Principles of Distributed Database Systems

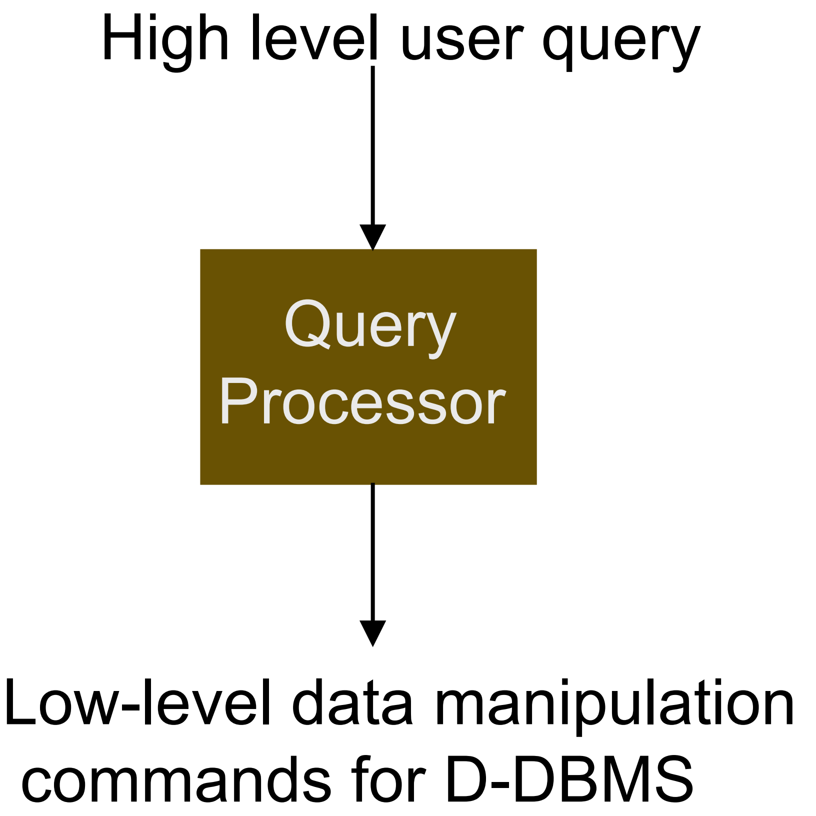
Outline

- Introduction
- Distributed and parallel database design
- Distributed data control
- Distributed Query Processing
- Distributed Transaction Processing
- Data Replication
- Database Integration – Multidatabase Systems
- Parallel Database Systems
- Peer-to-Peer Data Management
- Big Data Processing
- NoSQL, NewSQL and Polystores
- Web Data Management

Outline

- Distributed Query Processing
  - Query Decomposition and Localization
  - Join Ordering
  - Distributed Query Optimization
  - Adaptive Query Processing

Query Processing in a DDBMS



Query Processing Components

- Query language
  - SQL: “intergalactic dataspeak”
- Query execution
  - The steps that one goes through in executing high-level (declarative) user queries.
- Query optimization
  - How do we determine the “best” execution plan?
- We assume a homogeneous D-DBMS

Selecting Alternatives

```
SELECT   ENAME
FROM     EMP NATURAL JOIN ASG
WHERE    RESP = "Manager"
```

Strategy 1

$$\Pi_{\text{ENAME}}(\sigma_{\text{RESP="Manager"} \wedge \text{EMP.ENO=ASG.ENO}}(\text{EMP} \times \text{ASG}))$$

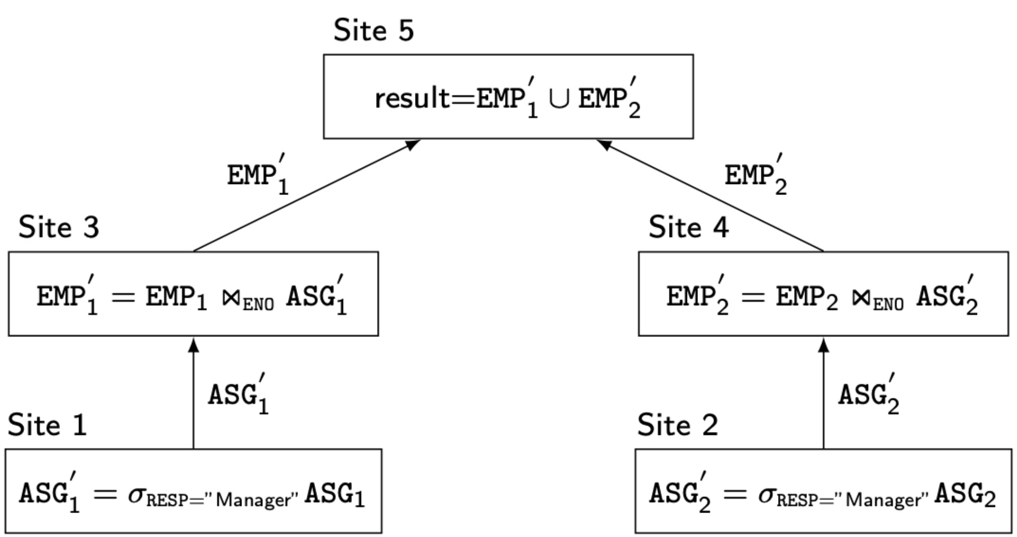
Strategy 2

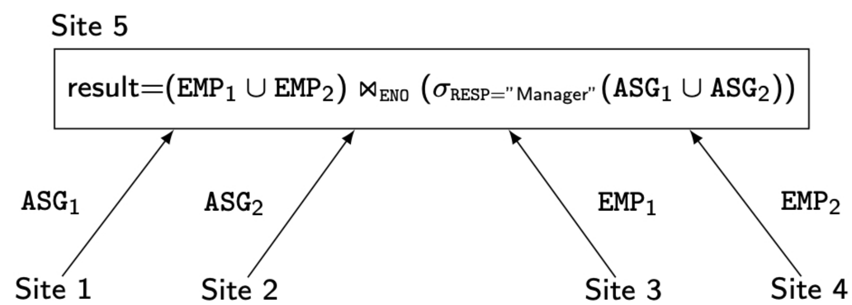
$$\Pi_{\text{ENAME}}(\text{EMP} \bowtie_{\text{ENO}} (\sigma_{\text{RESP="Manager"}}(\text{ASG}))$$

Strategy 2 avoids Cartesian product, so may be “better”

What is the Problem?

Site 1	Site 2	Site 3	Site 4	Site 5
$ASG_1=\sigma_{ENO\leq\text{“E3”}}(ASG)$	$ASG_2=\sigma_{ENO>\text{“E3”}}(ASG)$	$EMP_1=\sigma_{ENO\leq\text{“E3”}}(EMP)$	$EMP_2=\sigma_{ENO>\text{“E3”}}(EMP)$	Result





Cost of Alternatives

- Assume
  - *size*(EMP) = 400, *size*(ASG) = 1000
  - tuple access cost = 1 unit; tuple transfer cost = 10 units
- Strategy 1

produce ASG': (10+10) $*$ tuple access cost	20
transfer ASG' to the sites of EMP: (10+10) $*$ tuple transfer cost	200
produce EMP': (10+10) $*$ tuple access cost $*$ 2	40
transfer EMP' to result site: (10+10) $*$ tuple transfer cost	200
Total Cost	460

- Strategy 2

transfer EMP to site 5: 400 $*$ tuple transfer cost	4,000
transfer ASG to site 5: 1000 $*$ tuple transfer cost	10,000
produce ASG': 1000 $*$ tuple access cost	1,000
join EMP and ASG': 400 $*$ 20 $*$ tuple access cost	8,000
Total Cost	23,000

Query Optimization Objectives

- Minimize a cost function
  - I/O cost + CPU cost + communication cost
  - These might have different weights in different distributed environments
- Wide area networks
  - Communication cost may dominate or vary much
    - Bandwidth
    - Speed
    - Protocol overhead
- Local area networks
  - Communication cost not that dominant, so total cost function should be considered
- Can also maximize throughput

Complexity of Relational Operations

- Assume
  - Relations of cardinality *n*
  - Sequential scan

Operation	Complexity
Select Project (without duplicate elimination)	$O(n)$
Project (with duplicate elimination) Group	$O(n * \log n)$
Join Semi-join Division Set Operators	$O(n * \log n)$
Cartesian Product	$O(n^2)$

Types Of Optimizers

- Exhaustive search
  - Cost-based
  - Optimal
  - Combinatorial complexity in the number of relations
- Heuristics
  - Not optimal
  - Regroup common sub-expressions
  - Perform selection, projection first
  - Replace a join by a series of semijoins
  - Reorder operations to reduce intermediate relation size
  - Optimize individual operations

Optimization Granularity

- Single query at a time
  - Cannot use common intermediate results
- Multiple queries at a time
  - Efficient if many similar queries
  - Decision space is much larger

Optimization Timing

- Static
  - Compilation ➔ optimize prior to the execution
  - Difficult to estimate the size of the intermediate results⇒error propagation
  - Can amortize over many executions
- Dynamic
  - Run time optimization
  - Exact information on the intermediate relation sizes
  - Have to reoptimize for multiple executions
- Hybrid
  - Compile using a static algorithm
  - If the error in estimate sizes > threshold, reoptimize at run time

Statistics

- Relation
  - Cardinality
  - Size of a tuple
  - Fraction of tuples participating in a join with another relation
- Attribute
  - Cardinality of domain
  - Actual number of distinct values
- Simplifying assumptions
  - Independence between different attribute values
  - Uniform distribution of attribute values within their domain

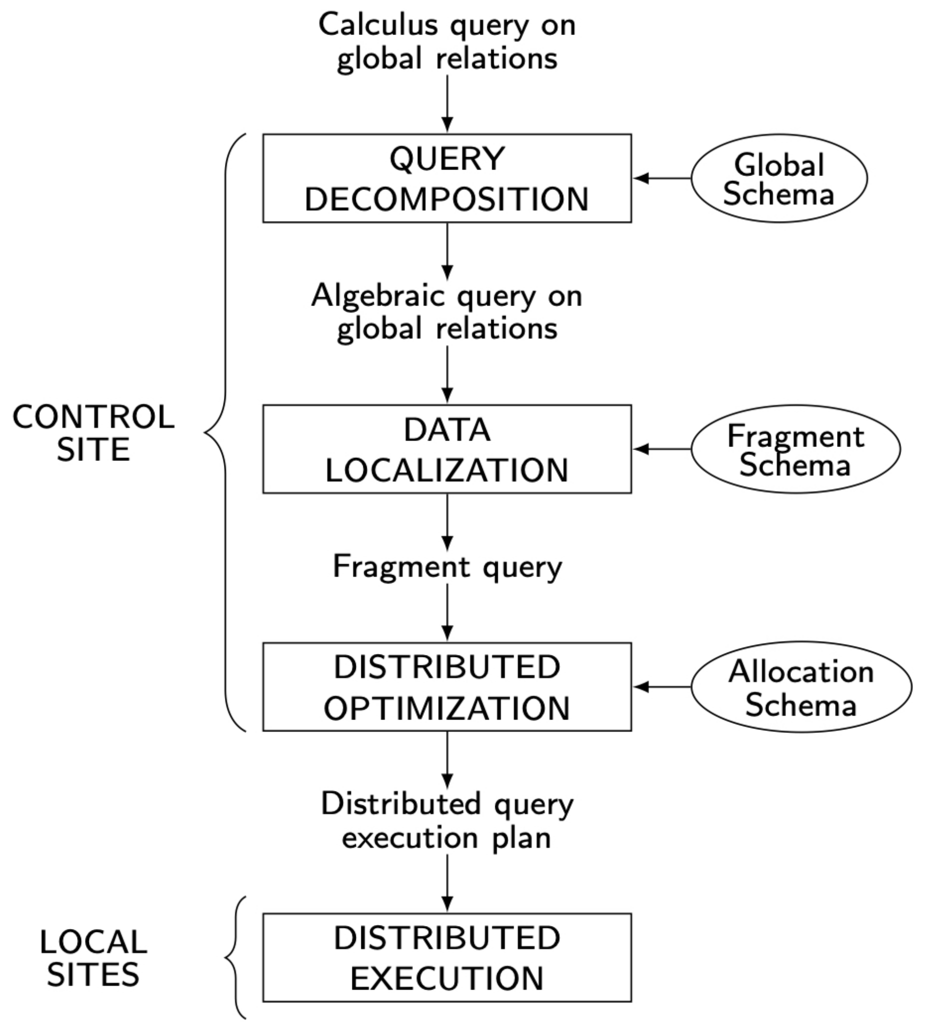
Optimization Decision Sites

- Centralized
  - Single site determines the “best” schedule
  - Simple
  - Need knowledge about the entire distributed database
- Distributed
  - Cooperation among sites to determine the schedule
  - Need only local information
  - Cost of cooperation
- Hybrid
  - One site determines the global schedule
  - Each site optimizes the local subqueries

Network Topology

- Wide area networks (WAN) – point-to-point
  - Characteristics
    - Relatively low bandwidth (compared to local CPU/IO)
    - High protocol overhead
  - Communication cost may dominate; ignore all other cost factors
  - Global schedule to minimize communication cost
  - Local schedules according to centralized query optimization
- Local area networks (LAN)
  - Communication cost not that dominant
  - Total cost function should be considered
  - Broadcasting can be exploited (joins)
  - Special algorithms exist for star networks

Distributed Query Processing Methodology



Outline

- Distributed Query Processing
  - Query Decomposition and Localization
  - Distributed Query Optimization
  - Join Ordering
  - Adaptive Query Processing

Step 1 – Query Decomposition

Same as centralized query processing

Input : Calculus query on global relations

- Normalization
  - Manipulate query quantifiers and qualification
- Analysis
  - Detect and reject “incorrect” queries
- Simplification
  - Eliminate redundant predicates
- Restructuring
  - Calculus query → algebraic query
  - Use transformation rules

Step 2 – Data Localization

Input: Algebraic query on distributed relations

- Determine which fragments are involved
- Localization program
  - Substitute for each global query its materialization program
  - Optimize

Example

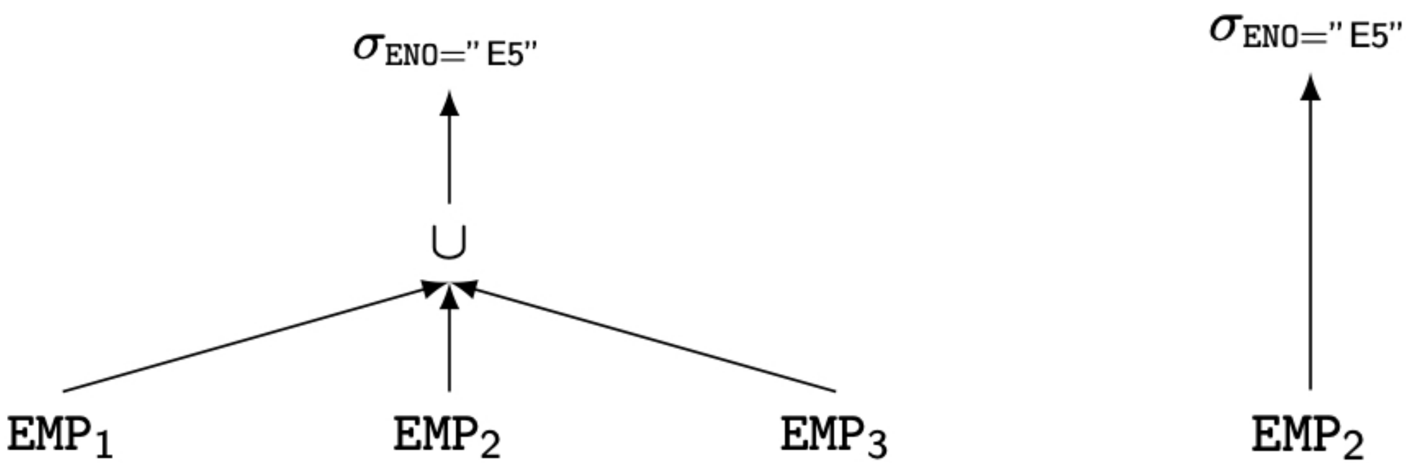
- Assume
  - EMP is fragmented as follows:
    - $EMP_1 = \sigma_{ENO \leq \text{"E3"}}(EMP)$
    - $EMP_2 = \sigma_{\text{"E3"} < ENO \leq \text{"E6"}}(EMP)$
    - $EMP_3 = \sigma_{ENO \geq \text{"E6"}}(EMP)$
  - ASG fragmented as follows:
    - $ASG_1 = \sigma_{ENO \leq \text{"E3"}}(ASG)$
    - $ASG_2 = \sigma_{ENO > \text{"E3"}}(ASG)$
- In any query
  - Replace EMP by $(EMP_1 \cup EMP_2 \cup EMP_3)$
  - Replace ASG by $(ASG_1 \cup ASG_2)$

Reduction for PHF

- Reduction with selection
  - Relation R and $F_R=\{R_1, R_2, \ldots, R_w\}$ where $R_j=\sigma_{p_j}(R)$

    $\sigma_{p_i}(R_j)=\varnothing$ if $\forall x$ in R: $\neg(p_i(x) \wedge p_j(x))$

```
SELECT *
        FROM   EMP
        WHERE  ENO="E5"
```



Reduction for PHF

- Reduction with join
  - Possible if fragmentation is done on join attribute
  - Distribute join over union

$$(R_1 \cup R_2) \bowtie S \Leftrightarrow (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

  - Given $R_i = \sigma_{p_i}(R)$ and $R_j = \sigma_{p_j}(R)$

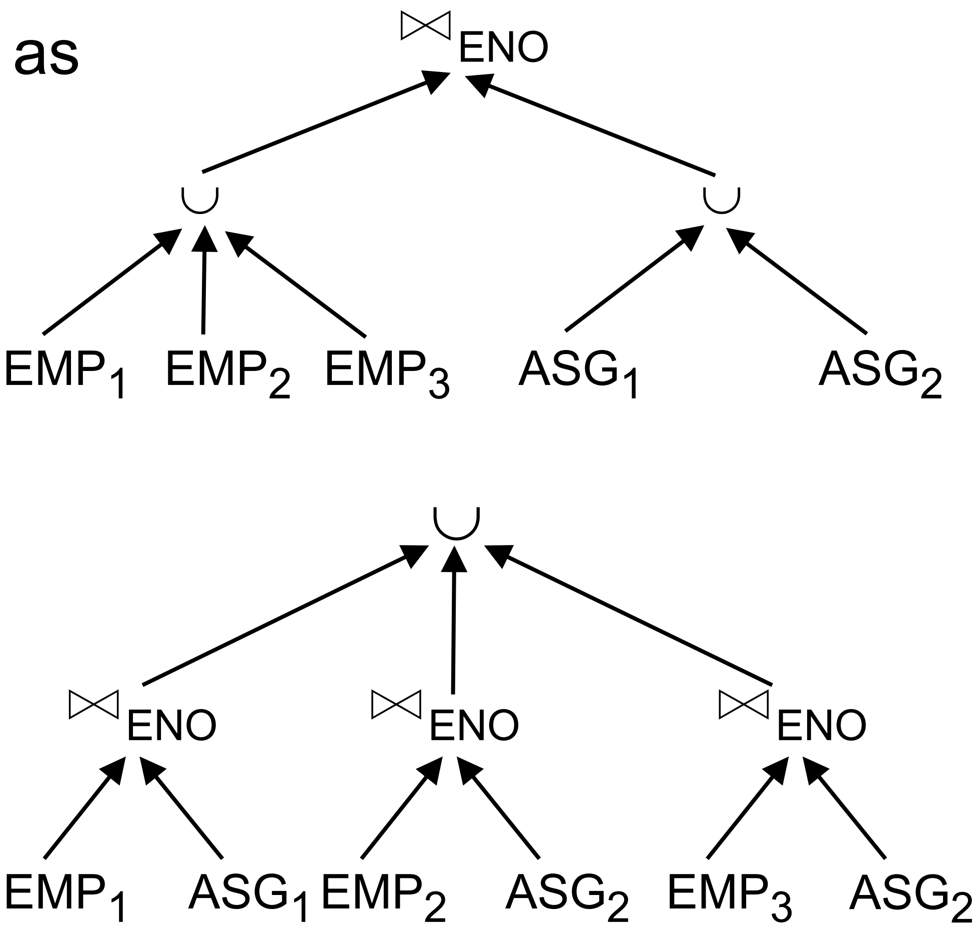
$$R_i \bowtie R_j = \emptyset \text{ if } \forall x \text{ in } R_i, \forall y \text{ in } R_j \text{: } \neg(p_i(x) \wedge p_j(y))$$

Reduction for PHF

- Assume EMP is fragmented as before and
  - ASG_1: $\sigma_{ENO \leq "E3"}(ASG)$
  - ASG_2: $\sigma_{ENO > "E3"}(ASG)$
- Consider the query

```
SELECT  *
FROM    EMP
NATURAL JOIN ASG
```

- Distribute join over unions
- Apply the reduction rule



Reduction for VF

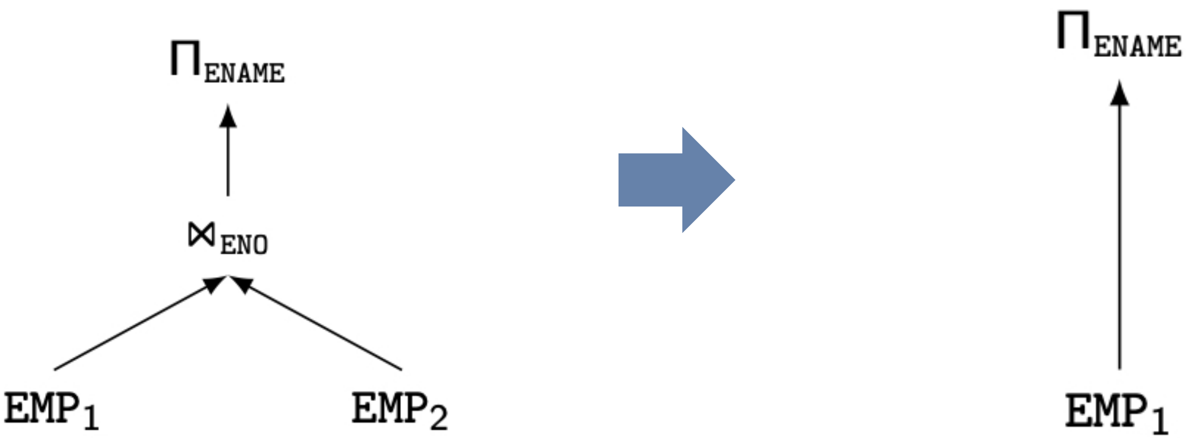
- Find useless (not empty) intermediate relations

Relation R defined over attributes $A = \{A_1, ..., A_n\}$ vertically fragmented as $R_i = \Pi_{A'}(R)$ where $A' \subseteq A$:

$\Pi_{D,K}(R_i)$ is useless if the set of projection attributes D is not in A'

Example: $EMP_1 = \Pi_{ENO,ENAME}(EMP)$; $EMP_2 = \Pi_{ENO,TITLE}(EMP)$

```
SELECT ENAME
FROM   EMP
```



Reduction for DHF

- Rule :
  - Distribute joins over unions
  - Apply the join reduction for horizontal fragmentation
- Example

  ASG_1: ASG $\ltimes_{ENO}$ EMP_1

  ASG_2: ASG $\ltimes_{ENO}$ EMP_2

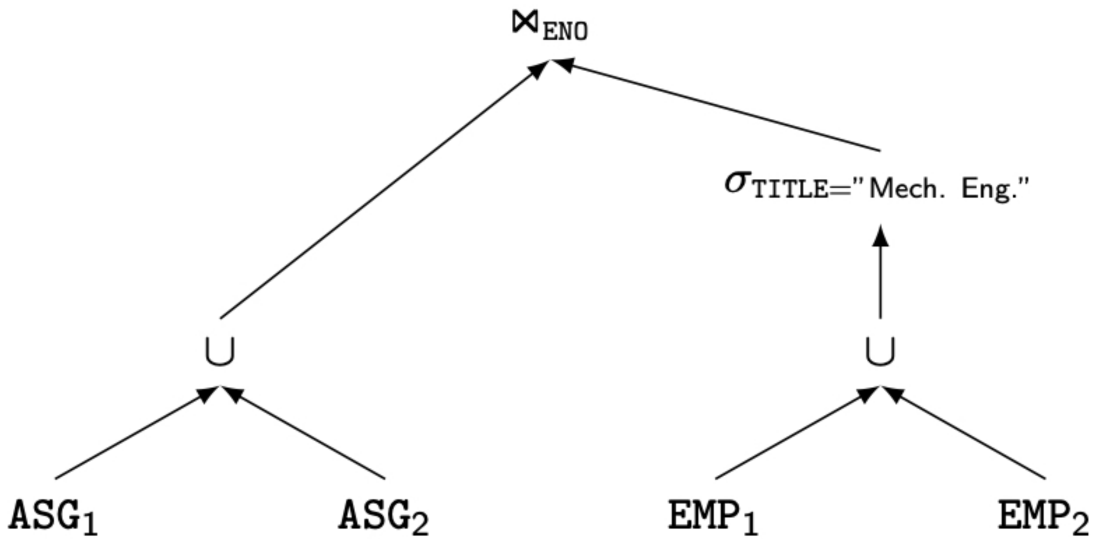
  EMP_1: $\sigma_{TITLE=\text{“Programmer”}}$ (EMP)

  EMP_2: $\sigma_{TITLE\neq\text{“Programmer”}}$ (EMP)
- Query

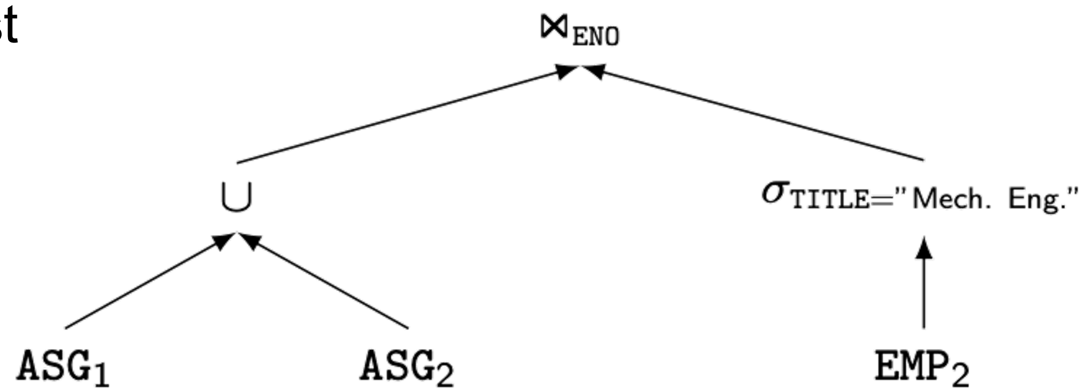
```
SELECT  *
FROM    EMP NATURAL JOIN ASG
WHERE   EMP.TITLE = "Mech. Eng."
```

Reduction for DHF

Generic query

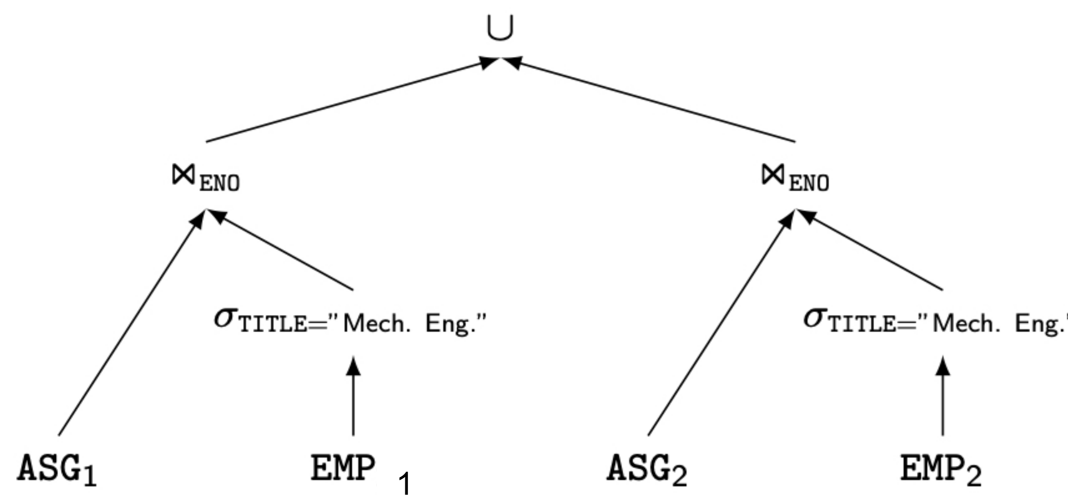


Selections first

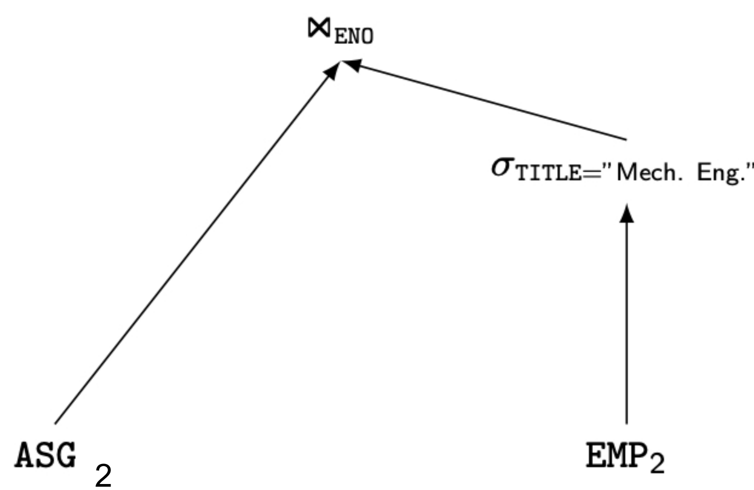


Reduction for DHF

Joins over unions



Elimination of the empty intermediate relations (left sub-tree)



Reduction for Hybrid Fragmentation

- Combine the rules already specified:
  - Remove empty relations generated by contradicting selections on horizontal fragments;
  - Remove useless relations generated by projections on vertical fragments;
  - Distribute joins over unions in order to isolate and remove useless joins.

Reduction for HF

Example

Consider the following hybrid fragmentation:

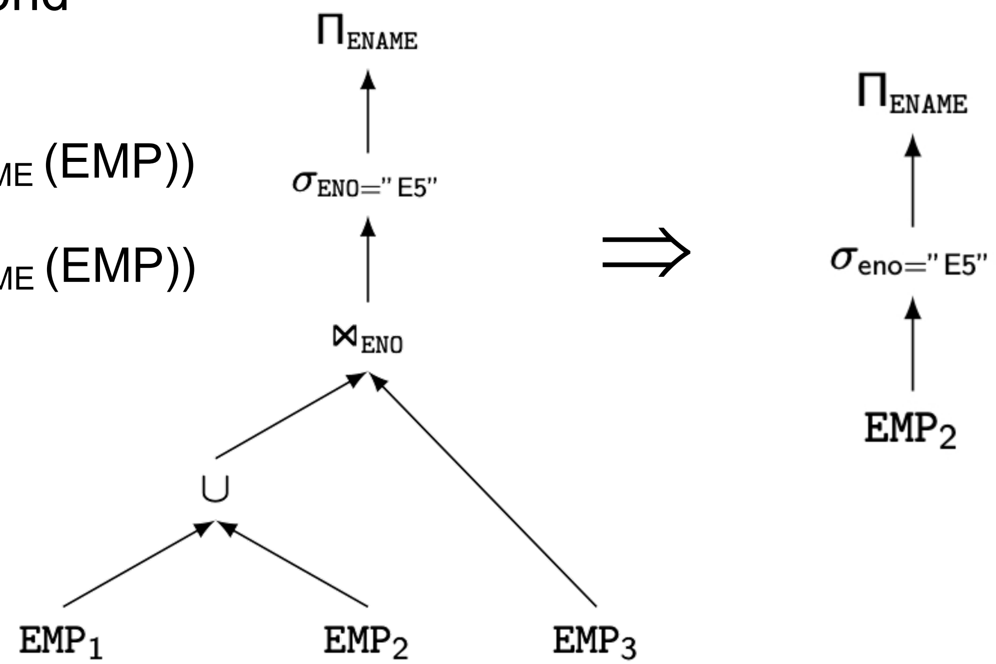
$EMP_1 = \sigma_{ENO\leq"E4"} (\Pi_{ENO,ENAME} (EMP))$

$EMP_2 = \sigma_{ENO>"E4"} (\Pi_{ENO,ENAME} (EMP))$

$EMP_3 = \sigma_{ENO,TITLE} (EMP)$

and the query

```
SELECT   ENAME
FROM     EMP
WHERE    ENO="E5"
```



Outline

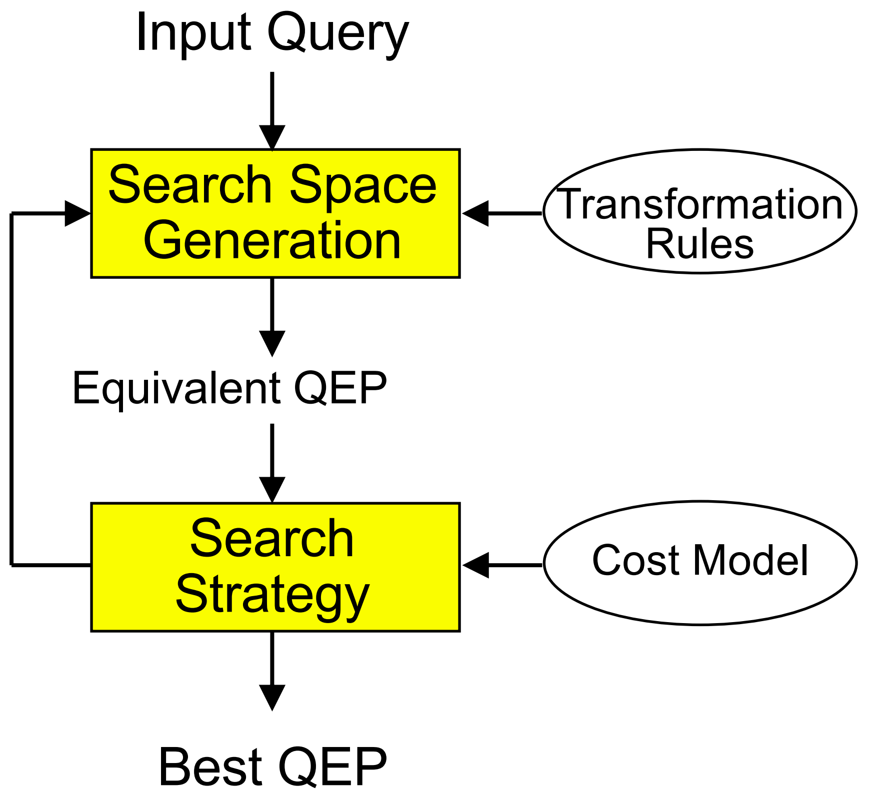
- Distributed Query Processing
  - Query Decomposition and Localization
  - Distributed Query Optimization
  - Join Ordering
  - Adaptive Query Processing

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, merge join or hash join

Query Optimization Process



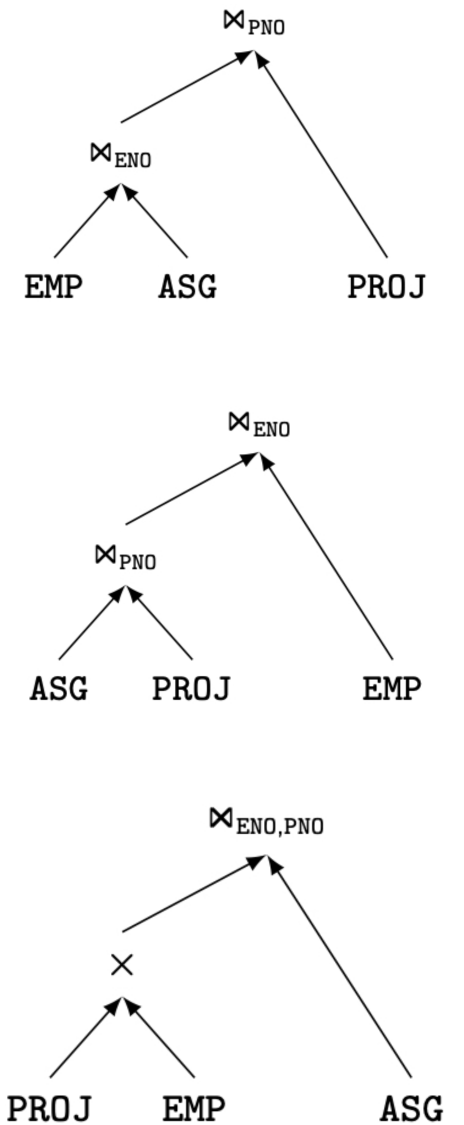
Components

- Search space
  - The set of equivalent algebra expressions (query trees)
- Cost model
  - 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,…)

Join Trees

- Characterize the search space for optimization
- For *N* relations, there are O(*N*!) equivalent join trees that can be obtained by applying commutativity and associativity rules

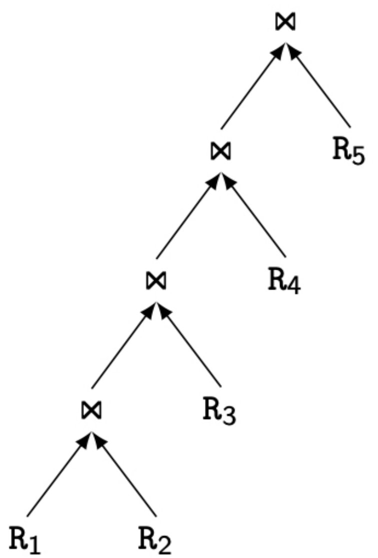
```
SELECT  ENAME,RESP
FROM    EMP
NATURAL JOIN ASG
NATURAL JOIN PROJ
```



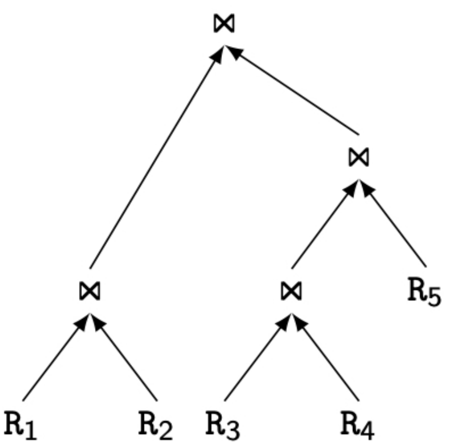
Join Trees

- Two major shapes
  - Linear versus bushy trees

Linear Join Tree



Bushy Join Tree

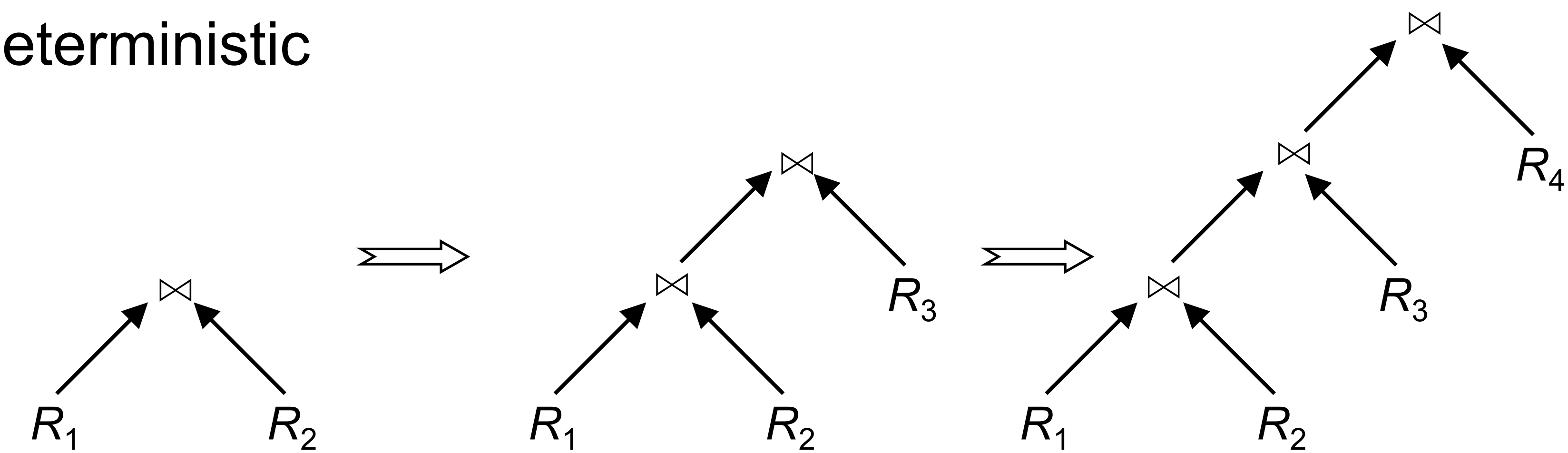


Search Strategy

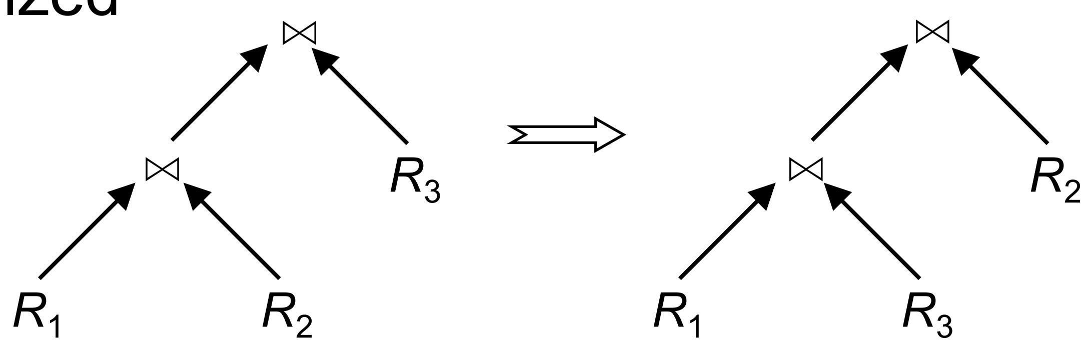
- How to “move” in the search space
- Deterministic
  - Start from base relations and build plans by adding one relation at each step
  - Dynamic programming: breadth-first
  - Greedy: depth-first
- Randomized
  - Search for optimalities around a particular starting point
  - Trade optimization time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

Search Strategies

- Deterministic



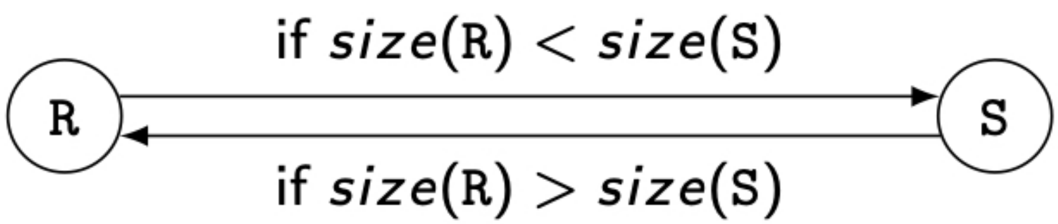
- Randomized



Outline

- Distributed Query Processing
  - Query Decomposition and Localization
  - Distributed Query Optimization
  - Join Ordering
  - Adaptive Query Processing

Join Ordering

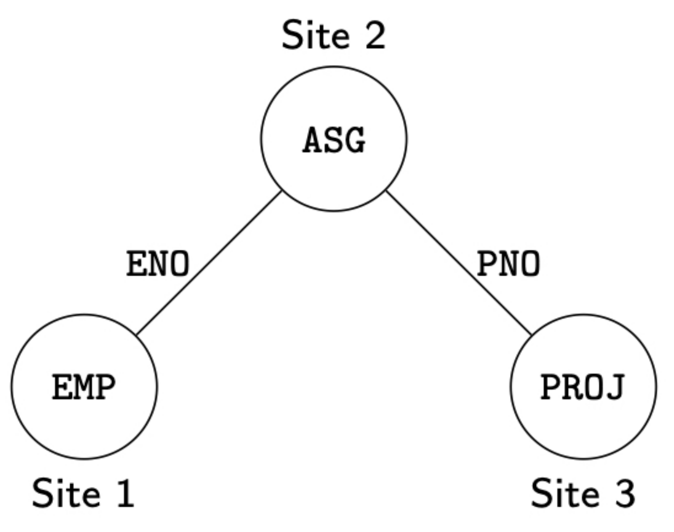


- Multiple relations more difficult because too many alternatives.
    - Compute the cost of all alternatives and select the best one.
        - Necessary to compute the size of intermediate relations which is difficult.
    - Use heuristics

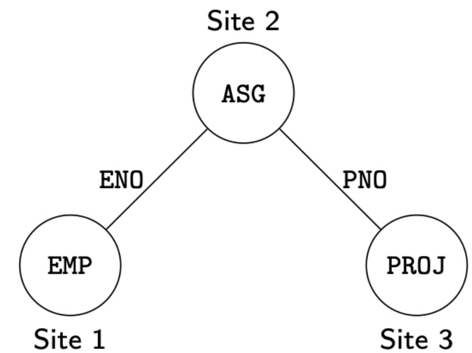
Join Ordering – Example

Consider

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



Join Ordering – Example



Execution alternatives

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

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

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

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

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

Semijoin-based Ordering

- 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-based Ordering

- 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

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

Full Reducer

- Optimal semijoin program that reduces each relation more than others
- How to find the full reducer?
  - Enumeration of all possible semijoin programs and select the one that has best size reduction
- Problem
  - For cyclic queries, no full reducers can be found
  - For tree queries, full reducers exist but the number of candidate semijoin programs is exponential in the number of relations
    - For chained queries, where relations can be ordered so that each relation joins only with the next relation, polynomial algorithms exist

Full Reducer – Example

Consider

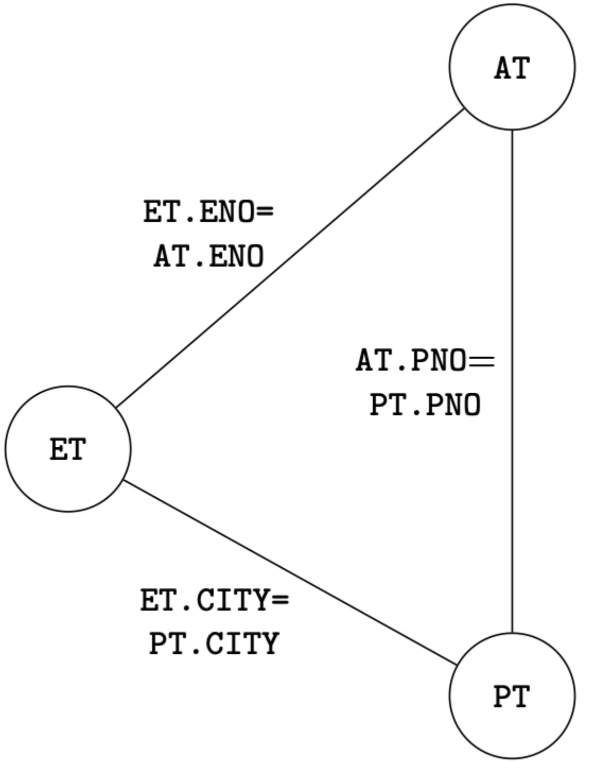
ET (ENO, ENAME, TITLE, CITY)

AT (ENO, PNO, RESP, DUR, CITY)

PT (PNO, PNAME, BUDGET, CITY)

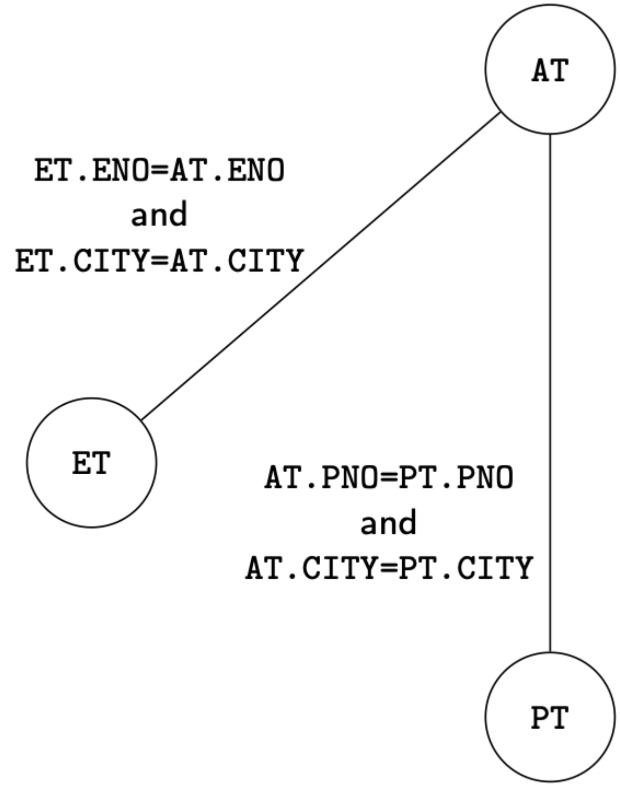
And the cyclic query

SELECT ENAME, PNAME
FROM ET **NATURAL JOIN** AT
NATURAL JOIN PT
NATURAL JOIN ET



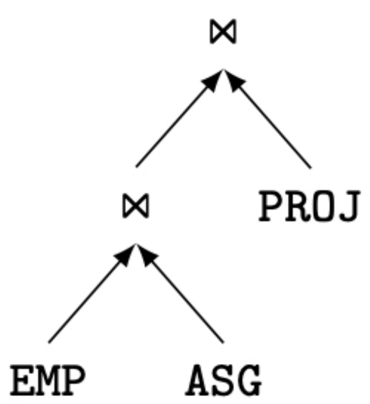
Full Reducer – example

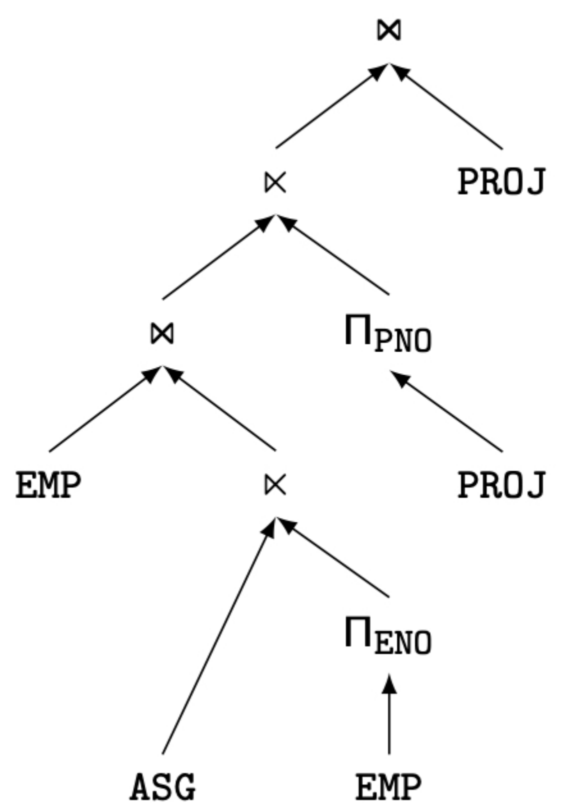
- Solution: transform the cyclic query into a tree
  - Remove one arc of the cyclic graph
  - Add appropriate predicates to other arcs such that the removed predicate is preserved by transitivity



Join versus Semijoin-based Ordering

- Semijoin-based induces more operators, but possibly on smaller operands





Distributed Cost Model

- 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 resource utilization and increases system throughput
  - Response Time
    - Do as many things as possible in parallel
    - May increase total time because of increased total activity

Total Time

Total time = CPU cost + I/O cost + com. Cost

The summation of all cost factors

CPU cost= unit instruction cost $*$ no.of instructions

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

com. cost = message initiation + transmission

Response Time

Response time = CPU time + I/O time + com. time

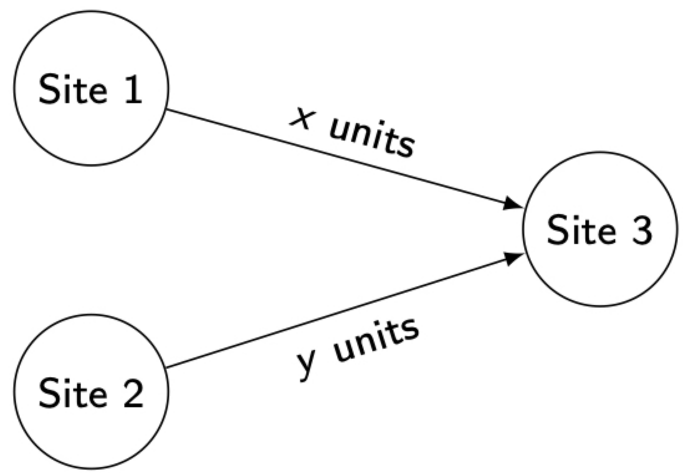
Must consider parallel execution

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

I/O time = unit I/O time * no. of seq I/Os

com. time = unit msg initiation time * no. of seq msgs
+ unit transmission time * no. of seq bytes

Example



- Consider communication cost only
  - Total time = 2 × msg 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}

Database Statistics

- Primary cost factor: size of intermediate relations
  - Need to estimate their sizes
- Make them precise $\Rightarrow$ 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 operator on relations
  - See centralized query optimization statistics

Distributed Query Optimization

- Dynamic approach
  - Distributed INGRES
  - No static cost estimation, only runtime cost information
- Static approach
  - System R*
  - Static cost model
- Hybrid approach
  - 2-step

Dynamic Approach

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'
    1. Determine processing site
    2. Determine fragments to move
5. Repeat 3 and 4

Dynamic Approach

Site 1	Site 2
EMP_1	EMP_2
ASG	PROJ

- q1 : **SELECT** EMP.ENAME **FROM** EMP **NATURAL JOIN** ASG **NATURAL JOIN** PROJ **WHERE** PNAME="CAD/CAM"

- possible strategies
  - Execute the entire query (EMP ⋈ ASG ⋈ PROJ) by moving EMP1 and ASG to site 2.
  - Execute (EMP ⋈ ASG) ⋈ PROJ by moving (EMP1 ⋈ ASG) and ASG to site 2, and so on.
- if size(EMP ⋈ ASG) > size(EMP1), strategy 1 is preferred to strategy 2.

Dynamic Approach

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

- Consider the query PROJ $\bowtie$ ASG
- With a point-to-point network,
  - the best strategy is to send each PROJi to site 3,
  - which requires a transfer of 3000 kbytes, versus 6000 kbytes if ASG is sent to sites 1, 2, and 4.
- With a 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.

Static Approach

- Cost function includes local processing as well as transmission
- Considers only joins
- “Exhaustive” search
- Compilation

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

$$internal \ltimes_A external$$

Static Approach – Vertical Partitioning & Joins

internal $\bowtie_A$ *external*

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

internal $\bowtie_A$ *external*

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 inner tuples)

   + no. of outer tuples fetched * cost(retrieving matching inner tuples from temporary storage)

   + cost(retrieving qualified outer 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

internal $\bowtie_A$ *external*

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

internal $\ltimes_A$ *external*

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)

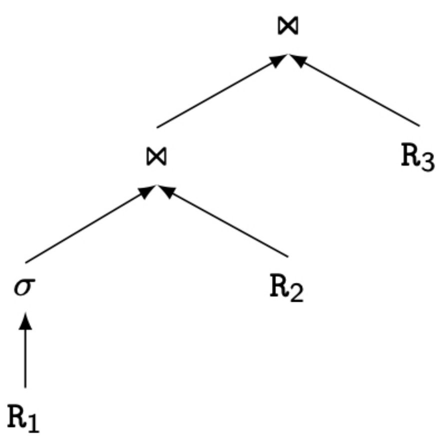
\+ no. of outer tuples fetched * cost(retrieving matching inner tuples for one outer value)

2-Step Optimization

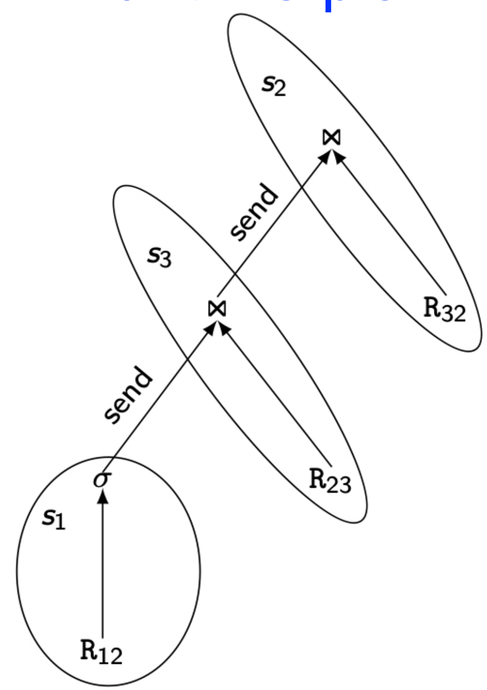
$\sigma(R_1) \bowtie R_2 \bowtie R_3$

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

Static plan



Runtime 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

$$\sigma(R_1) \bowtie R_2 \bowtie R_3 \bowtie R_4$$

$q_1,\quad q_2,\quad q_3,\quad q_4$

- 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

Outline

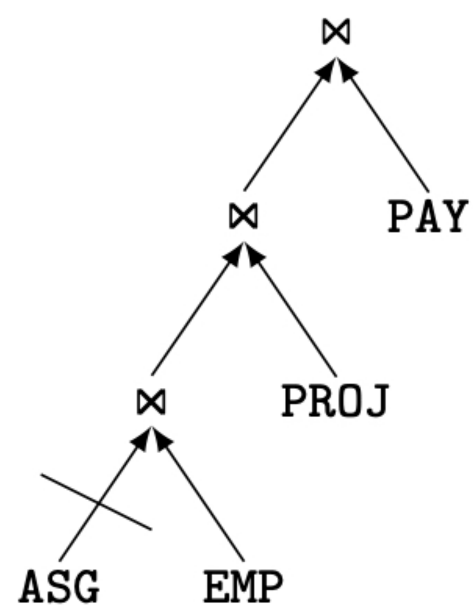
- Distributed Query Processing
  - Query Decomposition and Localization
  - Distributed Query Optimization
  - Join Ordering
  - Adaptive Query Processing

Adaptive Query Processing - Motivations

- Assumptions underlying query optimization
  - The optimizer has sufficient knowledge about runtime
    - Cost information
  - Runtime conditions remain stable during query execution
- Appropriate for systems with few data sources in a controlled environment
- Inappropriate for changing environments with large numbers of data sources and unpredictable runtime conditions

Example: QEP with Blocked Operator

- Assume ASG, EMP, PROJ and PAY each at a different site
- If ASG site is down, the entire pipeline is blocked
- However, with some reorganization, the join of EMP and PAY could be done while waiting for ASG



Adaptive Query Processing – Definition

- A query processing is adaptive if it receives information from the execution environment and determines its behavior accordingly
  - Feed-back loop between optimizer and runtime environment
  - Communication of runtime information between DDBMS components
- Additional components
  - Monitoring, assessment, reaction
  - Embedded in control operators of QEP
- Tradeoff between reactiveness and overhead of adaptation

Adaptive Components

- Monitoring parameters (collected by sensors in QEP)
  - Memory size
  - Data arrival rates
  - Actual statistics
  - Operator execution cost
  - Network throughput
- Adaptive reactions
  - Change schedule
  - Replace an operator by an equivalent one
  - Modify the behavior of an operator
  - Data repartitioning

Eddy Approach

- Query compilation: produces a tuple ⟨*D, P, C*, Eddy⟩
  - *D*: set of data sources (e.g. relations)
  - *P*: set of query predicates
  - *C*: ordering constraints to be followed at runtime
  - Eddy: *n*-ary operator between D and P
- Query execution: operator ordering on a tuple basis using Eddy
  - On-the-fly tuple routing to operators based on cost and selectivity
  - Change of join ordering during execution
    - Requires symmetric join algorithms such as Ripple joins

QEP with Eddy

$Q = (\sigma_p(R) \bowtie S \bowtie T)$

- $D = \{R, S, T\}$
- $P = \{\sigma_P(R), R \bowtie_1 S, S \bowtie_2 T\}$
- C = $\{S < T\}$ where < imposes S tuples to probe T tuples using an index on join attribute
  - Access to T is wrapped by $\bowtie$