the smallest fragments to execute (call MRQ')
2. Find the best execution strategy for MRQ'
   a) Determine processing site
   b) Determine fragments to move
3. Repeat 3 and 4

Static Approach

- 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

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 = cost(retrieving qualified outer tuples)
+ no. of outer tuples fetched * cost(retrieving qualified inner tuples)
+ msg. cost * (no. outer tuples fetched * avg. outer tuple size)/msg. size

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. 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 · (no. of outer tuples fetched *
avg. outer tuple size)/msg. size
+ msg. cost * (no. of inner tuples fetched *
avg. inner tuple size)/msg. size
+ no. of outer tuples fetched * cost(retrieving
inner tuples from temporary storage)

Static Approach – Vertical Partitioning & Joins

4. 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

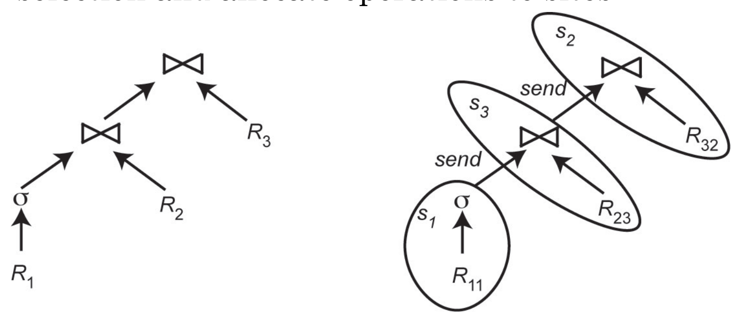
Total Cost = cost(retrieving qualified outer tuples)
+ msg. cost * (no. of outer tuples fetched)
+ no. of outer tuples fetched * no. of inner tuples fetched * avg. inner tuple size * msg. cost / msg. size)
+ no. of outer tuples fetched * cost(retrieving matching inner tuples for one outer value)

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 $S = \{s_1, s_2, \ldots, s_n\}$ 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 $S_q = \{s_1, s_2, \ldots, s_k\}$ 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 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 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 load(s_1)=2
- Iteration 2: select q_2, allocate to s_2, set load(s_2)=3
- Iteration 3: select q_3, allocate to s_1, set load(s_1) =3
- Iteration 4: select q_1, allocate to s_3 or s_4

sites	load	R_1	R_2	R_3	R_4
s_1	1	R_{11}		R_{31}	R_{41}
s_2	2		R_{22}		
s_3	2	R_{13}		R_{33}	
s_4	2	R_{14}	R_{24}		

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