
Distributed query optimization

Data Management for Big Data
2018-2019 (spring semester)

Dario Della Monica

These slides are a modified version of the slides provided with the book
Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

The original version of the slides is available at: extras.springer.com

Outline (distributed DB)

- Introduction (Ch. 1) ★
- Distributed Database Design (Ch. 3) ★
- Distributed Query Processing (Ch. 6-8) ★
 - Overview (Ch. 6) ★
 - Query decomposition and data localization (Ch. 7) ★
 - **Distributed query optimization (Ch. 8) ★**
- Distributed Transaction Management (Ch. 10-12) ★

★ Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Outline (today)

- Distributed query optimization (Ch. 8) [★]
 - Overview
 - Join Ordering in Localized Queries
 - Semijoin-based Algorithm
 - Distributed query optimization strategies
 - Hybrid approaches

[★] Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Distributed Query Optimization

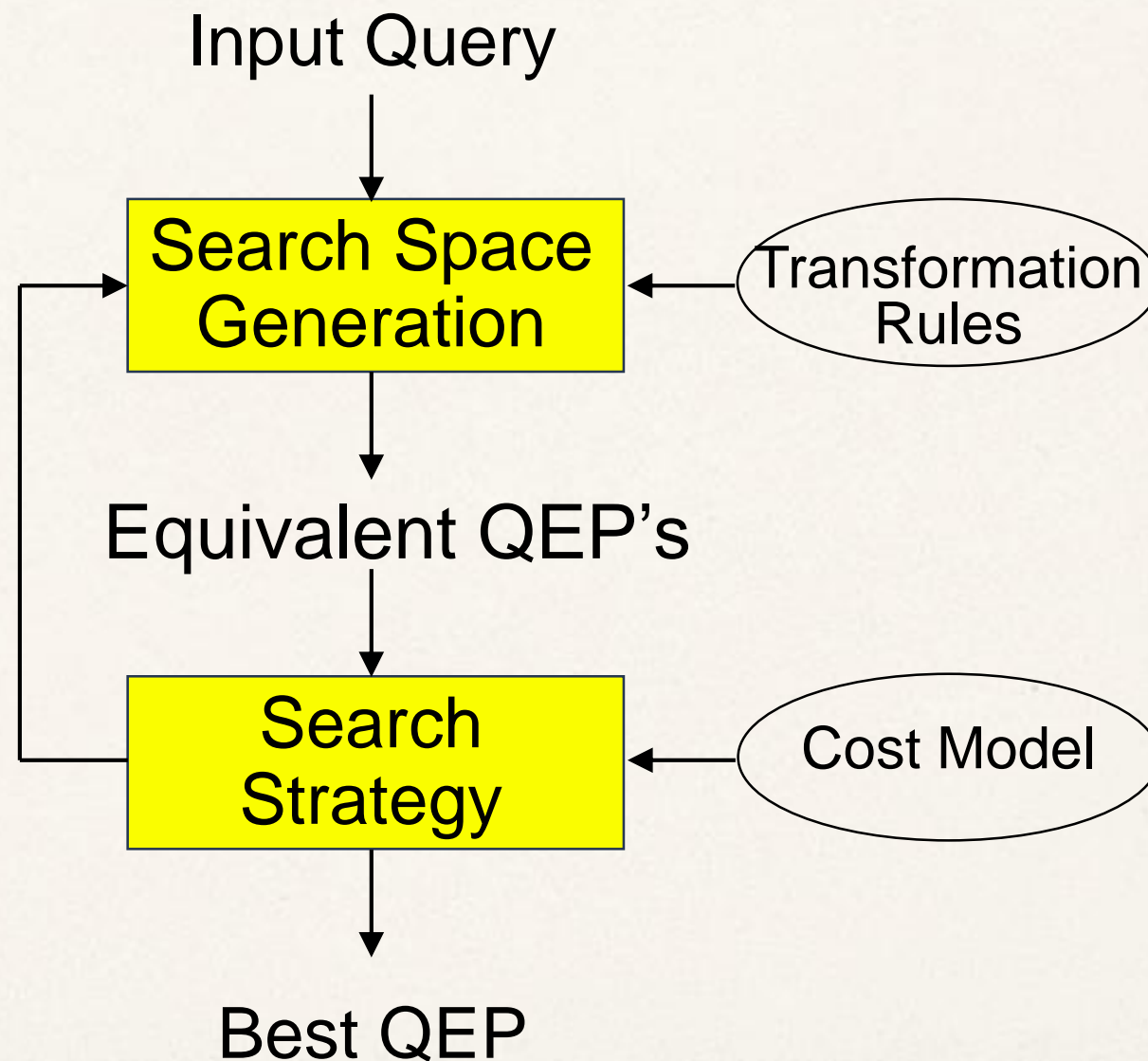
- In previous chapter (Ch. 7) ^{*}:
 - A distributed query is mapped into a query over fragments ([decomposition](#) and [data localization](#))
 - Simplification (“optimization”) independent from relation (fragment) statistics (e.g., cardinality)
- In this chapter (Ch. 8) ^{*}:
 - Optimization based on DB statistics (order of operations and operands, algorithm to perform simple operations) to produce a query execution plan (QEP)
 - ◆ In the distributed case a QEP is further extended with communication operations to support execution of queries over fragment sites
 - Once again: the problem is NP-hard, so not looking for the optimal solution
 - Statement of the problem
 - ◆ **Input:** Fragment query
 - ◆ **Output:** the *best* (not necessarily optimal) global schedule
 - Additional problems specific to the distributed setting
 - ◆ Where to execute (partial) queries? Which relation to ship where?
 - ◆ Choose between data transfer methods : ship-whole vs. fetch-as-needed
 - ◆ Decide on the use of semijoins (semijoins save on communication at the expense of more local processing)

^{*} Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Structure of the Optimizer

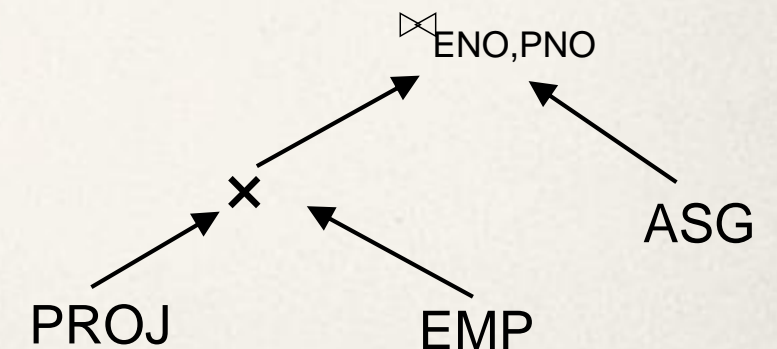
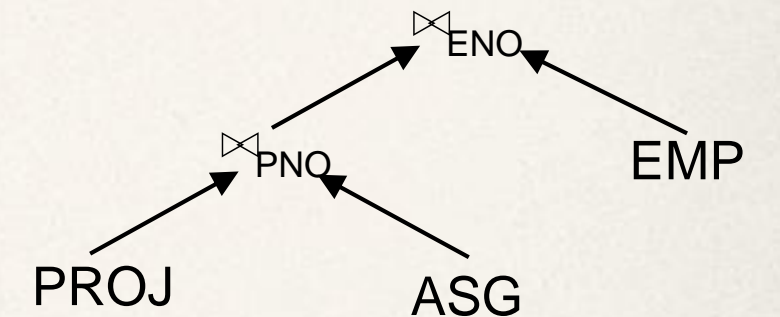
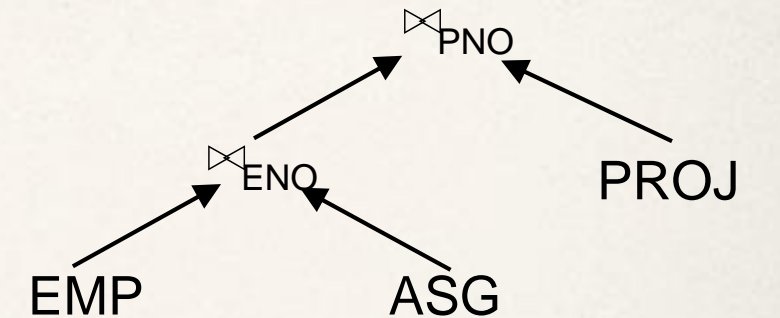
- Similar to the centralized case
 - Solution space (aka search space)
 - ◆ The set of equivalent QEP: algebra expressions enriched with implementation details **and communication choices**
 - Cost model
 - ◆ Cost prediction for local and global operations based on catalog statistics
 - ◆ Cost function (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
 - ✓ In early approach only communication costs were considered; due to fast communication technology, communication and I/O costs become comparable
 - Search algorithm (aka search strategy)
 - ◆ How do we move inside the solution space?
 - ✓ Exhaustive search, heuristic algorithms (iterative improvement, simulated annealing, genetic,...)
 - ◆ Goal is finding a good strategy according to the cost model
- **Difference** between centralized and **distributed** settings: **search space** and **cost model** (*search strategy* remains the same)

Query Optimization Process



Search Space

- Search space is large
 - N relations $\Rightarrow ((2(N-1))!)/((N-1)!)^*$ equivalent join trees (by join commutativity and associativity)
 - **SELECT** ENAME, RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO=ASG.ENO
AND ASG.PNO=PROJ.PNO
- Focus on join trees
- A difference
 - A good heuristics for centralized context: left-deep trees
 - In distributed context: non left-deep trees allow for parallelization



* In Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011: it is said $O(N!)$, which is incorrect

Centralized vs. Distributed Query Optimization

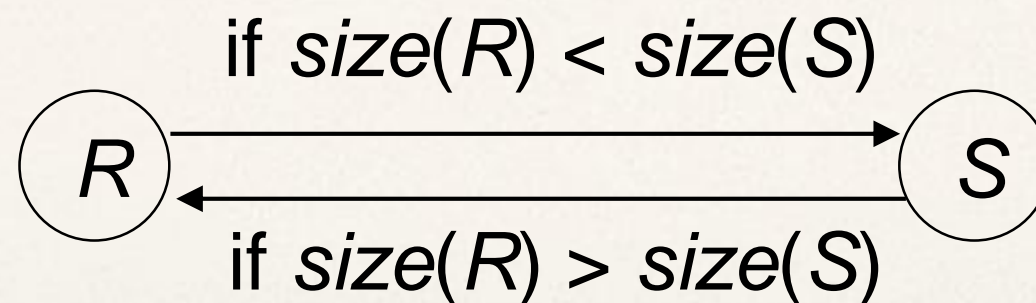
- Relation between centralized and distributed query optimization
 - Distributed query optimization (**DQO**) employs techniques and solutions from the centralized context
 - ♦ A distributed query is translated into local ones (localized queries): centralized query optimization (**CQO**) techniques
 - ♦ Distributed query optimization is a more general (and thus difficult) problem
 - ✓ Most solution to DQO extend solutions to CQO
 - We focus on communication costs (local CPU and I/O costs are ignored)
 - ♦ Clearly, cost of localized queries (handled with CQO techniques) is computed as in the centralized case (mainly I/O costs)

Join Ordering in Localized Queries

- Join ordering is important in centralized query optimization
- It is even more in distributed query optimization (reduce communication costs)
- Use of semijoins to reduce relation sizes (and thus communication costs) before performing join operations

Join Ordering – 2 relations

- We assume query to be already localized (i.e., on fragments)
 - Fragments are relations entirely stored at a single site
 - ♦ We often use “fragments” and “relations” indistinguishably (no technical reason to distinguish them)
- We first focus on ordering issues without using semijoins
 - Consider two relations only: $R \bowtie S$ (R and S are at different sites)
 - ♦ Move the smaller relation to the site of the larger one



Join Ordering – Multiple Relations

- Multiple relations case: more difficult because too many alternatives
- Goal is still transmit small operands (relations)
 - Compute the cost of all alternatives and select the best one
 - ♦ Necessary to compute the size of intermediate relations which is difficult
 - ✓ In distributed context it is even more because information may be not available on site

Join Ordering – Example

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

Execution alternatives:

1. $EMP \rightarrow \text{Site 2}$

Site 2 computes $EMP' = EMP \bowtie ASG$

$EMP' \rightarrow \text{Site 3}$

Site 3 computes $EMP' \bowtie PROJ$

2. $ASG \rightarrow \text{Site 1}$

Site 1 computes $EMP' = EMP \bowtie ASG$

$EMP' \rightarrow \text{Site 3}$

Site 3 computes $EMP' \bowtie PROJ$

3. $ASG \rightarrow \text{Site 3}$

Site 3 computes $ASG' = ASG \bowtie PROJ$

$ASG' \rightarrow \text{Site 1}$

Site 1 computes $ASG' \bowtie EMP$

4. $PROJ \rightarrow \text{Site 2}$

Site 2 computes $PROJ' = PROJ \bowtie ASG$

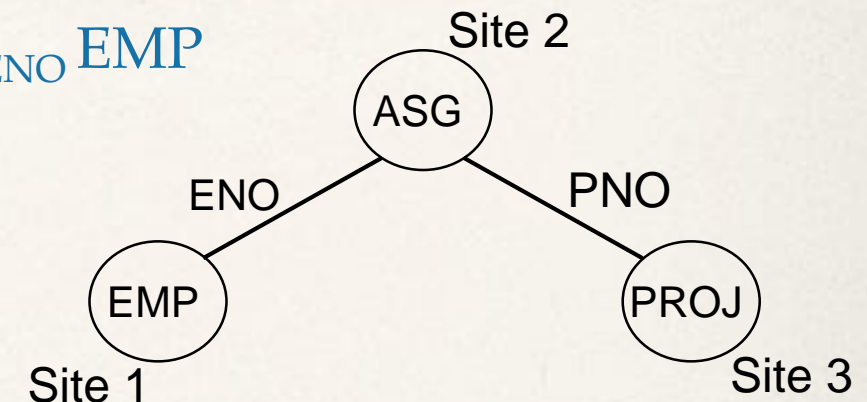
$PROJ' \rightarrow \text{Site 1}$

Site 1 computes $PROJ' \bowtie EMP$

5. $EMP \rightarrow \text{Site 2}$

$PROJ \rightarrow \text{Site 2}$

Site 2 computes $EMP \bowtie PROJ \bowtie ASG$



Join graph of distributed query

Semijoin Algorithms

- Semijoins can be used to reduce the sizes of operands to transfer (similar to what selections do)
 - Reduced communication costs
- Consider the join of two relations:
 - R (at site 1)
 - S (at site 2)
- Alternatives:
 1. Do the join $R \bowtie_A S$
 2. Perform one of the semijoin-based equivalent options

$$\begin{aligned} R \bowtie_A S &\Leftrightarrow (R \ltimes_A S) \bowtie_A S \\ &\Leftrightarrow R \bowtie_A (S \ltimes_A R) \\ &\Leftrightarrow (R \ltimes_A S) \bowtie_A (S \ltimes_A R) \end{aligned}$$

Tradeoff between

- a) cost to compute and send semijoin to other site (and then perform the join there)
- b) Cost to send the whole relation to other site (and then perform the join there)

Semijoin Algorithms – Example

- 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
$$size(\Pi_A(S)) + size(R \bowtie_A S) < size(R)$$
 - Only communication costs (time to transfer relations)

Semijoin Algorithms – Sum up

- Using semijoin is convenient if $R \bowtie_A S$ has high selectivity (select few tuples) and/or size of R is large
- It is bad otherwise, due to the additional transfer of $\Pi_A(S)$
- Cost of transferring $\Pi_A(S)$ can be reduced by using **bit arrays**
- **A disadvantage of using semijoin is the loss of indices**

Bit arrays

- Let h be a hash function that distributes possible values for A into n buckets:

$$h : \text{Dom}(A) \longrightarrow \{ 0, \dots, n-1 \}$$

- Bit array $BA[0 .. n-1]$ over relation S is defined as:

$$BA[i] = 1 \quad \text{iff} \quad \exists \text{ value } v \text{ for attribute } A \text{ in } S \text{ s.t. } h(v) = i$$

- Transfer BA (n bits) rather than $\Pi_A(S)$
- A tuple of R with value v for attribute A belongs to R' iff $BA[h(v)] = 1$
- R' is an (over-)approximation of $R \bowtie_A S$

Bit Arrays for Semijoins

R		S	
id_R	A	id_S	A
1	1	1	5
2	2	2	5
3	2	3	3
4	5	4	5
5	4	5	3
6	5		
7	4		
8	5		

R'		$\not\supseteq$	$R \bowtie_A S$	
id_R	A		id_S	A
1	1		4	5
4	5		6	5
6	5		8	5
8	5			

- Recall:
 - $BA[i] = 1$ iff \exists value v for attribute A in S s.t. $h(v) = i$
 - a tuple of R with value v for A belongs to R' iff $BA[h(v)] = 1$
- $h(x) = x \bmod 4$
- $n = 4$ (4 buckets)
- $h(1) = h(5) = 1$
- $BA[0] = 0$ (no value v occurs in $S.A$ s.t. $h(v) = 0$)
- $BA[1] = 1$ (due to occurrence of 5 for attribute A in S)
- $BA[2] = 0$ (no value v occurs in $S.A$ s.t. $h(v) = 2$)
- $BA[3] = 1$ (due to occurrence of 3 for attribute A in S)

R' contains tuple $\langle 1, 1 \rangle$ that does not belong to $R \bowtie_A S$

However, R' is a good approximation because h has only one conflict ($h(1) = h(5)$) among values for attribute A in R and S

$R' : R \bowtie_A S$ computed with bit array

Semijoins for Joins among Multiple Relations

- Semijoins to optimize joins among more than 2 operands

$$\text{EMP} \bowtie \text{ASG} \bowtie \text{PROJ} = \text{EMP}' \bowtie \text{ASG}' \bowtie \text{PROJ}$$

where $\text{EMP}' = \text{EMP} \bowtie \text{ASG}$

and $\text{ASG}' = \text{ASG} \bowtie \text{PROJ}$

- Each operand can be further reduced using more than one semijoin in cascade

$$\text{EMP}'' = \text{EMP} \bowtie (\text{ASG} \bowtie \text{PROJ})$$


We have

$$\text{size}(\text{ASG} \bowtie \text{PROJ}) \leq \text{size}(\text{ASG})$$

Therefore

$$\text{size}(\text{EMP}'') \leq \text{size}(\text{EMP}')$$

Semijoin
program



- Full reducer** for a relation is the semijoin program that reduces the relation **the most**
- Finding full reducer for a relation with exhaustive brute force approach
 - For cyclic queries full reducer cannot be found
 - ♦ Solution: break the cycle
 - With other queries: inefficient (NP-hard)
 - ♦ Solution: only use semijoin when problem is simple
 - ✓ e.g., for chained queries, where relations are in sequence and each one joins with the next one

Distributed Query Optimization

- We focus on optimization of joins
- The algorithm for optimizing a join is adapted from the one for the centralized case
- In distributed context
 - There is a coordinator (master site) where query is initiated
 - Coordinator chooses
 1. execution site and
 2. transfer method
 - Apprentice sites (where fragments are stored and queries are executed)
 - ♦ Apprentices behave as in the case of centralized query optimization in optimizing localized queries (over fragments) assigned to them
 - ✓ Choose best join ordering, join algorithm, and access method for relations

Choices of the Master Site

1. Choice of the execution sites

- E.g., $R \bowtie S$ can be executed:
 - ♦ at the site where R is stored
 - ♦ at the site where S is stored
 - ♦ at a third site (e.g., where a 3rd relation waits to be joined – allows for parallel transfer)

2. Transfer method

- *ship-whole*: relation is transferred to the join execution site entirely
 - ♦ In some cases (e.g., for outer relations or in case of merge join) there is no need to store the relation: join as it arrives, in pipelined mode
- *fetch-as-needed* (only needed tuples are transferred, i.e., tuples selected by the join):
 - ♦ equivalent to perform semijoin of one relation with tuple of the other one (to reduce size of the former) before executing the join
 - ♦ e.g., semi-join of inner relation wrt outer one (only needed tuples of inner relation are transferred)
 - ✓ tuples of the outer relation are sent (only the join attribute) to the site of the inner relation
 - ✓ matching tuples of the inner relation are sent to the site of the external relation to execute the join

Choices of the master produce 4 strategies (not all combinations are worth being considered)

Strategy 1 – *ship-whole/inner* site

1. *ship-whole/* site of *inner* relation: move outer relation (R) to the site of the inner relation (S)

- (a) Retrieve outer tuples
- (b) Send them to the inner relation site
- (c) Join them as they arrive

- $CT(x)$: communication time to transfer x bytes
- $LT(x)$: local processing time to perform op. x
- $s = card(S \bowtie_A R) / card(R)$: average number of tuples of S that match a tuple of R

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

Join is done as R comes because R is the outer relation

Strategy 2 – *ship-whole/outer* site

2. *ship-whole/* site of *outer* relation: move inner relation (S) to the site of outer relation (R)

Cannot join as S arrives; it needs to be stored

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

- $CT(x)$: communication time to transfer x bytes
- $LT(x)$: local processing time to perform op. x
- $s = card(S \bowtie_A R) / card(R)$: average number of tuples of S that match a tuple of R

Strategy 3 – *fetch-as-needed/outer* site

3. *fetch-as-needed/* site of *outer* relation

- (a) Retrieve tuples at outer relation (R) site
- (b) For each tuple of R , send join attribute values to inner relation (S) 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

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

- $CT(x)$: communication time to transfer x bytes
- $LT(x)$: local processing time to perform op. x
- $s = \text{card}(S \bowtie_A R) / \text{card}(R)$: average number of tuples of S that match a tuple of R

Strategy 4 – Move Both Relation at Third Site

4. move both inner (S) and outer (R) relations to another site

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

- $CT(x)$: communication time to transfer x bytes
- $LT(x)$: local processing time to perform op. x
- $s = card(S \bowtie_A R)/card(R)$: average number of tuples of S that match a tuple of R

Moving inner relation S first is better so we can then join as outer relation R arrives

Strategy comparison

$$PROJ \bowtie_{PNO} ASG$$

- PROJ (outer rel.) and ASG (inner rel.) are stored at different sites
- Index on PNO for relation ASG

- | | |
|--|--|
| 1. Ship whole PROJ at site of ASG | $CT (size(PROJ))$ |
| 2. Ship whole ASG at site of PROJ | $CT (size(ASG))$ |
| 3. Fetch tuples of ASG as needed at site of PROJ | $CT (length (A)) * card (PROJ)$
$+ CT (s * length (ASG)) * card (PROJ)$ |
| 4. Move both ASG and PROJ to a third site | $CT (size (ASG)) + CT (size (PROJ))$ |

- If there is no upper level operation then **4** is a bad choice
- If $size (PROJ) \gg size (ASG)$, then **2** is a good choice (if local processing time is not too bad compared with **1** and **3** (**1** and **3** can exploit index on ASG in their local processing))
- If PROJ is large/few tuples of ASG match, then **3** is better than **1**
- Otherwise, **1** is better than **3**

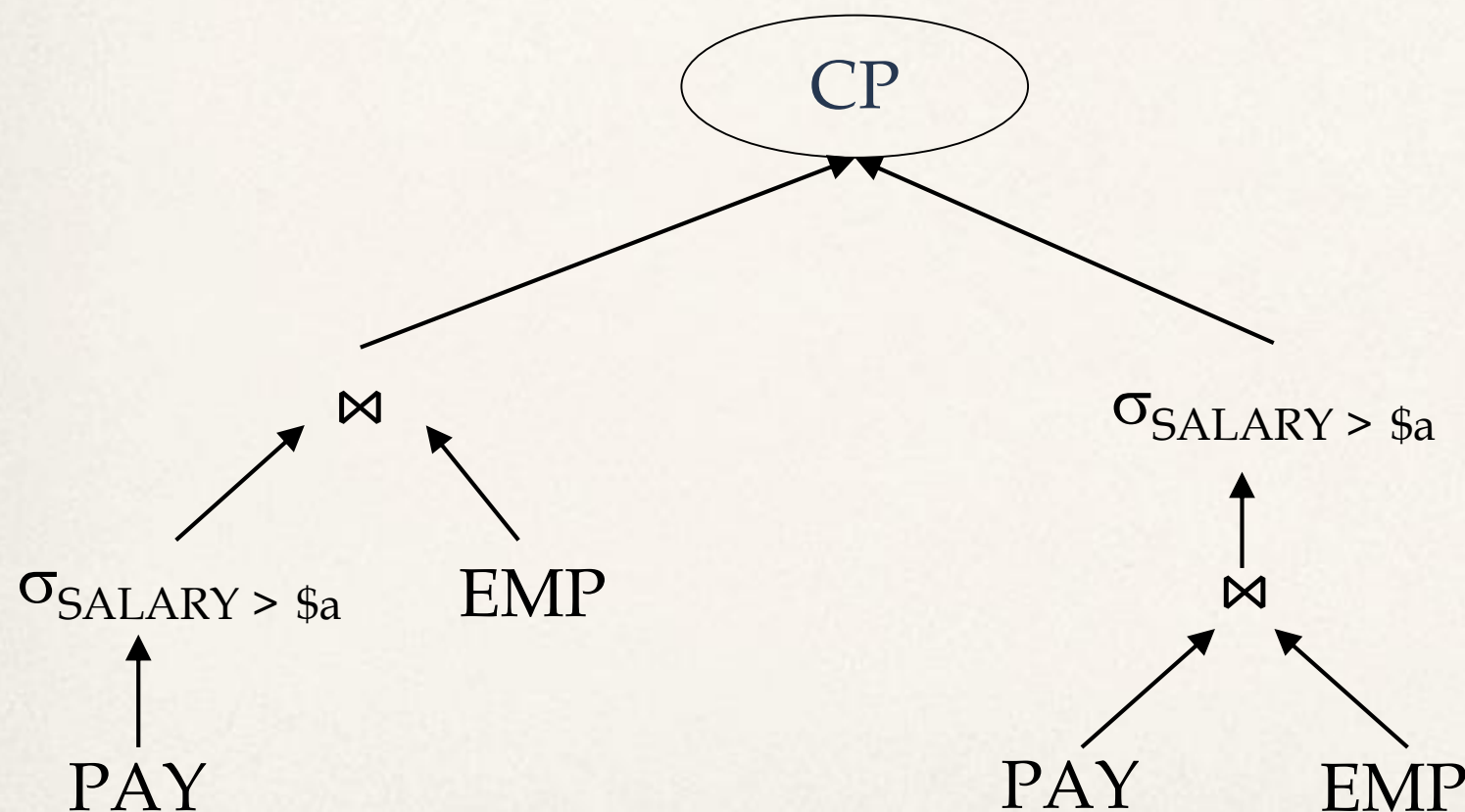
Hybrid approach

- So far, focus on *static* approaches, i.e., strategies (QEP, expressed as decorated trees) are evaluated and compared at compile time
- Advantages: query optimization is done once and used for several query executions
- Disadvantages: cost evaluation is not that accurate
 - it is not always done on exact values but on estimations based on statistics
 - ♦ e.g., size of intermediate results
 - some parameter of a query might be known only at runtime
- Problems of static query optimization are much more severe in the distributed context: more information variability at runtime
 - Sites may become unavailable or overloaded
 - Selection of site and fragment copy should be done at runtime to increase availability and load balancing
- An hybrid solution (some decisions are taken at runtime) is implemented by means of the CP (choose-plan) operator, which is resolved at runtime, when an exact plan comparison can be done

The CP (choose-plan) Operator

SELECT *
FROM EMP, PAY
WHERE SALARY > \$a

where \$a is a variable whose value is specified by the user at runtime

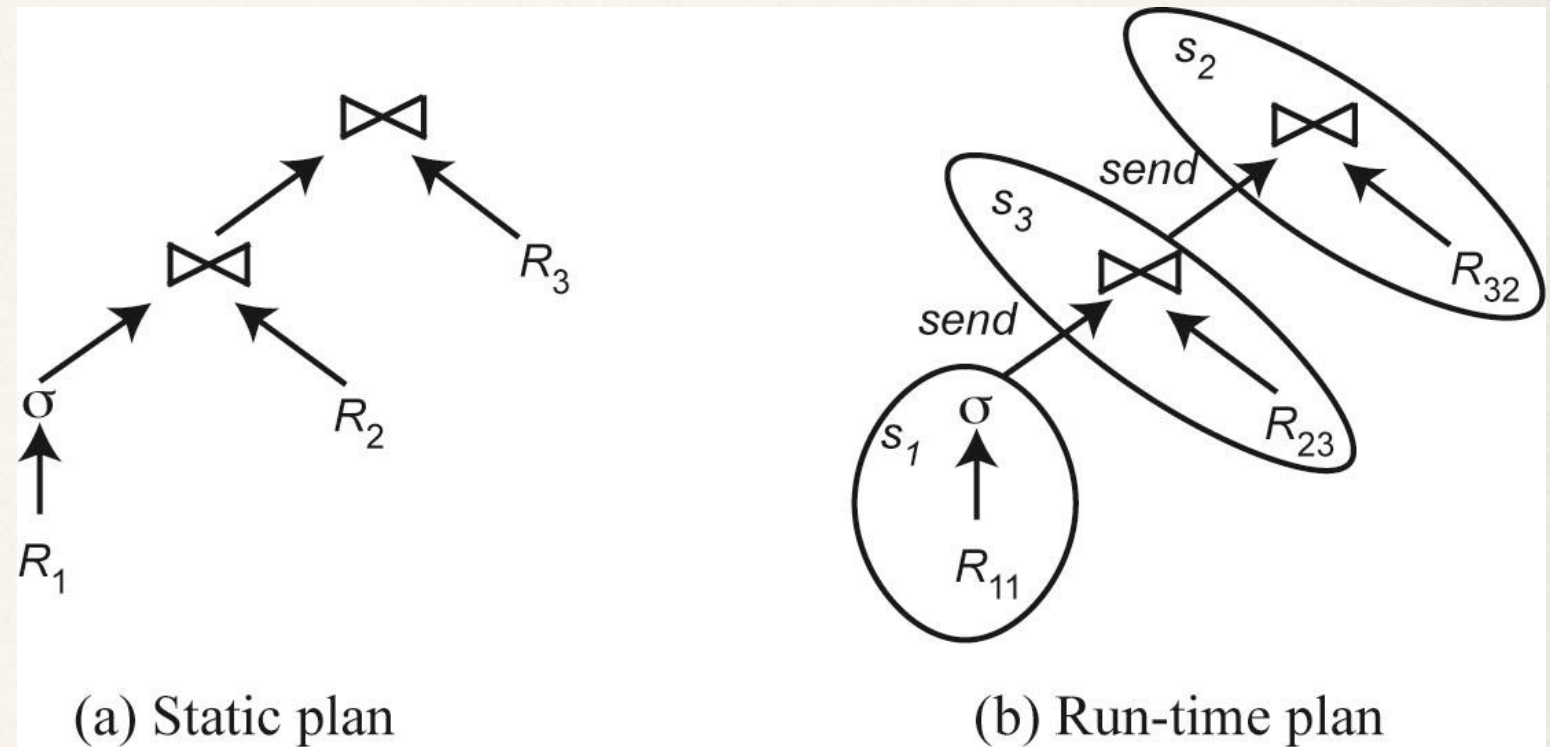


Normally, pushing σ inside \Join is a good heuristics, but it can be bad if selection rate of \Join is higher than the one of σ

2-Step Optimization

- **2-Step optimization**: a simpler approach (more efficient, less exhaustive) than the one based on CP operator; it reduces workload at runtime (no CP operator)
 - At runtime labels are added about site and fragment copy selection only

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



- Site (and copy) selection is done in a greedy fashion
 - best load balancing,
 - best benefit (# of queries already executed at the site, possible saving of communication costs as the site might have already data available)