Integration of Transactional Systems

Distributed Query Processing

Robert Wrembel
Poznań University of Technology
Institute of Computing Science
`Robert.Wrembel@cs.put.poznan.pl`
`www.cs.put.poznan.pl/rwrembel`

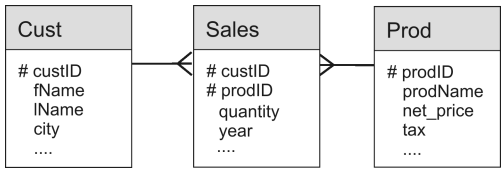
Relational calculus and algebra

- Relational calculus ⇨ declarative (SQL)
- Relational algebra ⇨ procedural
  - basic operators
    - selection
    - projection
    - cartesian product
    - union
    - set difference
  - derived operators
    - intersection
    - theta-join (natural, semi, equi, non-equi)

Query processor

- **Transforming a high-level query expressed in relational calculus into an equivalent lower-level query expressed in relational algebra ⇨ query execution plan**
  - **transformation correctness - the same query result**
  - **more efficient performance of a transformed query**

Example



select fName, lName from Cus c, Sales s where s.custID=c.custID and quantity=100

$$\Pi_{(fName,lName)}(\sigma_{(quantity=100 \wedge Sales.custID=Cust.custID)}(Cust \times Sales))$$

$$\Pi_{(fName,lName)}(Cust \bowtie_{Sales.custID=Cust.custID} (\sigma_{quantity=100}(Sales))$$

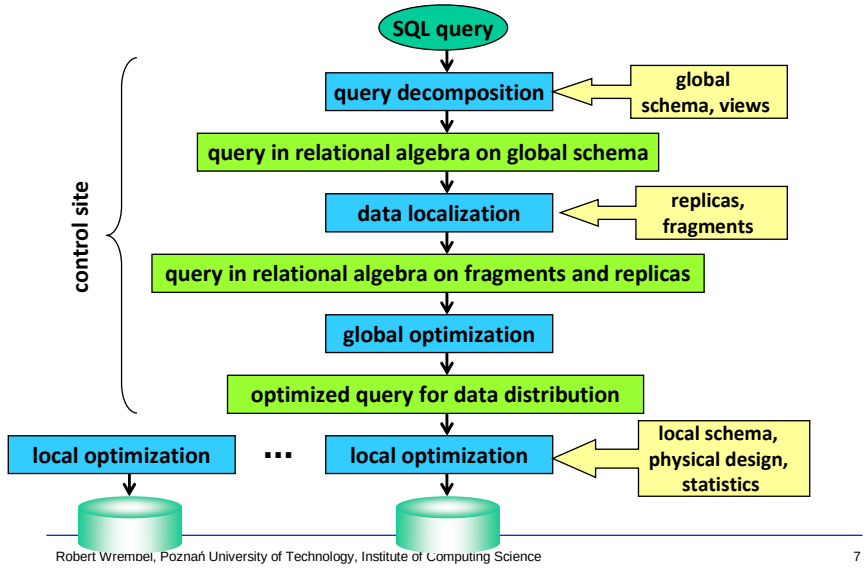
Query optimization "monent"

- Static
  - compilation time optimization
  - problem of estimating sizes of intermediate results ⇨ non-optimal execution
  - a query execution plan can be reused (cached)
  - R*
- Dynamic
  - run time optimization
  - exact sizes of intermediate results are known
  - for every query its execution plan has to be optimized ⇨ no plan sharing
  - Distributed INGRES
- Hybrid
  - compilation time optimization
  - if estimated sizes of intermediate results differ by a threshold from real ones ⇨ reoptimize at run time
  - MERMAID

Query optimization architectures

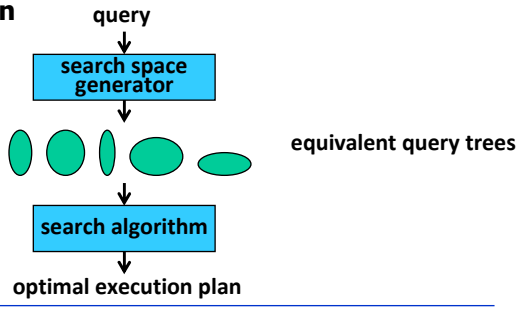
- Centralized architecture
  - access to statistics of all components of DDBS
  - central query optimizer ⇨ bottleneck
- Distributed architecture
  - sites cooperate in order to create optimal plan
  - higher network traffic (messages exchange)
- Hybrid architecture
  - one site determines global plan
  - local query is optimized locally on site

Layers of query processing



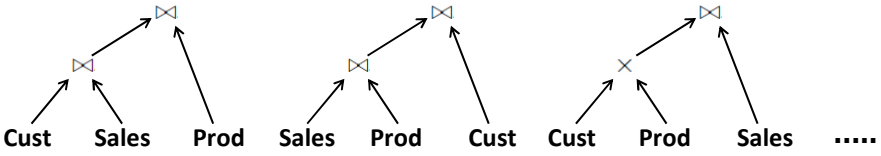
Optimization

- Global and local optimization
- Search space: the set of alternative query trees of an input query ⇨ obtained by applying transformation rules
  - the most costly are joins ⇨ different types of join trees
- Cost model: describes the cost of a query tree (execution plan)
- Search strategy: explores search space in order to find optimal execution plan



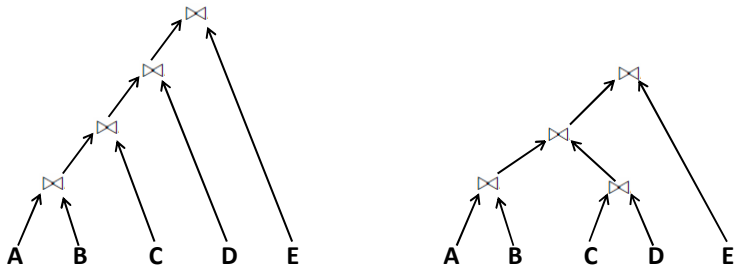
Join trees

```
select fName, lName, quantity, prodName
from Cust c, Sales s, Prod p
where s.custID=c.custID and s.prodID=p.prodID
```



Join trees

- Linear: at least one operand is a base relation
- Bushy: both operands may be intermediate relations
  - increased parallelism (distributed DBS)



Query optimization cost

- Optimizing
  - total execution time (cost)
    - reduces cost of every operation
  - response time (elapsed from query beginning to obtaining results)
    - total cost may be larger
    - operations may be performed in parallel
- Cost components
  - I/O
  - CPU
  - data transfer

```
Total_cost = CPU_cost + I/O_cost + communication_cost
CPU_cost = CPU_instr_cost * #instructions
I/O_cost = disk_I/O_cost * #accesses
communication_cost = #messages + data_transmission
```


Database statistics

- #rows in table (table cardinality, card(R))
- avg record length
- #distinct attribute values (attribute cardinality, card(Π_{Ai}(R))
- histograms (equi width, equi height)
- #db blocks
- ...

Estimating size of intermediate result

- **Size of a relation**

$$size(R) = card(R) * length(R)$$

- **Join selectivity factor**

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

- **Assumptions**
  - **attribute values are uniformly distributed**
  - **attributes are not correlated**

Estimating cardinalities

- **Selection** $card(\sigma_F(R)) = SF_\sigma(F) * card(R)$

$$SF_\sigma(A = value) = \frac{1}{card(\Pi_A(R))}$$

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

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

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

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

Estimating cardinalities

- **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: $card(R \cup S) = max\{card(R), card(S)\}$
- **Difference**
  upper bound: $card(R-S) = card(R)$
  lower bound: 0

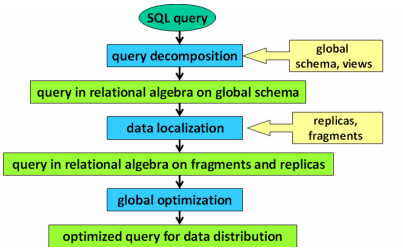
Estimating cardinalities

- **Join**
  - **upperbound: cardinality of cartesian product**
  - **simple case: A←PK of R, B←FK of S**
    - **upperbound when every tuple in R joins with tuples in S**

      $card(R \bowtie_{A=B} S) = card(S)$
  - **in general:** $card(R \bowtie S) = SF_{\bowtie} * card(R) * card(S)$
- **Maintain database statistics on cardinalities of relations and attributes**

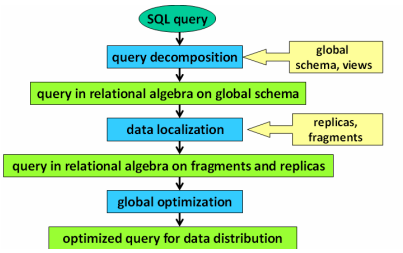
Query decomposition

- Query normalization (SQL)
- Syntactical and semantical analysis (SQL)
- Simplification (elimination of redundant predicates) (SQL)
- Transformation into algebraic representation
  - "optimization" of algebraic query



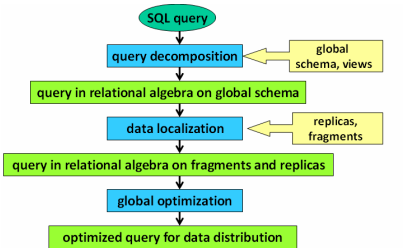
Data localization

- Finding the location of queried data based on data distribution statistics
  - eliminate useless fragments
  - use appropriate replicas
- Create "optimal" algebraic query



Global optimization

- Taking into consideration
  - cardinalities of fragments
  - communication, I/O, CPU costs
- Using
  - reordering operations (esp. joins)
  - semijoin reduction



Query decomposition
Normalization

- Lexical and syntactic correctness analysis
  - existence of attributes and relations, access rights
  - checking type compatibilities
- Transform to normalized (unified form)
  - conjunctive NF (more frequently used)

$$(p_{11} \vee p_{12} \vee \ldots \vee p_{1n}) \wedge \ldots \wedge (p_{m1} \vee p_{m2} \vee \ldots \vee p_{mn})$$

  - disjunctive NF

$$(p_{11} \wedge p_{12} \wedge \ldots \wedge p_{1n}) \vee \ldots \vee (p_{m1} \wedge p_{m2} \wedge \ldots \wedge p_{mn})$$

  - apply equvalence rules for logical operators

1. $p_1 \wedge p_2 \Leftrightarrow p_2 \wedge p_1$
2. $p_1 \vee p_2 \Leftrightarrow p_2 \vee p_1$
3. $p_1 \wedge (p_2 \wedge p_3) \Leftrightarrow (p_1 \wedge p_2) \wedge p_3$
4. $p_1 \vee (p_2 \vee p_3) \Leftrightarrow (p_1 \vee p_2) \vee p_3$
5. $p_1 \wedge (p_2 \vee p_3) \Leftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$
6. $p_1 \vee (p_2 \wedge p_3) \Leftrightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$
7. $\neg(p_1 \wedge p_2) \Leftrightarrow \neg p_1 \vee \neg p_2$
8. $\neg(p_1 \vee p_2) \Leftrightarrow \neg p_1 \wedge \neg p_2$
9. $\neg(\neg p_1) \Leftrightarrow p$

Query decomposition
Analysis

- Reject type incorrect normalized queries
- Reject semantically incorrect normalized queries
- Reject queries whose execution is not necessary

Query decomposition
Redundancy Removal

- Remove redundant predicates
- Apply rules
- Example

1. $p \wedge p \Leftrightarrow p$
2. $p \vee p \Leftrightarrow p$
3. $p \wedge true \Leftrightarrow p$
4. $p \vee true \Leftrightarrow true$
5. $p \wedge false \Leftrightarrow false$
6. $p \vee false \Leftrightarrow p$
7. $p \wedge \neg p \Leftrightarrow false$
8. $p \vee \neg p \Leftrightarrow true$
9. $p_1 \wedge (p_1 \vee p_2) \Leftrightarrow p_1$
10. $p_1 \vee (p_1 \wedge p_2) \Leftrightarrow p_1$

$$(\neg p_1 \wedge (p_1 \vee p_2) \wedge \neg p_2) \vee p_3$$

$p_1 \wedge (p_2 \vee p_3) \Leftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$

$$(\neg p_1 \wedge ((p_1 \wedge \neg p_2) \vee (p_2 \wedge \neg p_2))) \vee p_3$$

$p_1 \wedge (p_2 \vee p_3) \Leftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$

$$((\neg p_1 \wedge (p_1 \wedge \neg p_2)) \vee (\neg p_1 \wedge (p_2 \wedge \neg p_2))) \vee p_3$$

$p_1 \wedge (p_2 \wedge p_3) \Leftrightarrow (p_1 \wedge p_2) \wedge p_3$

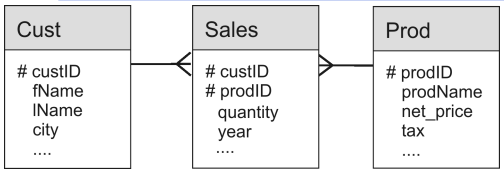
$$((\neg p_1 \wedge p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2 \wedge \neg p_2)) \vee p_3$$

$p \wedge \neg p \Leftrightarrow false$

$$((false \wedge \neg p_2) \vee (\neg p_1 \wedge false)) \vee p_3$$

$$false \vee false \vee p_3 = p_3$$

Query decomposition
Rewriting




```
select fName, lName
from Cust c, Sales s, Prod p
where s.custID=c.custID
and s.prodID=p.prodID
and c.city != 'Poznań'
and p.tax = 23
and (s.year=2008 or s.year=2009 )
```

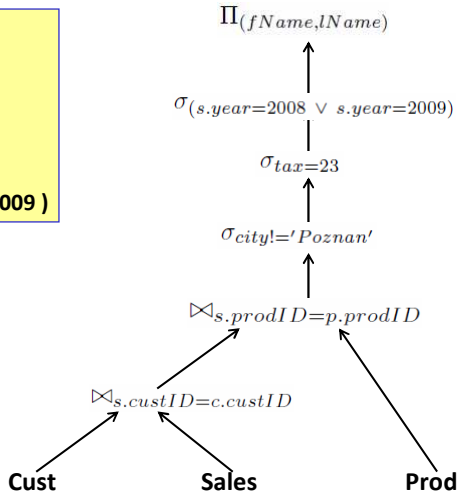
- Expressing query in relational algebra ⇨ operator tree
  - leaves ⇨ relations (FROM clause)
  - root ⇨ result with projected attributes (SELECT clause)
  - intermediate nodes ⇨ relational algebra operators
- Straighforward transformation

$$\Pi_{fName,lName}(\sigma_{s.custID=c.custID \wedge s.prodID=p.prodID \wedge c.city!='Poznan' \wedge p.tax=23 \wedge (s.year=2008 \vee s.year=2009)}(Cust \times Sales \times Prod))$$

- Optimized tree by applying transformation rules
  - multiple equivalent trees may be created

Query decomposition
Rewriting

```
select fName, lName
from Cust c, Sales s, Prod p
where s.custID=c.custID
and s.prodID=p.prodID
and c.city != 'Poznań'
and p.tax = 23
and (s.year=2008 or s.year=2009 )
```



Transformation rules

- Relations R, S, T
- R is composed of attributes A={A_1, A_2, ..., A_n}
- S is composed of attributes B={B_1, B_2, ..., B_n}

1. **Comutativity of binary operators**

$$R \times S \Leftrightarrow S \times R$$
$$R \bowtie S \Leftrightarrow S \bowtie R$$
$$R \cup S \Leftrightarrow S \cup R$$

2. **Associativity of binary operators**

$$(R \times S) \times T \Leftrightarrow R \times (S \times T)$$
$$(R \bowtie S) \bowtie T \Leftrightarrow R \bowtie (S \bowtie T)$$
$$(R \cup S) \cup T \Leftrightarrow R \cup (S \cup T)$$

3. **Grouping unary operators**
   - grouping subsequent projections
   - grouping subsequent selections

$$\Pi_{A_1,A_2,\ldots,A_m}(\Pi_{A_1,A_2,\ldots,A_m,A_{m+1},\ldots,A_{m+k}}(R)) \Leftrightarrow \Pi_{A_1,A_2,\ldots,A_m}(R)$$
$$\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) \Leftrightarrow \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$$

Transformation rules

4. **Commuting selection with projection**

$$\Pi_{A_1,\ldots,A_n}(\sigma_{p(A_p)}(\Pi_{A_1,\ldots,A_n,A_p}(R))) \Leftrightarrow \Pi_{A_1,\ldots,A_n}(\sigma_{p(A_p)}(R))$$

5. **Commuting selection with binary operators**

$$\sigma_{p(A_i)}(R \times S) \Leftrightarrow (\sigma_{p(A_i)}(R)) \times S$$
$$\sigma_{p(A_i)}(R \bowtie_{p(A_j,B_k)} S) \Leftrightarrow \sigma_{p(A_i)}(R) \bowtie_{p(A_j,B_k)} S$$
$$\sigma_{p(A_i)}(R \cup T) \Leftrightarrow \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$$

6. **Commuting projection with binary operators**

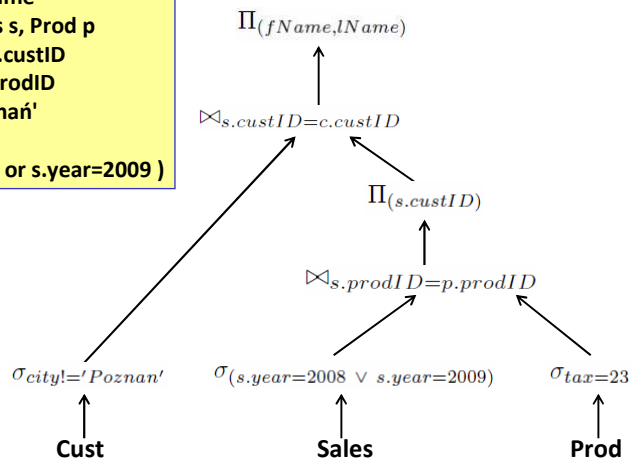
$$\Pi_C(R \times S) \Leftrightarrow \Pi_{A_1}(R) \times \Pi_{B_1}(S), where\ C = A_1 \cup B_1, A_1 \subseteq A, B_1 \subseteq B$$
$$\Pi_C(R \bowtie_{p(A_i,B_j)} S) \Leftrightarrow \Pi_{A_1}(R) \bowtie_{p(A_i,B_j)} \Pi_{B_1}(S)$$
$$\Pi_C(R \cup S) \Leftrightarrow \Pi_C(R) \cup \Pi_C(S)$$

7. **Commuting join with set operator**

$$(R \cup S) \bowtie T = (R \bowtie T) \cup (S \bowtie T)$$

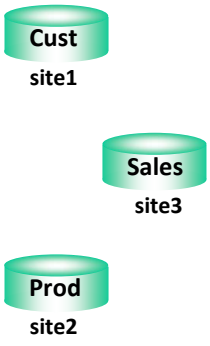
Equivalent query tree

```
select fName, lName
from Cust c, Sales s, Prod p
where s.custID=c.custID
and s.prodID=p.prodID
and c.city != 'Poznań'
and p.tax = 23
and (s.year=2008 or s.year=2009 )
```



Distributed query optimization
Problem

```
select lName, prodName, quantity
from Cust c, Sales s, Prod p
where s.custID=c.custID and s.prodID=p.prodID
and c.city != 'Poznań' and p.tax = 23 and (s.year=2008 or s.year=2009 )
```

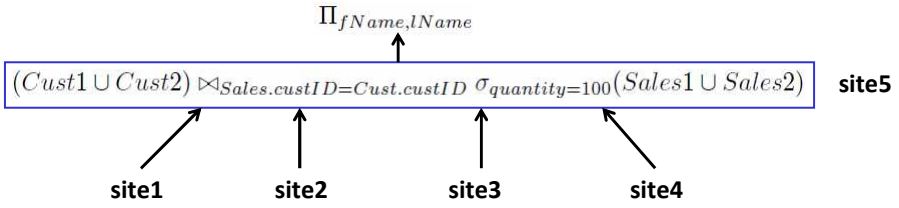


- Query optimizer needs to consider (global view on the system)
  - sizes of relations
  - sizes of intermediate results
  - communication costs (network throughput)
  - computation power of sites (CPU, I/O, memory)
  - data structures at sites (indexes, partitions, clusters, ...)
  - power of query optimizer (join algorithms, cost based, rule based, search space generation and searching)
  - availability and location of fragments
  - availability and location of replicas

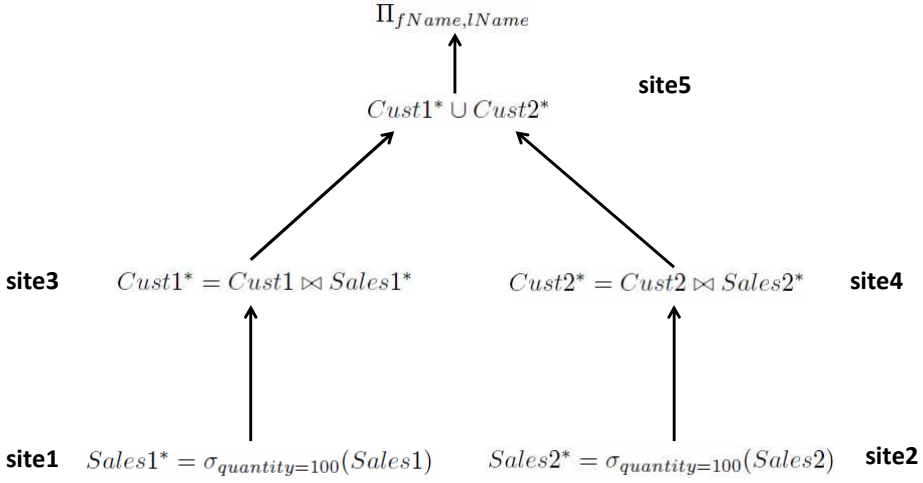
Distributed DB example

- Horizontal fragmentation of Cust and Sales
  - site1 ⇨ Cust1: custID<=1000
  - site2 ⇨ Cust2: custID>1000
  - site3 ⇨ Sales1: custID<=1000
  - site4 ⇨ Sales2: sustID>1000
  - site5 ⇨ executes query

$$\Pi_{(fName,lName)}(Cust \bowtie_{Sales.custID=Cust.custID} (\sigma_{quantity=100}(Sales))$$



Distributed DB example



Data localization

- DB is fragmented (partitioned)
- Global relation is represented by a reconstructng mechanism ⇨ reconstruction program that reconstructs the relation from its fragments
- Naive approach: construct generic query tree where each relation is represented by its reconstruction program ⇨ not optimal tree ⇨ apply reduction techniques
  - reduction for primary horizontal fragmentation
  - reduction for vertical fragmentation
  - reduction for derived fragmentation
  - reduction for hybrid fragmentation

Data localization: RPHF

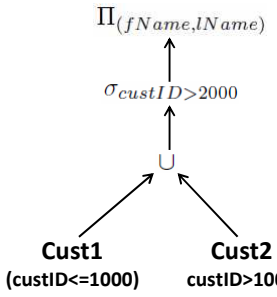
- Reduction for Primary Horizontal Fragmentation
- Cust fragmented into
  - Cust1: custID<=1000
  - Cust2: custID>1000
  - reconstruction program $Cust = Cust1 \cup Cust2$
- In generic query tree Cust is replaced by its reconstruction program
- After building a query tree, find out subtrees that produce emtpy relations (no results)
  - reduction with selection
  - reduction with join

Data localization: RPHF

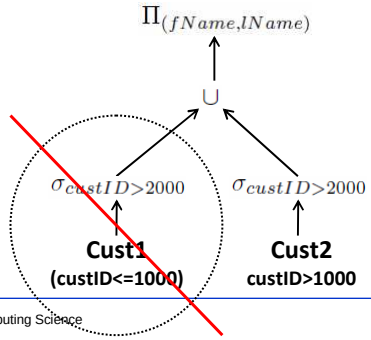
Reduction with selection

```
select fName, lName
from Cust c
where custID>2000
```

➲ Generic query tree



➲ Reduced query tree



Data localization: RPHF

Reduction with join

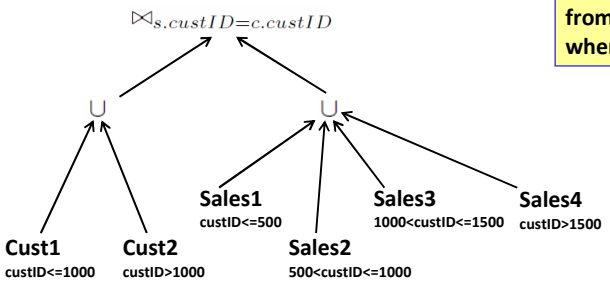
- ➲ Assumption: joined relations are fragmented according to a join attribute
- ➲ Distributing joins over unions + eliminating empty joins ⇨ applicable when the number of joins of fragments is small
- ➲ Possible parallel computation of joins of fragments
- ➲ Cust fragmented into
  - Cust1: custID<=1000
  - Cust2: custID>1000
- ➲ Sales fragmented into
  - Sales1: custID<=500
  - Sales2: 500<custID<=1000
  - Sales3: 1000<custID<=1500
  - Sales4: custID>1500

Data localization: RPHF

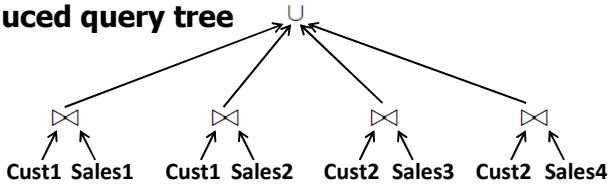
Reduction with join

➲ Generic query tree

```
select *
from Cust c, Sales s
where s.custID=c.custID
```



➲ Reduced query tree



Data localization: RVF

➲ Reduction for Vertical Fragmentation

➲ Reconstruction program: join

➲ Vertical fragments that have no attribtes in common (except PK) with the list of projected attributes in a query are useless

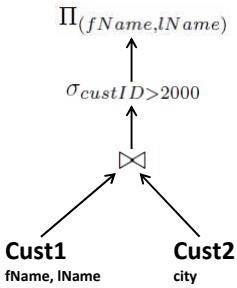
➲ Cust fragmented as follows

- Cust1 = $\Pi_{custID,fName,lName}$(Cust)
- Cust2 = $\Pi_{custID,city}$(Cust)

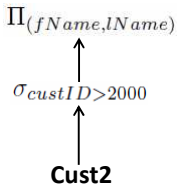
Data localization: RVF

```
select fName, lName
from Cust
where custID>2000
```

➲ Generic query tree



➲ Reduced query tree

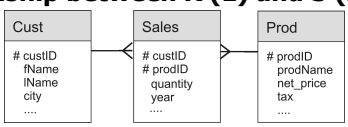


Data localization: RDF

➲ Reduction for Derived Fragmentation

➲ Derived fragmentation: fragments of R and S that have the same join attribute values are located at the same site

- tuples of S are placed based on tuples of R
- 1:m relationship between R (1) and S (m)



➲ Prod fragmented as follows

- Prod1 = (tax<=7%)
- Prod2 = (tax>7%)

➲ Derived fragmentation of Sales based on Prod

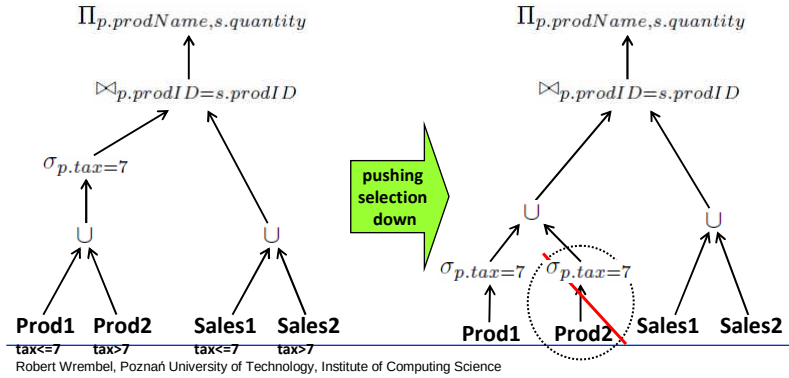
$$Sales1 = Sales \triangleright\!\triangleleft_{prodID} Prod1$$
$$Sales2 = Sales \triangleright\!\triangleleft_{prodID} Prod2$$

semijoin PK-FK

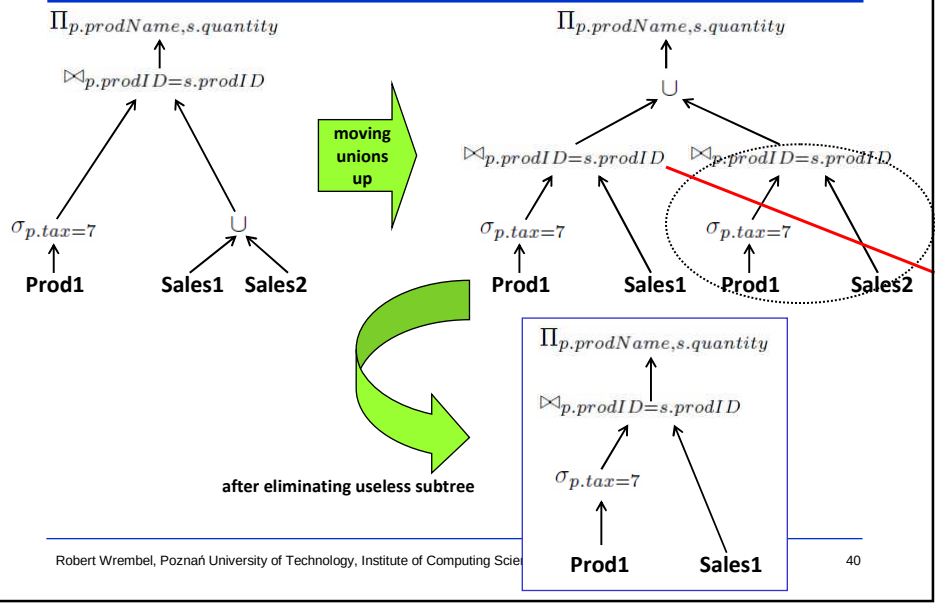
Data localization: RDF

```
select  p.prodName, s.quantity
from Prod p, Sales s
where p.prodID=s.prodID
and p.tax=7
```

➲ **Generic query tree**



Data localization: RDF



Data localization: RHF

- **Reduction for Hybrid Fragmentation**
- **Hybrid fragmentation: horizontal + vertical + derived ⇨ support SPJ queries**
- **Query graph optimization rules**
  - **remove horizontal fragments fragmentation whose predicates contradict with query predicates**
  - **remove vertical fragments that have no attribtes in common with projected attributes in a query**
  - **distribute joins over unions of fragments and remove useless joins**

Data localization: RHF

- **Prod fragmented (horizontally and vertically) as follows**

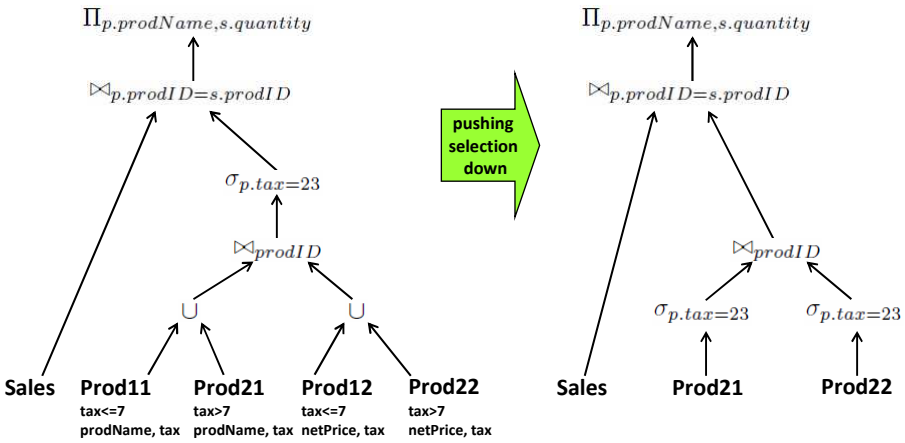
$$Prod11 = \Pi_{(prodID,prodName,tax)}(\sigma_{tax\leq 7}(Prod))$$
$$Prod12 = \Pi_{(prodID,netPrice,tax)}(\sigma_{tax\leq 7}(Prod))$$
$$Prod21 = \Pi_{(prodID,prodName,tax)}(\sigma_{tax>7}(Prod))$$
$$Prod21 = \Pi_{(prodID,netPrice,tax)}(\sigma_{tax>7}(Prod))$$

- **Sales not fragmented**

```
select  p.prodName, s.quantity
from Prod p, Sales s
where p.prodID=s.prodID
and p.tax=23
```

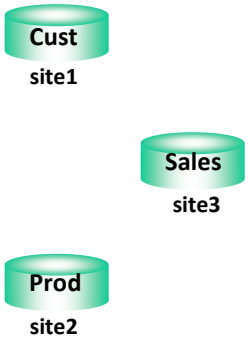
Data localization: RHF

- Generic query tree



Distributed joins

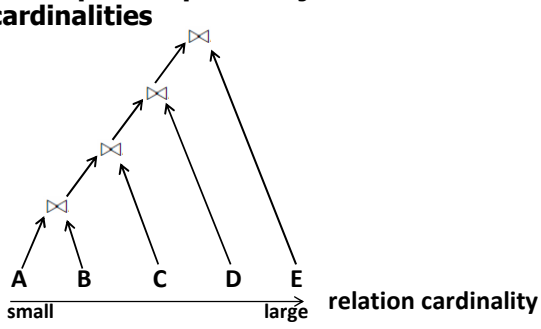
- Ordering joins



- Possible join strategies
  - send Cust to Sales, join, send result to Prod
  - send Sales to Cust, join, send result to Prod
  - send Sales to Prod, join, send result to Cust
  - send Prod to Sales, join, send result to Cust
  - send Prod and Cust to Sales, join
  - send Prod and Sales to Cust, join
  - send Cust and Sales to Prod, join
- Rule: send a smaller relation to a bigger one

Join order

- **Join order is determined by:**
  - **either the cardinality of relations ⇨ sort ascendingly relations by their cardinality and join them "from the smallest to the largest"**
  - **or the cardinalities of all the possible join sequences ⇨ create the search space of possible joins and estimate their cardinalities**



Using semijoins

- **Use semijoin to decrease the size (cost) of intermediate relation**

Strategy 1: $R \bowtie_{A_m} S \Leftrightarrow (R \triangleright \triangleleft_{A_m} \Pi_{A_m}(S)) \bowtie_{A_m} S$

Strategy 2: $R \bowtie_{A_m} S \Leftrightarrow R \bowtie_{A_m} (S \triangleright \triangleleft_{A_m} \Pi_{A_m}(R))$

Strategy 3: $R \bowtie_{A_m} S \Leftrightarrow (R \triangleright \triangleleft_{A_m} \Pi_{A_m}(S)) \bowtie_{A_m} (S \triangleright \triangleleft_{A_m} \Pi_{A_m}(R))$

- **Example: strategy**

(1) send $Cust_1 = \Pi_{custID}(Cust)$ to site3
(2) at site3 compute $Sales_1 = Sales \triangleright \triangleleft_{custID} Cust_1$
(3) send $Sales_1$ to site1
(4) at site1 compute $Cust \triangleright \triangleleft_{custID} Sales_1$

Using semijoins

- Example

$$(Cust \triangleright \triangleleft_{custID} Sales) \bowtie_{custID} (Sales \triangleright \triangleleft_{prodID} Prod) \bowtie_{prodID} Prod$$

...

$$(Cust \triangleright \triangleleft_{custID} (Sales \triangleright \triangleleft_{prodID} Prod)) \bowtie_{custID} (Sales \triangleright \triangleleft_{prodID} Prod) \bowtie_{prodID} Prod$$

- Multiple sequences of semijoins
  - the number of sequences grows expotentially with the number of relations
  - finding one optimal sequence is an NP-hard problem

Using semijoins

- Beneficial when the total amount of transmitted data is smaller than with join ⇨ reducing the cardinality of an intermediate result
- Intermediate results cannot profit from additional data structures as base relations do
- Optimalization of Π_{Ai} transmission ⇨ encode in a bit array (bitmap) ← data size reduction

R* Algorithm

- **Master site (where a query is initiated)**
  - **global optimization of a query**

```
input: query tree QT
output: minimum cost strategy strat
begin
   {for each relation Ri ∈QT
       for each access path APij to Ri
          {compute cost(APij)}
       best_APi:=APij with minimum cost
   }
   {for each order (Ri1, Ri2, ..., Rin): i=1, ..., n!
       build strategy ((...((bestAPi1 ⋈ Ri2) ⋈ Ri3) ⋈ Ri4)... ⋈ Rin)
       compute cost of the strategy
   }
   strat:= strategy with minimum cost
   {for each site k storing relation in QT
      send local strategy to k
   }
end
```

access method (index, full scan, ...) ⇨ use statistics and cost formulas

select join sequence, join algorithm, relation transfers - join site ⇨ estimate cardinalities, complexity of join algorithms

R* Algorithm

- **Data transfer methods**
- **Ship-whole**
  - **the whole relation R is transfered to the join site**
  - **better when most of rows in R join**
  - **better for smal R**
- **Fetch-as-needed**
  - **outer relation is sequentially read**
  - **join value v is sent to the site of inner relation S**
  - **inner tuples joining with v are sent back to the outer relation**
  - **better than few rows of S join**

R* Algorithm

- Build strategy ⇨ all possible scenarios for transferring relations/tuples between sites
- Cost model includes
  - local processing cost (I/O for retrieving relations/tuples)
  - communication cost (amount of data transferred between sites)
- Strategies that can be applied by the algorithm
  1. transfer entire external relation to the site of an internal relation
  2. transfer entire internal relation to the site of an external relation
  3. fetch-as-needed tuples from an internal relation
  4. move an internal and external relation to a third site

Hill climbing algorithm

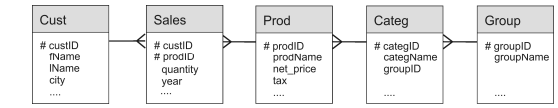
- No semijoins, no replication, no fragmentation
- Heuristic for searching a solution space
  - local minimum can be obtained
  - global minimum may not be obtained ⇨ the first step eliminates more costly query trees that might lead to a final query tree with a global minimum cost
- Uses: query graph, location of relations, statistics

HC algorithm

1. **Compute initial query plan (IQP^0)**
   - **select the site where the final result will be computed ⇨ site with relation of the greatest cardinality involved in the query**
   - **compute data transfer cost of relations from all other nodes independently**

```
select lName, prodName, quantity
from Cust c, Sales s, Prod p, Categ c, Group g
where s.custID=c.custID and s.prodID=p.prodID and ...
and c.categID=1
```

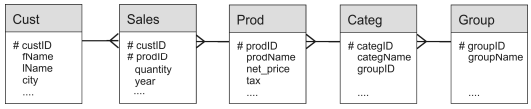
assumption: uniformly distributed data



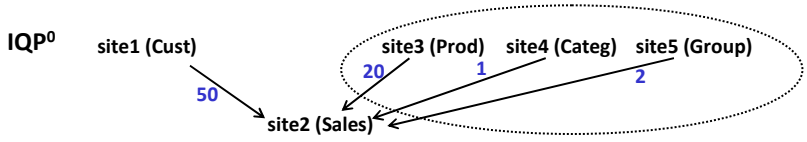
card(Cust)=50 site1
card(Sales)=2000 site2
card(Prod)=20 site3
card(Categ)=5 site4
card(Group)=2 site5

HC Algorithm

1. **Compute initial query plan (IQP^0)**



card(Cust)=50 site1
card(Sales)=2000 site2
card(Prod)=20 site3
card(Categ)=5 site4
card(Group)=2 site5

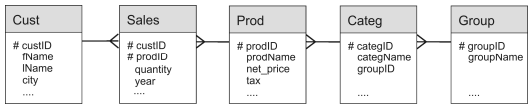


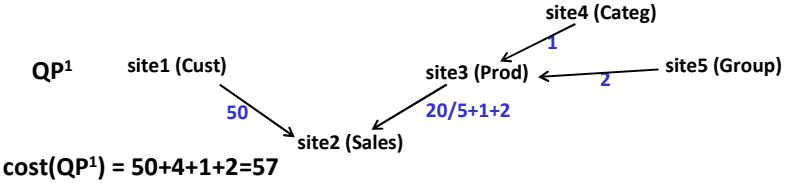
initial cost(IQP^0) = 50+20+1+2=73

HC algorithm

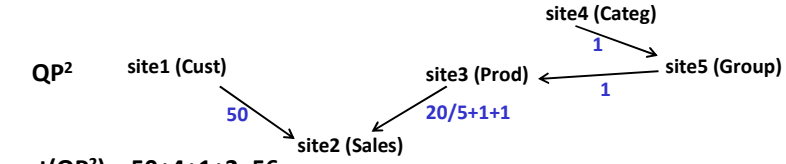
2. Alter initial query plan IQP^0 into QP^i (i=1,2,...n) ⇨ QP^i exploits transfer of one relation to the other site for the purpose of joining it with the remote relation
   - compute cost(QP^i) (include data transfer time and local processing time)
   - find min{cost(QP^i)}
   - if cost(IQP^0)> min{cost(QP^i)} replace IQP^0 with QP^i
   - recursively apply step 2 on QP^i until all joins are resolved

HC algorithm





cost(QP^1) = 50+4+1+2=57



cost(QP^2) = 50+4+1+2=56

Bibliography

- M. T. Özsu, P. Valduriez: Principles of Distributed Database Systems. Prentice Hall,1991
- T. Conolly, C. Begg: Database Systems - a Practical Approach to Design, Implementation, and Management. Adison-Wesley, 2002
- K. Stocker, D. Kossmann, R. Braumandi, A. Kemper: Integrating semi-join-reducers into state-of-the-art query processors. Proc. of ICDE, 2001