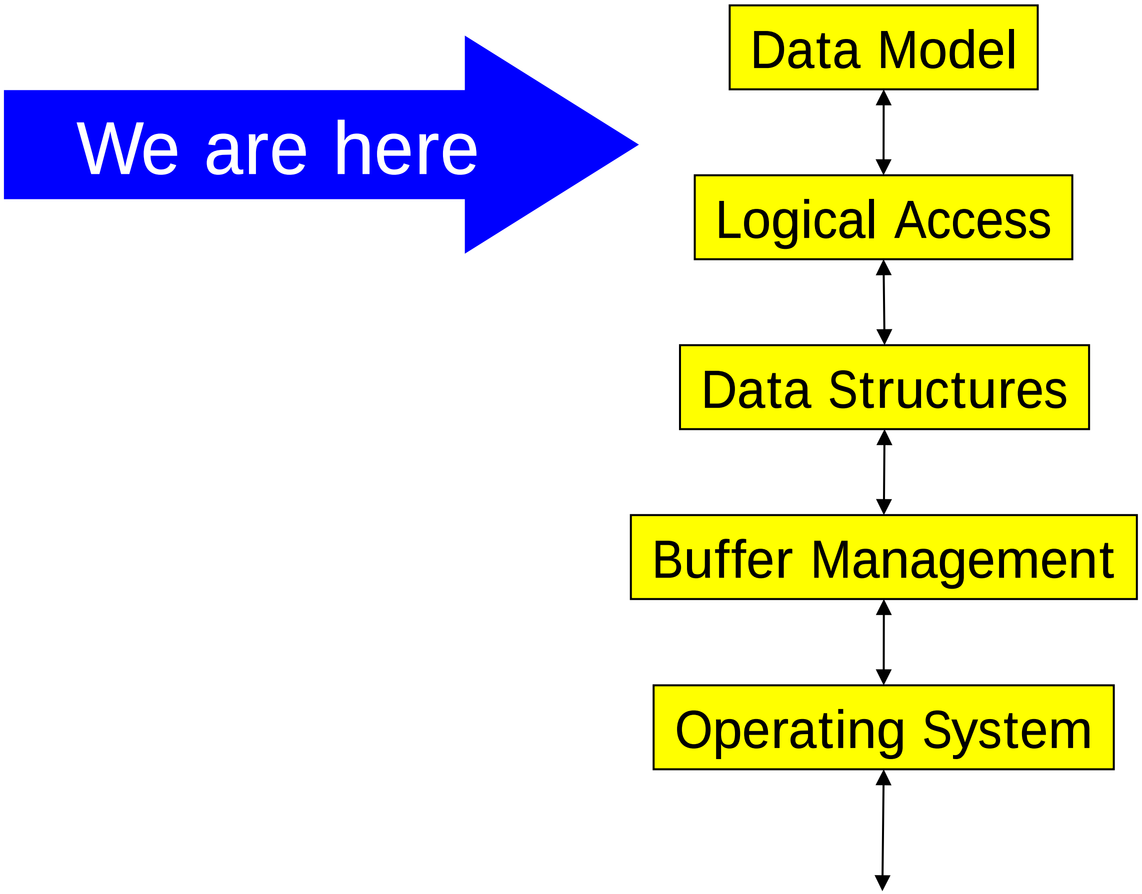
Datenbanksysteme II: Query Optimization

Ulf Leser

5 Layer Architecture



Content of this Lecture

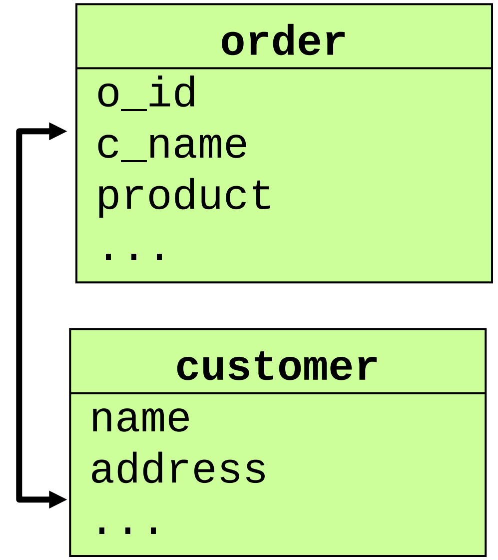
- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Is Optimization Worth It?

- Goal: Find cheapest way to compute a query result
  - Generate and judge different physical plans to answer the query
  - All QEPs must be semantically equal
- Optimization costs time
  - Some steps are exponential
    - E.g. join order: 10 joins – potentially 3^{10} steps
  - Finding the best plan might take more time than executing an arbitrary plan
    - And usually we don't even find the best plan
- Why bother?

Example

```
SELECT C.name, C.address
FROM   customer C, order O
WHERE  C.name = O.c_name AND
        O.product = „coffee“
```

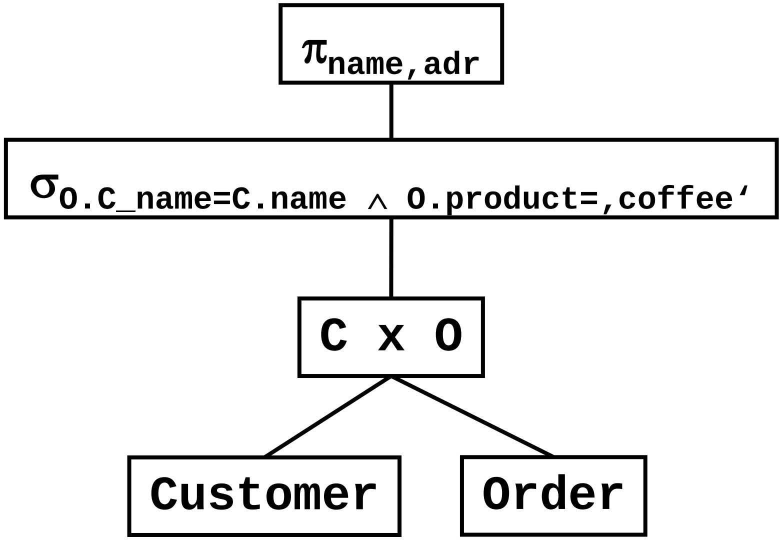


- Assumptions
  - 1:n relationship between C and O
  - |C|=100, 5 tuples per block, b(C)=20
  - |O|=10.000, 10 tuples per block, b(O) = 1.000
  - Result size: 50 tuples
  - Intermediate results
    - (C.name, C.address): 50 per block
    - Join result (C,O) with full tuples: 3 per block
  - Small main memory

First Attempt

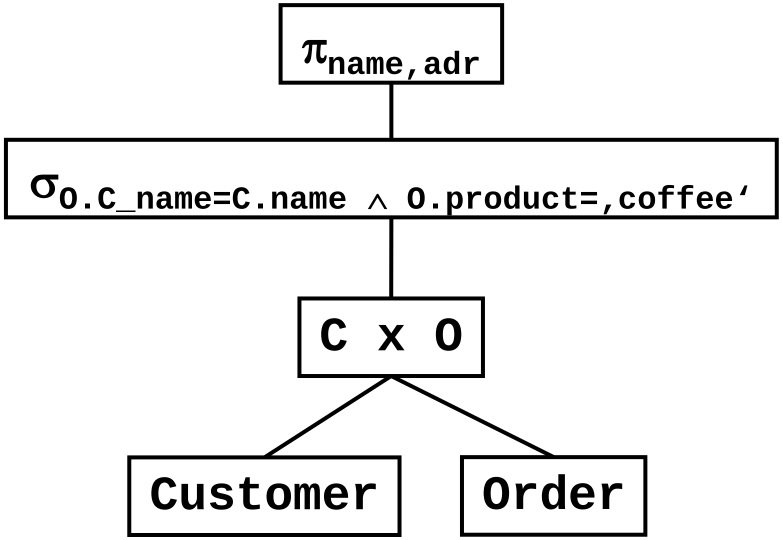
- Translate in relational algebra expression
  - $\pi_{\texttt{name,adr}}(\sigma_{\texttt{O.C\_name=C.name} \wedge \texttt{O.product='coffee'}}(\texttt{C x O}))$

- Interpret query „from inner to outer“
  - No optimization at all
  - Full materialization of intermediate results (no buffering, no pipelining)



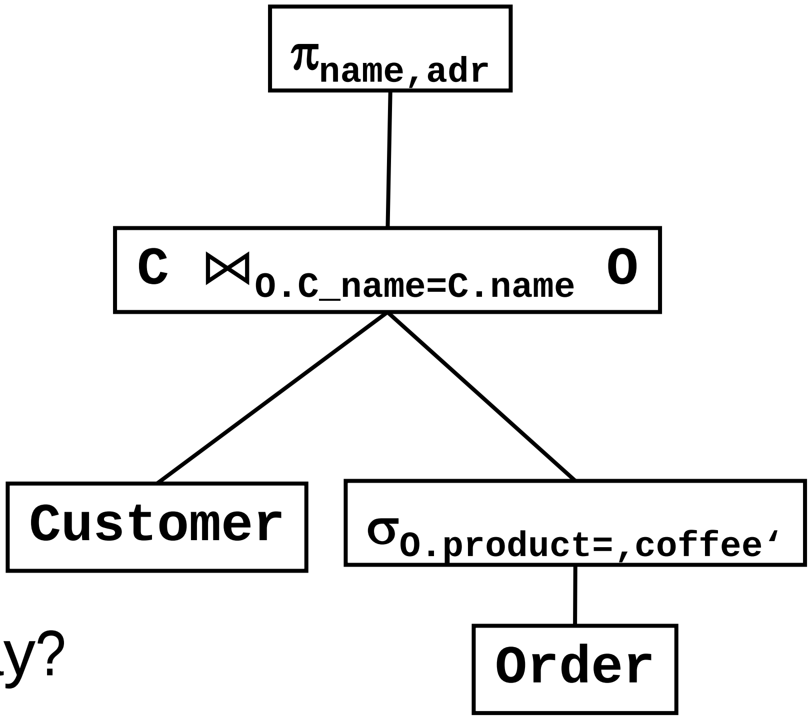
Cost

- Compute cross-product
  - Reads: b(C)*b(O)=20.000
  - Writes: 100*10.000/3 ~ 333.000
- Compute selections
  - Reads: 333.000
  - Writes: 50/3 ~ 17
- Compute projection
  - Reads: 17
  - Writes: 50/50 ~ 1
- Altogether: ~ 686.000 IO
  (and 333.000 blocks required on disk)



Use Term Rewriting

- Rewrite into: $\pi_{\texttt{name,adr}}(\texttt{C} \bowtie_{\texttt{O.C\_name=C.name}}(\sigma_{\texttt{O.product=,coffee'}}(\texttt{O})))$
- Compute selection on O
  - Reads: 1.000, writes: 50/10 = 5
- Compute join using BNL
  - Reads: 5 + b(C)*5 = 105
  - Writes: 50/3 ~ 17
- Compute projection
  - Reads: 17, writes: 50/50 ~ 1
- Altogether: 1.145
  (requiring 17 blocks on disk)
- Maybe there is an ever better way?



Better Plan

- Push projection
  - $\pi_{\texttt{name,adr}}(\pi_{\texttt{name,adr}}(\texttt{C}) \bowtie_{\texttt{O.C\_name=C.name}}(\sigma_{\texttt{O.product='coffee'}}(\texttt{O})))$
- Compute selection on O
  - Reads: 1.000, writes: 50/10 = 5
- Compute projection on C
  - Reads b(C)=20, writes 100 / 50 = 2
- Compute join using nested loop
  - Reads: 2 + 2*5 = 12, writes: 50/3 ~ 17
- Compute projection
  - Reads: 17, writes: 50/50 ~ 1
- Altogether: 1.080 (requiring 17 blocks on disk)

Even Better – Use Indexes

- Indexes on (O.product, O.C_name) and (C.name, C_address)
- Compute selection on O using index
  - Reads: Roughly between 5 and 10
    - Height of index plus consecutive blocks for 50 TIDs with product='coffee'
    - Number of blocks depends on fill degree of B-tree
    - Assume 10 pointer in an index node: height = 4
  - Writes: 50/10 = 5
- Sort intermediate result
  - Read and writes: ~ 5*log(5) ~ 15
    - Very conservative estimation
  - Result has 5 blocks

Even Better – Use Indexes

- ...
- Compute join
  - Reads: 20 + 5 = 25
    - Using sort-merge – read C.name in sorted order using index
  - Writes: 50/3 ~ 17
- Compute projection
  - Reads: 17, writes: 50/50 ~ 1
- Altogether: between 85 and 90
  (requiring 17 blocks on disk)
- Even better?

Comparison

	Read/Write	Temp space
Naive	687.000	333.000
Optimized, no index	1.080	17
With index	85-90	17

- Reduction by a factor of ~8.000
- Conclusion: DB should invest some time in optimization

Steps in Optimization

- Parsing, view expansion, subquery rewriting
- Query minimization (maybe)
- Expression/tree generation
- Plan optimization
  - Algebraic term rewriting (logic optimization)
  - Cost estimation (cost-based optimization)
  - Plan instantiation (physical optimization)
  - Plan enumeration and pruning
  - Note: Steps are interleaved
- Selection of best plan
- Code generation (compilation or interpretation)

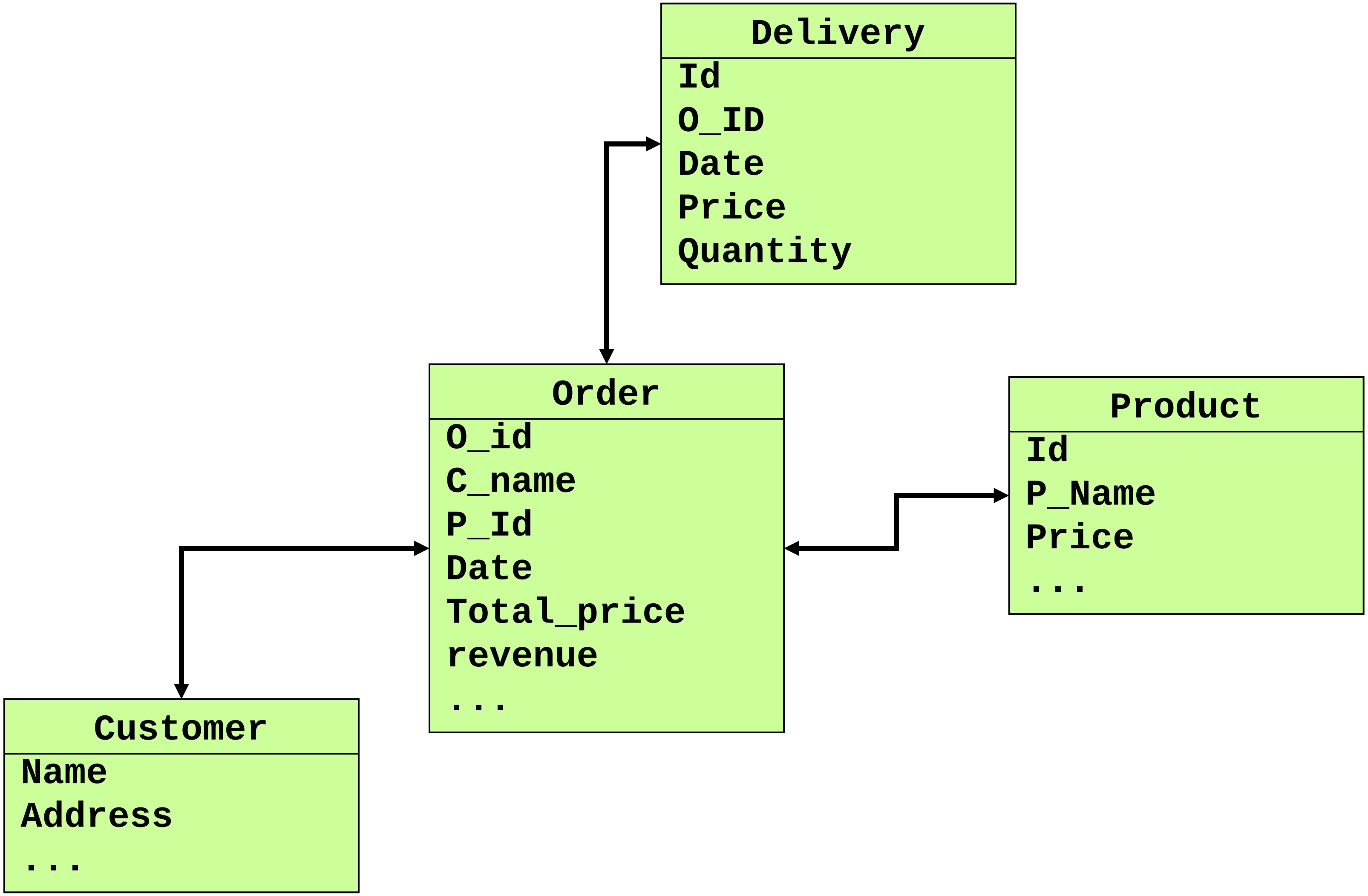
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Subquery Rewriting

- No equivalent in relational algebra: IN, EXISTS, ALL
  – Generate subtrees during parsing
  – For optimization, a single tree with only relational operations is easier to handle
  – But: Transformation not always easy, not always advantageous
- We look at four cases of IN
  – Uncorrelated without aggregation
  – Uncorrelated with aggregation
  – Correlated without aggregation
  – Correlated with aggregation
- See literature for EXISTS, ALL, MINUS, INTERSECT, ...

Example


Delivery
Id
O_ID
Date
Price
Quantity
Order
O_id
C_name
P_Id
Date
Total_price
revenue
...
Product
Id
P_Name
Price
...
Customer
Name
Address
...

Uncorrelated Subquery without Aggregation

```
SELECT o_id
FROM order
WHERE p_id IN (SELECT id
               FROM product
               WHERE price<1)
```

- Option 1: Compute subquery and materialize result
  - Advantageous if subquery appears more than once
- Option 2: Rewrite into join
  - Allows global optimization (i.e. index join)
  - Be careful with duplicates
    - Assuming id is PK of P, example is fine
    - Otherwise, we need to introduce a DISTINCT

```
SELECT o.o_id
FROM   order o, product p
WHERE  o.p_id = p.id AND
        p.price < 1
```

Uncorrelated Subquery with Aggregation

```
SELECT o_id
FROM   order
WHERE  p_id IN (SELECT max(id)
                FROM product)
```

- (Only) option: Compute subquery and materialize result
- Rewriting not possible
- Other way of expression this: User-defined table functions
  - This would allow formulation as join
  - But overall even harder to optimize
- Third way: Use view (two queries)

Correlated Subquery without Aggregation

```
SELECT o.o_id
FROM   order o
WHERE  o.o_id IN (SELECT d.o_id
                  FROM   delivery d
                  WHERE  d.o_id = o.o_id AND
                         d.date-o.date<5)
```

- Subquery materialization not possible
- Naïve computation requires one execution of subquery for each tuple of outer query
- Solution: Rewrite into join
  - Again: Caution with duplicates (if o:d is 1:n, DISTINCT required)

```
SELECT DISTINCT o.o_id
FROM   order o, delivery d
WHERE  o.o_id = d.o_id AND
       d.date-o.date<5
```

Correlated Subquery with Aggregation

```
SELECT o.o_id
FROM   order o
WHERE  o.total_price != (SELECT sum(price*quantity)
                         FROM   delivery d
                         WHERE  d.o_id = o.o_id)
```

- Materialization not easily possible
  - Note that there is only one join condition
- Rewrite into join not possible
- Naïve computation requires one execution of subquery for each tuple of outer query
- Solution: Rewrite into two queries

Correlated Subquery with Aggregation

```
SELECT o.o_id
FROM order o
WHERE o.total_price != (SELECT sum(price*quantity)
                        FROM delivery d
                        WHERE d.o_id = o.o_id)
```

- New inner query

```
CREATE VIEW all_sums AS
SELECT o_id, sum(price*quant) as tp
FROM delivery
GROUP BY o_id
```

- New outer query

```
SELECT o.o_id
FROM order o
WHERE o.total_price !=
  (SELECT tp
   FROM all_sums
   WHERE all_sums.o_id = o.o_id)
```

Can be Combined

```
SELECT o_id, sum(price*quant) as tp
FROM delivery
GROUP BY o_id
```

```
SELECT o.o_id
FROM order o
WHERE o.total_price !=
  (SELECT tp
   FROM all_sums
   WHERE all_sums.o_id = o.o_id)
```

```
SELECT o.o_id
FROM order o, all_sums
WHERE o.total_price != all_sums.tp
```

Improvements

- Inner query can be computed and materialized once
- Inner query will use (efficient) full table scan instead of multiple queries with condition on join attribute

Query Minimization 1

- Especially important when views are involved or queries are created automatically

```
CREATE VIEW good_business
SELECT  C.name, O.O_id, O.revenue
FROM    customer C, order O
WHERE   C.name = O.name AND O.revenue>1.000
```

  – Find very good customers using view as first filter

```
SELECT name                SELECT C.name
FROM   good_business       FROM   customer C, order O
WHERE  revenue>5.000       WHERE  C.name = O.name AND
                                  O.revenue>1.000 AND
                                  O.revenue>5.000
```

- Goal: Remove redundant conditions

Query Minimization 2

- Especially important when views are involved or queries are created automatically

```
CREATE VIEW good_business
SELECT  C.name, O.O_id, O.revenue
FROM    customer C, order O
WHERE   C.name = O.name AND O.revenue>1.000
```

  - Find goods from good businesses

```
SELECT G.name, O.good       SELECT C.name, o2.good
FROM   good_busi G,order O  FROM   custom C,ord O1,ord O2
WHERE  G.o_id = O.o_id      WHERE  C.name=O1.name AND
                                   O1.revenue>1000 AND
                                   O1.o_id=O2.o_id
```

- Remove redundant joins

Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Equivalence of Relational Algebra Expressions

- Definition
  Let E_1 und E_2 be two relational algebra expressions over a schema S. E_1 and E_2 are called equivalent iff
  - *E_1 and E_2 contain the same relations $R_1 \ldots R_n$*
  - *For any instances of S, E_1 and E_2 compute the same result*

- We generate equivalent expressions by applying certain rewrite rules
- We will see some rules (there exist more: literature)

Rules for Joins and Products

- Assume
  - E_1 , E_2 , E_3 relational expressions
  - *Cond*, *Cond1, Cond2* are join conditions

- Rule 1: Joins and Cartesian-products are commutative

$$\mathbf{E_1} \bowtie_{Cond} \mathbf{E_2} \equiv \mathbf{E_2} \bowtie_{Cond} \mathbf{E_1}$$

$$\mathbf{E_1} \times \mathbf{E_2} \equiv \mathbf{E_2} \times \mathbf{E_1}$$

- Rule 2: Joins and Cartesian-products are associative

$$(\mathbf{E_1} \bowtie_{Cond1} \mathbf{E_2}) \bowtie_{Cond2} \mathbf{E_3} \equiv \mathbf{E_1} \bowtie_{Cond1} (\mathbf{E_2} \bowtie_{Cond2} \mathbf{E_3})$$

Requirement: E_3 joins with E_2 (and not with E_1)

$$(\mathbf{E_1} \times \mathbf{E_2}) \times \mathbf{E_3} \equiv \mathbf{E_1} \times (\mathbf{E_2} \times \mathbf{E_3})$$

For Projection and Selection

- Assume
  - $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ be attributes of E
  - *Cond1* und *Cond2* conditions on E

- Rule 3: Cascading projections

  If $A_1, \ldots, A_n \supseteq B_1, \ldots, B_m$, then

  $$\Pi_{\{B1,\ldots,Bm\}}(\Pi_{\{A1,\ldots,An\}}(E)) \equiv \Pi_{\{B1,\ldots,Bm\}}(E)$$

- Rule 4: Cascading selections

  $$\sigma_{Cond1}(\sigma_{Cond2}(E)) \equiv \sigma_{Cond2}(\sigma_{Cond1}(E)) \equiv \sigma_{Cond1 \text{ and } Cond2}(E)$$

For Projection and Selection

- Assume
  - $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ be attributes of E
  - *Cond1* und *Cond2* conditions on E

- Rule 5a. Exchange of projection and selection

$$\pi_{\{A1,\ldots,An\}}(\sigma_{Cond}(E)) \equiv \sigma_{Cond}(\pi_{\{A1,\ldots,An\}}(E))$$

Requirement: *Cond* contains only attributes $A_1, \ldots, A_n$

- Rule 5b. Injection of projection

$$\pi_{\{A1\ldots An\}}(\sigma_{Cond}(E)) \equiv \pi_{\{A1\ldots An\}}(\sigma_{Cond}(\pi_{\{A1\ldots An, B1\ldots Bm\}}(E)))$$

Requirement: *Cond* contains only attributes $A_1 \ldots A_n$ and $B_1 \ldots B_m$

Joins and Projection/Selection

- Rule 6. Exchange of selection and join

$$\sigma_{Cond} (E_1 \bowtie_{Cond1} E_2) \equiv \sigma_{Cond} (E_1) \bowtie_{Cond1} E_2$$

Requirement: *Cond* contains only attributes of E1

- Rule 7. Exchange of selection and union/difference

$$\sigma_{Cond} (E_1 \cup E_2) \equiv \sigma_{Cond} (E_1) \cup \sigma_{Cond} (E_2)$$

$$\sigma_{Cond} (E_1 - E_2) \equiv \sigma_{Cond} (E_1) - \sigma_{Cond} (E_2)$$

Joins and Projection/Selection

- Rule 9. Exchange of projection and join:

$$\Pi_{\{A1,\ldots,An,B1,\ldots,Bm\}}(E_1 \bowtie_{Cond} E_2) \equiv \Pi_{\{A1,\ldots,An\}}(E_1) \bowtie_{Cond} \Pi_{\{B1,\ldots,Bm\}}(E_2)$$

Requirement: Cond contains only attributes $A_1 \ldots A_n$, $B_1 \ldots B_m$ and $A_1 \ldots A_n$ appear in E_1 , resp. $B_1 \ldots B_m$ in E_2

- Rule 10. Exchange of projection and union:

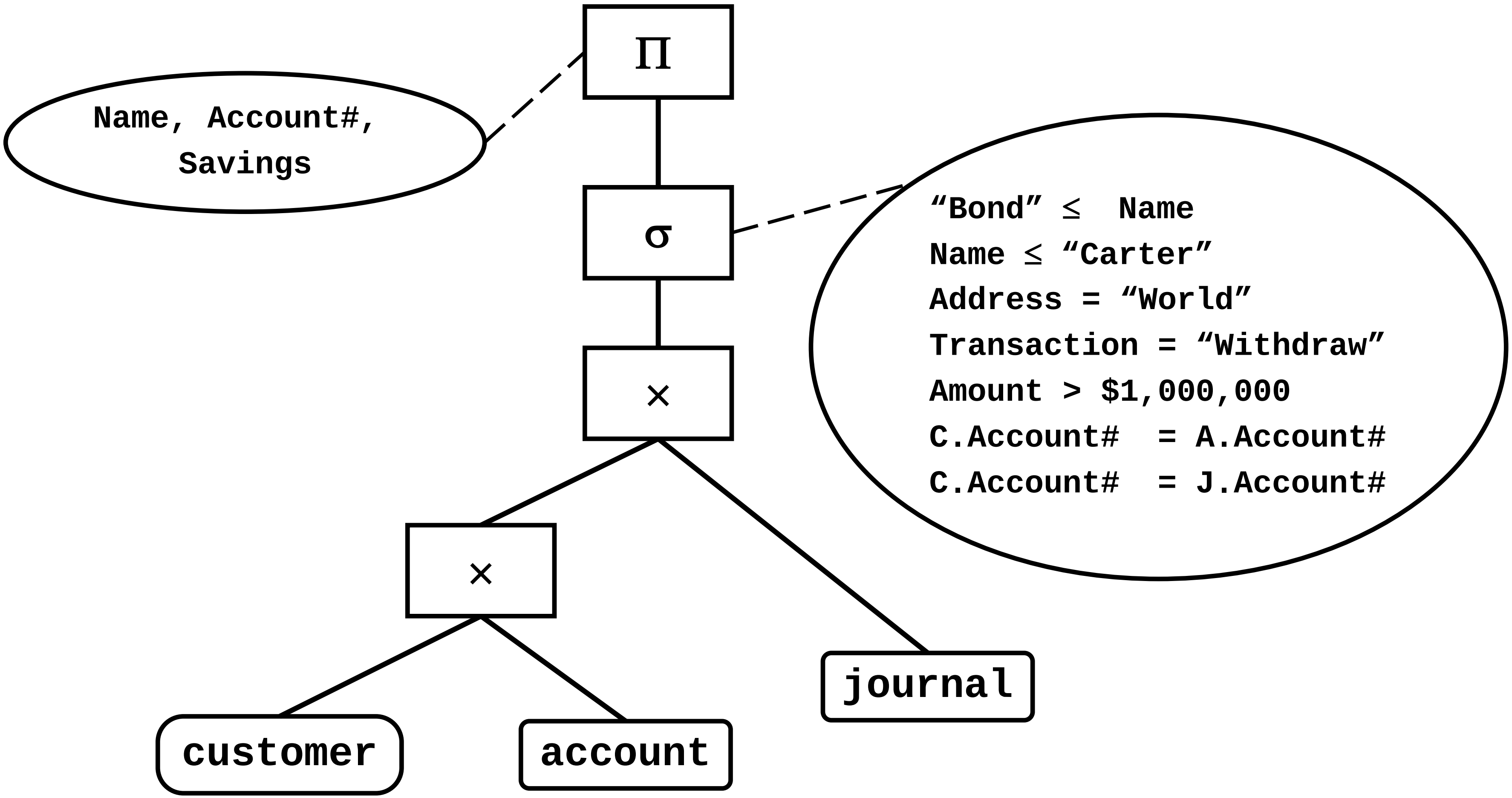
$$\Pi_{\{A1,.,An\}}(E_1 \cup E_2) \equiv \Pi_{\{A1,..,An\}}(E_1) \cup \Pi_{\{A1..,An\}}(E_2)$$

Example

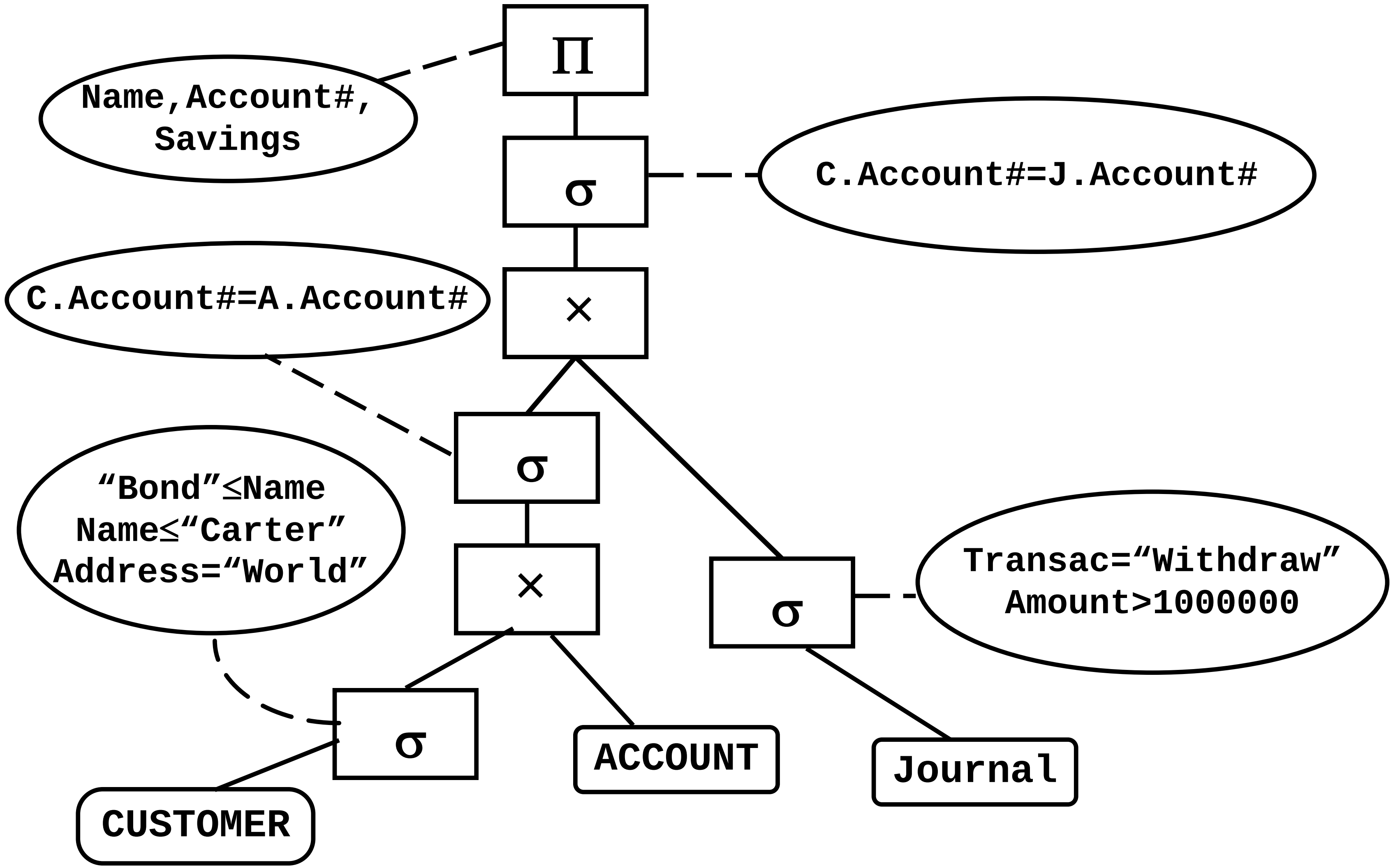
- Query on CUSTOMER database

```
SELECT    Name, Account#, Savings
FROM      customer C, account A, journal J
WHERE     “Bond” ≤ Name ≤ “Carter”          and
          Address = “World”                 and
          Transaction = “Withdraw”          and
          Amount > 1,000,000                and
          C.Account# = A.Account#           and
          C.Account# = J.Account#
```

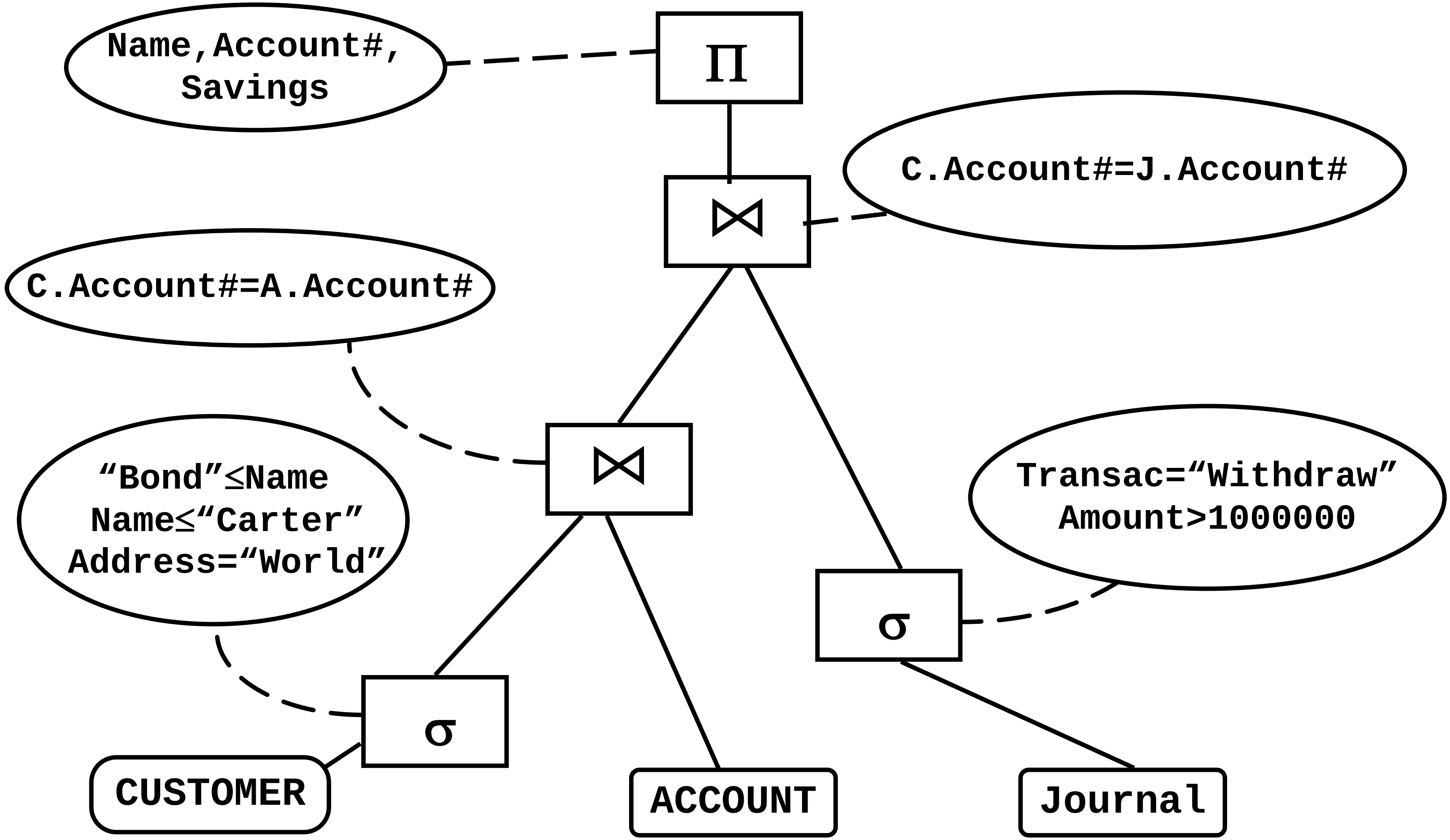
Initial Operator Tree



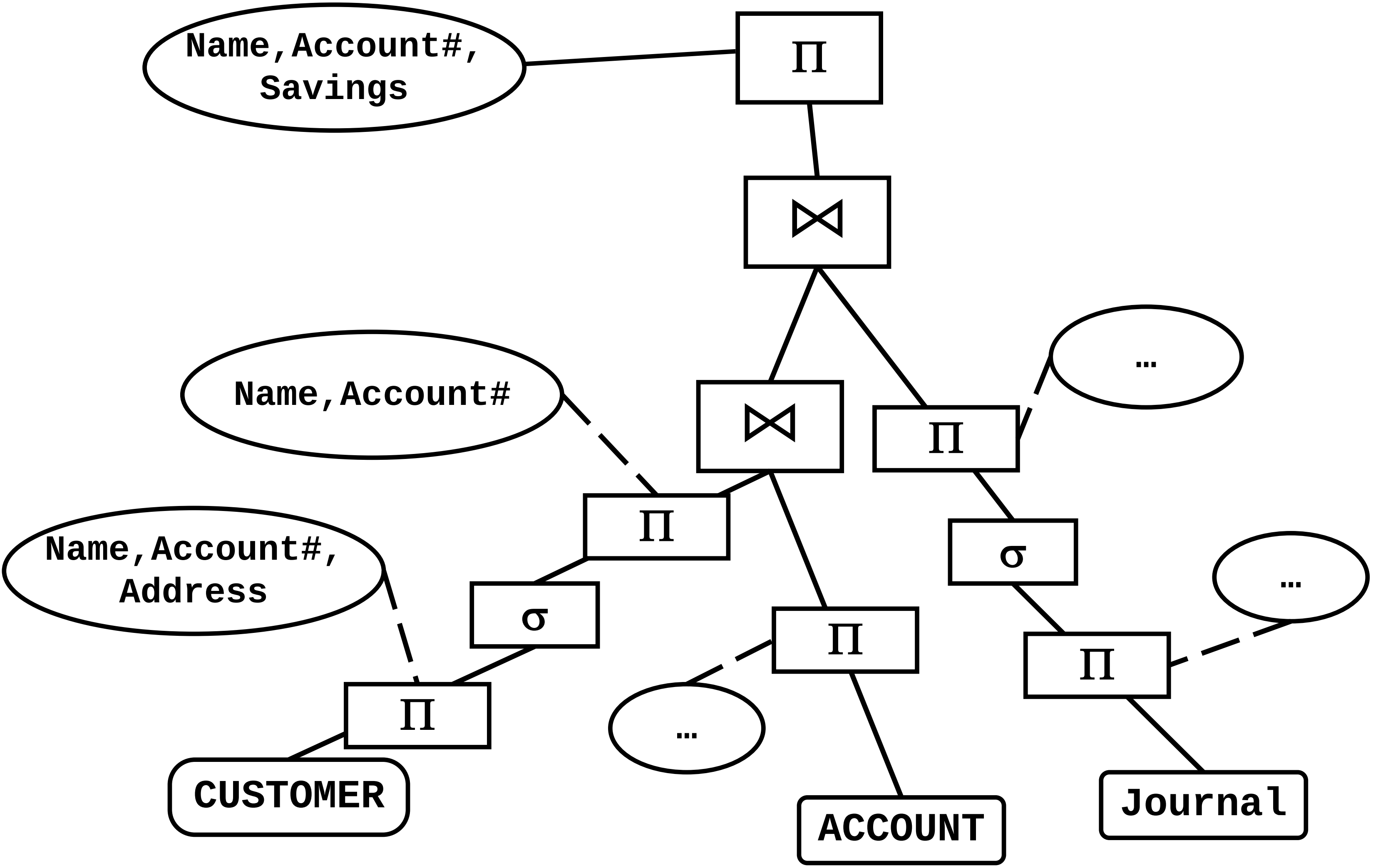
Breaking and Pushing Selections



Introduce Joins



Pushing Projections

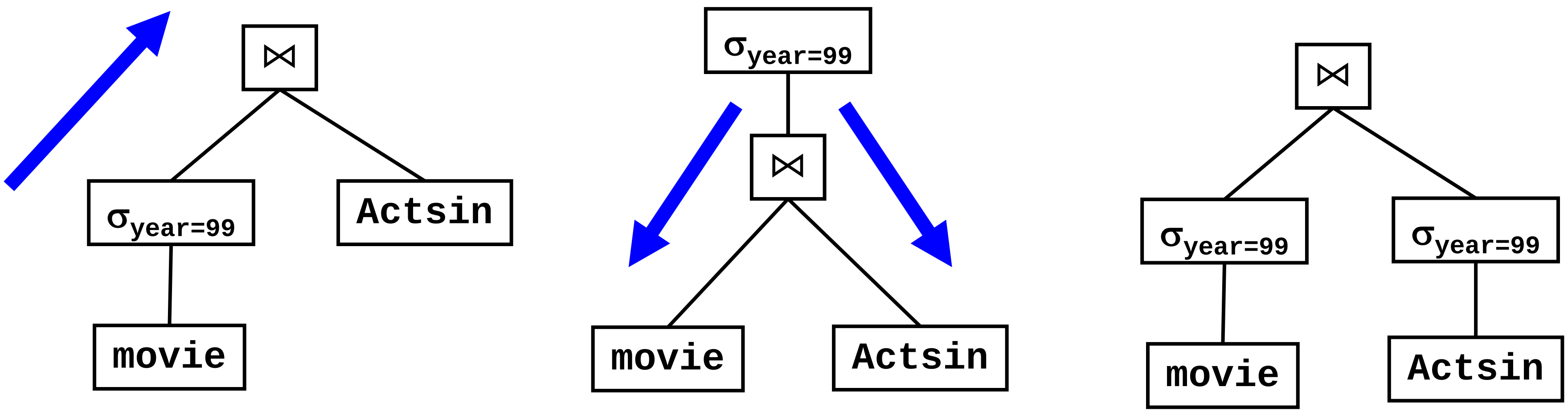


Caution

- Sometimes, pushing up selections is good
  - Especially for conditions on join attributes
- Example

```
CREATE VIEW movies99 AS
SELECT title, year, studio
FROM   movie WHERE year=1999
```

```
SELECT m.title, a.name
FROM   movies99 m, actsin a
WHERE  m.title=a.title AND
       m.year=a.year
```



Term Rewriting: Algebraic Optimization

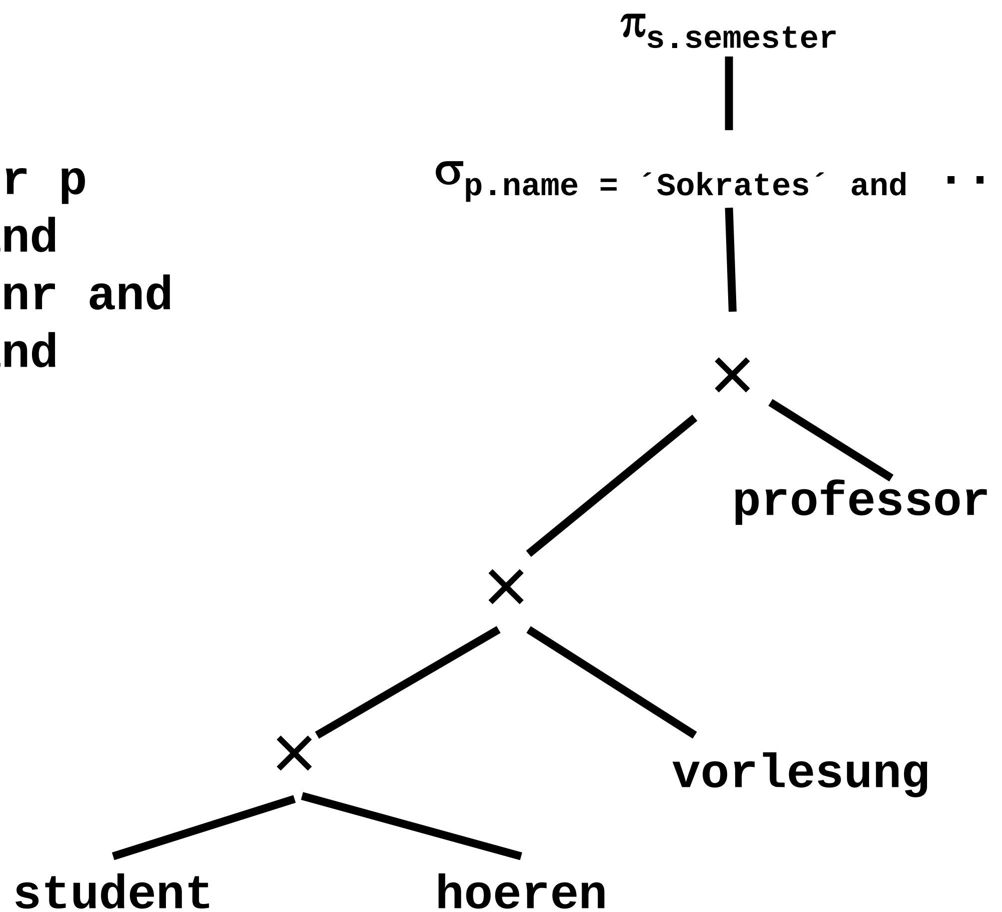
- Usually there infinitely many rewrite steps
  - But not infinitely many different plans
  - Rewritings often go back and forth
- General heuristic: Minimize intermediate results
  - Less IO if materialization is necessary
  - Less input for operations that are higher in the plan
- Option1: Rule-based
  - Use heuristics for selecting order of rule application
  - Based on experience – rules that are beneficial in most cases
  - Simple to implement, fast optimizer
  - But: Unusual queries lead to bad plans

A Simple Rule-Based Optimizer

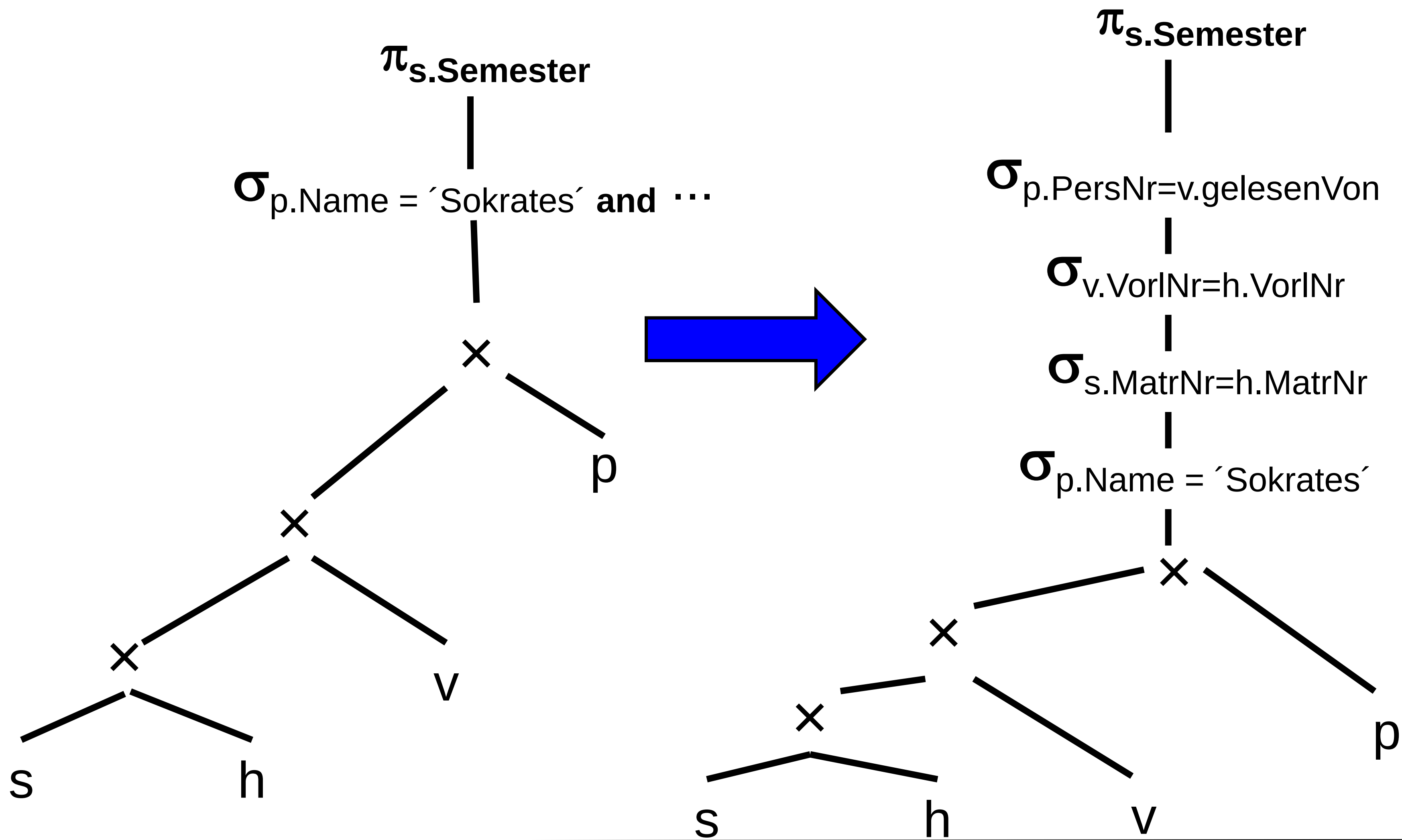
- Down – break and push down
  - Break combined selections into many simple selections
  - Break combined projections into many simple projections
  - Push selects/projects as much down the tree as possible
  - Introduce add. projections as deep in the tree as possible
- Up – merge operations
  - Replace selection and Cartesian product with join
  - Merge simple selections into combined selections
  - Merge simple projections into combined projections
- Physical
  - If there is a condition on an indexed attribute – use the index
    - Conflicts with break / merge patterns
  - For a join over PK-FK relationships: Use sort-merge
  - Other joins: Use hash join

Another Example

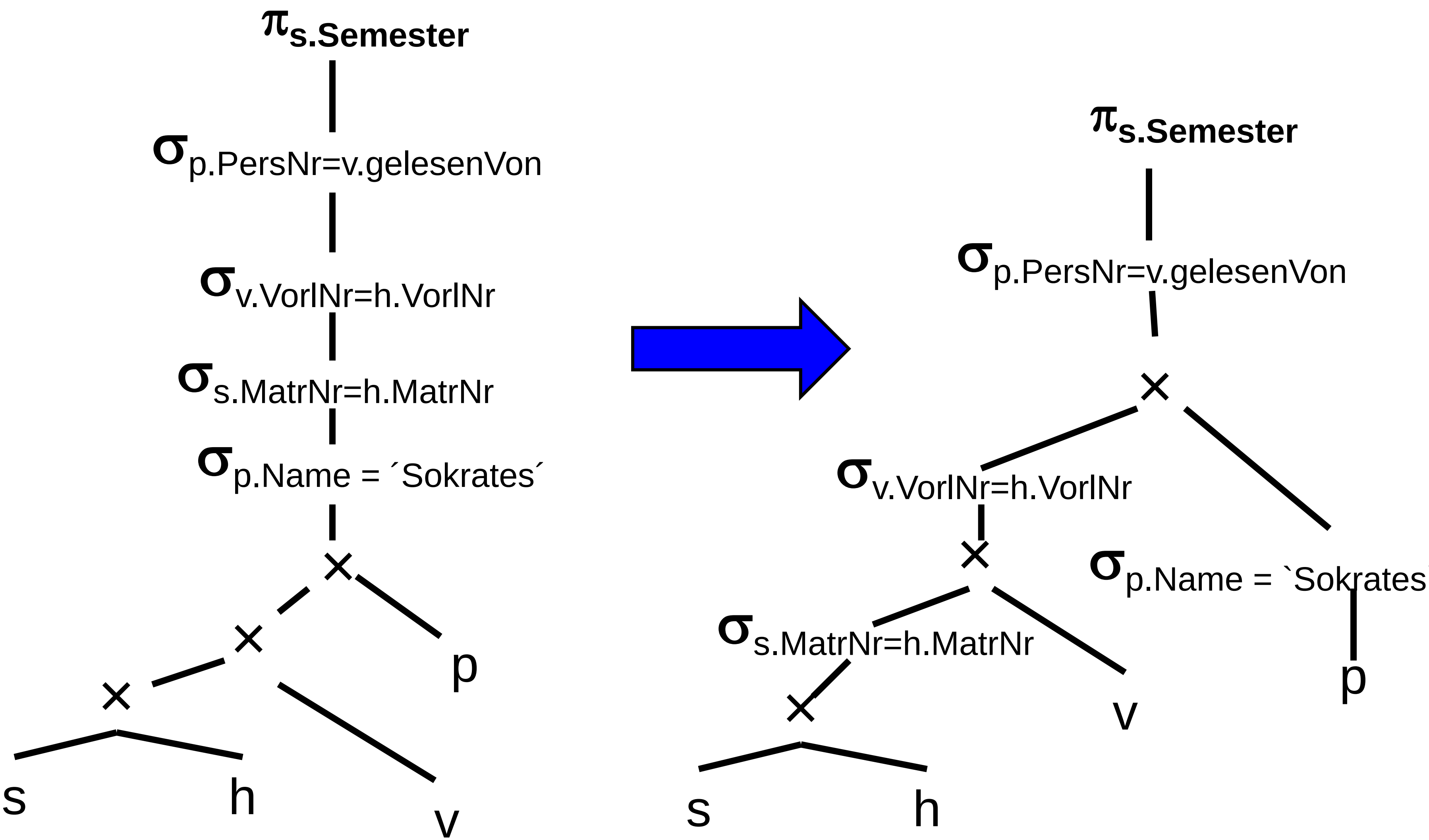
```
SELECT s.Semester
FROM   student s, hoeren h
       vorlesung v, professor p
WHERE  p.name = “Sokrates” and
       v.gelesenvon = p.persnr and
       v.vorlnr = h.vorlnr and
       h.matrnr = s.matrnr
```



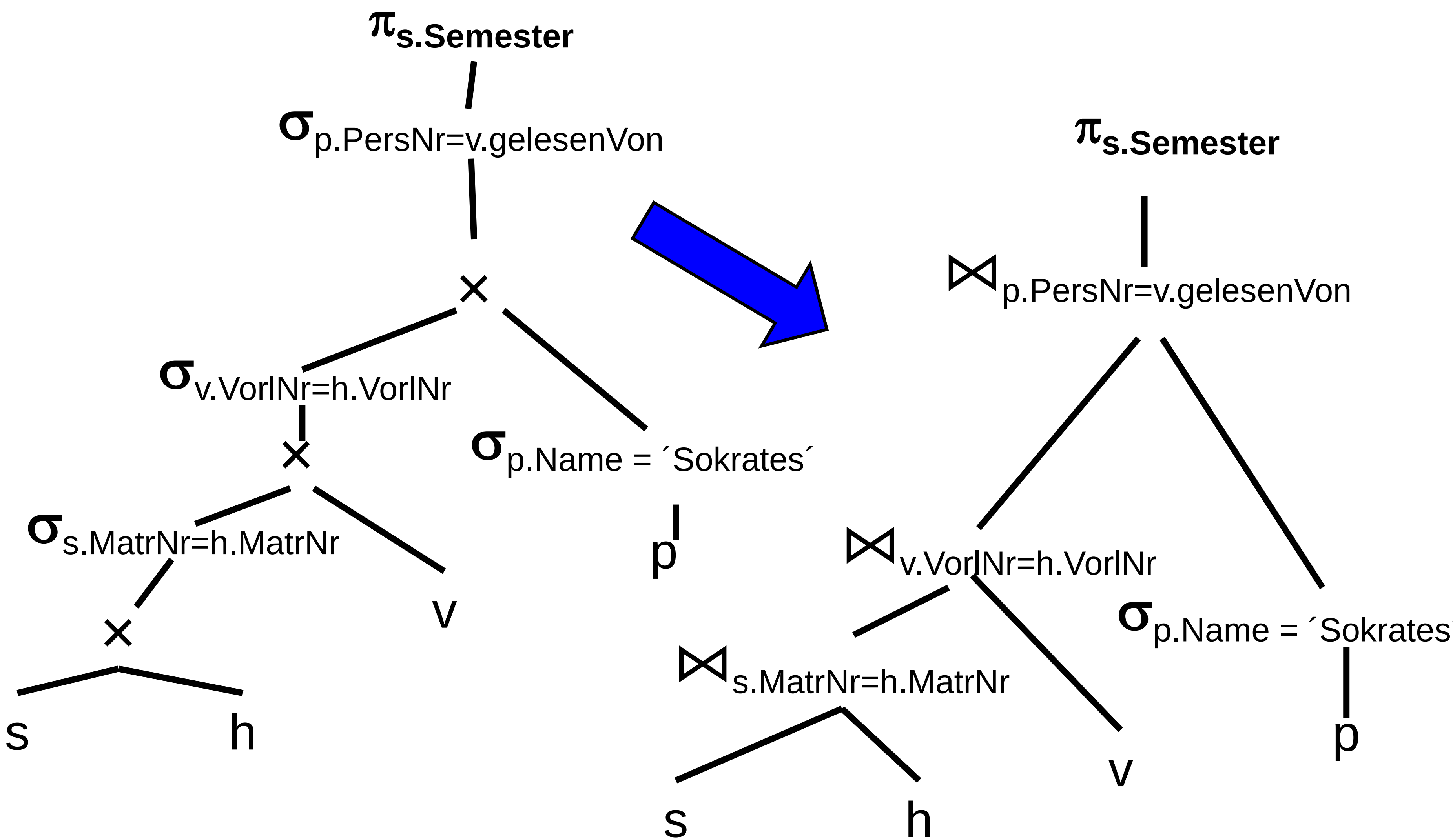
Break Up Selections



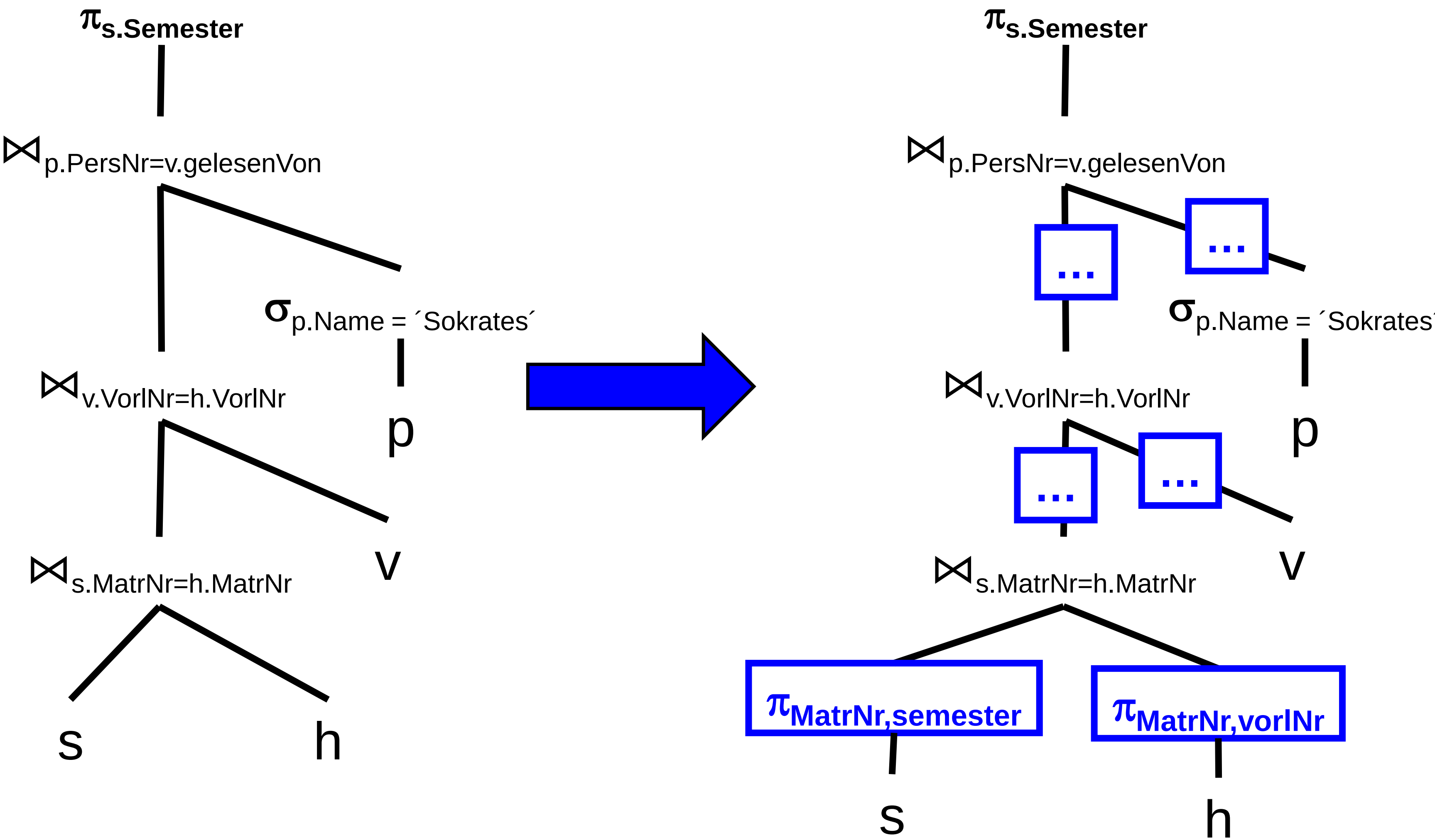
Push Selections



Rewrite Product+Selection into Joins



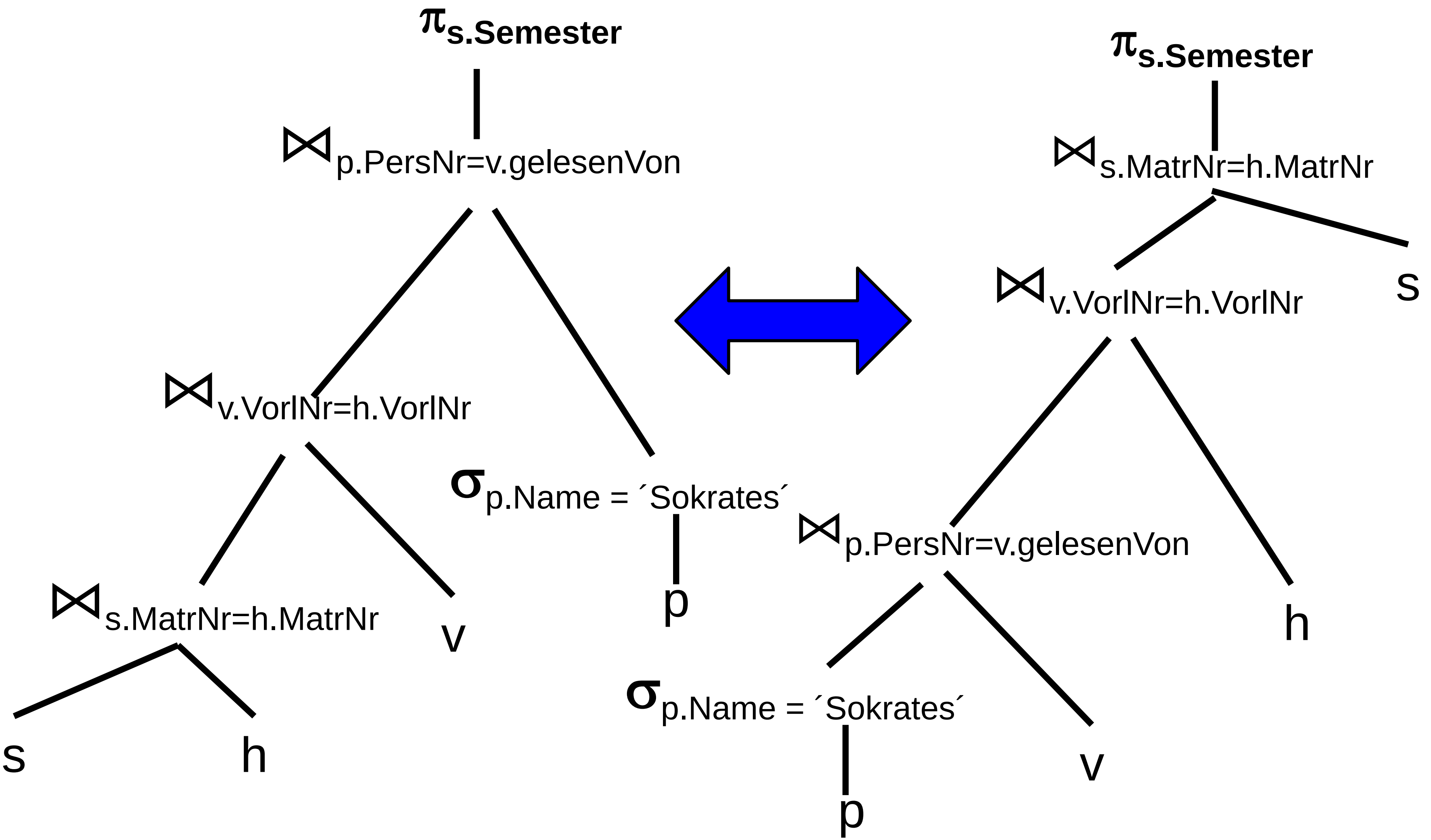
Introduce Additional Projections



Limitations

- Rule-based optimization is data-independent
  - Optimal selection of operators impossible without estimates about size of results (cardinality, width)
  - No rules for order of join processing
  - Rules are partly contradictory
    - E.g. Conjunctive selections and composite indexes
  - Benefit of indexes depends on selectivity
  - ...
- Option 2: Cost-based optimization
  - Estimate effect of rewritings on size of intermediate results (SIR)
  - Different optimization goals
    - Greedy: Chose next rewrite with greatest saving in SIR
    - Global: Chose plan with overall smallest SIR
    - Bound: Chose plan with smallest maximal SIR

Order of Joins: Indistinguishable



Join Order – Does it Matter?

- Assume uniform distributions
  - There are 1.000 students, 20 professors, 80 courses
  - Each professor gives 4 courses
  - Each student listens to 4 courses
  - Each course is followed by 50 students (4000 “hören” tuples)

Join Order – Does it Matter?

```
SELECT   s.Semester
FROM     student s, hoeren h
         vorlesung v, professor p
WHERE    p.name = “Sokrates” and
         v.gelesenvon = p.persnr and
         v.vorlnr = h.vorlnr and
         h.matrnr = s.matrnr
```

- Compute $\sigma_{Sokrates}(P) \bowtie (V \bowtie (S \bowtie H))$
  - Inner join: 1000*4 = 4000 tuples
  - Next join: Again 4000 tuples
  - Last join selects only 1/20 of intermediate results = 200
  - Intermediate result sizes: 4000 + 4000 + 1 = 8001
- Compute $S \bowtie (H \bowtie (\sigma_{Sokrates}(P) \bowtie V))$
  - Inner join selects 4 tuples
  - Next join generates 50*4= 200 tuples
  - Last join: No change
  - Intermediate result sizes: 1 + 4 + 200 = 205

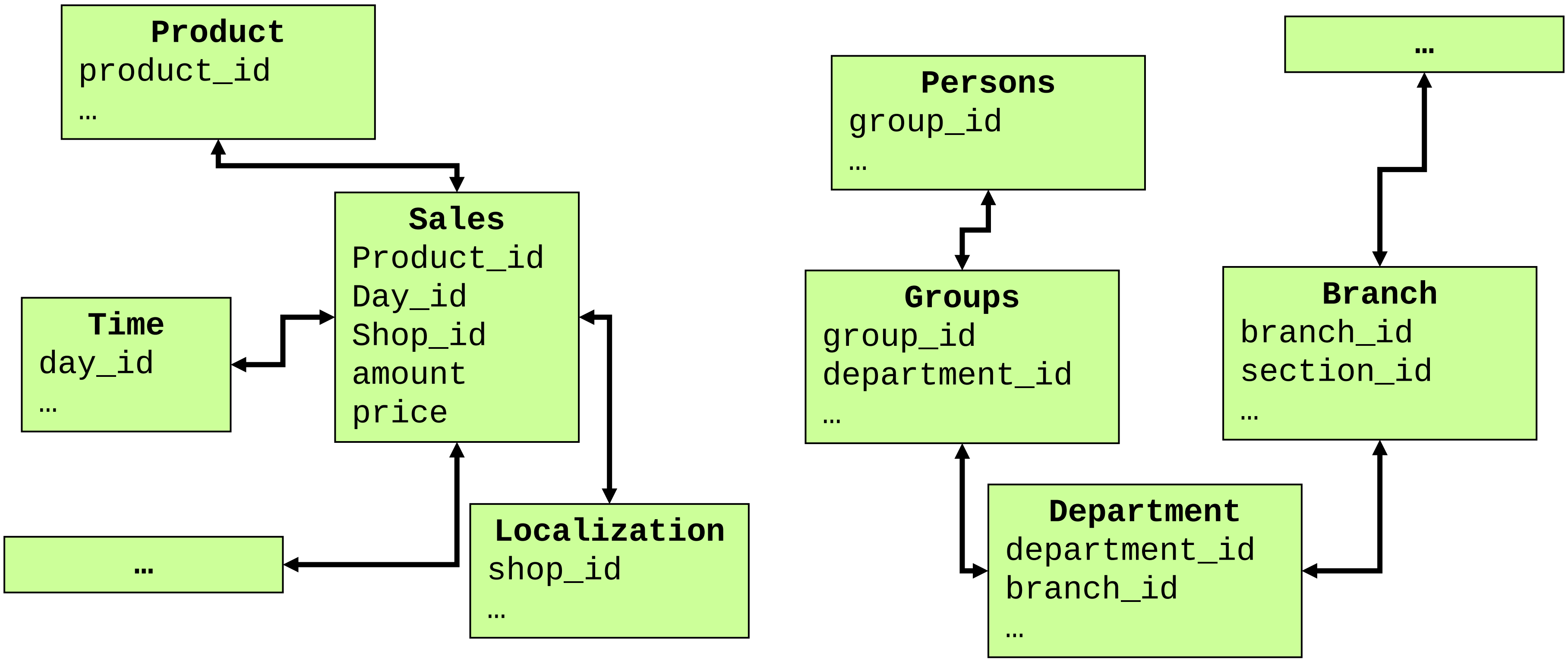
Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Optimizing Join Order

- From the relation algebra perspective, join is associative and commutative - reordering doesn't change result
- But execution times of different orders differ tremendously
  - If there are at least two joins, e.g. $R \bowtie (S \bowtie T) \equiv (S \bowtie R) \bowtie T$
- Join versus cross-product
  - Depending on join conditions, many orders involve intermediate cross-products
  - Most join-order algorithms disregard any plan containing a cross-product – which heavily reduces the search space
  - In the following, we assume that no order involves a cross-product
- Given n relations, there are n! possible orders

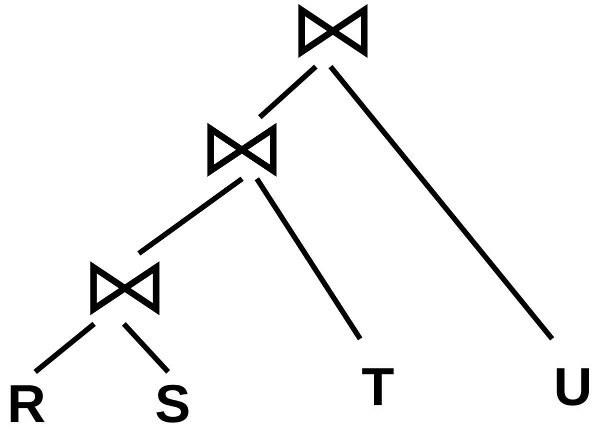
Query Types



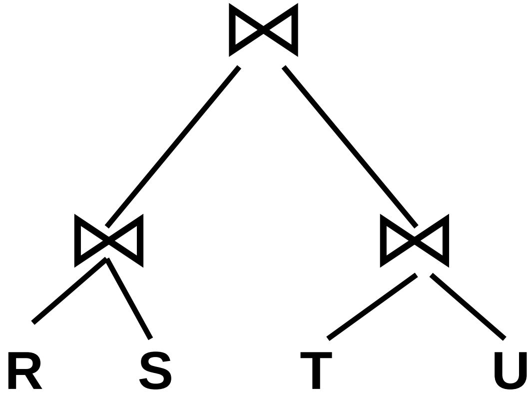
- $((S \bowtie R) \bowtie T) \bowtie L$
- $((S \bowtie L) \bowtie R) \bowtie T$
- …

- $((P \bowtie G) \bowtie T) \bowtie B$
- $(P \bowtie G) \bowtie (T \bowtie B)$
- …

Left/Right-deep versus Bushy Join Trees



Left-deep join tree



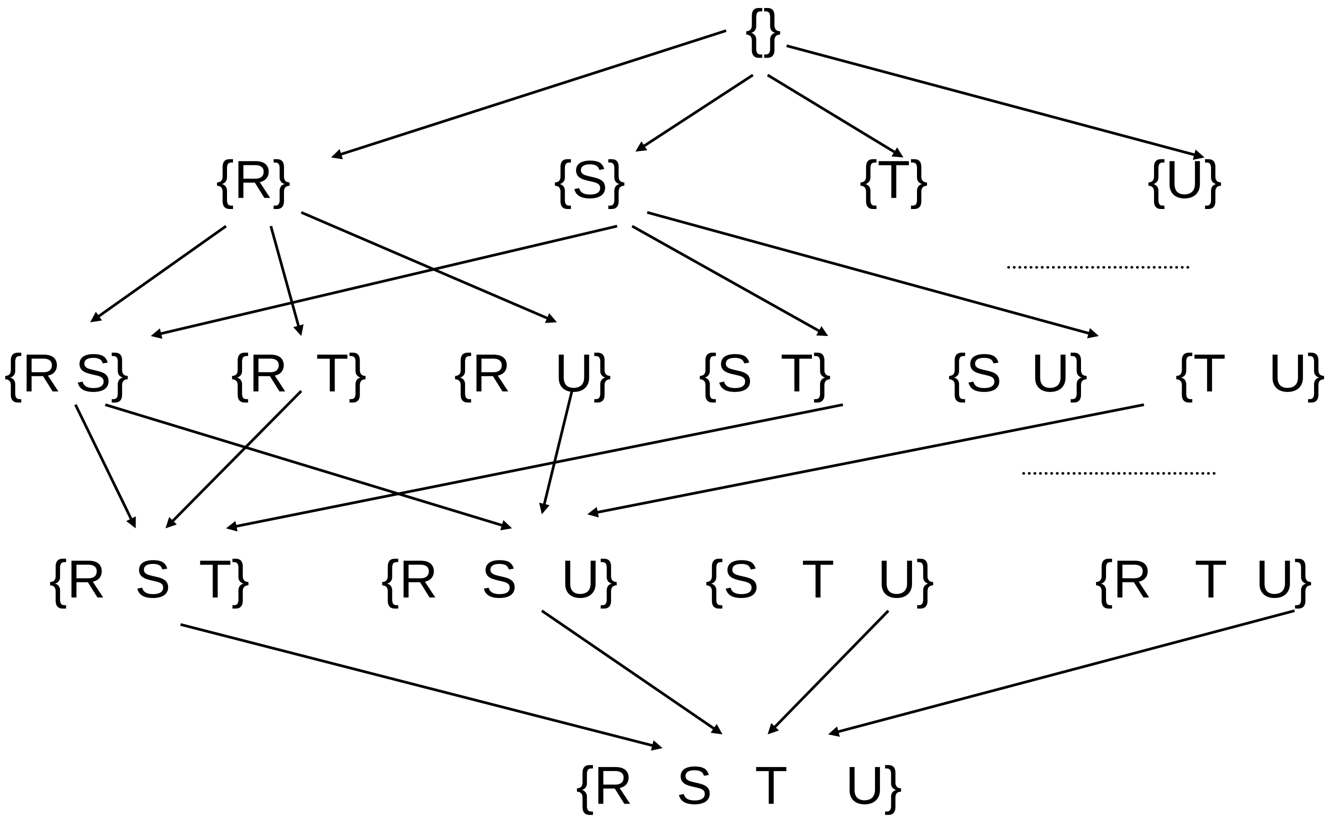
Bushy join tree

- There is one left-deep tree topology, but still O(n!) orders
- There are $(2n-3)!/(2^{n-2}*(n-2)!)$ unordered binary trees with n leaves, and for each O(n!) orders
  - Some are equivalent

Choosing a Join Order

- Typical first heuristic: Consider only left-deep trees
  - Can be pipelined efficiently
  - Usually generates among the best plans
- But there are still O(n!) possible orders
- Second Heuristic: Use dynamic programming with pruning
  - Generate plans bottom up: Plans for pairs, triples, ...
  - For each join group, keep only best plan
  - Use these to enumerate possibilities for larger join groups
  - Prune all subplans containing a cross product

Join Groups



- There are (n over i) join groups with i elements

Details

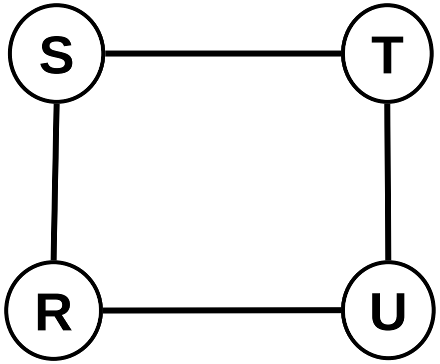
- Create a table containing for each join group
  - [Prune if this would involve a Cartesian product]
  - Estimated size of result (how: later)
  - Optimal cost for computing this group
    - For now, we simply take sum of sizes of intermediate results so far
  - Optimal plan for computing this group

Induction

- Induction over plan length = sizes of join groups
  - i=1: Consider every relation in isolation
    - Size = Size of relation
    - Cost = 0 (access costs are fixed for all plans anyway)
      - Not true is pushing of selections is considered
  - i=2: Consider each pair of relations
    - Size: Estimated size of “joining” both relations (might be product)
    - Cost = 0 (no intermediate result so far due to previous assumption)
    - Fix join method to use (e.g.: BNL with smaller relation as inner relation)
      - This method will never change again
  - i=3: Consider each pair in each triple and join with third relation
    - Consider only chosen methods for pairs involved
    - ...

Example 1

- We join four relations R, S, T, U
- Four join conditions



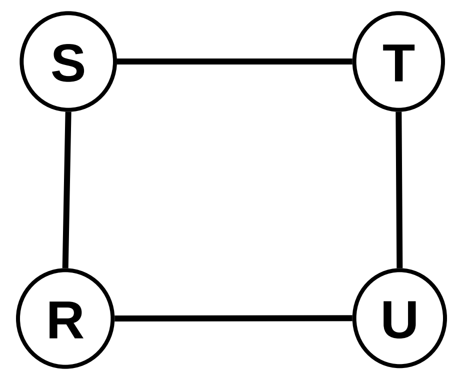
	{R}	{S}	{T}	{U}
Kardinalität	1000	1000	1000	1000
Kosten	0	0	0	0
Optimaler Plan	scan(R)	scan(S)	scan(T)	scan(U)

Example 2

	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
Kardinalität	5000	~~1M~~	10000	2000	~~1M~~	1000
Kosten	0	~~0~~	0	0	~~0~~	0
opt. Plan	R ⋈ S	~~R ⋈ T~~	R ⋈ U	S ⋈ T	~~S ⋈ U~~	T ⋈ U

Prune products

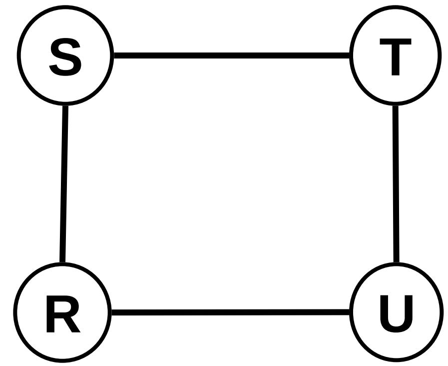
	{R,S,T}	{R,S,U}	{R,T,U}	{S,T,U}
Kardinalität	10000	50000	10000	2000
Kosten	2000	5000	1000	1000
opt. Plan	(S⋈ T)⋈ R	(R⋈ S)⋈ U	(T⋈ U)⋈ R	(T⋈ U)⋈ S



Better than
S⋈(T⋈R) and (R⋈S)⋈T

Example 3

	{R,S,T}	{R,S,U}	{R,T,U}	{S,T,U}
Kardinalität	10000	50000	10000	2000
Kosten	2000	5000	1000	1000
opt. Plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$



Plan	Kosten
$((S \bowtie T) \bowtie R) \bowtie U$	12k
$((R \bowtie S) \bowtie U) \bowtie T$	55k
$((T \bowtie U) \bowtie R) \bowtie S$	11k
$((T \bowtie U) \bowtie S) \bowtie R$	3k

(Hopefully) optimal left-deep plan

Algorithm

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

for $i = 1$ **to** n **do** {
    $optPlan(\{R_i\}) = accessPlans(R_i)$
    $prunePlans(optPlan(\{R_i\}))$
}
for $i = 2$ **to** n **do** {
    for all $S \subseteq \{R_1, \ldots, R_n\}$ **such that** $|S| = i$ **do** {
        $optPlan(S) = \emptyset$
        for all O **such that** $S \cup X = O$
            $optPlan(S) = optPlan(S) \cup joinPlans(optPlan(O), X)$
            $prunePlans(optPlan(S))$
        }
    }
}
return $\text{optPlan}(\{R_1, \ldots, R_n\})$

Enumerate physical plans for accessing R_i

Prune all except one

Prune all except one

Dynamic Programming

- DP here is a heuristic
  - Assumption of DP: Any subplan of an optimal plan is optimal
  - True for computing shortest paths, edit distance, ...
- But not true for join-order
  - Using a sort-merge join early in a plan might not be optimal for this particular join group - but result is sorted
  - Later joins can profit and also use sort-merge without sorting one intermediate relation again
  - Optimal plan might involve Cartesian product
- Solution (for sort order)
  - Keep different “optimal” plans for each join group
  - System R: One “optimal” plan per interesting sort order
    - Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G. (1979). "Access Path Selection in a Relational Database Management System". SIGMOD 1979

Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Ingredients

- We can evaluate different access paths for a single relation
- We can generate various equivalent relational algebra terms for computing a query
- We can optimize join order
  - Given selectivity estimates
- Query optimization =

  Search space (space of all possible plans) +

  Search strategy (algorithm to enumerate plans) +

  Cost functions for pruning plans (still missing)

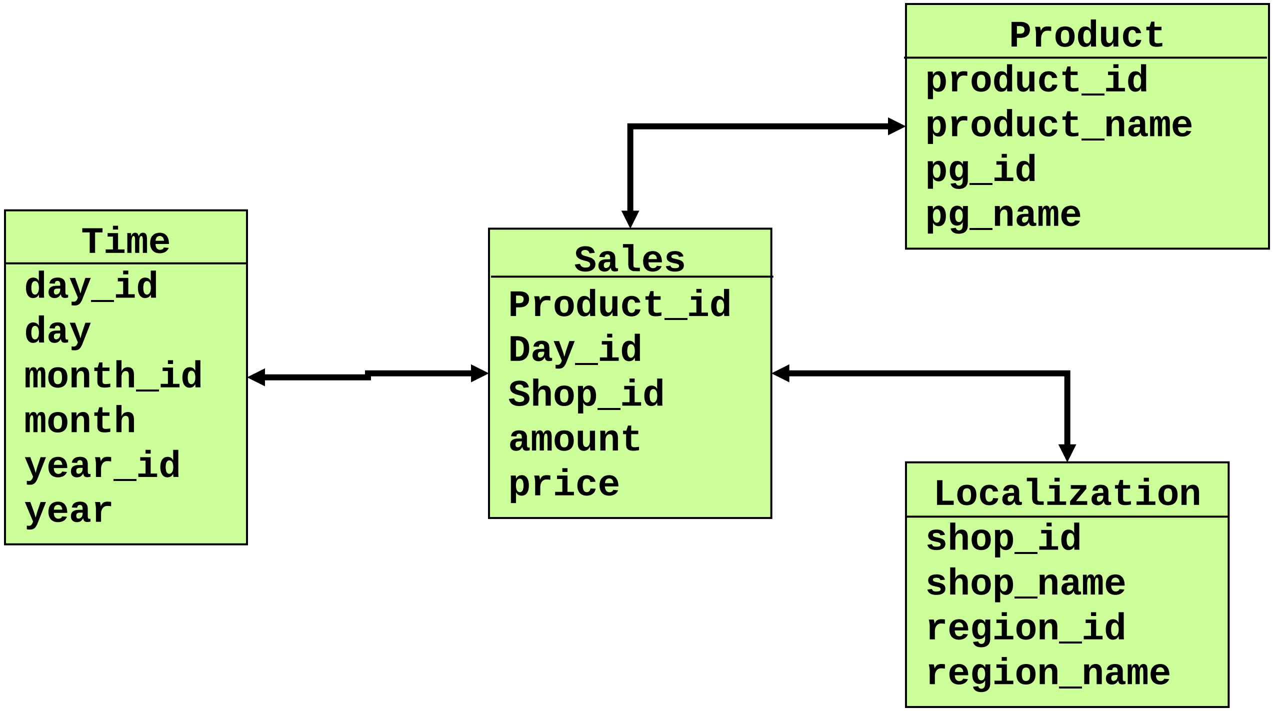
Search Strategies

- Searching a huge search space for a good (optimal) solution is a common computer science problem
  - Exhaustive search
    - Guarantees optimal result, but often too expensive
  - Heuristic method
    - Greedy/Hill-Climbing: only use one alternative for further search
  - Genetic optimization
    - Generate some good plans
    - Build combinations
  - Simulated annealing
  - ...
- Many join-order algorithms: Steinbrunn, Moerkotte, Kemper (1997). "Heuristic and randomized optimization for the join ordering problem." *VLDB Journal: 191-208.*

Content of this Lecture

- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Star Join



- Typische Anfrage gegen Star Schema
  - Aggregation und Gruppierung
  - Bedingungen auf den Werten der Dimensionstabellen
  - Joins zwischen Dimensions- und Faktentabelle

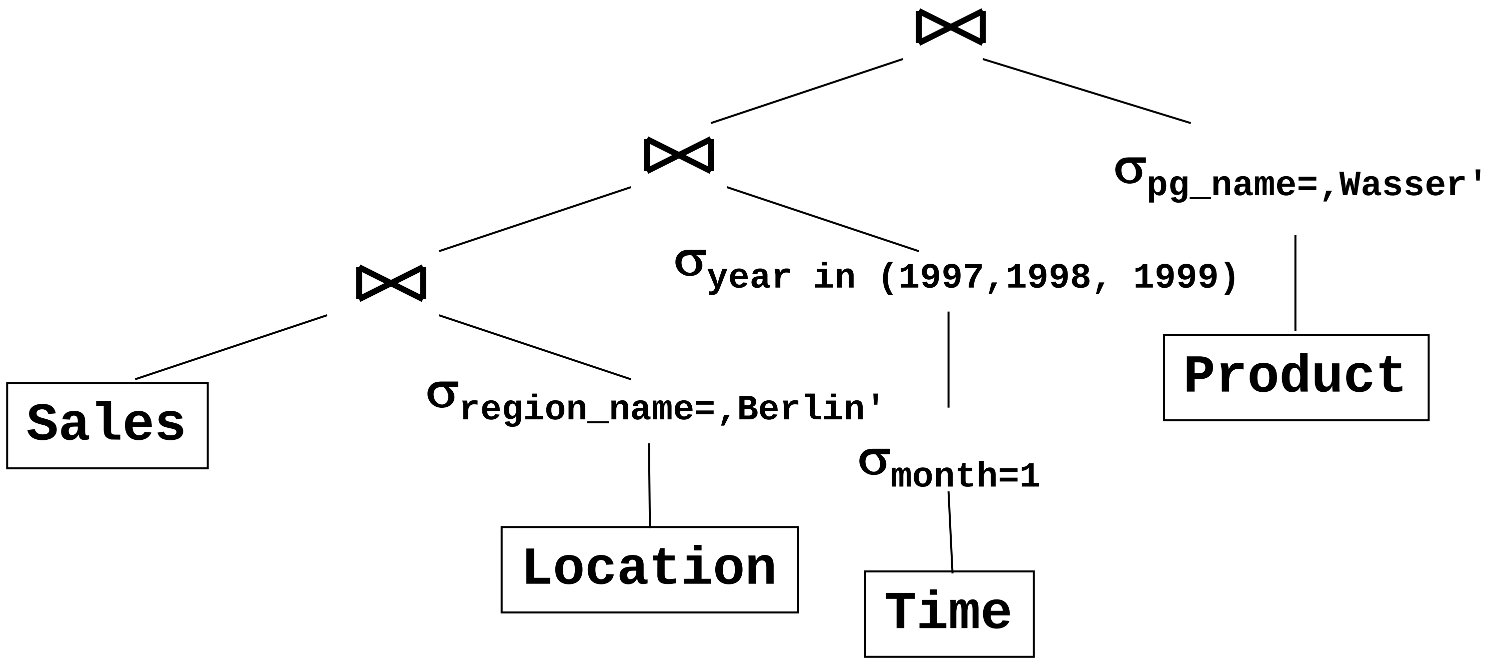
Beispielquery

- Alle Verkäufe von Produkten der Produktgruppe ‚Wasser' in Berlin im Januar der Jahre 1997, 1998, 1999, gruppiert nach Jahr

```
SELECT T.year, sum(amount*price)
FROM Sales S, Product P, Time T, Localization L
WHERE  P.pg_name=‚Wasser' AND
       P.product_id = S.product_id AND
       T.day_id = S.day_id AND
       T.year in (1997, 1998, 1999) AND
       T.month = ‚1' AND
       L.shop_id = S.shop_id AND
       L.region_name=‚Berlin'
GROUP BY T.year
```

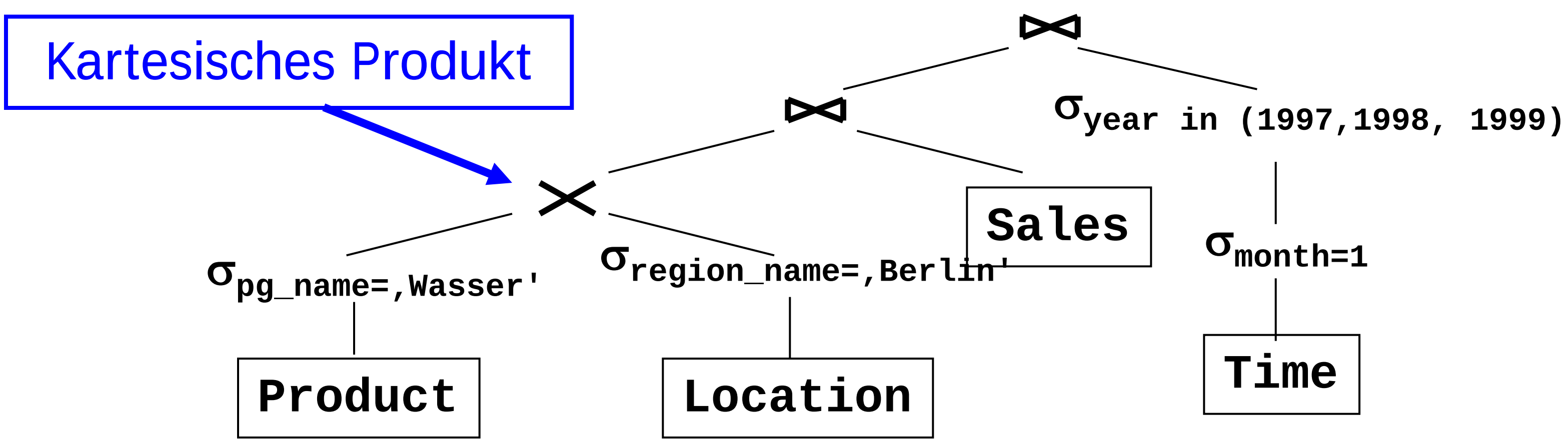
Anfrageplanung

- Anfrage enthält 3 Joins über 4 Tabellen
- Zunächst 4! left-deep join trees
  - Aber: Nicht alle Tabellen sind mit allen gejoined
- Nur 3! beinhalten kein Kreuzprodukt



Heuristiken

- Typisches Vorgehen
  - Auswahl des Planes nach Größe der Zwischenergebnisse
  - Keine Beachtung von Plänen, die kartesisches Produkt enthalten



Abschätzung von Zwischenergebnissen

```
SELECT T.year, sum(amount*price)
FROM Sales S, Product P, Time T, Localization L
WHERE     P.pg_name=‚Wasser‘ AND
          P.product_id = S.product_id AND
          T.day_id = S.day_id AND
          T.year in (1997, 1998, 1999) AND
          T.month = ‚1‘ AND
          L.shop_id = S.shop_id AND
          L.region_name=‚Berlin‘
GROUP BY T.year
```

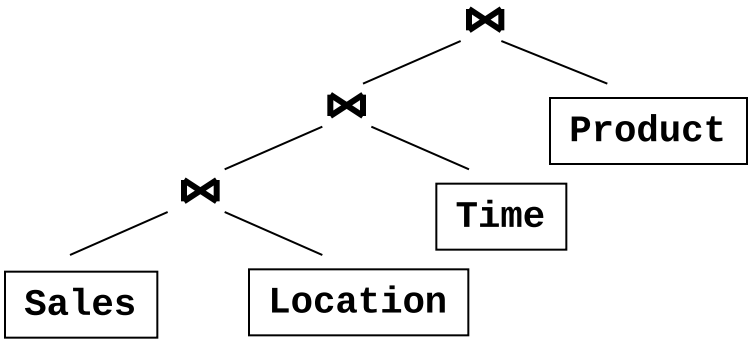
Annahmen

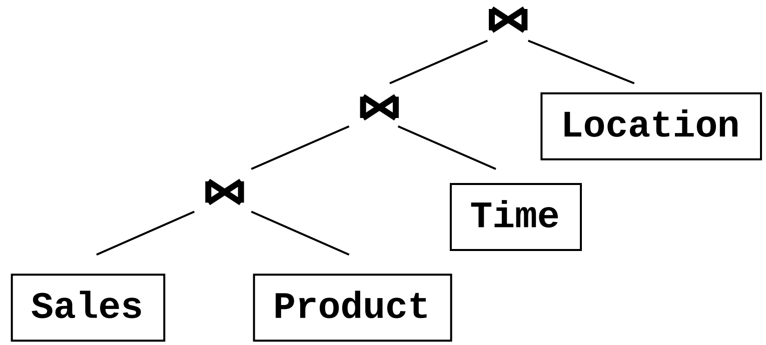
- M= |S| = 100.000.000
- 20 Verkaufstage pro Monat
- Daten von 10 Jahren
- 50 Produktgruppen a 20 Produkten
- 15 Regionen a 100 Shops
- Gleichverteilung aller Verkäufe

Größte des Ergebnis

- Selektivität Zeit
  - 60 Tage:
    (M / (20*12*10)) * 3*20
- Selektivität ‚Wasser‘
  - 20 Produkte
    (M / (20*50)) * 20
- Selektivität ‚Berlin‘
  - 100 Shops
    (M / (15*100)) * 100
- Gesamt
  - 3.333 Tupel
- Selektivität: 0,00003%

Left-deep Pläne

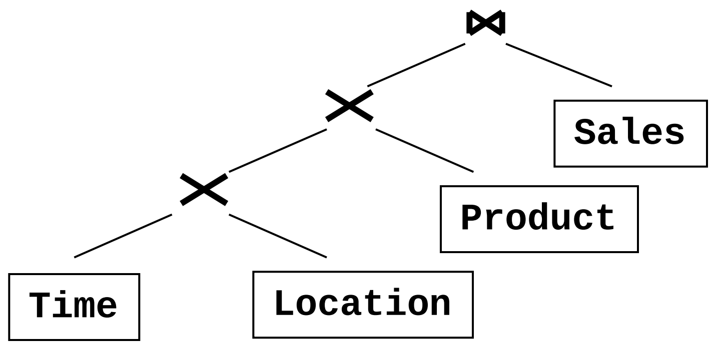




	Zwischen-ergebnis
1. Join (M / 15)	6.666.666
2. Join ($\lvert J_1 \rvert$*3/120)	166.666
3. Join ($\lvert J_2 \rvert$/50)	3.333

	Zwischen-ergebnis
1. Join (M / 50)	2.000.000
2. Join ($\lvert J_1 \rvert$*3/120)	50.000
3. Join ($\lvert J_2 \rvert$/ 15)	3.333

Plan mit kartesischen Produkten



	Zwischenergebnis
1. Time x Location (3*20 * 100)	6.000
2. … x Product ($\|P_1\|$*20)	120.000
3. … ⋈ Sales	3.333

- Es gibt mehr „Zellen" als Verkäufe
- Nicht an jedem Tag wird jed. Produkt in jed. Shop verkauft

STAR Join in Oracle (v7)

- STAR Join Strategie in Oracle v7
  - Kartesisches Produkt aller Dimensionstabellen
  - Zugriff auf Faktentabelle über Index
    - Hohe Selektivität für Anfrage wichtig
    - Zusammengesetzter Index auf allen FKs muss vorhanden sein
    - Sonst „nur“ kleinere Zwischenergebnisse, aber trotzdem teurer Scan
- Aber: Nicht immer gut
  - Daten für 3 Monate, 10 Jahre, 5 Regionen, 10 Produktgruppen
  - Größe des kartesischen Produkts:
    3*20*10 * 5*100 * 10*20 = 60.000.000

STAR Join in Oracle 8i – 9i

- Möglichkeit der (komprimierten) Bitmapindexe lässt kartesisches Produkt weniger vorteilhaft erscheinen
- Phasen
  1. Berechnung aller FKs in Faktentabelle gemäß Dimensionsbedingungen einzeln für jede Dimension
  2. Anlegen/laden von Join-Bitmapindexen auf allen FK Attributen der Faktentabelle
  3. Merge (AND) aller Bitmapindexe
  4. Direkter Zugriff auf Faktentabelle über TID
  5. Join nur der selektierten Fakten mit Dimensionstabellen zum Zugriff auf Dimensionswerte
- Zwischenergebnisse sind nur (komprimierte) Bitlisten

Gesamtplan

Phase 2

Phase 1

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                                   LOCATION
   HASH JOIN
    TABLE ACCESS FULL                                  TIME
    HASH JOIN
     TABLE ACCESS FULL                                 PRODUCT
     PARTITION RANGE ALL
      TABLE ACCESS BY LOCAL INDEX ROWID                SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP INDEX SINGLE VALUE                     SALES_L_BJIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                          PRODUCT
           BITMAP INDEX RANGE SCAN                     SALES_P_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                          TIME
           BITMAP INDEX RANGE SCAN                     SALES_TIME_BIX
```