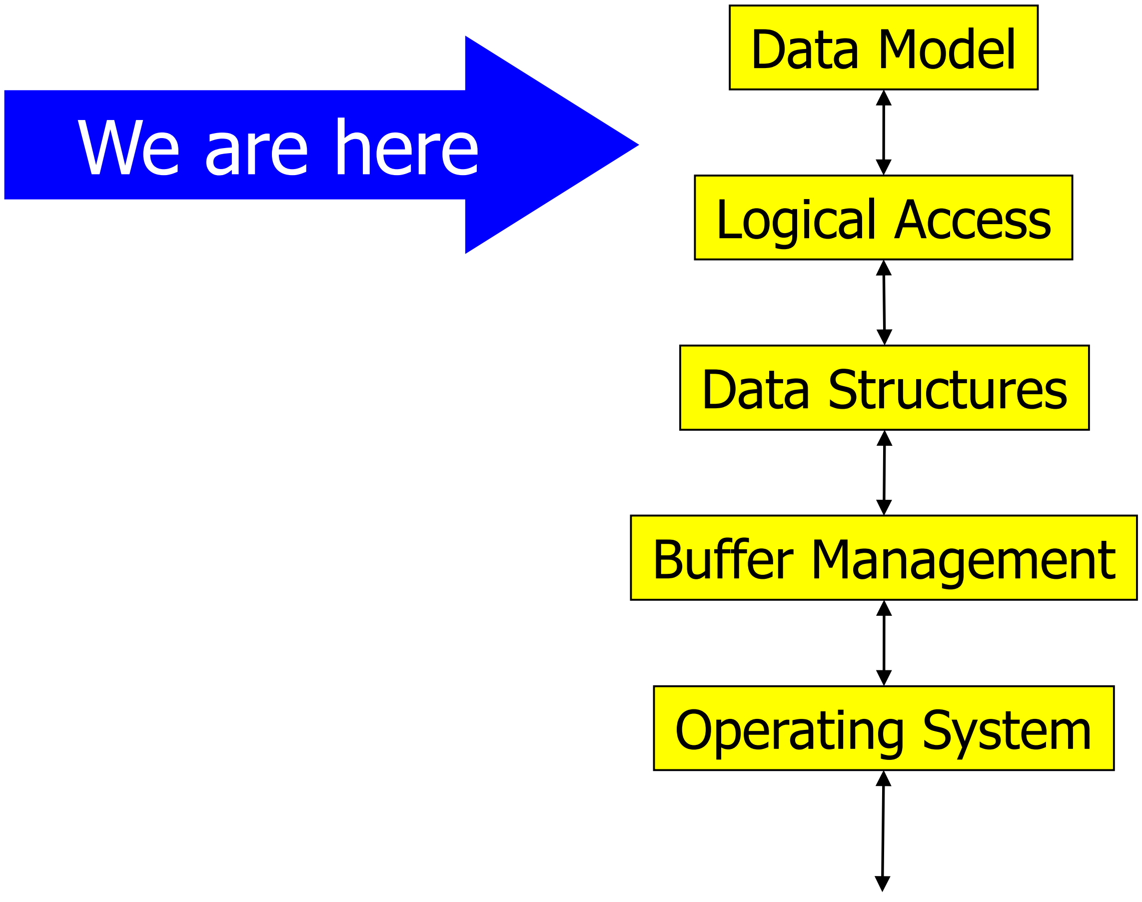
Datenbanksysteme II: Query Optimization

Ulf Leser

5 Layer Architecture



Content of this Lecture

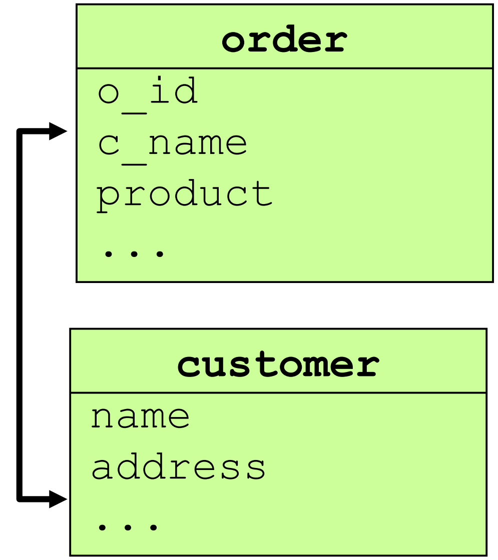
- Introduction
- Rewriting Subqueries
- Algebraic Term Rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Is Optimization Worth It?

- Goal: Find fastest way to compute a query result
  - Generate and assess different physical plans to answer the query
  - All plans must be semantically equivalent – always the same result
- Optimization itself costs time
  - Some steps have exponential complexity
    - 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 find the best plan anyway
- Why bother?

Example

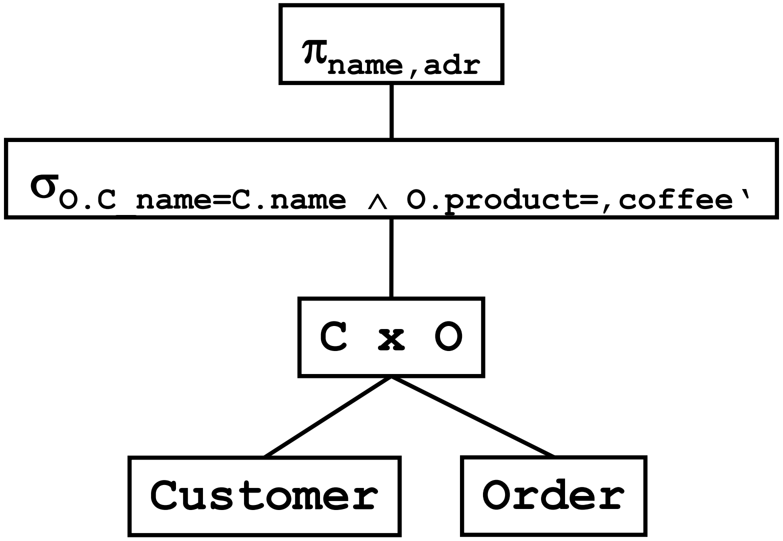
```
SELECT C.name, C.address
FROM   customer C, order O
WHERE  C.name = O.c_name AND
       O.product = „coffee"
```



- 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

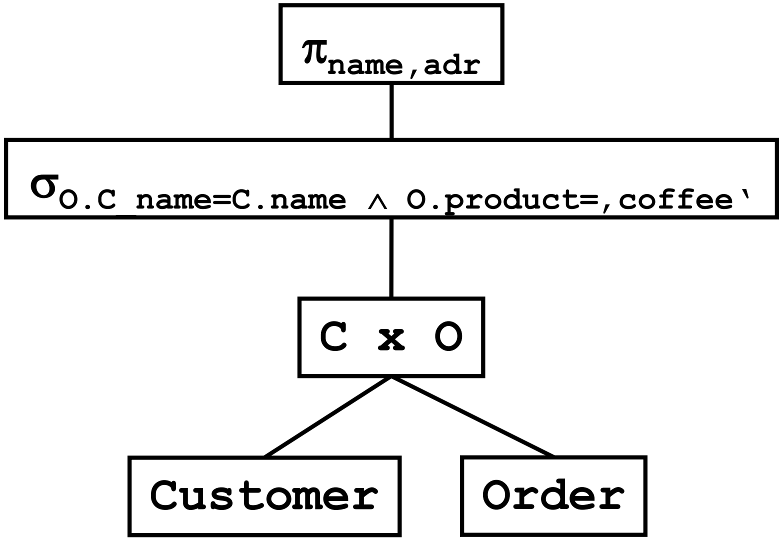
First Attempt

- Translate in relational algebra
  - $\pi_{\texttt{name,adr}}(\sigma_{\texttt{O.C\_name=C.name} \wedge \texttt{O.product='coffee'}}(\texttt{C x O}))$
- Interpret query „from inner to outer"
  - No optimization yet
- Assume materialization of intermediate results
  - No caching, no pipelining



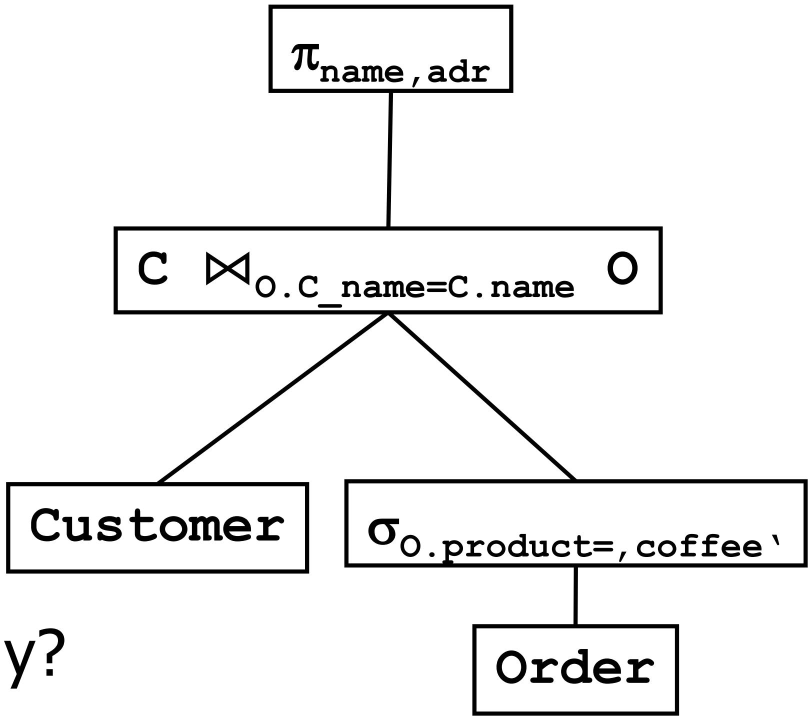
Cost

- Compute cross-product (block-nested-loop)
  - 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
  - 333.000 blocks temp space required on disk



Query Rewriting

- Rewrite into: $\pi_{name,adr}(C \bowtie_{O.C\_name=C.name}(\sigma_{O.product='coffee'}(O)))$
- 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
  - 17 blocks temp space
- 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
  - Less space needed due to projection: Assume 6 per block
  - Reads: 2 + 2*5 = 12, writes: 50/6 ~ 9
- Compute projection
  - Reads: 9, writes: 50/50 ~ 1
- Altogether: 1.064
  - 9 blocks temp space

Even Better – Use Indexes

- Assume indexes on (O.product, O.C_name) and on (C.name, C.address)
- Compute selection on O using index
  - Reads: Roughly between 5 and 10 blocks
    - 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
- Due to the index, result already sorted by c.name
- What about a SM-Join?

Even Better – Use Indexes

- …
- Compute join with sort merge
  - Read C.name in sorted order using index
  - Read O.c_name in sorted order using index
  - Reads: 20 + 5 = 25
  - Writes: 50/3 ~ 17
- Compute projection
  - Reads: 17, writes: 50/50 ~ 1
- Altogether: between 85 and 90
  (requiring 17 blocks on disk)

Comparison

	Read/Write	Temp space
Naive	686.000	333.000
Optimized, no index	1.064	9
With index	~90	17

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

Steps in Optimization

- Parsing, view expansion, subquery rewriting
- Query minimization (maybe)
- Plan optimization
  - Algebraic query rewriting (logical optimization)
  - Cost estimation (cost-based optimization)
  - Plan instantiation (physical optimization)
  - Plan enumeration and pruning
  - Note: Steps are executed in an interleaved fashion
- Selection of best plan
  - According to cost model
- Code generation (compilation or interpretation)

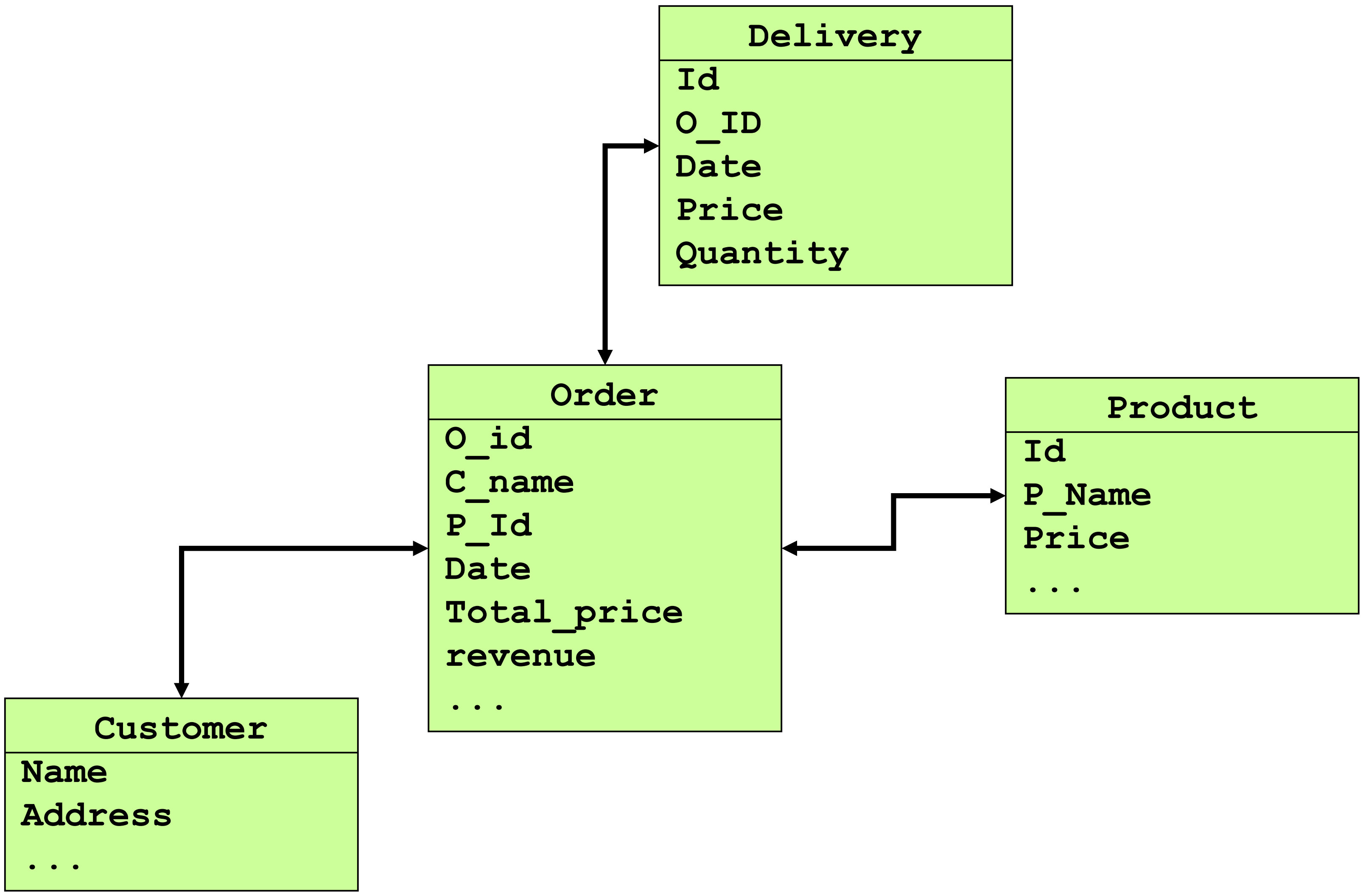
Content of this Lecture

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

Subquery Rewriting

- No equivalent in relational algebra: IN, EXISTS, ALL, …
  - Generate subtrees with non-relational root node
  - For optimization, a fully relational tree is easier to handle
  - Transformation not always possible / advantageous
- We look at four cases of IN
  - A subquery p is called correlated if it refers to a variable declared in the outer query
  - Uncorrelated without aggregation
  - Uncorrelated with aggregation
  - Correlated without aggregation
  - Correlated with aggregation
- See literature for other predicates

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 (hence order:product is 1:n), 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 expressing such functionality: User-defined table functions
  - This would allow formulation as join
  - But even harder to optimize
- Third way: Use view (two queries)
  - Will look like a join, but same optimization problem change after view expansion

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)
```

- For correlated sqs, full materialization is impossible
- 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 not 1:n, DISTINCT required)

```
SELECT 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 NOT IN (SELECT sum(price*quantity)
                             FROM   delivery d
                             WHERE  d.o_id = o.o_id)
```

- Materialization not possible (correlation)
- Rewrite into join not possible (aggregation)
- Naïve computation requires one execution of subquery for each tuple of outer query
- Solution: Rewrite into two queries
  - That are optimized in isolation

Correlated Subquery with Aggregation

```
SELECT o.o_id
FROM   order o
WHERE  o.total_price NOT IN (SELECT sum(price*quantity)
                             FROM   delivery d
                             WHERE  d.o_id = o.o_id)
```

- Query 1
  - Computes inner query result for all tuples of o
  - Can be materialized

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

- Query 2

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

Always Better?

- Be careful
- This rewriting only pays off when many OID's are required
- Counter example

```
SELECT o.o_id
FROM   order o
WHERE  o.total_price NOT IN (SELECT sum(price*quantity)
                              FROM   delivery d
                              WHERE  d.o_id = o.o_id)
       AND o.total_price> SOME_VERY_LARGE_PRICE
```

  - Materialization computes sums for many OIDs that are never used
    - And need a lot of space for the materialization
  - Nested execution probably better

Subquery rewriting Wrap-Up

- Some subqueries with IN can be rewritten in single SPJ queries, some not
  - A syntactical rewrite is always possible using views
  - This doesn't help the optimizer, but the developer
- Same holds true for other "unusual" predicates
  - Many detailed rules; see literature, such as
    - Seshadri et al. (1996). Complex query decorrelation. ICDE
    - Elhemali et al. (2007). Execution strategies for SQL subqueries. SIGMOD
- Special problems occur when subqueries appear multiple times in a single query
  - Syntax: Use "WITH" predicate
  - Optimization: Detection of repeated query fragments

Content of this Lecture

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

Query Minimization 1

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

```
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
```

- Optimization: Remove redundant condition

Query Minimization 2

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

```
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
```

- Optimization: Remove redundant joins

Techniques (sketch)

- Group conjunctive conditions with constants per attribute and compute minimal intervals (or find contradictions)
  - Different techniques for OR, XOR, NOT
- Equi-Joins: Build join graph, compute transitive closure, and find minimal spanning tree
  - Be careful with join attributes – must all be the same
  - "Minimal" already assumes a cost estimate (later)
  - Different MST's – different plans – different runtimes
- Theta-Joins: Translate into propositional logical formula and test for soundness
- …

Content of this Lecture

- Introduction
- Rewriting Subqueries
- Query Minimization
- 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*
- Optimizers generate equivalent expressions by applying provably correct rewrite rules
  - Testing if two query are equivalent is a different topic
- We look at a few such rules
  - There exist more (see literature)

Rules for Joins and Products

- Assume
  - E_1 , E_2 , E_3 are relational expressions (queries)
  - *Cond*, *Cond1*, *Cond2* are (equi-)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)

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

Projections and Selections

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

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

Projections and Selections Part 2

- Assume
  - $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ are attributes of E
  - *Cond1* und *Cond2* are 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 and $B_1 \ldots B_m$ appear in E_2

- Rule 10. Exchange of projection and union:

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

Cartesian Product and Joins

- Rule 11: Turn Cartesian Products and *cond* into join

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

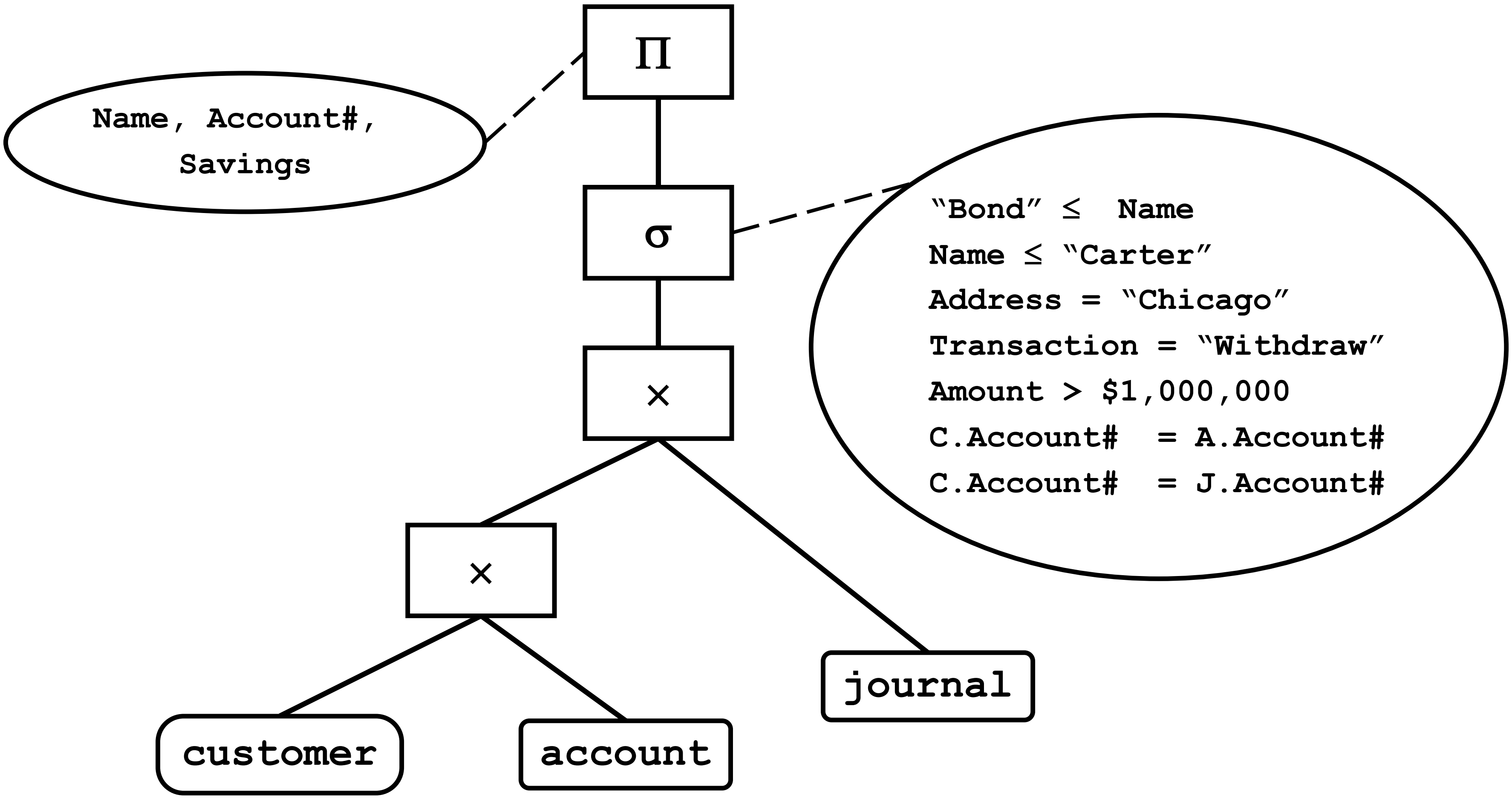
Requirement: Cond is a join condition between E_1 and E_2

Example

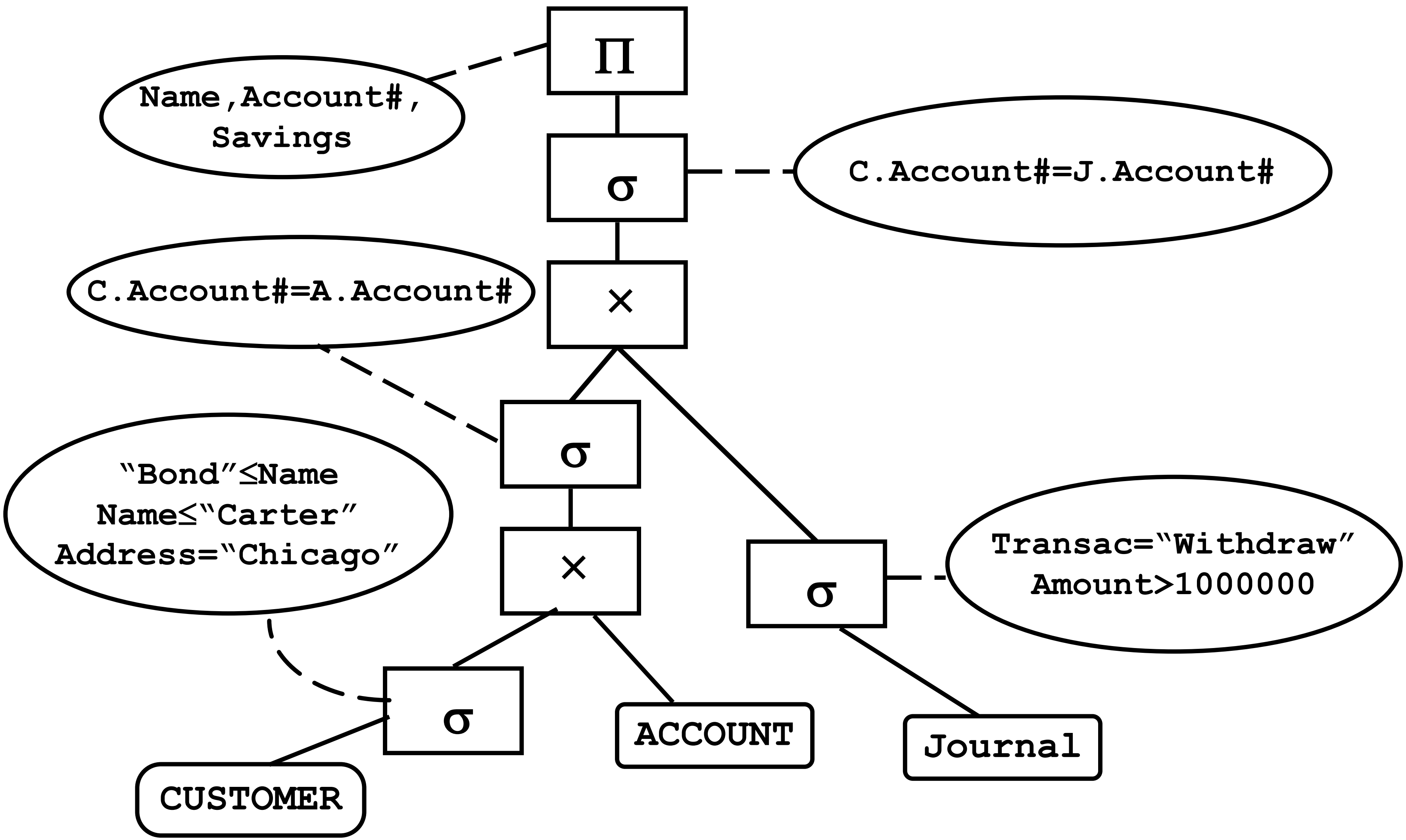
- Query on a CUSTOMER database

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

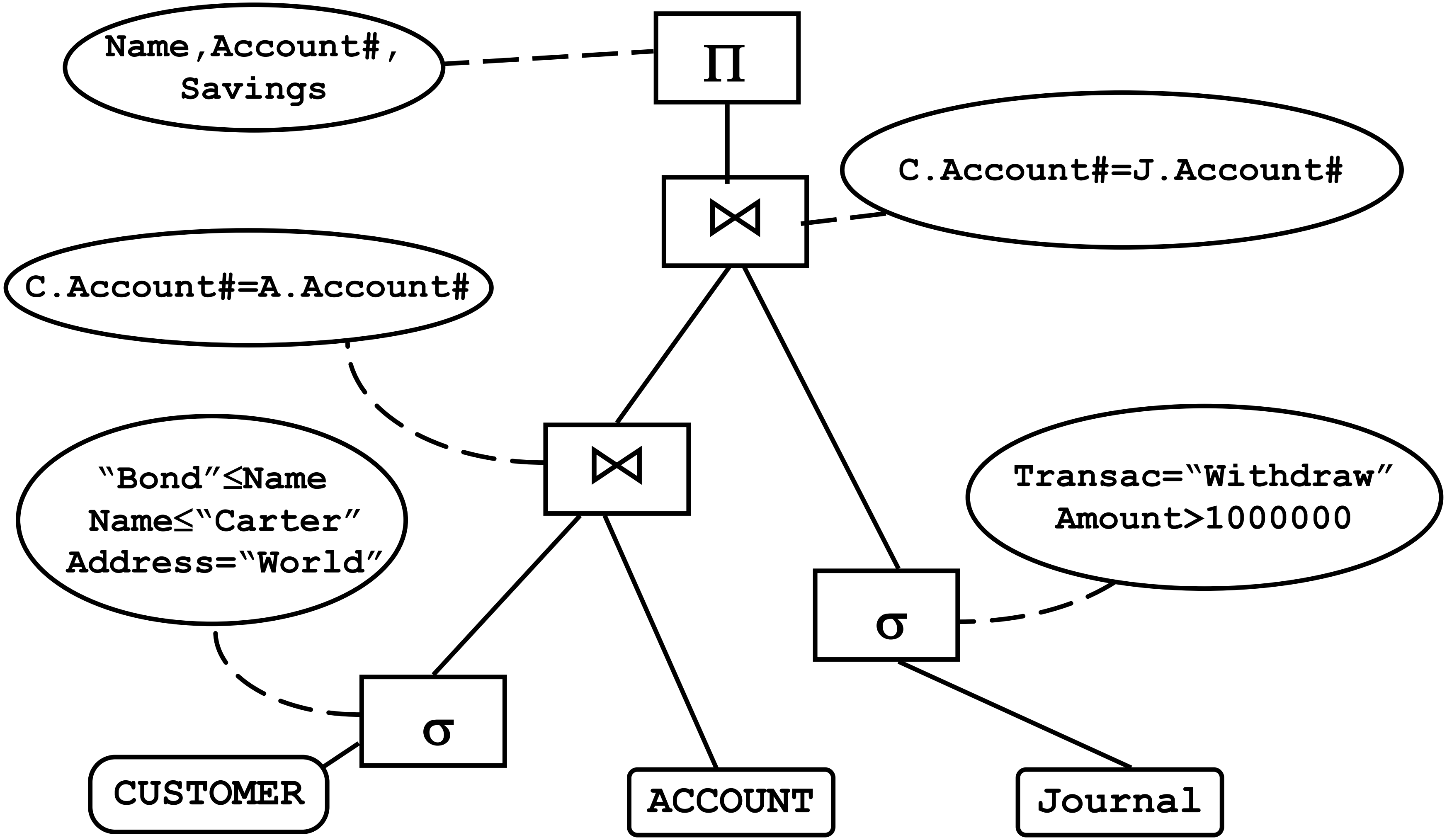
Initial Operator Tree



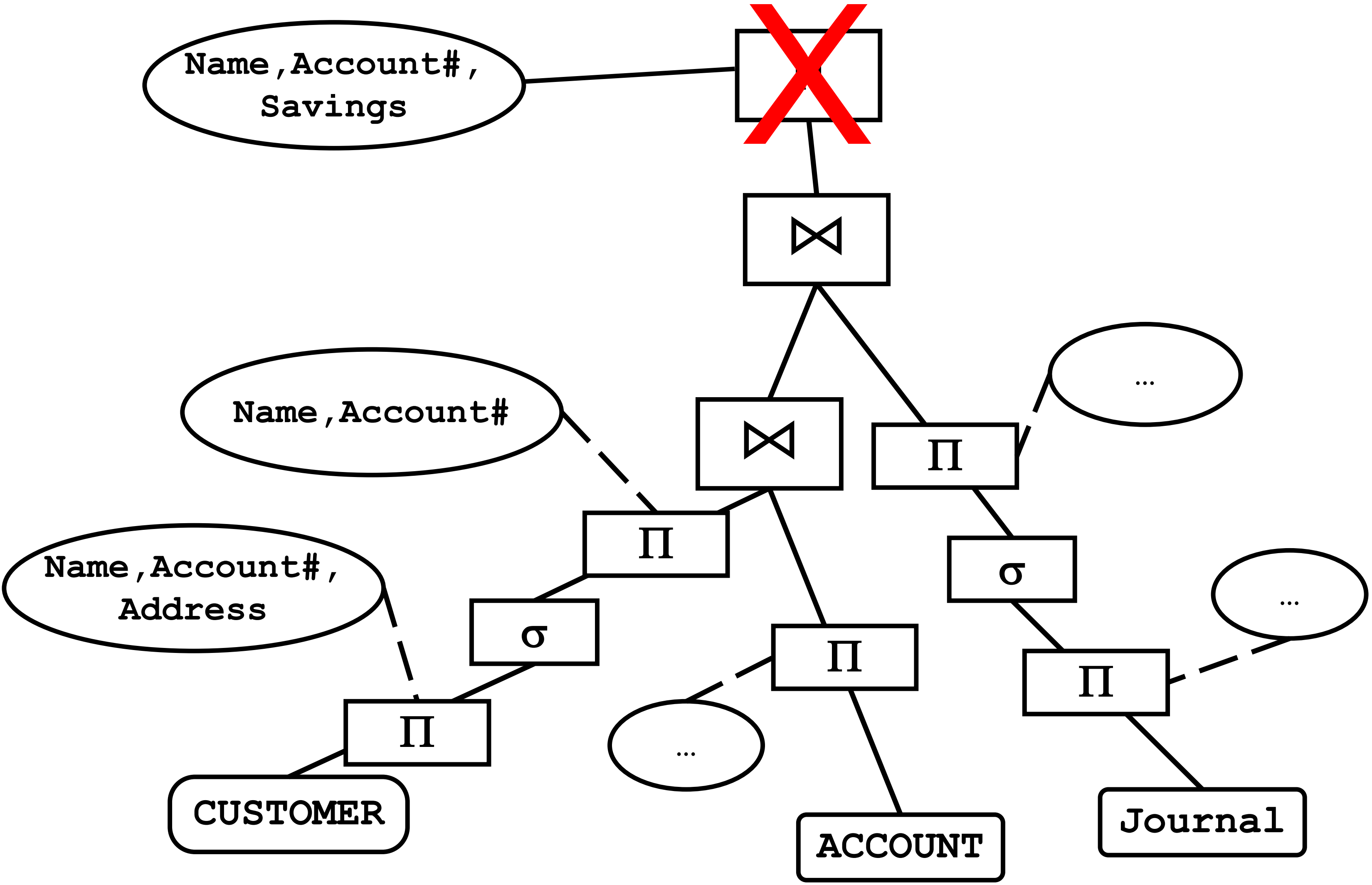
Breaking and Pushing Selections



Introduce Joins



Pushing Projections

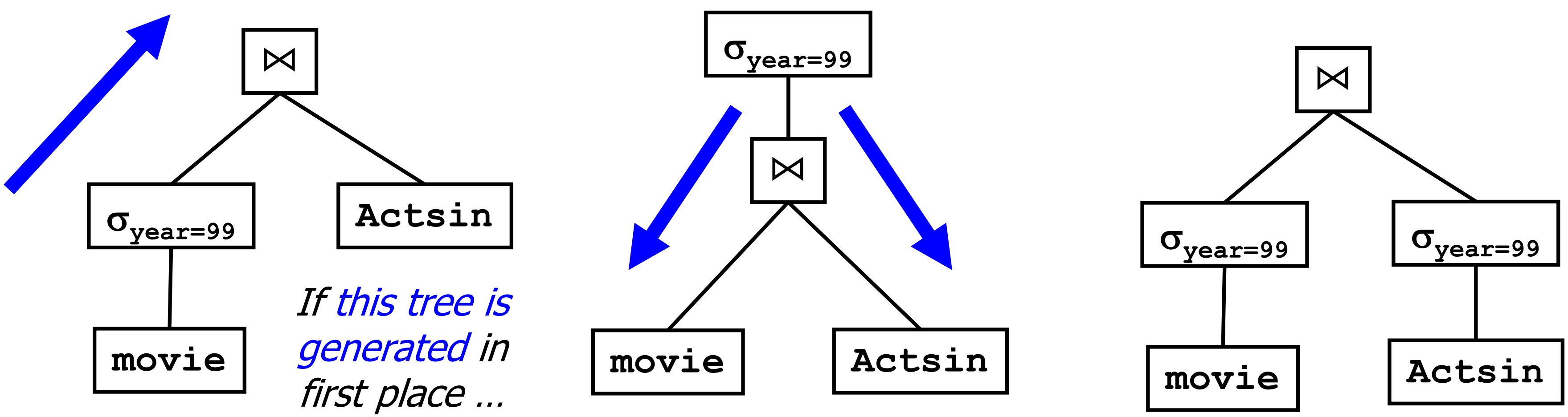


Caution

- Sometimes, pushing up selections also is beneficial
  - Especially for conditions on join attributes
- Example (assume both actsin and movie have a year attribute)

```
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
```



Content of this Lecture

- Introduction
- Rewriting Subqueries
- Query Minimization
- Algebraic Term Rewriting
  - Rule based rewriting
  - Cost based rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Term Rewriting: Algebraic Optimization

- Usually there are infinitely many rewrite steps
  - But not infinitely many different plans
  - Rewritings may go back and forth
- Give it a goal: What is a beneficial rewriting?
- General heuristic: Minimize size of intermediate results
  - Less IO if materialization is necessary
  - Less work for operations that are higher in the plan

- Option 1: Rule-based
  - Old school, simple
- Option 2: Cost-Based
  - State-of-the-art, more complex

Rule Based Query Optimization (RBO)

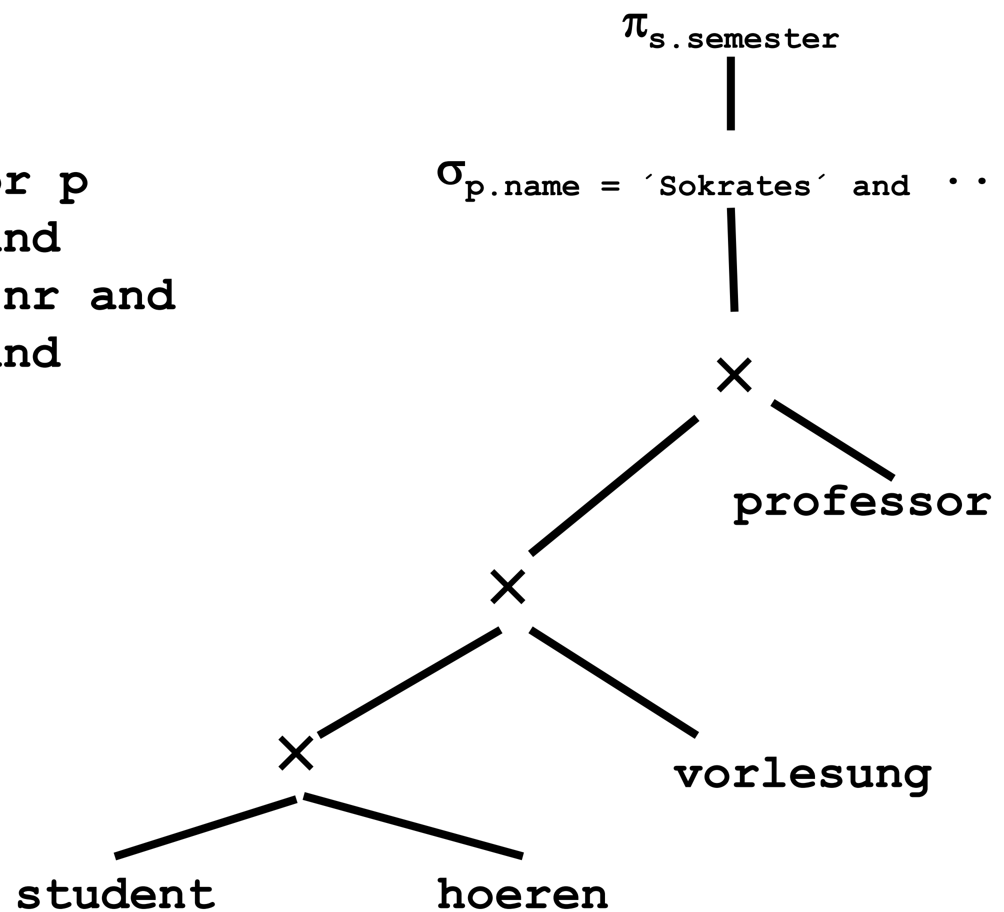
- Goal: Find a fixed order in which rewrite steps are applied such that the final plan is faster than the original plan
- Rule-based optimization
  - Rules typically disregard the concrete database instance
    - That's why RBO fails to achieve SOTA results
  - Use heuristics for prioritizing rewrite rule
  - Based on experience – rules that are beneficial in most cases
  - Simple to implement, fast optimization
  - But: Most real instances lead to non-optimal plans
    - Though hopefully still better than the original plan

A Simple Rule-Based Optimizer

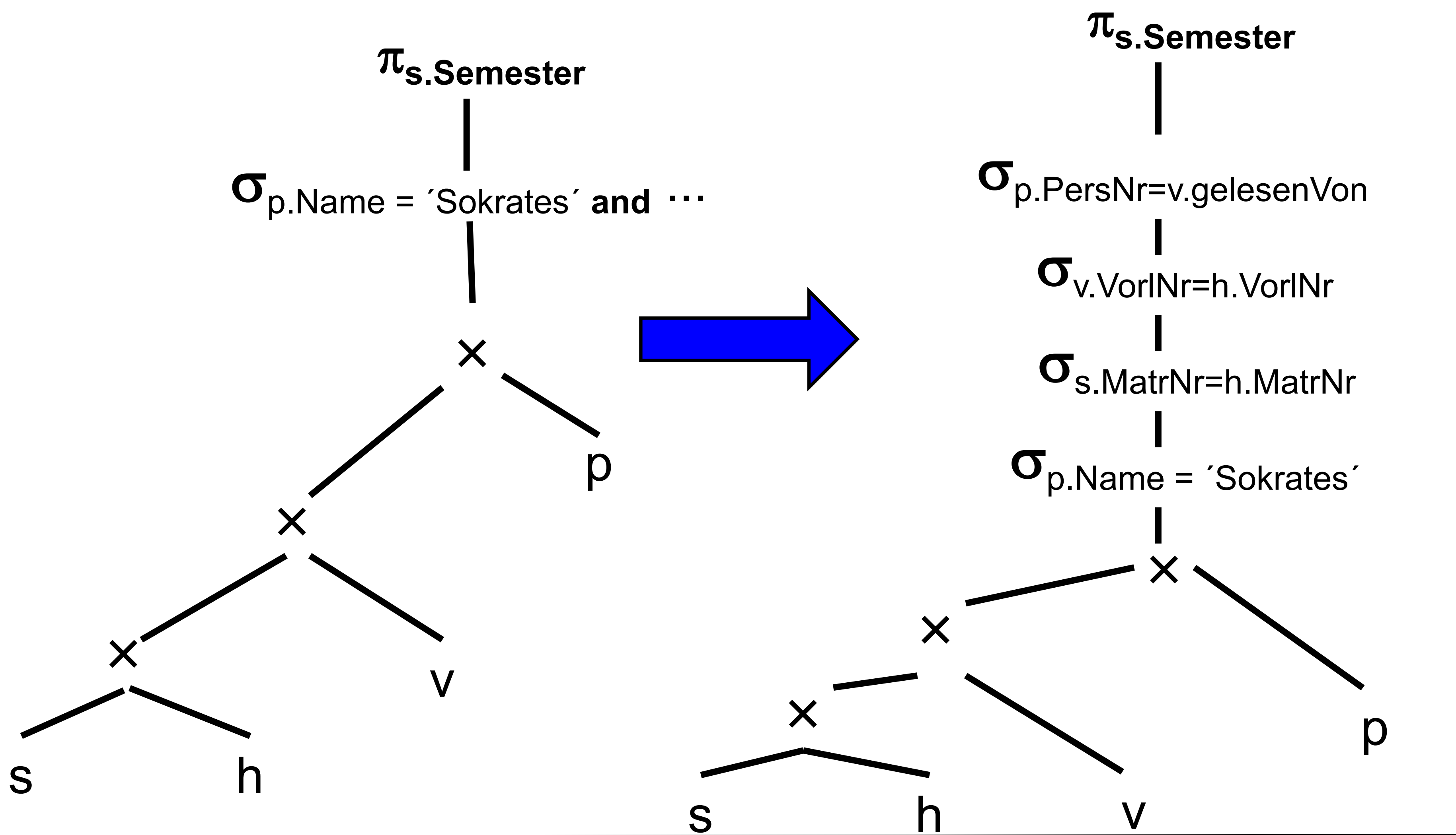
- First down: Break and push down conditions/projections
  - Break conjunctive selections into sets of atomic selections
  - Break combined projections into atomic projections
  - Push selects/projects as deep down the tree as possible
- Then up: Merge operations
  - Replace selection and Cartesian product with join
  - Merge neighboring atomic selections into combined selections
  - Merge neighboring atomic projections into combined projections
- Avoid Cartesian Products (if possible)
  - Choose other join order, start optimization again
- Finally physical: Choose concrete implementations
  - If there is a condition on an indexed attribute – use the index
  - For a join over PK-FK relationships: Use sort-merge
  - Other joins: Use hash join

Example

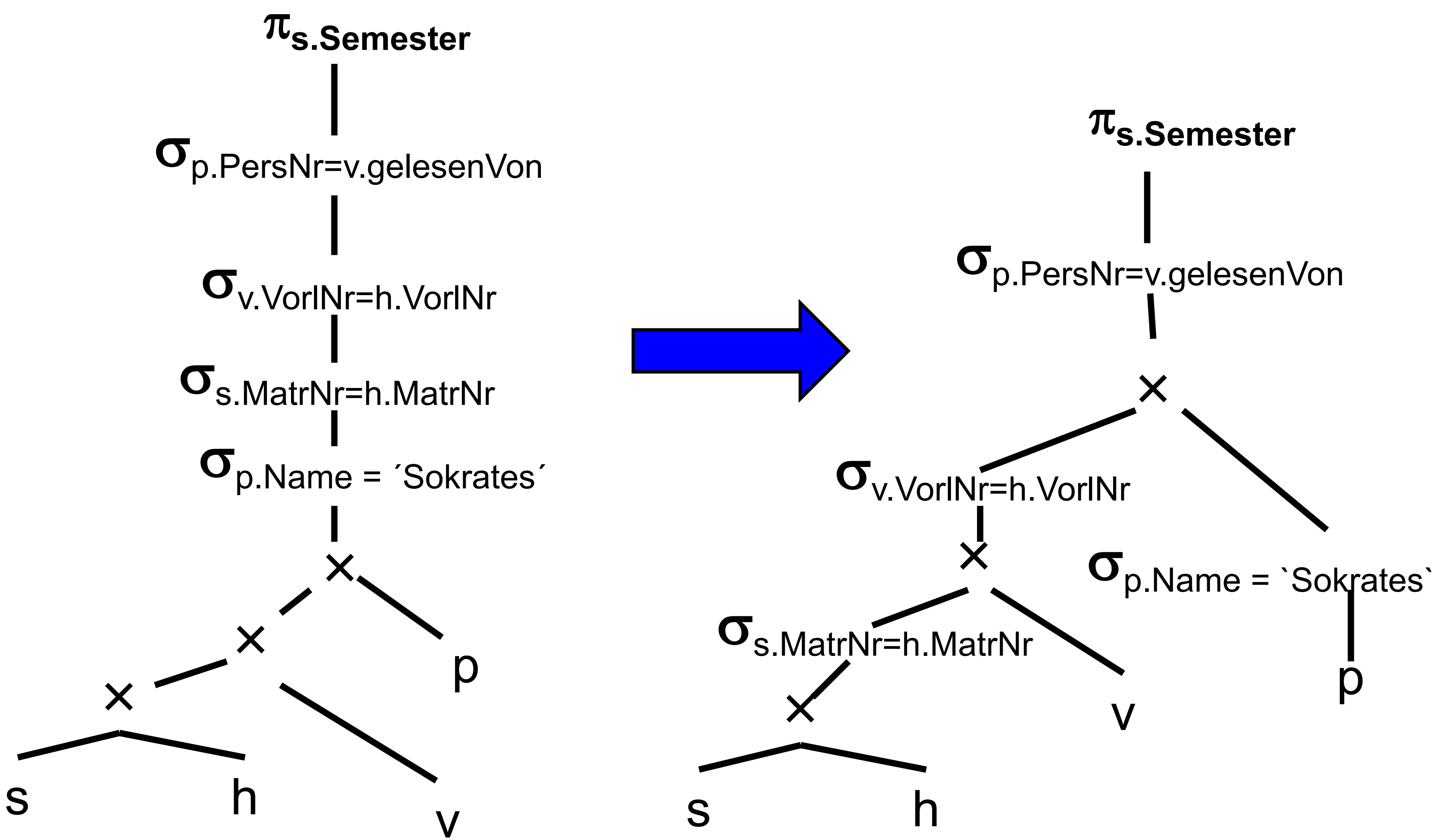
```
SELECT s.Semester
FROM   student s, hoeren h
       vorlesung v, professor p
WHERE  p.name = "Sokrates" and
       v.gelesenvon = p.persnr and
       v.vorlnr = h.vorlnr and
       h.matrnr = s.matrnr
```



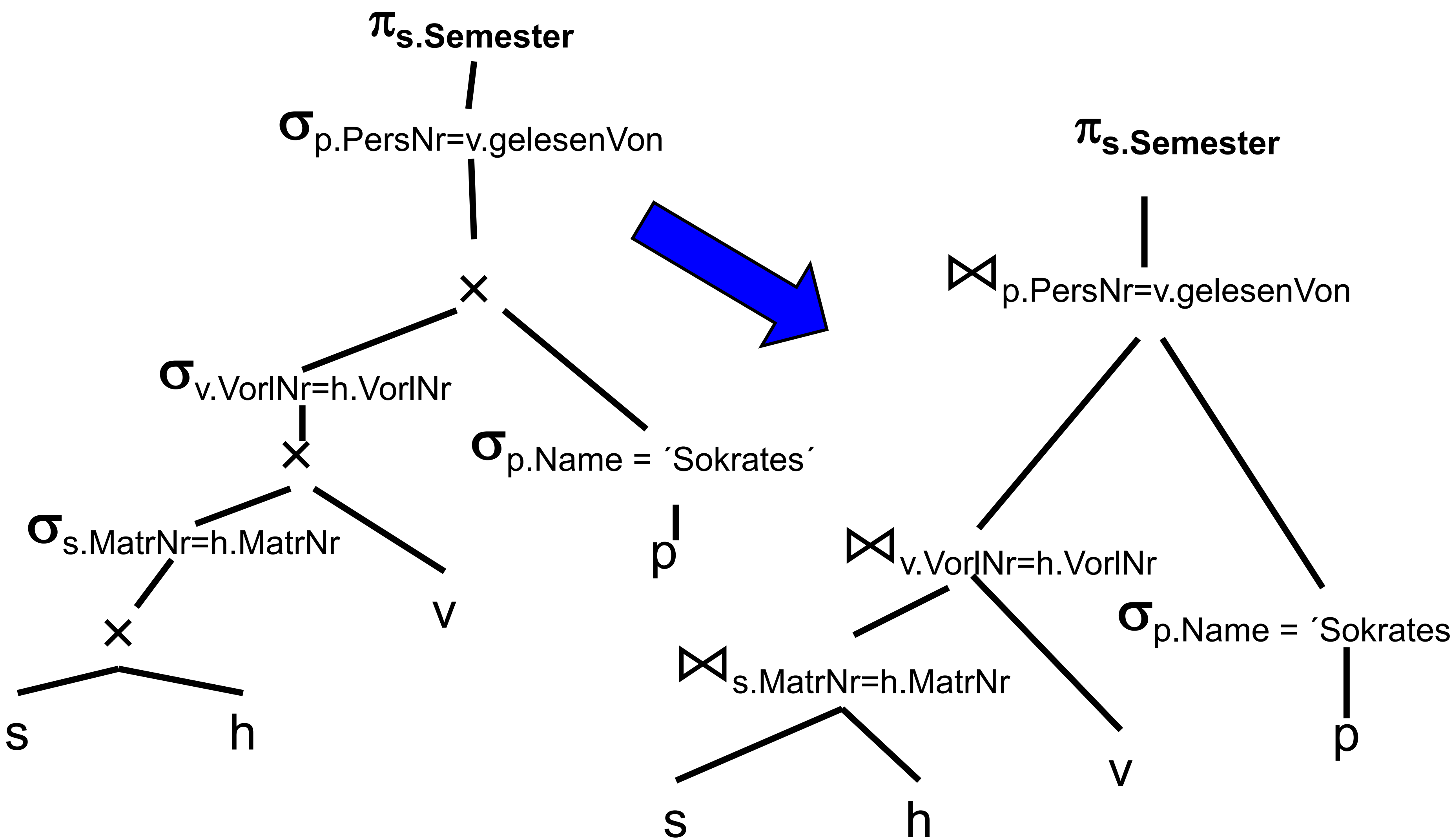
Break Up Selections



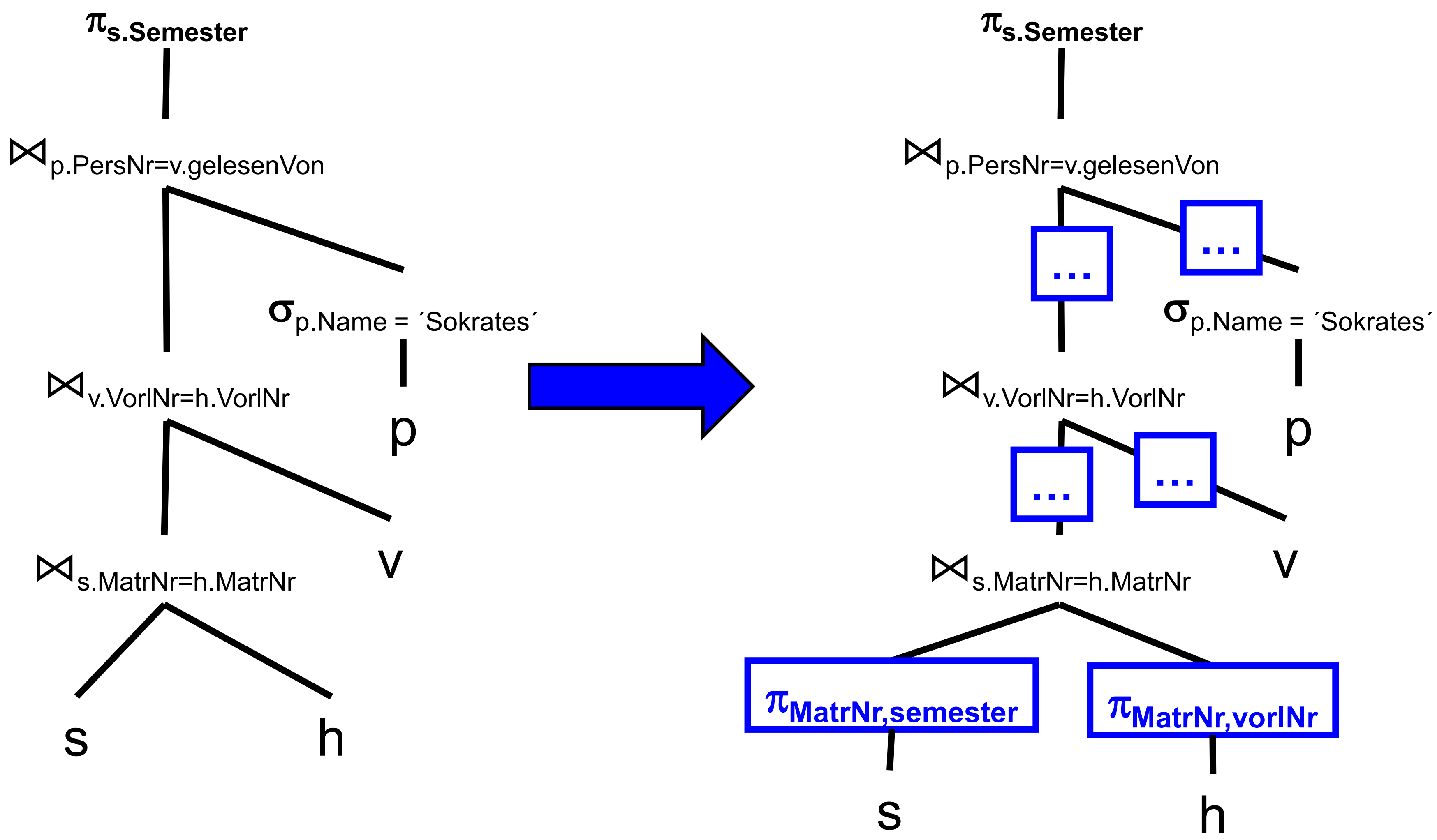
Push Selections



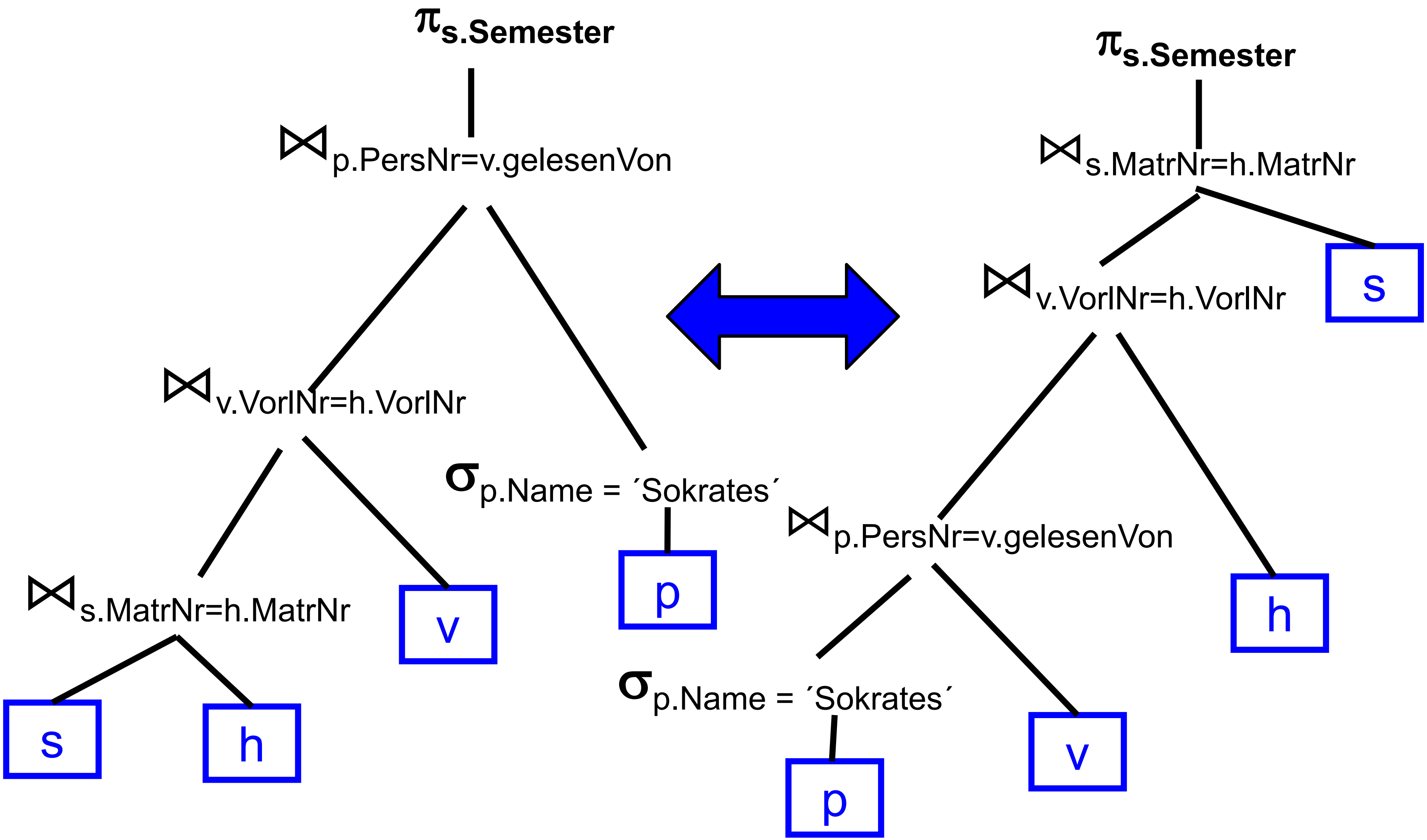
Rewrite Product+Selection into Joins



Break and Push Projections



Order of Joins: Indistinguishable



Limitations

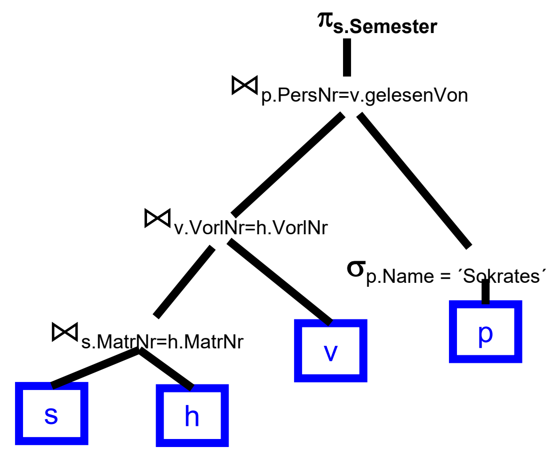
- RBO is data-independent
- Optimal selection of operators impossible without estimates about size of results (cardinality, width)
  - Best index, best join method, best join order – all depend on the concrete input and output of an operation
- No rules for order of joins
- Rules are partly contradictory
  - E.g. Conjunctive selections and composite indexes

Join Order – Does it Matter?

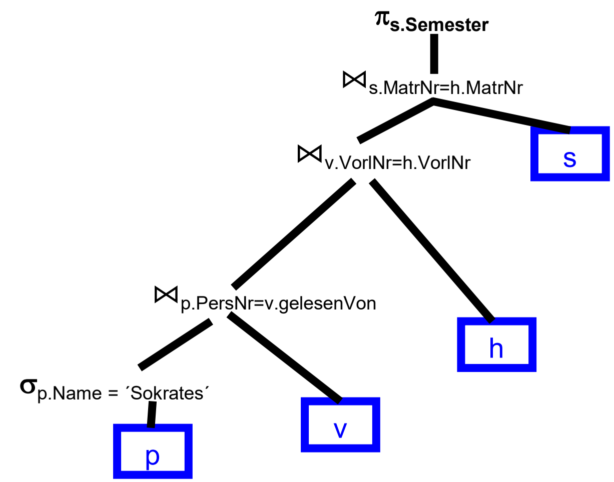
- Assume uniform distributions
  - There are 1.000 students, 20 professors, 80 courses
  - Each professor gives 4 courses
  - Each student listens to 4 courses
  - Each course is followed by 50 students (4000 “hören” tuples)

Join Order – Does it Matter?

- Compute $\sigma_{\text{Sokrates}}(P)\bowtie(V\bowtie(S\bowtie H))$
  - 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 + 200 = 8200



- Compute $S\bowtie(H\bowtie(\sigma_{\text{Sokrates}}(P)\bowtie V))$
  - Inner join selects 4 tuples
  - Next join generates 50*4= 200 tuples
  - Last join: No change
  - Intermediate result sizes: 4 + 200 + 200 = 404



Content of this Lecture

- Introduction
- Rewriting Subqueries
- Query Minimization
- Algebraic Term Rewriting
  - Rule based rewriting
  - Cost based rewriting
- Optimizing Join Order
- Plan Enumeration
- A counter-example

Cost-Based Query Optimization (CBO)

- Goal: Find the plan that is cheapest among all possible plans given a cost model
  - “Possible” – Created by a finite sequence of rewrite rules
- Cost-based optimization
  - Use a clever algorithm to enumerate possible plans
  - Estimate effect of all individual rewritings regarding a cost model
  - Use this to compute a cost per (sub-)plan
  - Prune parts of the search space wherever possible
    - When it is clear that they will not find better plans
  - Choose cheapest
- Variations in optimization goal
  - Global: Chose plan with smallest sum of intermediate results
  - Bound: Chose plan with smallest maximal intermediate result

Enumerating Query Plans

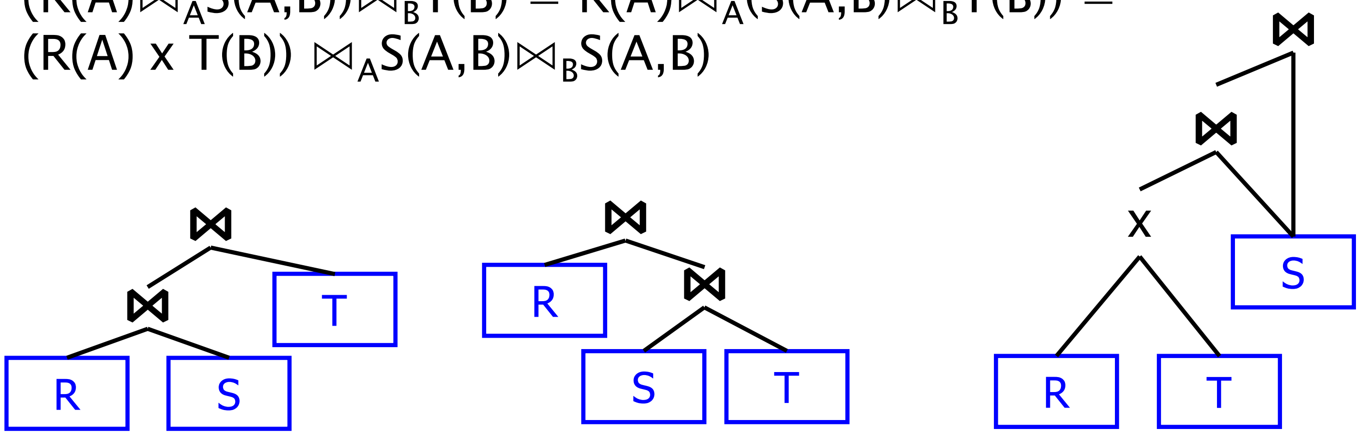
- Assume a plan P of size p=|P| with j joins
  - Size: Number of predicates in the plan (~nodes in the tree)
- Rewritings may …
  - Merge / break selections/projections (into atomic form)
    - Creates up to c different plans, when c is length of longest predicate
  - Move a selection/projection up/down the tree
    - Creates up to p different plans per predicate
  - Change order of joins (or Cartesian products)
    - Need to consider concrete join predicates
    - Creates in worst case more than j! different plans (see later)
- Typical plan enumeration strategy
  - Push predicates as deep as possible
  - Find best join order

Content of this Lecture

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

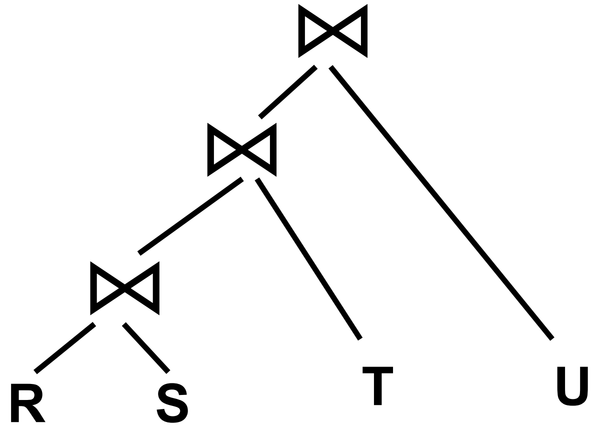
Optimizing Join Order

- Possible / reasonable join orders
  - Depending on join conditions, many orders involve intermediate cross-products
    - $(R(A) \bowtie_A S(A,B)) \bowtie_B T(B) = R(A) \bowtie_A (S(A,B) \bowtie_B T(B)) = (R(A) \times T(B)) \bowtie_A S(A,B) \bowtie_B S(A,B)$

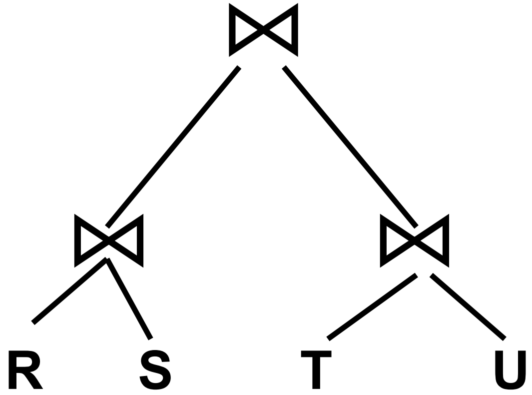


  - Most join-order algorithms disregard any plan containing a cross-product – which heavily reduces the search space
  - In the following, we assume that no order involves a Cartesian Product (e.g., all tables join on the same attribute)

Left/Right-deep versus Bushy Join Trees



Left-deep join tree



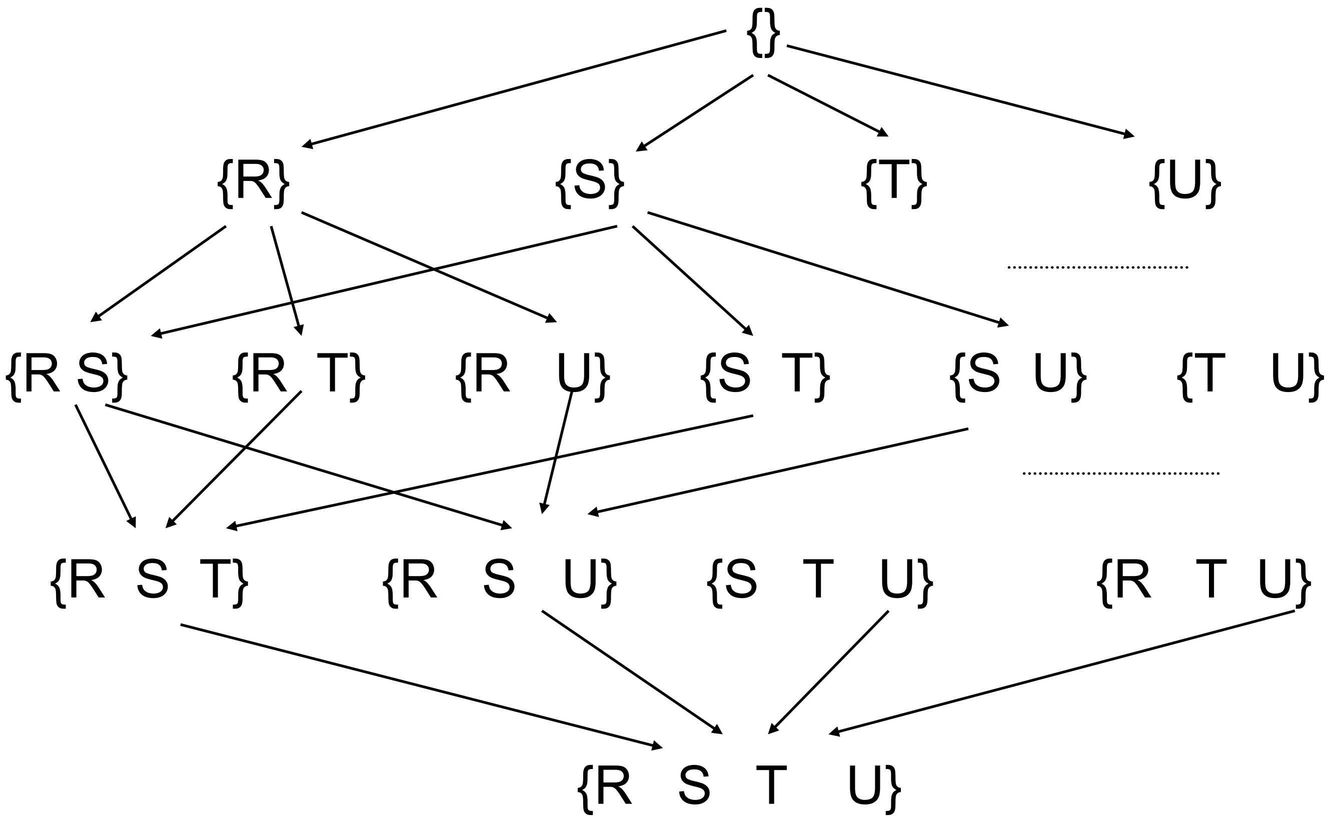
Bushy join tree

- 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
  - Used, for instance, in Oracle
  - Can be pipelined efficiently
  - Usually generates among the best plans
  - But suboptimal if parallel execution is possible
- 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 Cartesian Product
  - Still is a heuristic - later

Join Groups



- There are (n over i) join groups with i elements
- Within each join group, there are many different orderings

Details

- Create a table containing for each join group

- Prune if this would involve a Cartesian product
- Estimated size of result (how: next lecture)
  - Cost of this operator
- Minimal cost for computing the inputs to this group
  - Minimal cost of "getting there"
  - We use sum of intermediate result sizes in the subtree of this group
- Optimal plan for computing this group
  - Executable plan of "getting there" with minimal cost

Induction

- Induction over sizes of join groups
  - i=1: Consider every relation in isolation
    - Size = Size of relation
    - Cost = 0 (access costs of leaf nodes are identical for all plans)
    - Plan: Table access
  - i=2: Consider each pair of joined relations
    - Size: Estimated size of join result
    - Cost = 0 (sum of all inputs is identical - ignore)
    - Plan: Physical join method
      - E.g.: BNL with smaller relation as inner relation)
      - This method will never change again

Induction

- Induction over sizes of join groups
  - i=1: Consider every relation in isolation
    - Size = Size of relation
    - Cost = 0 (access costs of leaf nodes are identical for all plans)
    - Plan: Table access
  - i=2: Consider each pair of joined relations
    - Size: Estimated size of join result
    - Cost = 0 (sum of all inputs is identical - ignore)
    - Plan: Physical join method
      - 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
    - …

Induction

- Induction over sizes of join groups
  - …
  - i=3: Consider each pair in each triple and join with third relation
    - Loop-up minimal cost for all involved pairs (from table)
    - For each pair, add its cost and cost of joining with the third relation
    - Choose plan with lowest cost
    - …

	R,S	S,T	R,T
Size	500	1300	200
Cost	0	0	0
Plan	HJ	HJ	HJ

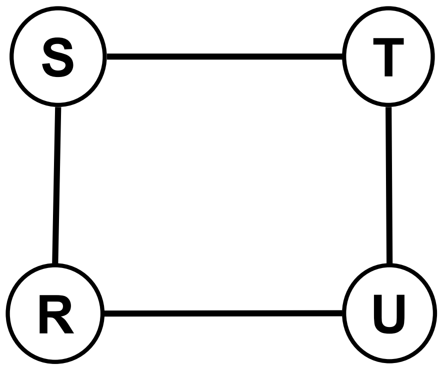
$(R \bowtie S) \bowtie T)$: 500+0

$(S \bowtie T) \bowtie R)$: 1300+0

$(R \bowtie T) \bowtie S)$: 200+0

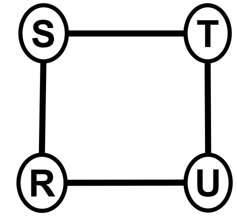
Example 1

- 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

Estimate somehow

Prune CPs

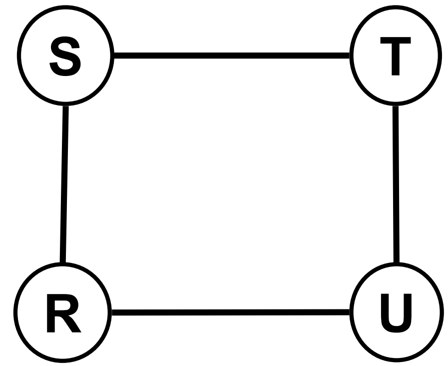
...

	{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⋈(RxT) 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⋈T)⋈R	(R⋈S)⋈U	(T⋈U)⋈R	(T⋈U)⋈S



Plan	Kosten
((S⋈T)⋈R)⋈U	12k
((R⋈S)⋈U)⋈T	55k
((T⋈U)⋈R)⋈S	11k
((T⋈U)⋈S)⋈R	3k

Best 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 ∪ X = O***
      $optPlan(S) = optPlan(S) \cup joinPlans(optPlan(O),$ ***X)***
      $prunePlans(optPlan(S))$
    }
  }
}
return optPlan($\{R_1, \ldots, R_n\}$)

Enumerate physical plans for accessing R_i

Prune all except one

Prune all except one

Dynamic Programming

- DP is a heuristic for join order optimization
- Issue 1: Main DP assumption broken
  - Assumption: Any subplan of an optimal plan is optimal
  - Not true: Optimal plan might involve Cartesian Products
    - Example later
- Issue 2: Inaccuracies of the cost model
  - Optimizers can only work as good as their inputs – cardinality estimates
  - These often are not very accurate (next lecture)

Dynamic Programming

- DP is a heuristic for join order optimization
- Issue 3: Effect of sorting on choice of join methods
  - Decisions on join method are taken early and are never revised
  - But it might pay off to perform a more costly sort-merge-join early because the order can also be exploited in all future joins
  - Requires choice of a suboptimal plan for small join groups
  - Solution: Keep different "optimal" plans for each join group
  - System R: One 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
- Query Minimization
- 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)

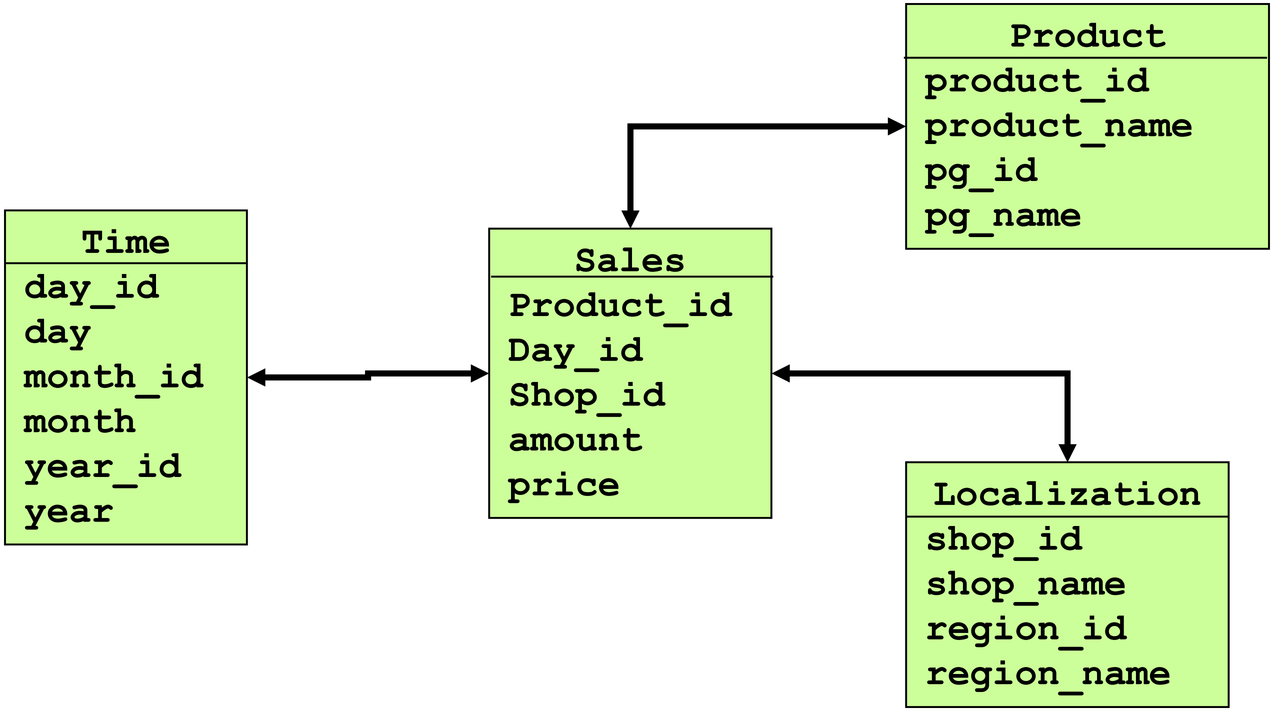
Search Strategies

- 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
- Query Minimization
- 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

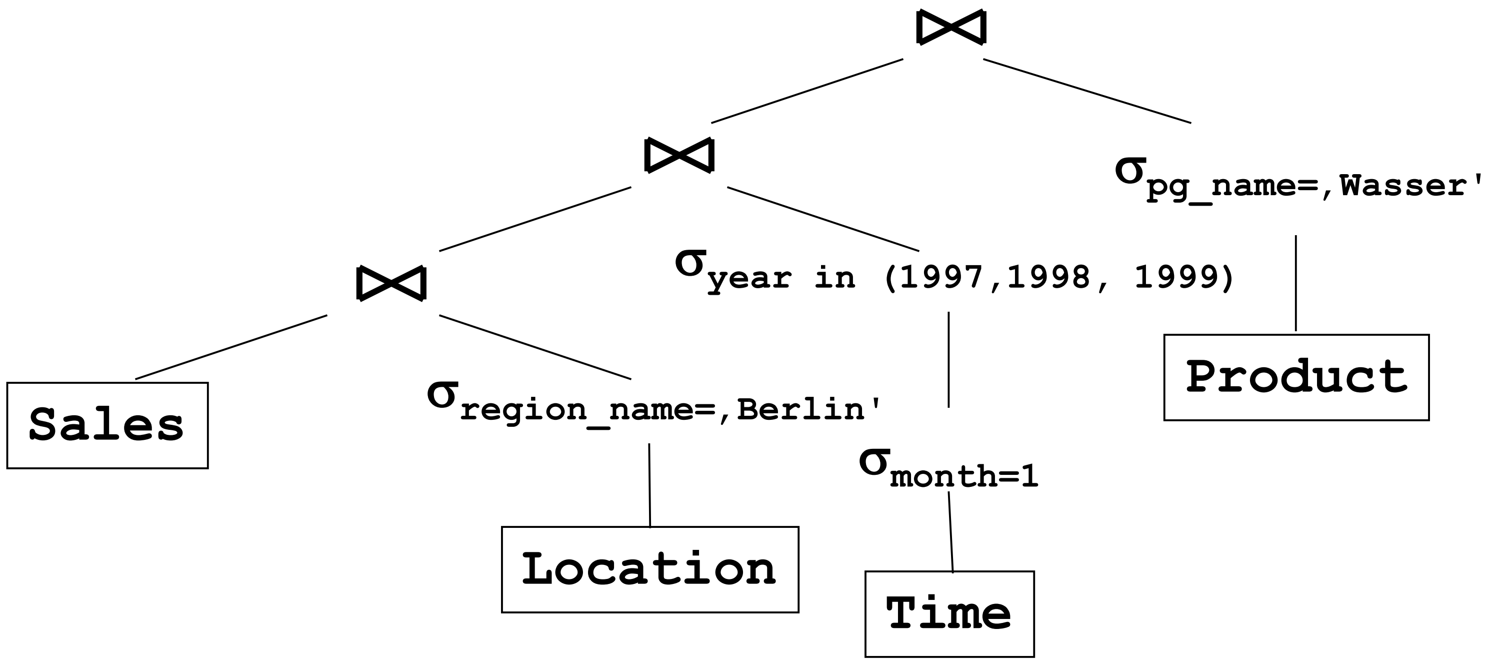
Beispielquery

- 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
```

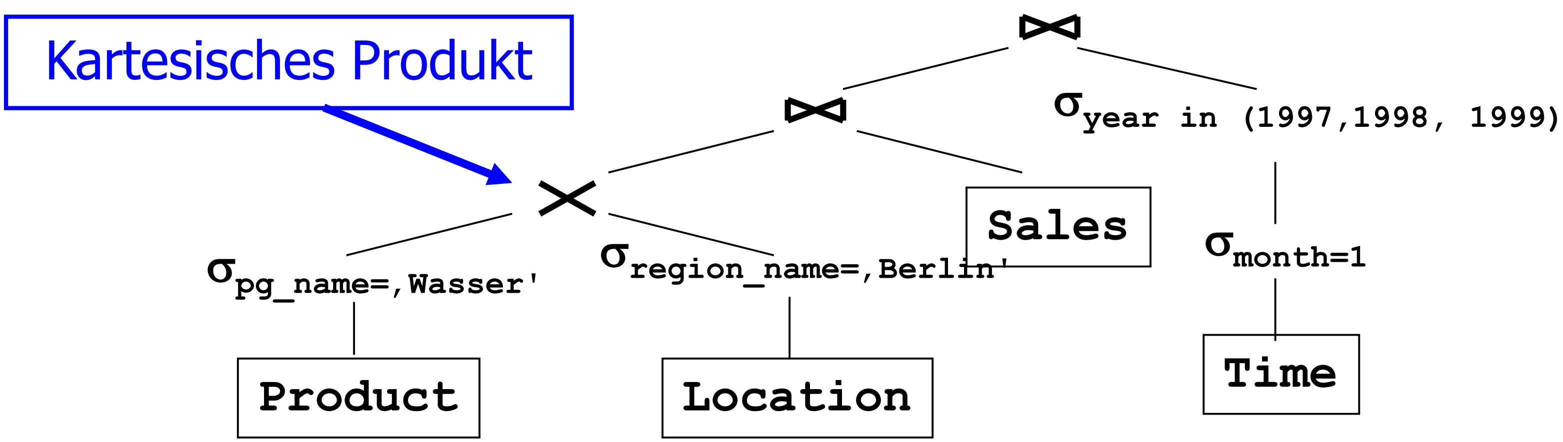
Anfrageplanung

- Anfrage enthält 3 Joins über 4 Tabellen
- Zunächst 4! left-deep join trees
  - Aber: Nicht alle Tabellen sind mit allen gejoined
- Star-Join: 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
```

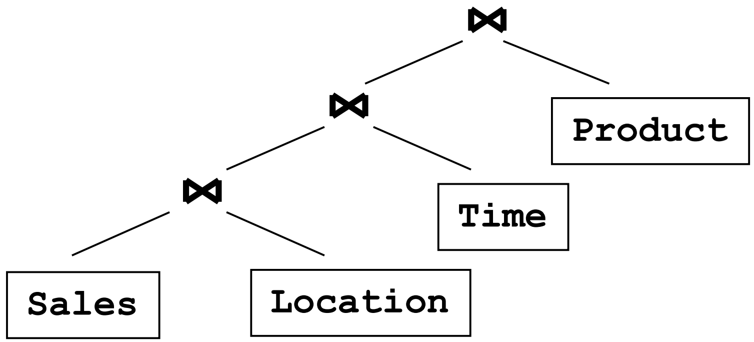
Annahmen

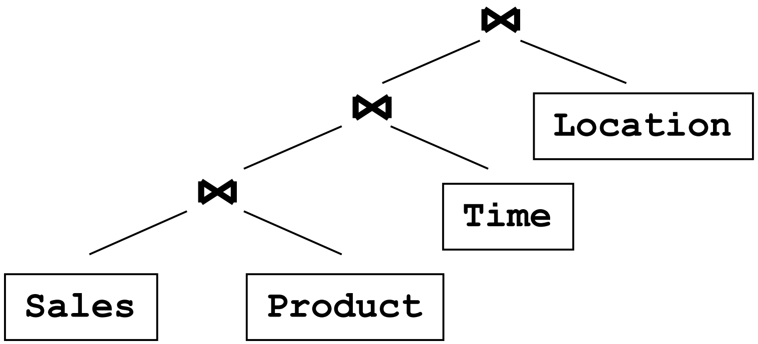
- M= |S| = 100.000.000
- 20 Verkaufstage pro Monat
- Daten von 10 Jahren
- 50 Produktgruppen a 20 Produkten
- 15 Regionen a 100 Shops
- Gleichverteilung aller Verkäufe

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

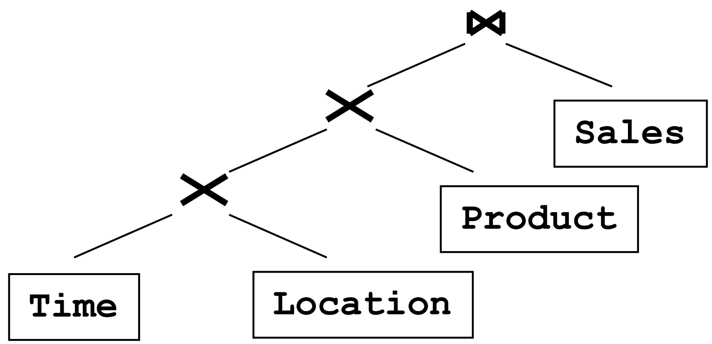




	Zwischen-ergebnis
1. Join (M / 15)	6.666.666
2. Join ($\|J_1\|$*3/120)	166.666
3. Join ($\|J_2\|$/50)	3.333

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

- Wie optimiert man Star-Joins?
- Siehe Modul „Data Warehousing“