SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (3/6)

- operator models
  - push-model
  - pull-model
- operator implementations
  - general idea
  - join algorithms for relational and multidimensional data
  - other operators
- query processing
  - scanning & “naive plans“
  - canonical plan computation

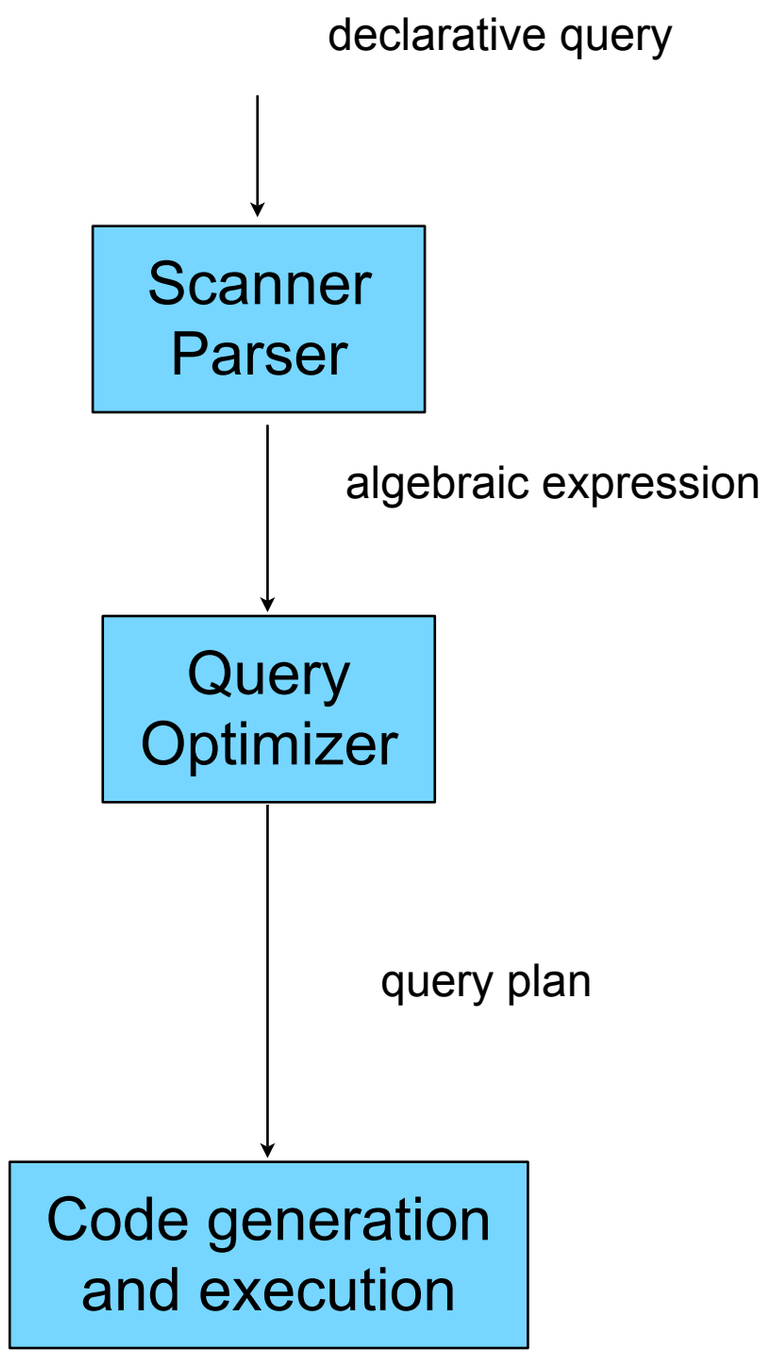
Topics (4/6)

- query optimization
  - query rewrite
  - cost-based
- data recovery
  - single versus multiple instance
  - ARIES
- parallelization of data and queries
  - horizontal partitioning
  - vertical partitioning
  - replication
  - map-reduce
  - multi-cores

Query Optimization.

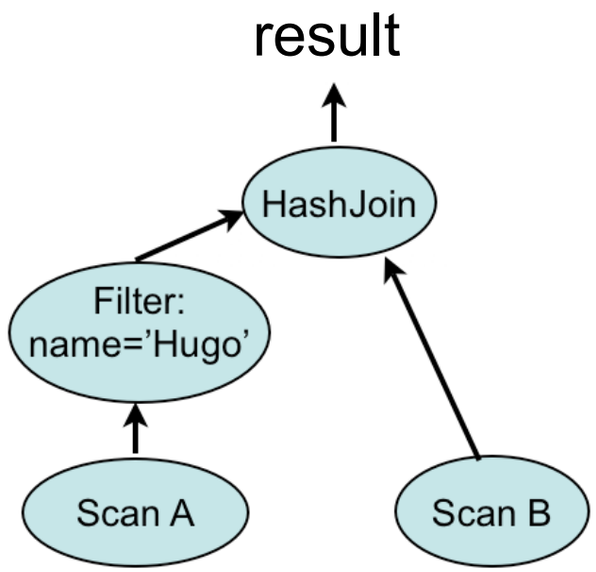
Introduction.

Motivation & Overview



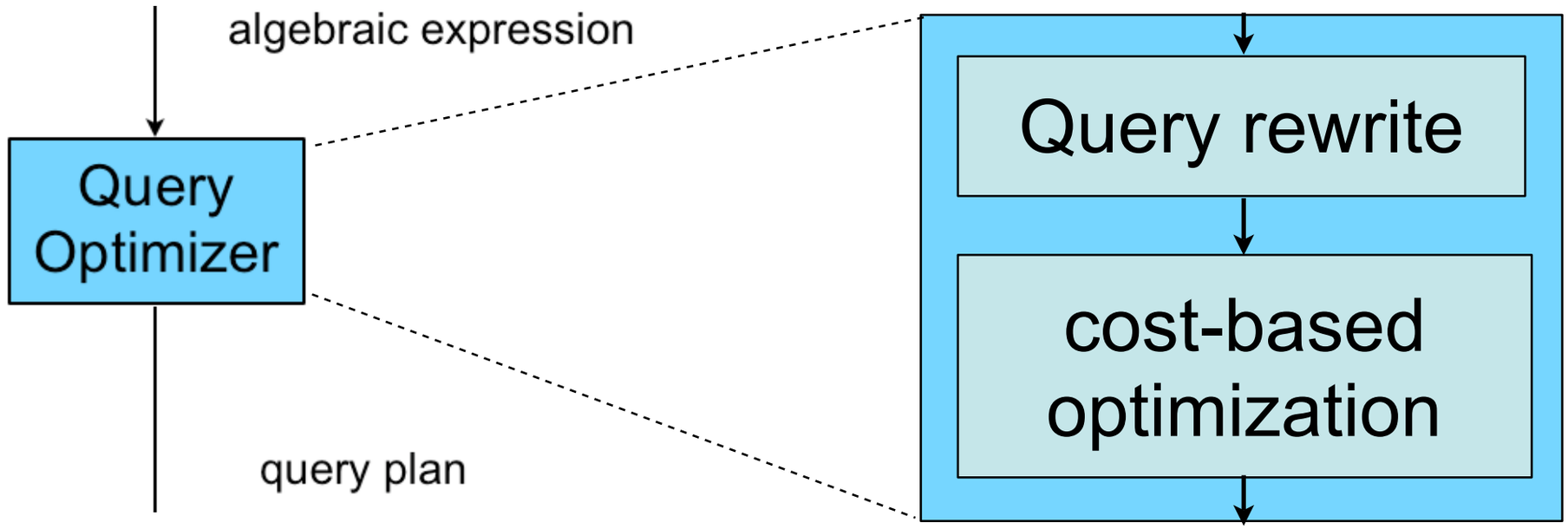

```
SELECT title
FROM A,C
WHERE A.name ='Hugo' AND A.id = C.dz
```

$$\prod_{\texttt{title}} \left(\sigma_{\texttt{A.name='Hugo' and A.id=B.dz}} \left(\texttt{A} \times \texttt{B} \right) \right)$$



Query Optimizer

- Query optimization consists of two phases



- Query rewrite
  - rewrites query using rules
- Cost-based optimization
  - estimates cost of different plans using cost models
  - picks plan that has expected lowest costs

Query Optimization.

Query Rewrite.

Query Rewrite

- Goals
  - transform queries based on a rule set
  - perform optimizations that are (almost) guaranteed to improve the plan (independent of size and distribution of data)
  - prepare query in a way such that the cost-based optimizer has many options for generating plans
- Examples
  - simplification of expressions
  - generation of join predicates
  - unnesting of views or subqueries
  - predicate move around

Query Rewrite in Practice

- exact rules are not published by the DBMS vendors (some example rules, however, available for IBM DB2 in Piharesh et.al, SIGMOD 1992)
- rules are extended from time to time (e.g., Rao et.al. SIGMOD 2004)
- rules may be applied in any order (however, order may play a role...)
- exact strategies of rule engines not published
- but: some core rules are published

Scanner/Parser

- Input: query in SQL

```
SELECT A1,...,An
FROM R1,...,Rk
WHERE P;
```

- the scanner/parser translates this into:

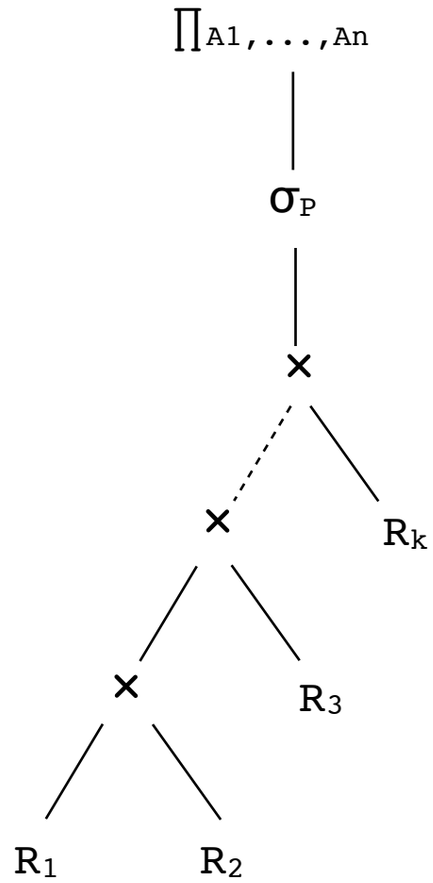
$$\prod_{A1,\ldots,An}(\sigma_{P}\ (R_1 \times \ldots \times R_k))$$

project filter input

Canonical Form of a Query

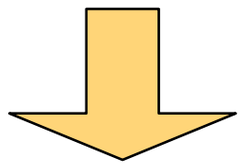
- Input: $\prod_{A1,...,An}(\sigma_P\ (R_1 \times ... \times R_k))$

- this corresponds to the following query tree:

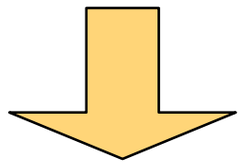


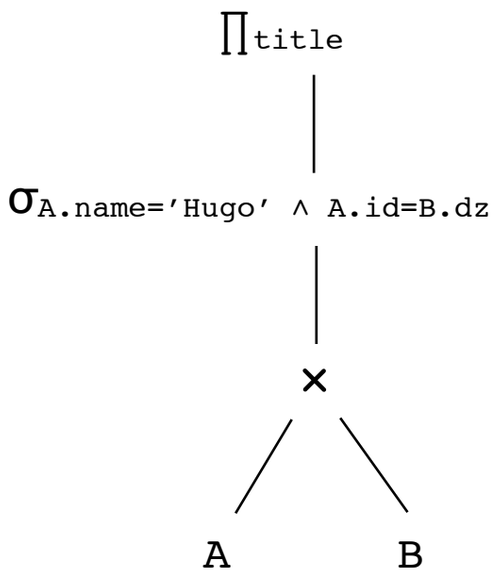
Example

```
SELECT title
FROM A,B
WHERE A.name ='Hugo' AND A.id = B.dz;
```



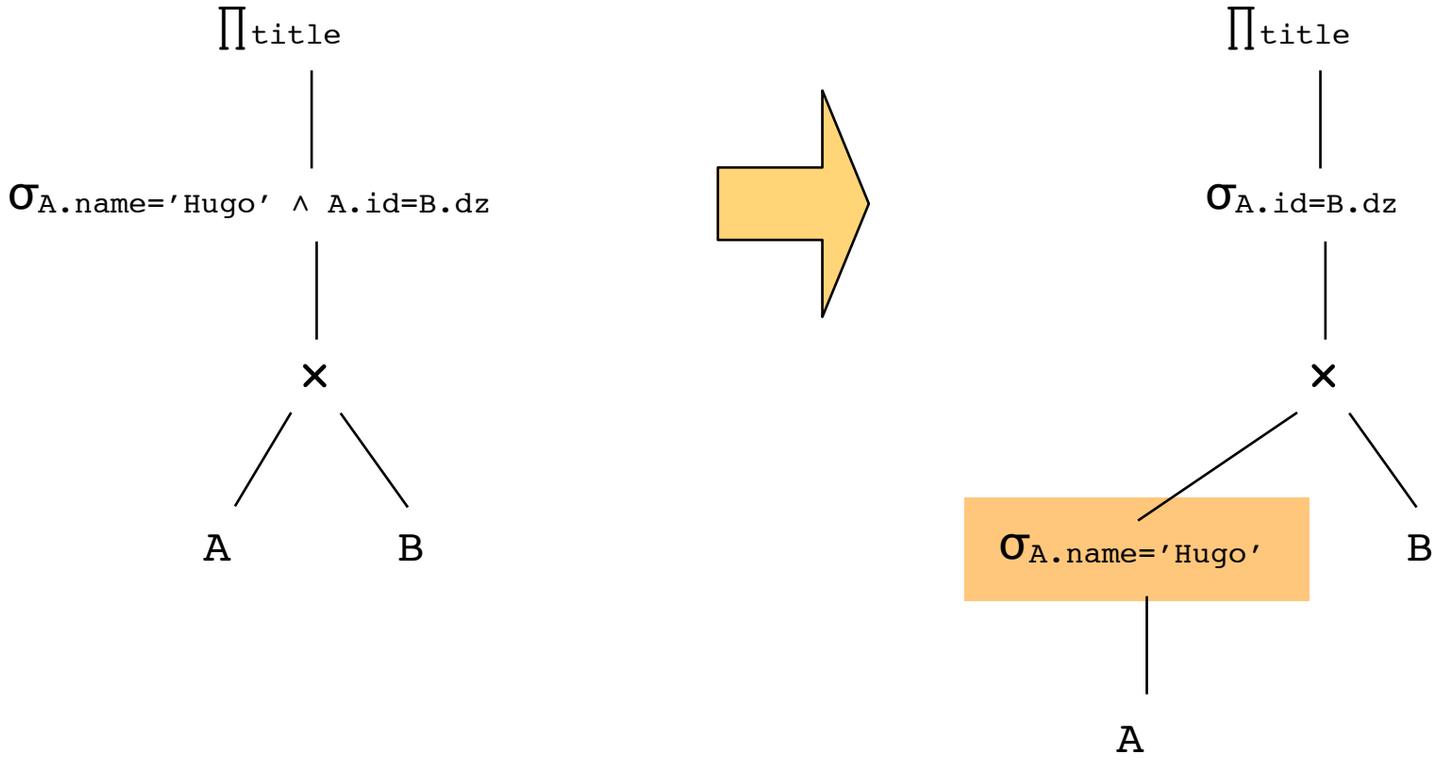
$$\prod_{\text{title}}(\sigma_{\text{A.name='Hugo'} \wedge \text{A.id=B.dz}}\ (A \times B))$$





Example: Rule Application (1/3)

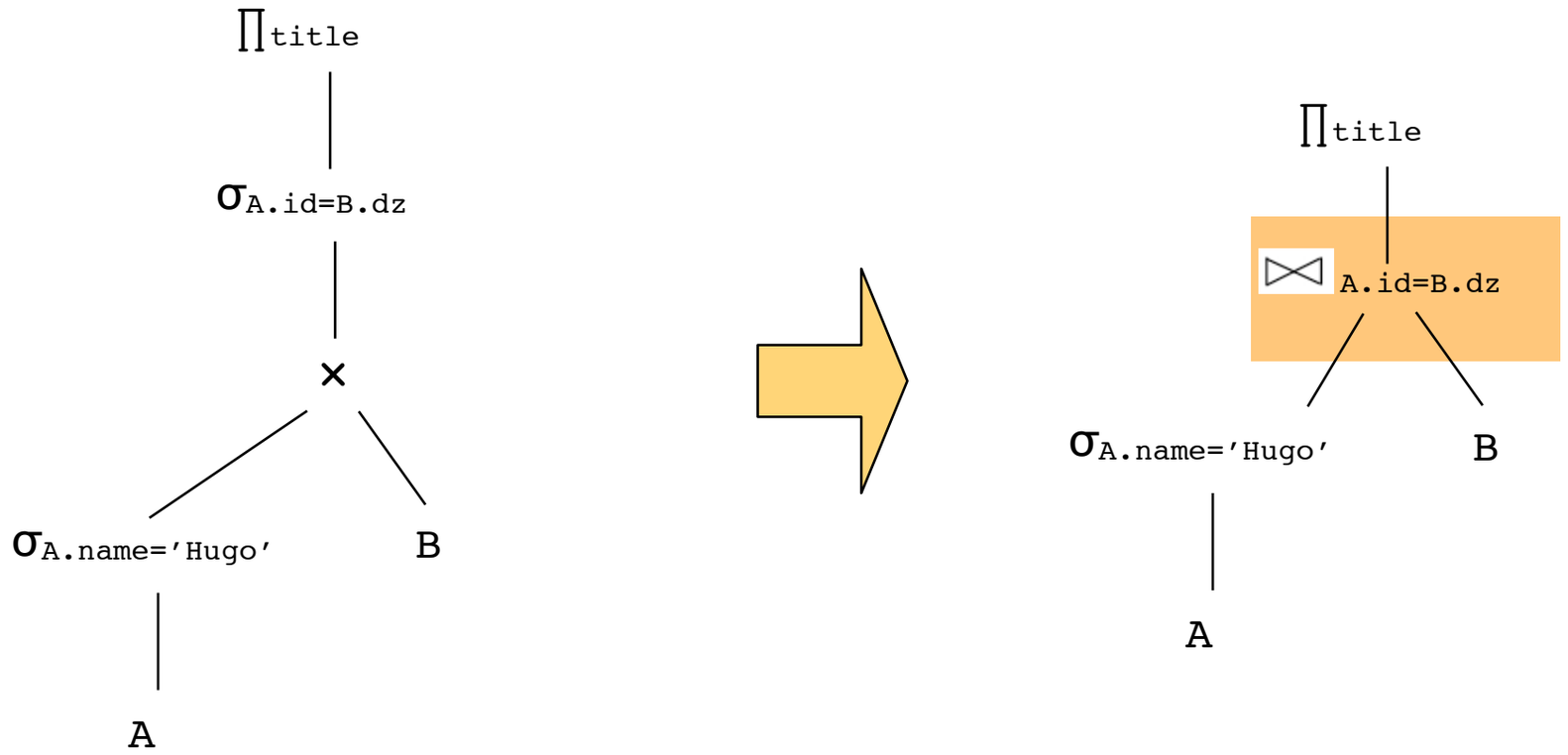
- now, we may apply rules to the tree
- Rule 7: push-down selection predicates



- effect: less tuples participate in the cross product

Example: Rule Application (2/3)

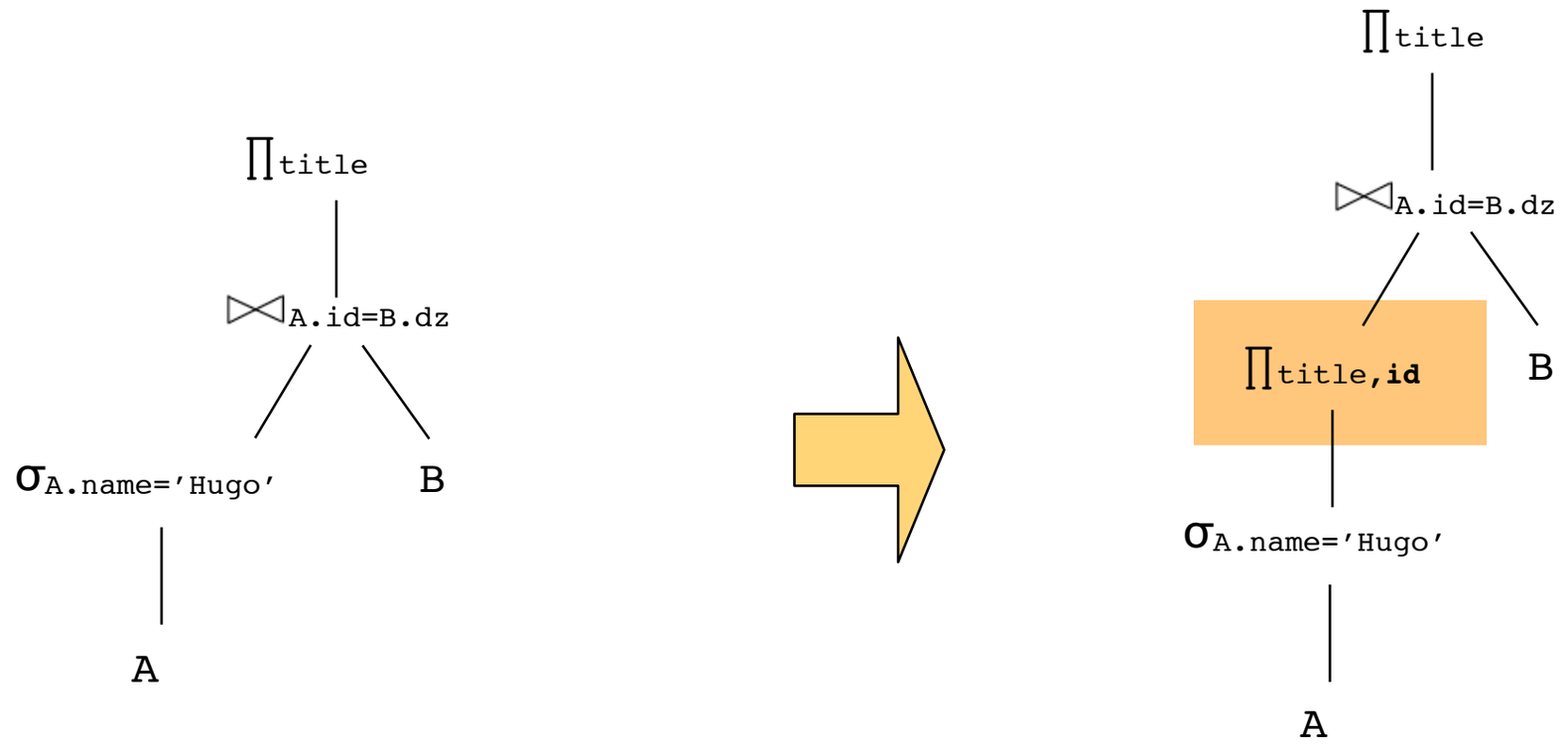
- Rule 11: combine selections and cross products into a join (if possible)



- effect: eliminates cross product ($\Theta(n^2)$!!)

Example: Rule Application (3/3)

- Rule 8: push-down projections through joins



- effect: input to join becomes smaller (row less wide)

Equivalences of the Relational Algebra

1. join, union and intersection are commutative:

$$\begin{aligned} R_1 \bowtie R_2 &= R_2 \bowtie R_1 \\ R_1 \cup R_2 &= R_2 \cup R_1 \\ R_1 \cap R_2 &= R_2 \cap R_1 \\ R_1 \times R_2 &= R_2 \times R_1 \end{aligned}$$

2. selections may be swapped with one another:

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

3. join, union, intersection and cross product are associative:

$$\begin{aligned} R_1 \bowtie (R_2 \bowtie R_3) &= (R_1 \bowtie R_2) \bowtie R_3 \\ R_1 \cup (R_2 \cup R_3) &= (R_1 \cup R_2) \cup R_3 \\ R_1 \cap (R_2 \cap R_3) &= (R_1 \cap R_2) \cap R_3 \\ R_1 \times (R_2 \times R_3) &= (R_1 \times R_2) \times R_3 \end{aligned}$$

Equivalences of the Relational Algebra

4. conjunctions in a selection predicate may be split into multiple selection predicates; likewise multiple selections may be replaced by a single conjunction:

$$\sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots))$$

5. nested projections may be eliminated:

$$\Pi_{l_1}(\Pi_{l_2}(\ldots(\Pi_{l_n}(R))\ldots)) = \Pi_{l_1}(R)$$

where it must hold that:

$$l_1 \subseteq l_2 \subseteq \ldots \subseteq l_n \subseteq \mathcal{R} = sch(R)$$

Equivalences of the Relational Algebra

6. projections may be pushed down through a selection if the projection does not remove any attribute needed to evaluate the selection. Therefore it holds:

$$\Pi_l(\sigma_p(R)) = \sigma_p(\Pi_l(R)), \text{ falls } attr(p) \subseteq l$$

7. selections may be pushed down through joins (or cross products) if it only contains attributes of one of the inputs to the join (cross product).

For instance: if predicate *p* only contains attributes of R_1, then it holds that

$$\sigma_p(R_1 \bowtie R_2) = \sigma_p(R_1) \bowtie R_2$$

Equivalences of the Relational Algebra

8. Similarly, projections may be pushed through a join.

However, we have to take care that attributes appearing in the join predicate are not removed.

$$\Pi_l(R_1 \bowtie_p R_2) = \Pi_l(\Pi_{l_1}(R_1) \bowtie_p \Pi_{l_2}(R_2)) \quad \text{where}$$

$$l_1 = \{A|A \in \mathcal{R}_1 \cap l\} \cup \{A|A \in \mathcal{R}_1 \cap attr(p)\} \quad \text{and}$$

$$l_2 = \{A|A \in \mathcal{R}_2 \cap l\} \cup \{A|A \in \mathcal{R}_2 \cap attr(p)\}$$

9. selections may be pushed through set operations like union, intersection and minus:

$$\begin{aligned} \sigma_p(R \cup S) &= \sigma_p(R) \cup \sigma_p(S) \\ \sigma_p(R \cap S) &= \sigma_p(R) \cap \sigma_p(S) \\ \sigma_p(R - S) &= \sigma_p(R) - \sigma_p(S) \end{aligned}$$

Equivalences of the Relational Algebra

10. projections may be pushed down through union:

$$\Pi_l(R_1 \cup R_2) = \Pi_l(R_1) \cup \Pi_l(R_2)$$

Note: for intersection and minus this is not allowed.

11. a selection and a cross product may be combined into a join if the selection predicate is a join predicate.

For instance, for an equi join it holds:

$$\sigma_{R_1.A_1=R_2.A_2}(R_1 \times R_2) = R_1 \bowtie_{R_1.A_1=R_2.A_2} R_2$$

Equivalences of the Relational Algebra

12. predicates may be rewritten.

For instance, a disjunction may be rewritten into a conjunction using DeMorgan's law. This may then be exploited later on to apply Rule 4 (splitting selections):

$$\neg(p1 \wedge p2) = (\neg p1) \vee (\neg p2)$$

$$\neg(p1 \vee p2) = (\neg p1) \wedge (\neg p2)$$

Summary of Most Important Rules

1. split selections
2. push down selections and projections down the tree as far as possible
3. combine selections and cross products into joins
4. if possible: insert additional projections

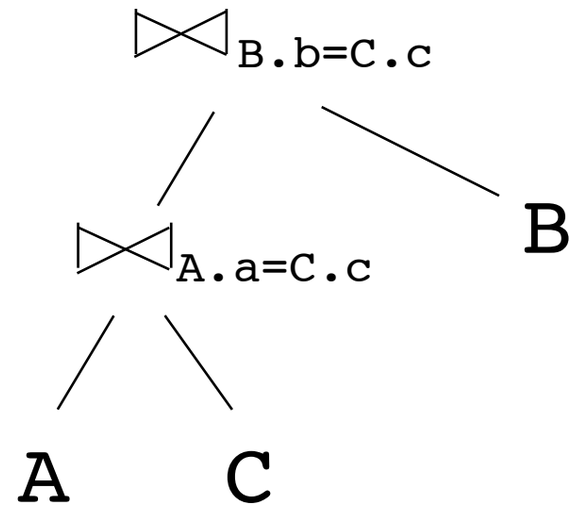
Query Rewrite: Add Additional Join Predicates

```
SELECT *
FROM A,B,C
WHERE A.a=B.b AND B.b=C.c;
```

is rewritten to

```
SELECT *
FROM A,B,C
WHERE A.a=B.b AND B.b=C.c AND A.a=C.c;
```

As a consequence, the following plan becomes feasible:



(If not, we would have a cross product instead of the join of A and C!)

QR: Simplification of Expressions

```
SELECT *
FROM rectangles
WHERE rectangles.width * rectangles. width < 100;
```

is rewritten to

```
SELECT *
FROM rectangles
WHERE rectangles. width < 10;
```

Discussion

- only works if the following integrity constraint holds: rectangles.width ≥ 0
- predicate evaluation becomes cheaper
- allows optimizer to use an index

QR: Unnesting ANJ

- we consider three classes of subqueries
  - Type **A**: subquery is constant and returns a single value
  - Type **N**: subquery is constant and returns a table
  - Type **J**: subquery depends on an outer query and returns a table
- Discussion
  - there exist many more specialized subquery types and many specialized rules to optimize them
  - we will concentrate on the most important cases

QR: Type A

```
SELECT *
FROM Emp e
WHERE e.salary = (SELECT max(salary) FROM Emp);
```

constAnt

is rewritten to

```
define m = SELECT max(salary) FROM Emp;

SELECT *
FROM Emp e
WHERE e.salary = m;
```

Note

- “define” is not part of SQL but a feature of the internal representation of a query
- advantage: instead of n iterations over the Emp table only two

QR: Type N

```
SELECT *
FROM Emp e
WHERE e.dno IN(SELECT d.dno
               FROM Dept d
               WHERE d.name='Research');
```

table iNdependent of e

is rewritten to

```
SELECT *
FROM Emp e, Dept d
WHERE e.dno=d.dno AND d.name='Research';
```

Note

- typically this only works if functional dependencies exist among the tables
- in general, a subquery of type N may be replaced by semijoins and antijoins

QR: Type J

```
SELECT *
FROM Emp e
WHERE e.dno IN (SELECT d.dno
                FROM Dept d
                WHERE d.budget > e.salary*10);
```

table dependent on e

is rewritten to

```
SELECT *
FROM Emp e, Dept d
WHERE e.dno=d.dno AND d.budget > e.salary*10;
```

Discussion

- rewrite similar to subqueries of type N
- however, in some cases there are differences

QR: FROM Clause

```
SELECT *
FROM Emp e, (SELECT d.dno
             FROM Dept d
             WHERE d.name='Research') d
WHERE e.dno=d.dno;
```

is rewritten to

```
SELECT e.*
FROM Emp e, Dept d
WHERE e.dno=d.dno AND d.name='Research';
```

Discussion

- in many cases unnesting inside a FROM-clause is simpler as less cases exist (e.g., no all quantifiers)

QR: Existential Quantifiers

```
SELECT *
FROM Emp e
WHERE e.salary EXISTS (SELECT m.salary
                       FROM Manager m
                       WHERE p(e,m));
```

is rewritten to

```
SELECT *
FROM Emp e
WHERE 0 < (SELECT COUNT(m.salary)
           FROM Manager m
           WHERE p(e,m));
```

Discussion

- similar for `NOT EXISTS`
- depending on complexity of p other transformations are possible (Type A or J)

QR: Predicate Movearound

```
SELECT *
FROM Emp e, (SELECT name, d.dno
               FROM Dept
               WHERE dno = e.dno
                     AND boss = e.name AND boss like '%strange%') d
WHERE e.dno BETWEEN 1 and 10;
```

is rewritten to

```
SELECT *
FROM Emp e, (SELECT name, d.dno
               FROM Dept
               WHERE dno = e.dno
                     AND boss = e.name AND boss like '%strange%'
                     AND e.dno BETWEEN 1 and 10) d
WHERE e.dno BETWEEN 1 and 10
      AND e.name like '%strange%';
```

Discussion

- use this if unnesting fails (or is not implemented)
- similar rules for `DISTINCT`

Query Optimization.

Internal Query Representation.

Internal Representation of a Query

- How is a query internally stored by the query optimizer?
- principal concepts
  - each subquery of a query (e.g., view) is represented by a separate query block
  - a query block has a header containing:
    - does the output have to be ordered?
    - duplicate elimination: necessary, allowed, forbidden
    - required attributes of this query block
  - a query block has a body containing:
    - operator tree
    - join graph, i.e., data sources (inputs) are nodes, possible joins are edges
  - dependencies among variables belonging to different query blocks are also represented by edges

Internal Representation of a Query

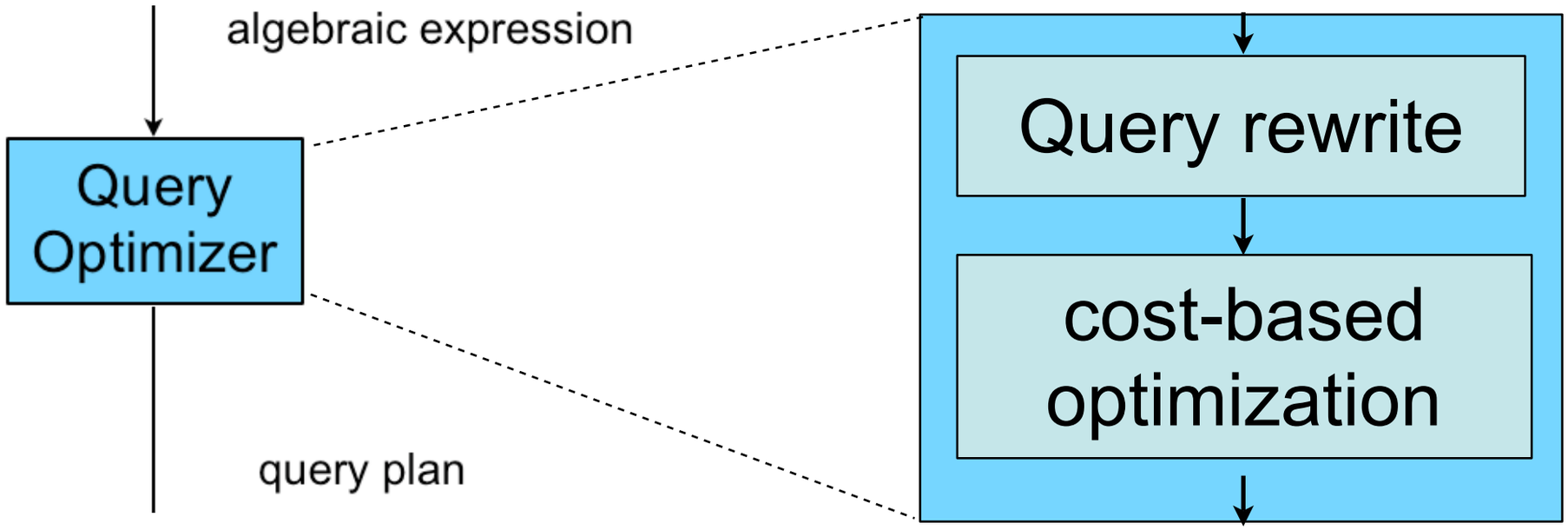
- **parser**
  - creates the first internal representation of a query
- **query rewrite**
  - transforms internal representations iteratively and creates the “best“ internal representation of a query
  - splits query into blocks (only one select clause per block)
- **cost-based optimizer**
  - is applied for each query block creates a physical plan

Query Optimization.

Cost-based Optimization.

Query Optimizer

- Query optimization consists of two phases



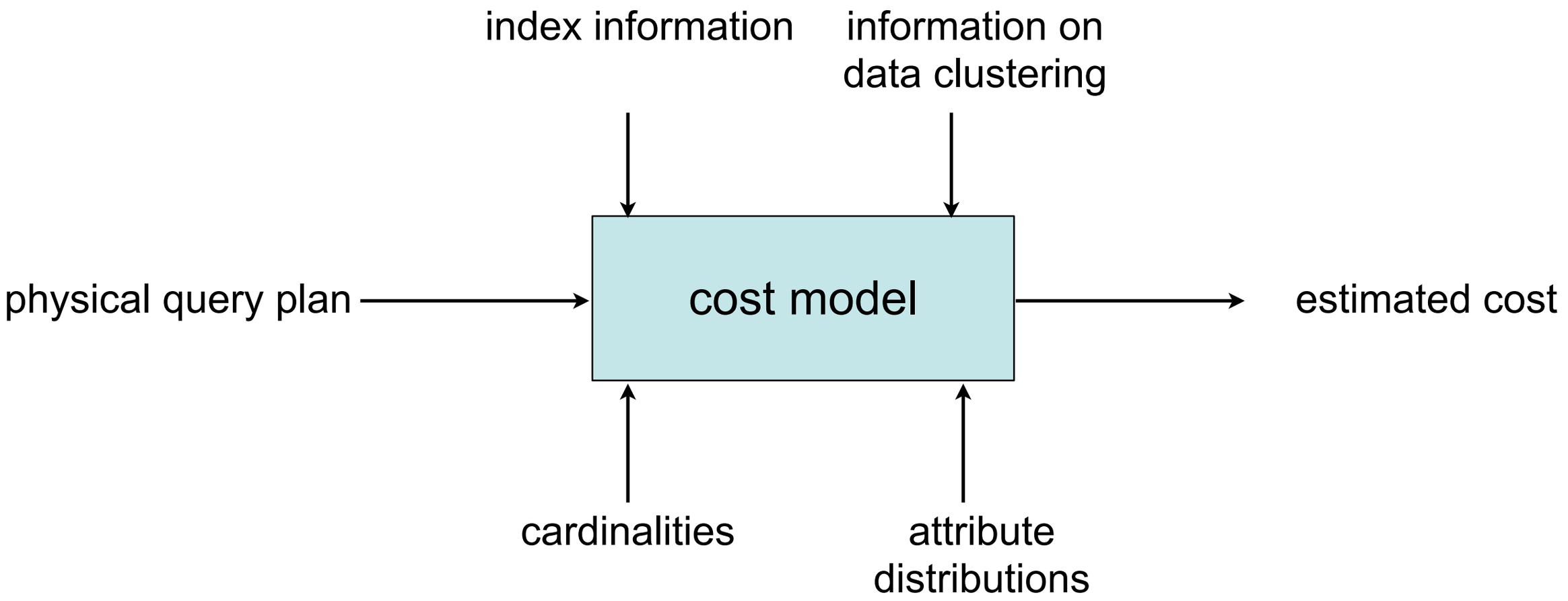
- Query rewrite
  - rewrites query using rules
- Cost-based optimization
  - estimates cost of different plans using cost models
  - picks plan that has expected lowest costs

Cost-Based Optimization

- Core idea
  - enumerate all possible physical plans (candidate plans)
  - try to estimate the cost of each candidate plan
  - pick the candidate plan with the estimated lowest cost

What are the costs of a query plan?

Cost Models


index information
information on data clustering
physical query plan
cost model
estimated cost
cardinalities
attribute distributions

Selectivities

- very simple cost model
  selectivity of an operaton = #tuples output / #tuples input
- selectivity using a predicate p

$$sel_p := \frac{|\sigma_p(R)|}{|R|}$$

- **Terminology:**
  - "high selectivity" = sel_P small
    i.e., many tuples are removed by the selection
  - "low selectivity" = sel_P big
    i.e., only few tuples are removed by the selection

Estimate Selectivities

- Examples
  - p(r) = (r.key%2==0)
  - assuming uniformly distributed keys in R it follows: $sel_p \approx 0.5$

$$sel_p := \frac{|\sigma_p(R)|}{|R|}$$

- $$sel_{R.key=42} := \frac{1}{|R|}$$ R.key = key of R

- $$sel_{R.attr} := \frac{1}{C}$$ where C is the cardinality of attribute R.attr

Terminology

"high selectivity" = sel_P small
"low selectivity" = sel_P big

Join Selectivities

$$sel_{R \bowtie S} := \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| * |S|}$$

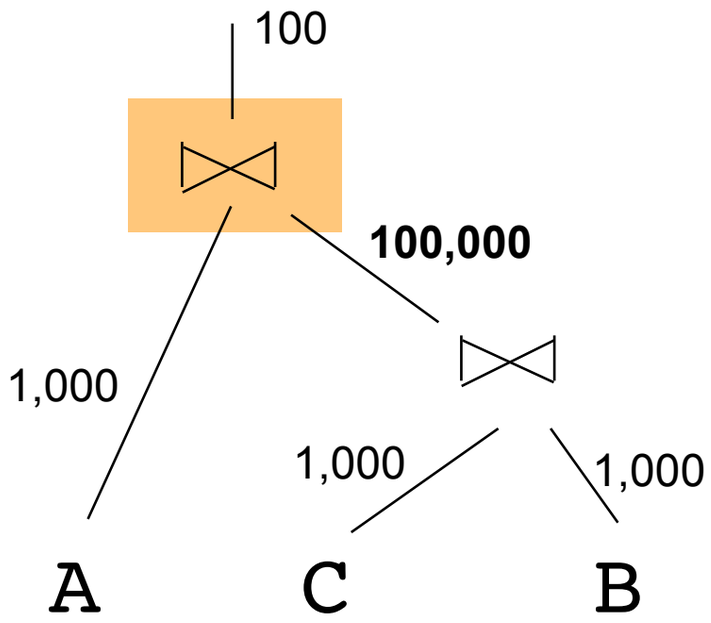
$$sel_{R.key=S.B} := \frac{1}{|R|}$$

equi join of R and S
using foreign key in S

Example: Join Order

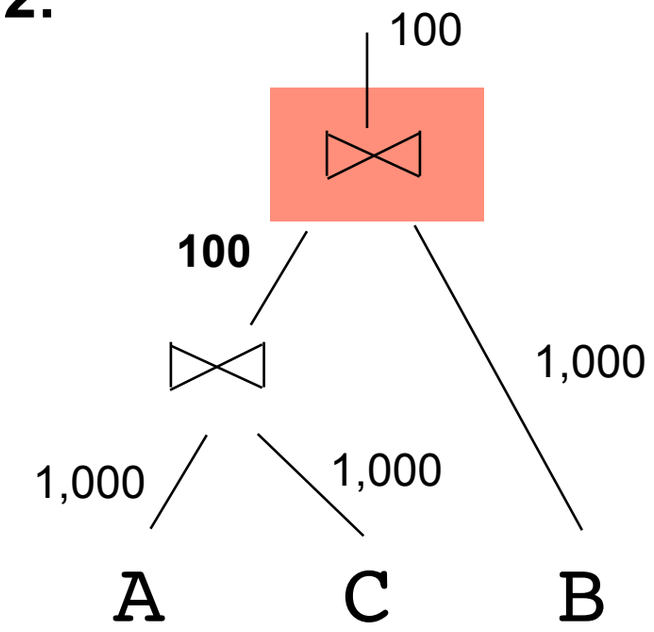
$|A|=|B|=|C|=1{,}000$

Plan 1:



$$sel_{C \bowtie B} := \frac{|C \bowtie B|}{|C| * |B|} = \frac{100{,}000}{|1{,}000| * |1{,}000|} = 0.1$$

Plan 2:



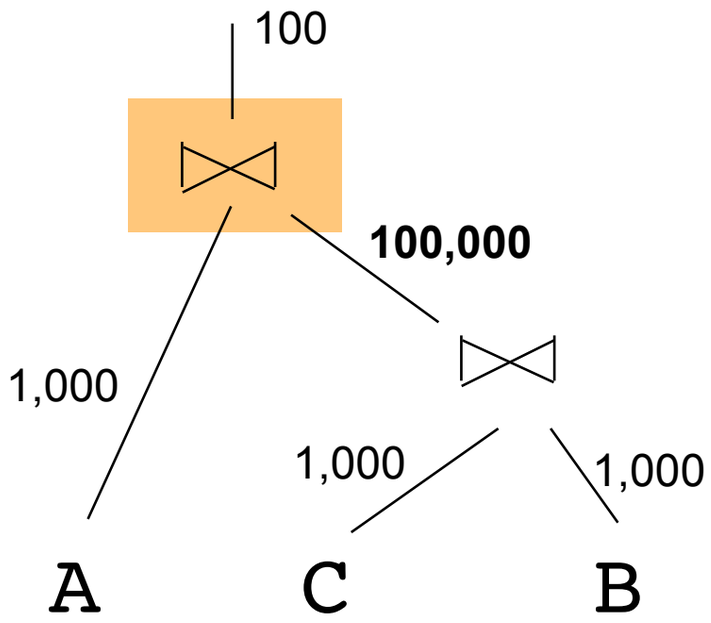
$$sel_{A \bowtie C} := \frac{|A \bowtie C|}{|A| * |C|} = \frac{100}{|1{,}000| * |1{,}000|} = 0.0001$$

- Plan 1: Top-level join has to process **1,000*100,000** tuples.
- Plan 2: Top-level join has to process **100*1,000** tuples.

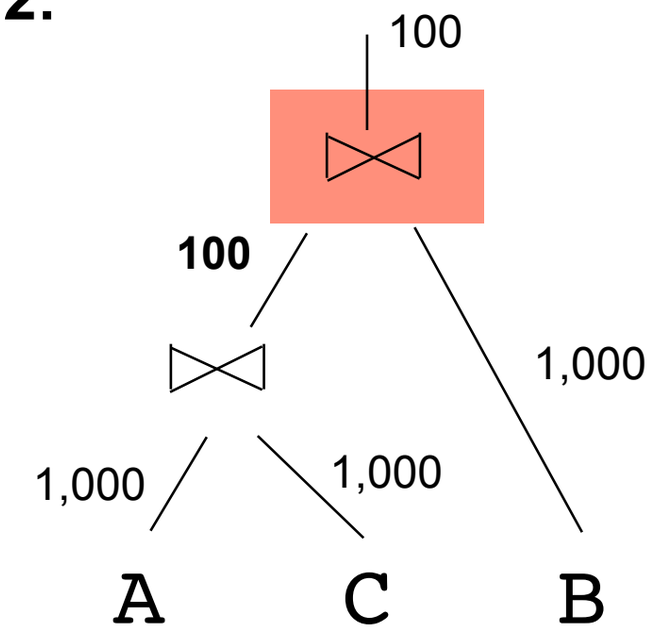
Example: Join Order

|A|=|B|=|C|=1,000

Plan 1:



Plan 2:



- consequences
  - joins having a high selectivity should be executed first
  - same problem for **intersections** in search engines
- effect
  - sizes of intermediate results will be decreased
  - following operators have to process smaller data sets
- However: join order should **not only** be determined based on selectivities

Discussion: Join Selectivities

- in reality join selectivities are hard to estimate
- distributions of R and S have a considerable impact on the true size of the join result

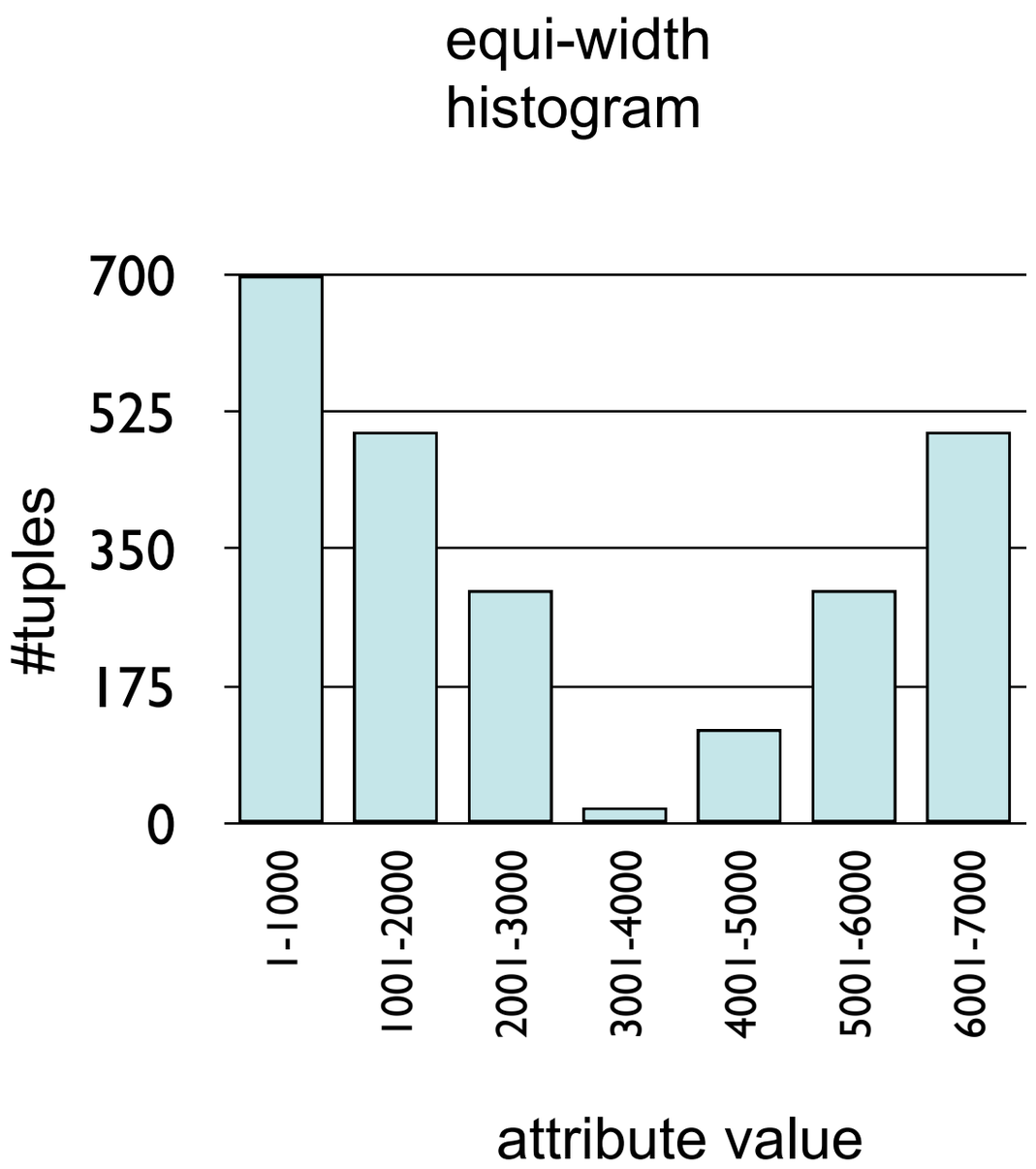
- Examples
  - R={1,3,5,7,9,...}
  - S={2,4,6,8,...}
  - |R ⋈ S| = 0
  - however:
  - R={1,3,5,7,9,...,N}
  - S={1,3,5,7,9,...,N}
  - |R ⋈ S| = N
  - etc.

How to Make Better Estimates?

- **Goal: provide optimizer with better selectivity estimations**
- **parametric distributions**
  - try to determine a function that approximates data distribution
  - e.g., as a combinations of different functions (kernels, wavelets)
- **histograms**
  - partition data domain into intervals and count number of tuples per interval
- **sampling**
  - draw a sample of the data set and compute statistics based on the sample
  - statistically, the sample represents the entire data set
  - could even try to execute joins on samples to understand join selectivity

Histograms

- **Idea**:
  partition data domain into intervals and count number of tuples per interval
- Advantage: easy to implement
- Variants
  - **equi-width**: all intervals have the same length
  - **equi-depth**: all intervals contain the same number of tuples



Sampling

- **Idea**: draw a sample of the data set and compute statistics based on the sample,
- statistically, the sample represents the entire data set
- two optimization goals
  - compute sample in a way such that I/O-effort is saved later on
  - compute sample in a way such that distribution of the sample is equal to the distribution of the data set (real sample)
- simple implementation

  let k be the fractional size of the sample to draw $0 < k \leq 1$

  ForEach r in R:

  x = uniformly distributd random number in-between 0 and 1

  If $x \leq k$:

  SAMPLE = SAMPLE $\cup$ {r}

  Question: does this save I/O-effort?

Statistic Computation

- **Note**
  - statisitics are not computed by the DBMS automatically and are not automatically updated (in case data changes)
  - however, the cost-based optimizer may only find an efficient plan if statistics are up-to-date...
- Therefore, the DBA has to explicitly trigger statistic computation.
- Oracle

```
ANALYZE TABLE xy COMPUTE STATISTICS FOR
TABLE;
```

- IBM DB2

```
RUNSTATS ON TABLE...
```

I/O-Cost Model

- our cost model so far: selectivities of operators
- This is too simple!!
- Better cost models exist:
- simple I/O-cost model
  - ignore cost for cpu-effort
  - every write and read operation of a block is counted as 1 cost unit
  - cost of a query plan := sum of cost units
  - estimated selectivities are just parameters in this model
- improved I/O-cost model
  - like simple cost model
  - but: each seek counts 10 cost units
  - cost of a query plan := sum of cost units for read, write, and seek

Cost Estimation for a Selection

- B: number of records per page
- N: size of data set (number of records)
- n: = ⌈N/B⌉, size of data set (number of pages)

- cost for
  - Full-table-scan (FTS): $cost_{FTS} = 10 + n$
  - random acces to n pages: $cost_{random} = 10*n + n$
  - Index-access+ISAM (att=42):
    assumptions:
    - index clustered on att. therefore logarithmic cost to walk down the tree =: t.
    - then ISAM.
    - in total: $cost_{Index+ISAM} = t*10 + \lceil sel_p * n \rceil$

Cost Estimation for a Simple Join

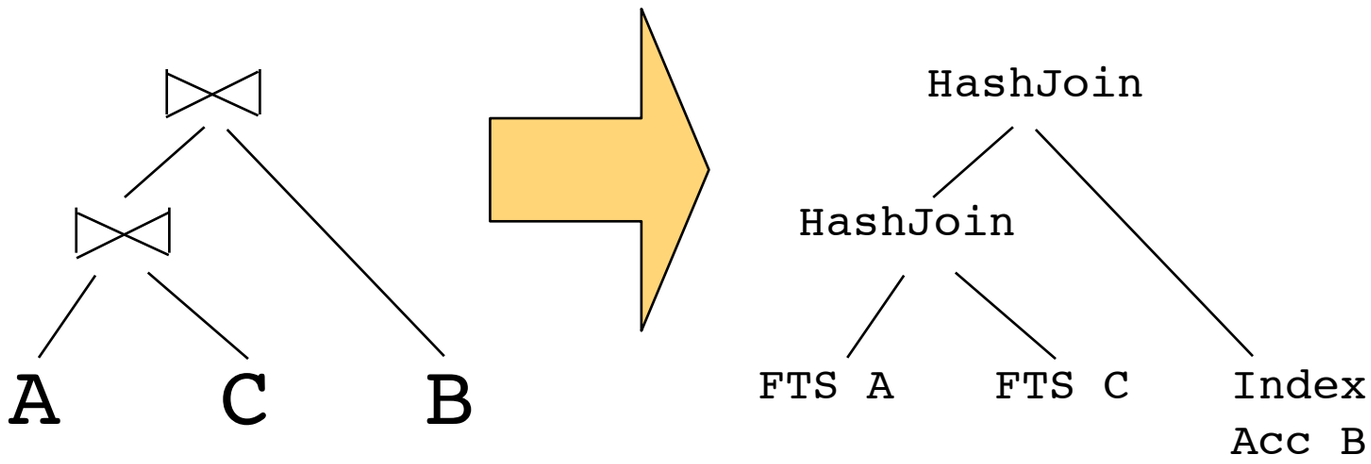
- page-oriented nested-loops join:
  - assumption: reserve m-k pages for the outer relation R (outer loop)
  - reserve k pages for relation S (inner loop)

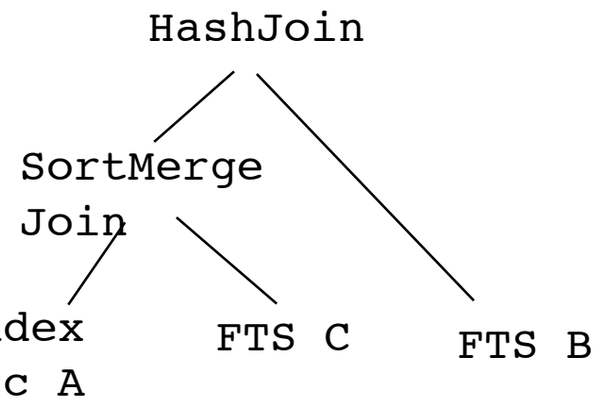
  - R will be read once: $cost_R = 10 * \lceil n_R/(m-k) \rceil + n_R$
  - S will be read $\lceil n_R/(m-k) \rceil$ times:

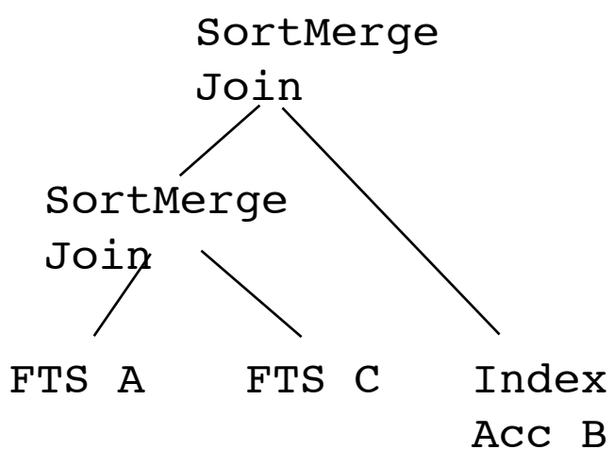
    $cost_S = \lceil n_R/(m-k) \rceil * (10+n_S)$
  - In total: $n_R + \lceil n_R/(m-k) \rceil * (20+n_S)$

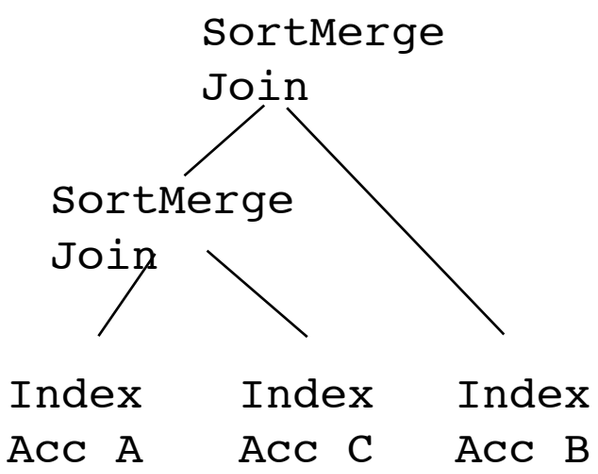
Plan Enumeration

- now we are able to enumerate different plans
- for each plan we may estimate its cost



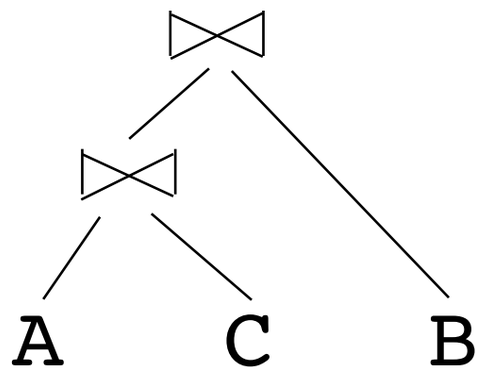






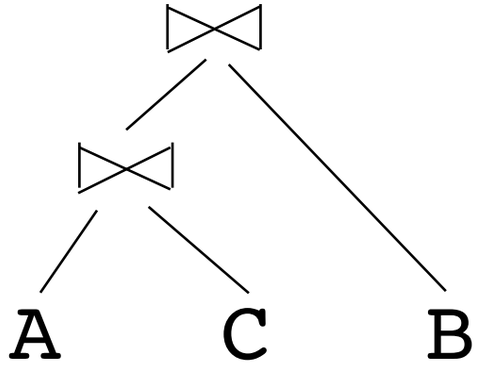
...

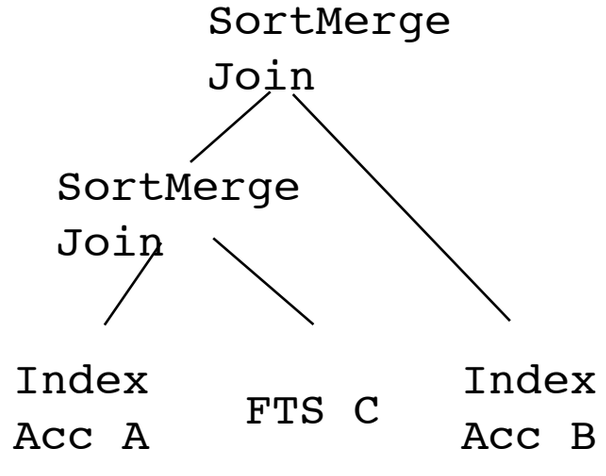
Cost of Plan Enumeration



- Assume...
  - ...each leaf may be implemented by two different access paths (FTS or IndexAccess).
  - ...each join may be implemented by either a HashJoin, Sort-Merge-Join, or a nested-loops join.
- For join order (A ⋈ C) ⋈ B this already results in $2^3 * 3^2 = 72$ plans.
- Moreover: 72 plans for join order A ⋈ (C ⋈ B).
- Question: why do we again consider the join order anyway? (we already did this for join selectivities...)

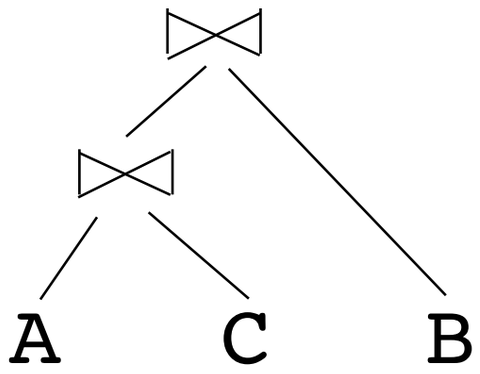
Join Order (Reloaded)

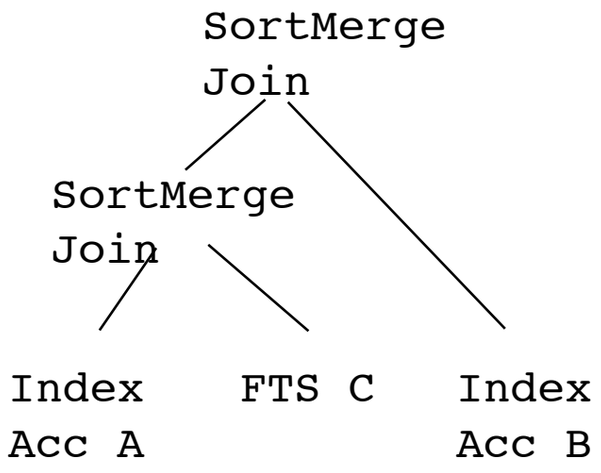




- Question: why do we again consider the join order?
- assume tables A and B are accessed by an index
- the indexes deliver tuples already sorted by the join predicate
- therefore the following sort-merge join only needs to sort table C!
- all other sort operations may be skipped!
- this may have the effect that a plan having joins producing large intermediate results may be more efficient!
- these **interesting orders** may allow the optimizer to pick a more efficient plan!

Join Order (Revolutions)





- Consequences
  - the final decision on a join order should be taken by the cost-based optimizer
  - in particular, decision on which join operator to choose depends on interesting orders created by subplans

Dynamic Programming

- bottom-up computation: compute optimal plan for one input
- plans for i inputs are based on optimal plans for i-1 inputs
- subplans are building blocks for larger plans
- example:
  - A $\bowtie$ B and B $\bowtie$ A compute the same results, however, only one of these plans is kept
  - let‘s assume that A $\bowtie$ B is cheaper and kept
  - then it holds that every plan having A $\bowtie$ B as a building block is cheaper than a plan having B $\bowtie$ A as a building block.
  - e.g.: C $\bowtie$ (A $\bowtie$ B) has to be cheaper than C $\bowtie$ (B $\bowtie$ A)
- Note: we make the assumption that the cost of the entire plan may be obtained by adding up the cost for subplans (costs are cumulative)!

Dynamic Programming

- Counter example: A sort-merge-join B and A hash-join B
- in this case the sorted output w.r.t. the join (interesting order) key may be exploited by a following operator (e.g., to compute an aggregate or for another join)
- Consequence: the decision for one of these plans may not be made locally
- therefore both plans have to be kept

- Literature: Simmen, D., Shekita, E., and Malkemus, T. Fundamental techniques for order optimization. ACM-SIGMOD, 1996, 57– 67.

Discussion

- Note: problem to find the optimal plan is NP
- exponential runtime: $O(3^v)$
- exponential memory consumption: $O(2^v)$
- v is the number of input relations
- delivers plan only at the end
- finds ‘optimal’ plan (under a given cost model)

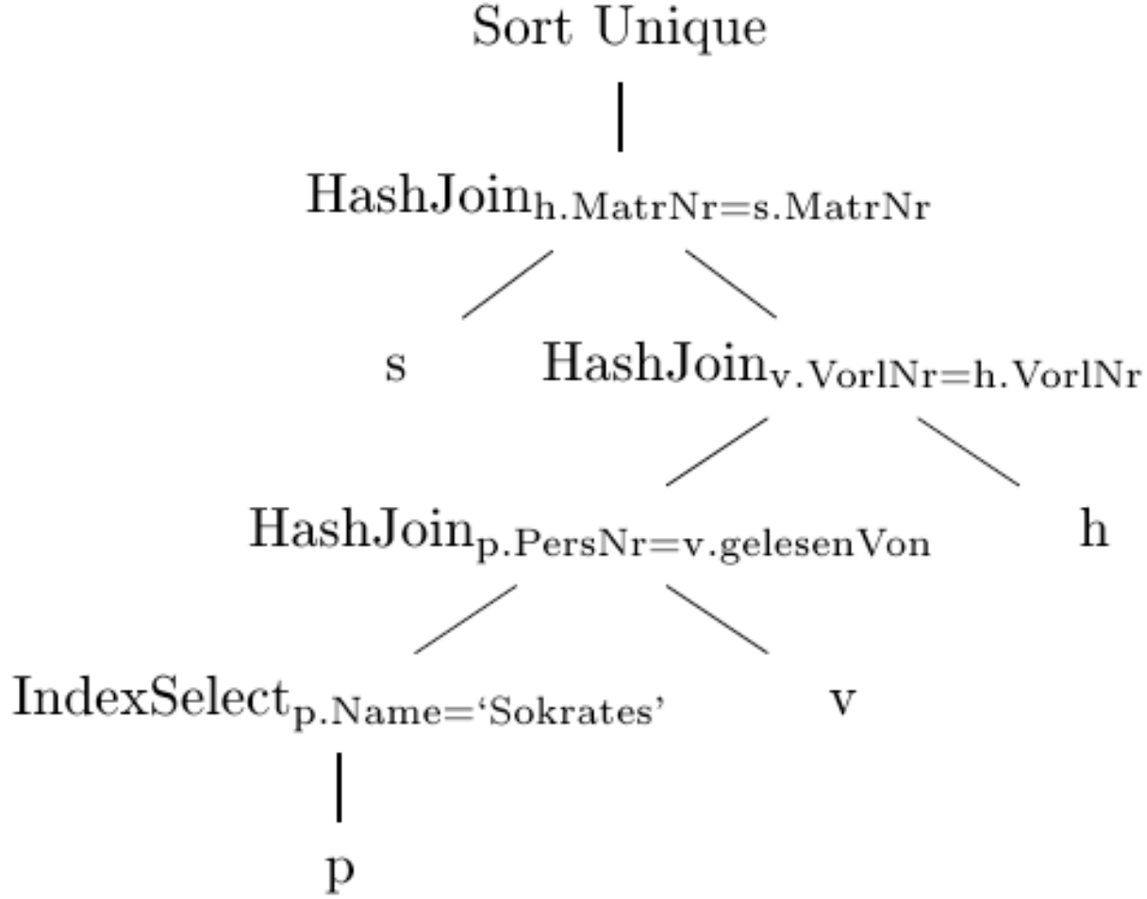
- Literature (overview): Donald Kossmann, Konrad Stocker: Iterative dynamic programming: a new class of query optimization algorithms. ACM Trans. Database Syst. 25(1): 43-82 (2000)

explain plan

```
explain plan for
  SELECT ...
```

result:

```
SELECT STATEMENT    Cost = 37710
  SORT UNIQUE
HASH JOIN
  TABLE ACCESS FULL STUDENTEN
  HASH JOIN
    HASH JOIN
      TABLE ACCESS BY ROWID PROFESSOREN
        INDEX RANGE SCAN PROFNAMEINDEX
      TABLE ACCESS FULL VORLESUNGEN
    TABLE ACCESS FULL HOEREN
```



Next Topic: Transaction Management and Crash Recovery.