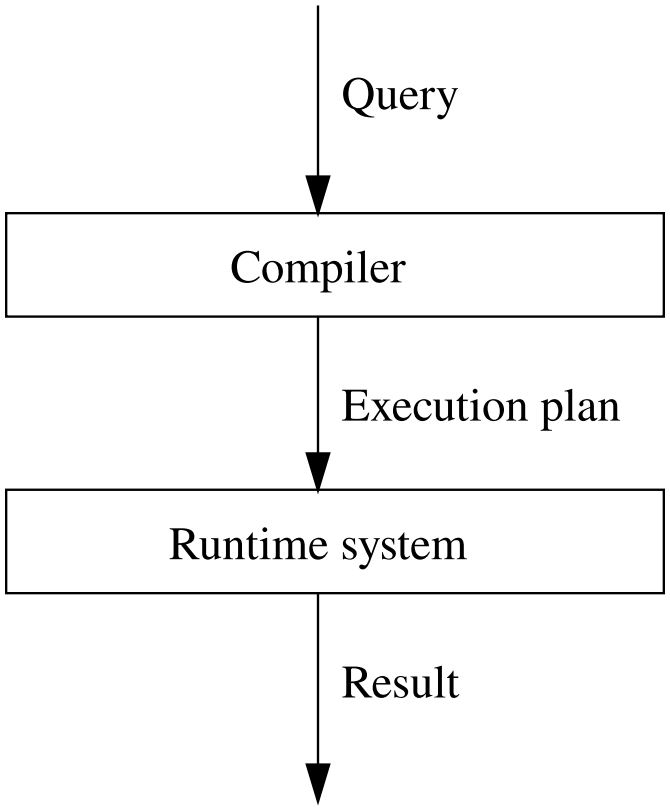
Query Processing

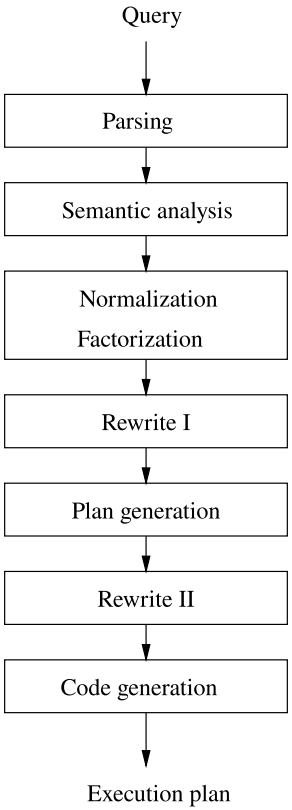
Query Processing


Query
Compiler
Execution plan
Runtime system
Result

Translation

- SQL is declarative, but at some point the query has to be translated into something procedural for the runtime system
- The DBMS translates SQL into an internal representation
- A widespread approach is the translation into relational algebra

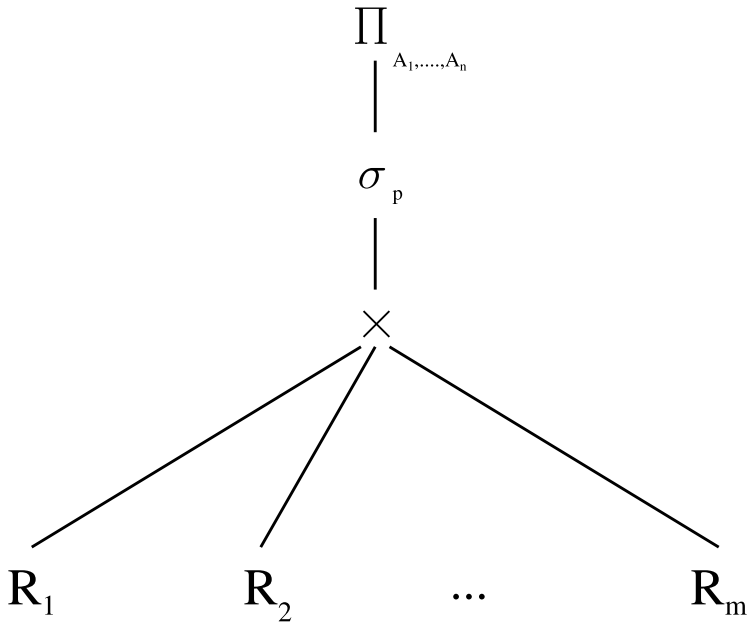
Translation(2)



Canonical Translation

- There is a standard translation from SQL into relational algebra
- Algebra expressions are often represented graphically, too

```
select A_1, …, A_n
from   R_1, …, R_m
where  p
```



Extended Translation

Extensions to the “classical” canonical translation:

- `select a, sum(d) as s from ... group by a, b, c`
  - $\Pi_{a,s}\left(\Gamma_{a,b,c;s:\sum(d)}(C)\right)$
  - where C is the canonical translation of the inner part
  - mind the projection clause!
- `select ... having p`
  - $\sigma_p(C)$, where C is the translation including group by
- `select ... order by a, b, c`
  - $sort_{a,b,c}(C)$, where C is the translation including having
  - $sort$ did not appear in relational algebra because relations are unsorted

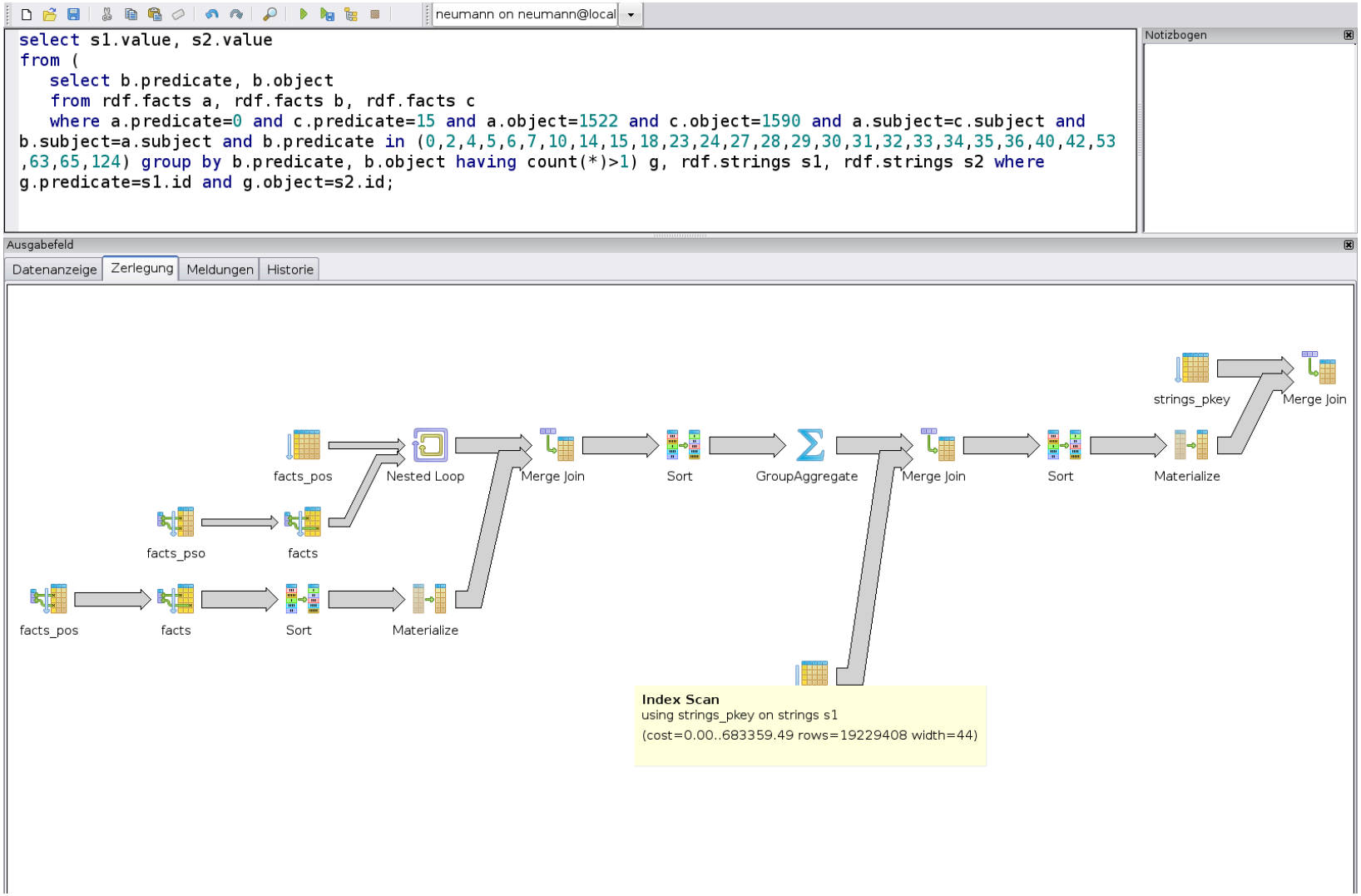
Optimization

- The canonical plan is not very efficient (e.g., it contains cross products)
- The DBMS has an optimizer to transform a plan into a more efficient form
- Finding a plan is a very hard problem, still subject of current research

Optimization(2)

- What does an ordinary user have to do with query optimization?
- The DBMS optimizer sometimes produces suboptimal plans
- Most DBMSs give the user insight into the generated plans
- The user can analyze the generated plan and, if necessary, restructure the query or give the DBMS hints for execution

Visualizing Plans



- The DBMS can estimate the cost of executing an operator with the help of cost models and statistics
- Optimizing a plan uses heuristics; looking at all possible plans is far too expensive
- Optimization can take place on different levels:
  - Logical level
  - Physical level

- Starting point is the relational algebra expression resulting from the canonical translation
- Optimization: trans-formation of relational algebra expressions into equivalent expressions (that lead to a faster execution plan)
- Transformations should be chosen such that the outputs of the individual operators become as small as possible

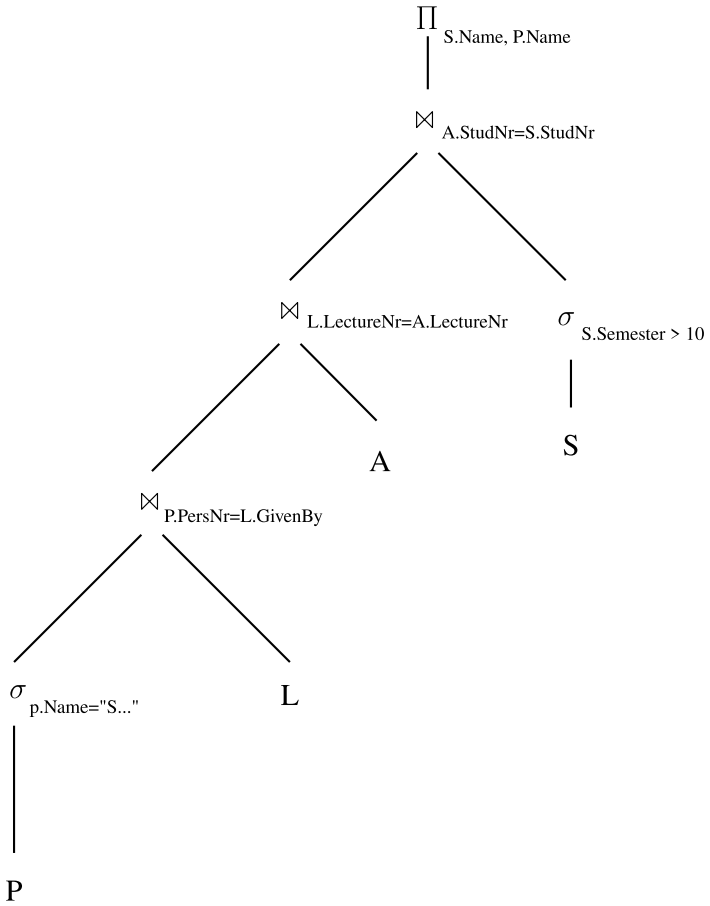
Logical Level(2)

- Basic techniques:
  - Breaking up selections
  - Pushing selections “down” in the plan
  - Combining selections and cross products into joins
  - Determining the join order
  - Inserting projections
  - Pushing projections “down” in the plan

Example Query

```
select S.Name, P.Name
from   Students S, attend A, Lectures L, Professors P
where  S.StudNr = A.StudNr
and    A.LectureNr = L.LectureNr
and    L.GivenBy = P.PersNr
and    S.Semester > 10
and    P.Name = 'Sokrates';
```

(Optimized) Query Plan



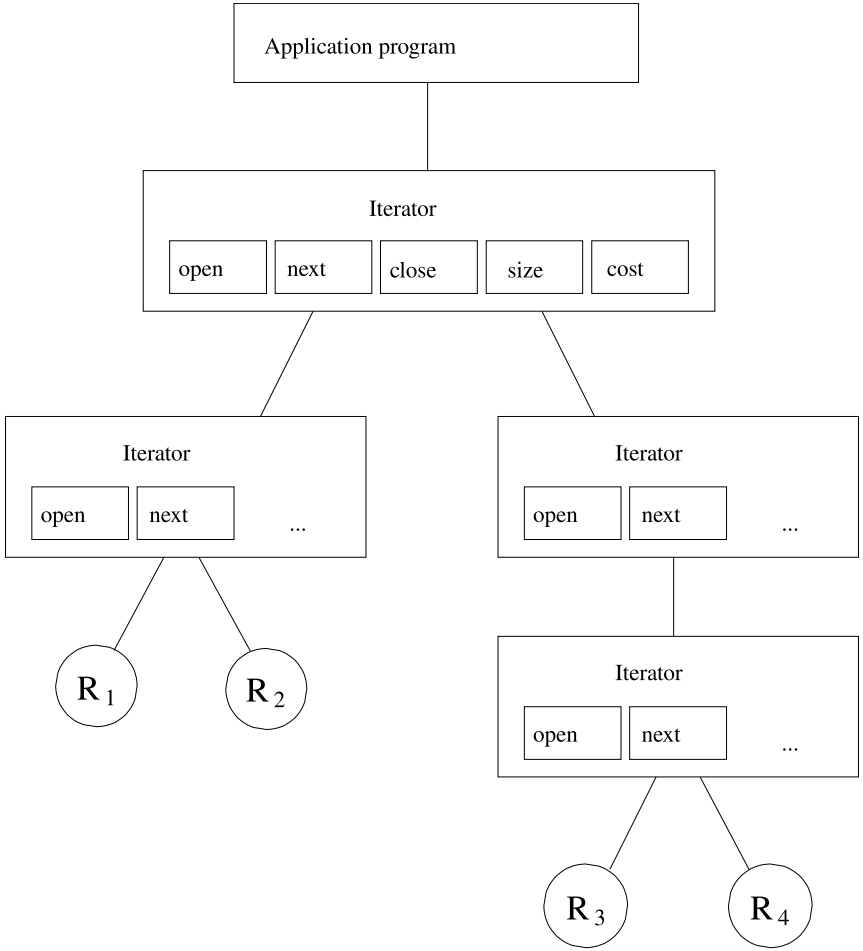
Excerpt of Rules

$$
\begin{array}{ccc}
R_1 \bowtie R_2 & = & R_2 \bowtie R_1 \\
R_1 \cup R_2 & = & R_2 \cup R_1 \\
R_1 \cap R_2 & = & R_2 \cap R_1 \\
R_1 \times R_2 & = & R_2 \times R_1 \\
R_1 \bowtie (R_2 \bowtie R_3) & = & (R_1 \bowtie R_2) \bowtie R_3 \\
R_1 \cup (R_2 \cup R_3) & = & (R_1 \cup R_2) \cup R_3 \\
R_1 \cap (R_2 \cap R_3) & = & (R_1 \cap R_2) \cap R_3 \\
R_1 \times (R_2 \times R_3) & = & (R_1 \times R_2) \times R_3 \\
\sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(R) & = & \sigma_{p_1}\left(\sigma_{p_2}\left(\ldots\left(\sigma_{p_n}(R)\right)\ldots\right)\right) \\
\Pi_{l_1}\left(\Pi_{l_2}\left(\ldots\left(\Pi_{l_n}(R)\right)\ldots\right)\right) & = & \Pi_{l_1}(R) \\
 & \text{with} & l_1 \subseteq l_2 \subseteq \ldots \subseteq l_n \subseteq \mathcal{R} = \text{sch}(R) \\
\Pi_l\left(\sigma_p(R)\right) & = & \sigma_p(\Pi_l(R)), \text{ if } \text{attr}(p) \subseteq l
\end{array}
$$

Physical Optimization

- One distinguishes between logical algebra operators and physical algebra operators
- Physical algebra operators are the implementations of the logical ones
- There can be several physical operators for one logical operator
- Optimization on the physical level means choosing one of these operators, deciding whether indexes should be used, materializing intermediate results, etc.

Iterator Concept



Implementing Selection

* **iterator** Scan_p

- **open**
  - Open input
- **next**
  - Fetch next tuples until one satisfies the condition p
  - Return this tuple
- **close**
  - Close input

Implementing Selection(2)

iterator IndexScan_p

- **open**
  - Look up the first tuple in the index that satisfies the condition
  - Open input
- **next**
  - Return next tuple if it still satisfies the condition p
- **close**
  - Close input

Implementing Join

- Set difference and intersection can be implemented analogously to the join
- here only equi-joins

Nested-loop join:

for each $r \in R$

  for each $s \in S$

    if $r.A = s.B$ **then**

      $res := res \cup (r \times s)$

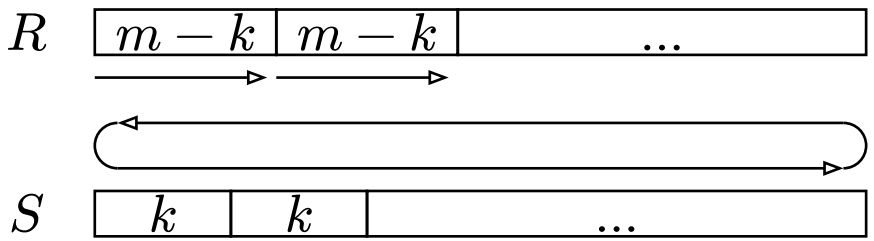
Implementing Join(2)

iterator NestedLoop$_p$

- **open**
  - Open the left input
- **next**
  - Right input closed?
    - Open it
  - Request tuples from the right until condition p is satisfied
  - If the right input is exhausted in the meantime
    - Close right input
    - Fetch next tuple of the left input
    - Restart **next**
  - Return the join of the current left and current right tuple
- **close**
  - Close both inputs

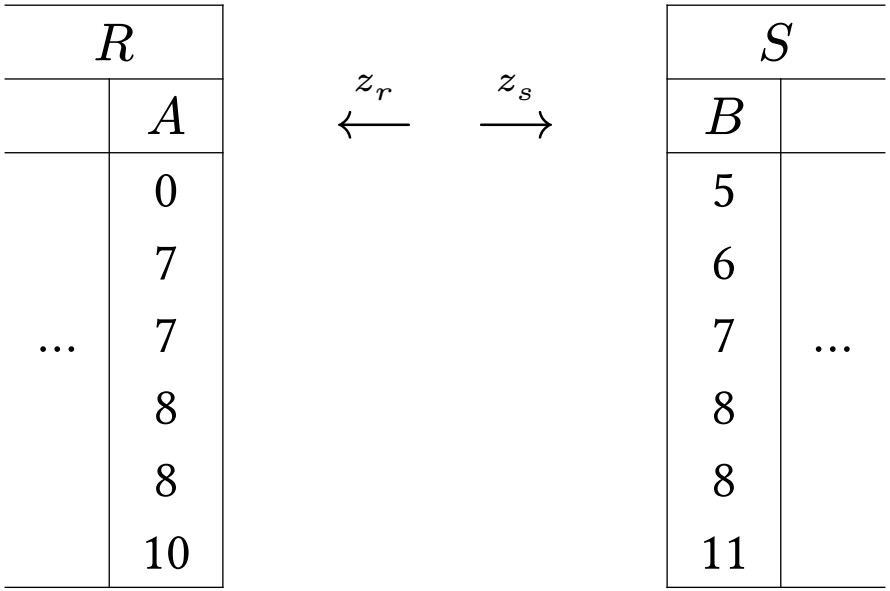
- Relations are stored *page-wise*
- There are m buffer frames available in main memory:
  - k for the inner loop of the nested loop
  - $m - k$ for the outer one

Join of R and S:



- Prerequisite: R and S are sorted (sort them first if necessary)

Example:



(Sort-)Merge-Join(2)

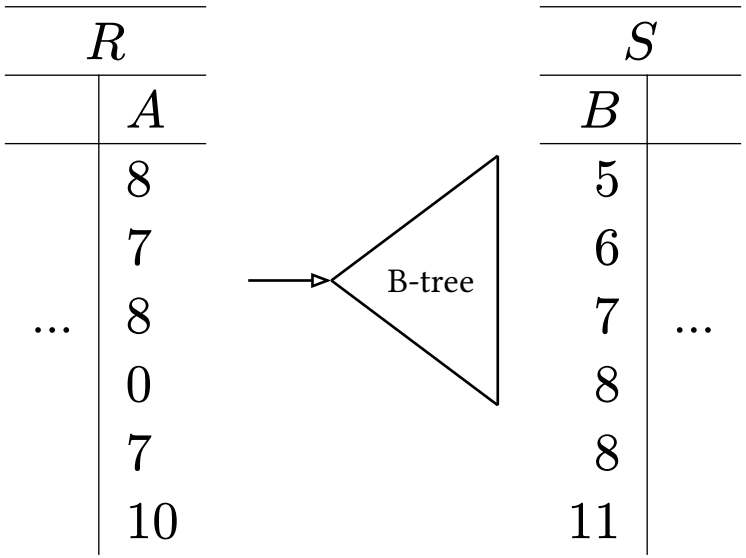
iterator MergeJoin$_p$

- **open**
  - Open both inputs
  - Set *cur* to left input
  - Mark right input
- **close**
  - Close both inputs

iterator MergeJoin_p

- **next**
  - While condition not satisfied
    - Set *cur* to the input with the smallest current value in the join attribute
    - Call **next** on *cur*
    - Mark other input
  - Combine left and right tuple
  - Advance other input
  - Condition no longer satisfied or other input exhausted?
    - Advance *cur*
    - Value of the join attribute in *cur* changed?
      ⇒ No: reset other input to the mark
      ⇒ Otherwise: mark the other input

Index-Join



Index-Join(2)

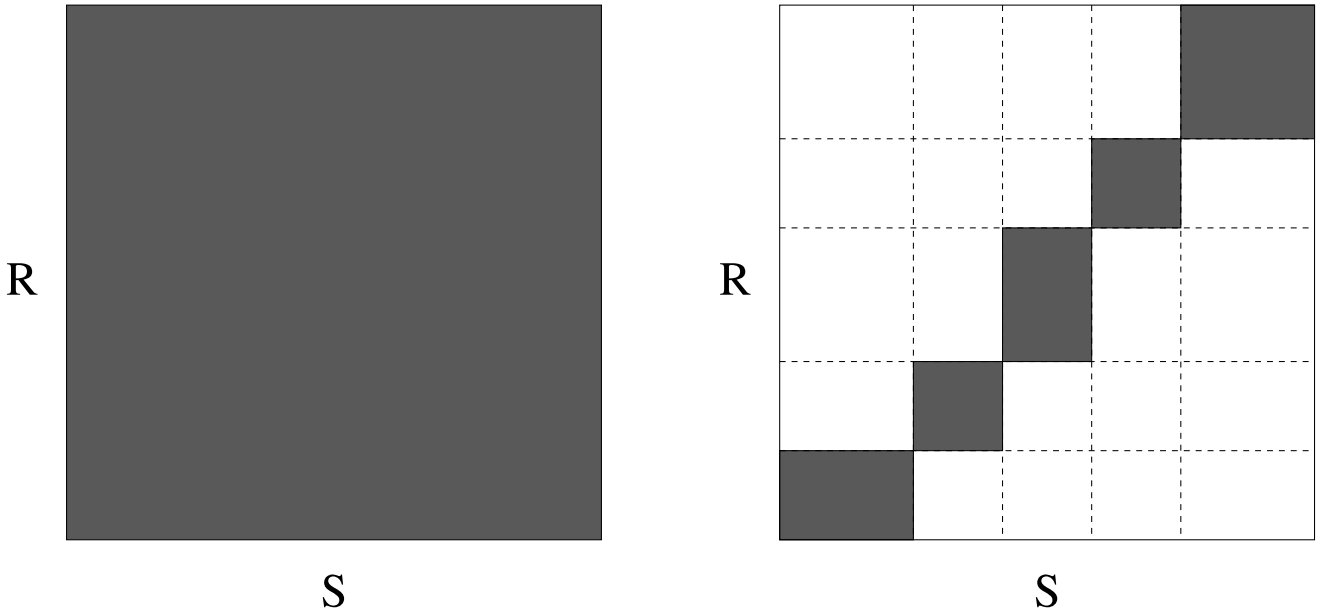
iterator IndexJoin$_p$

- **open**
  - Open the left input
  - Fetch first tuple from left input
  - Look up join attribute value in the index
- **next**
  - Produce join if the index yields a (further) tuple for this value
  - Otherwise advance left input and look up join attribute value in the index
- **close**
  - Close the input

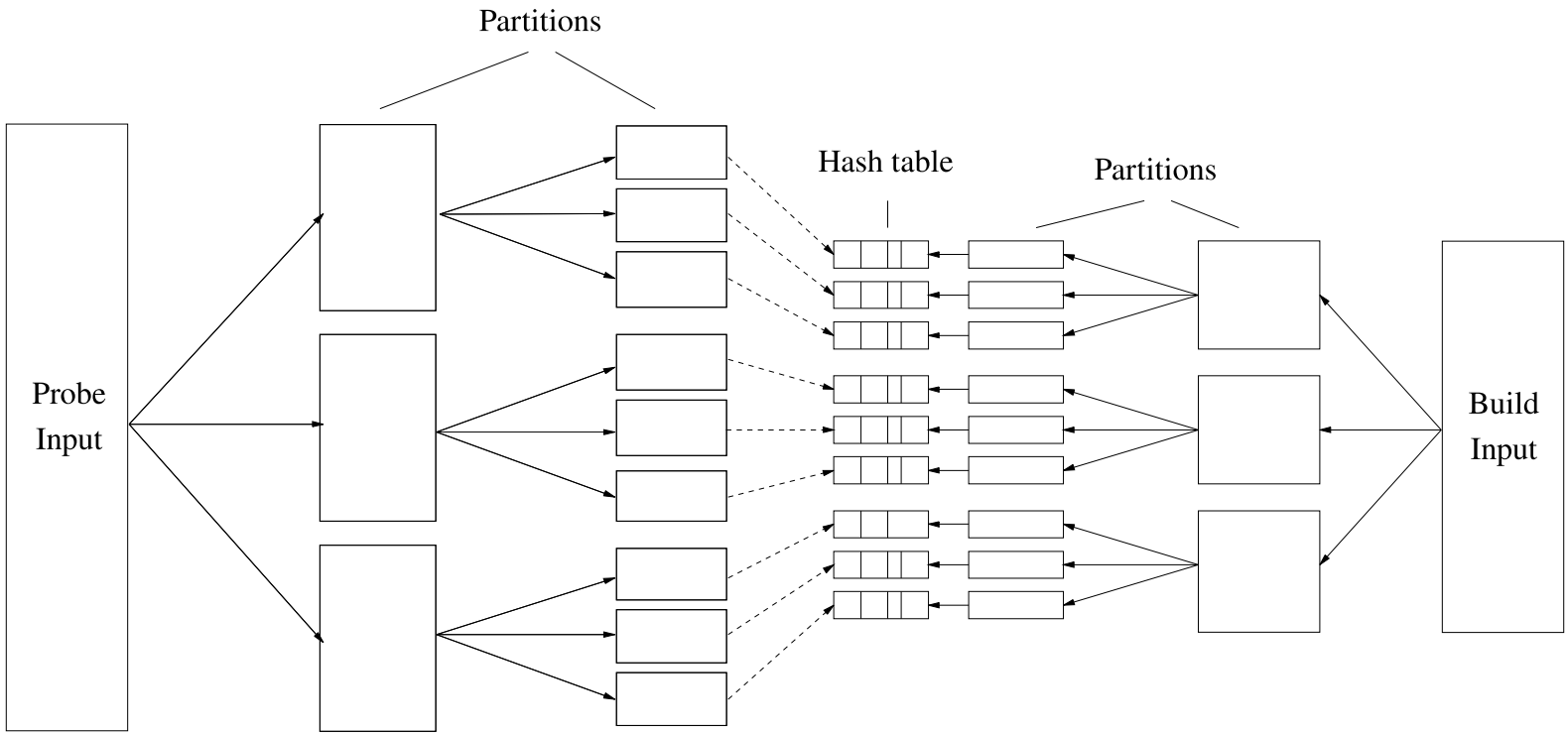
Index-Join(3)

- Disadvantages of the index join:
  - no index structures exist on intermediate results
  - creating them temporarily is usually too expensive
  - index lookups are usually too expensive

- Idea: partitioning the relations
- Building *main-memory* index structures per partition

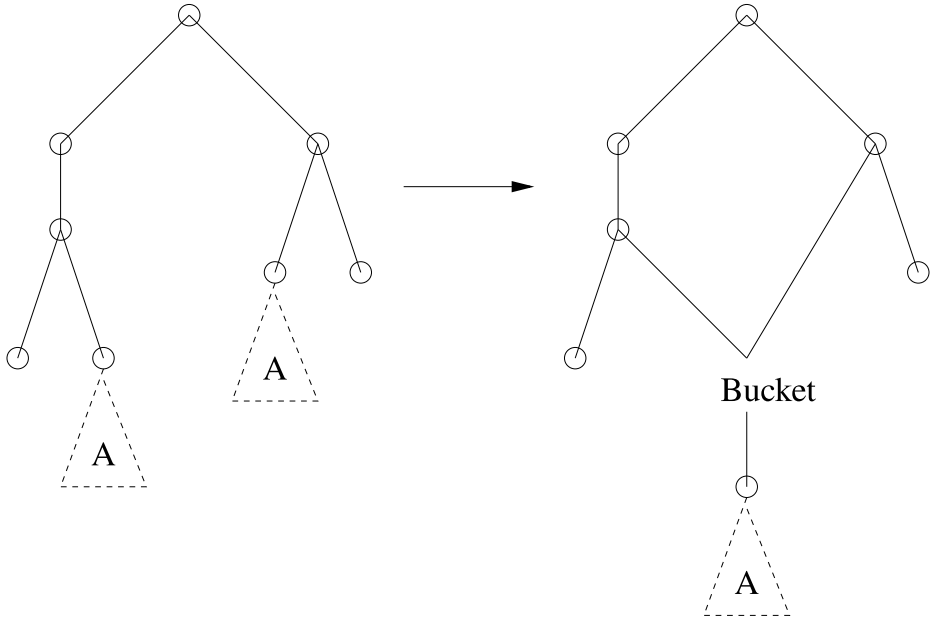


Partitioning Relations

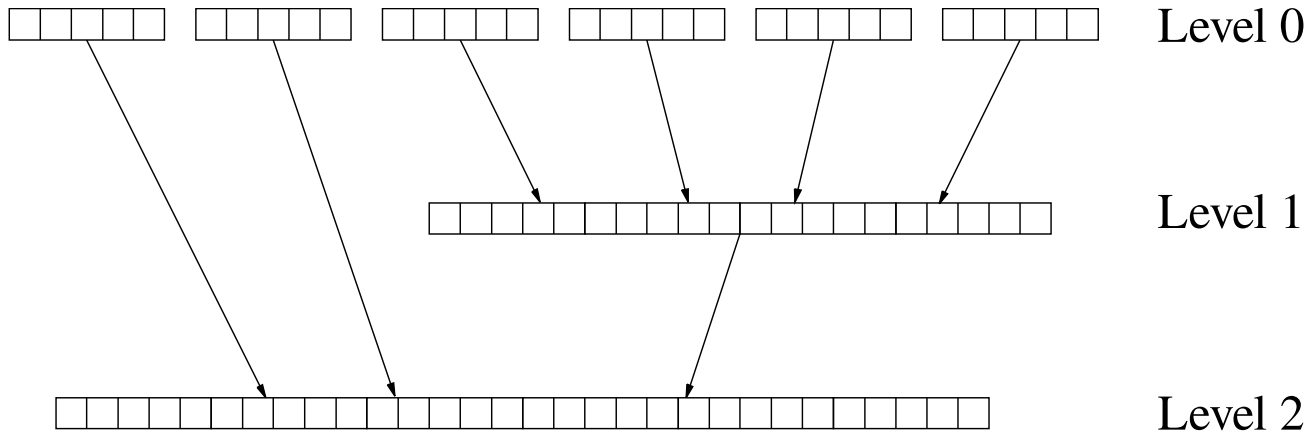


Materialization

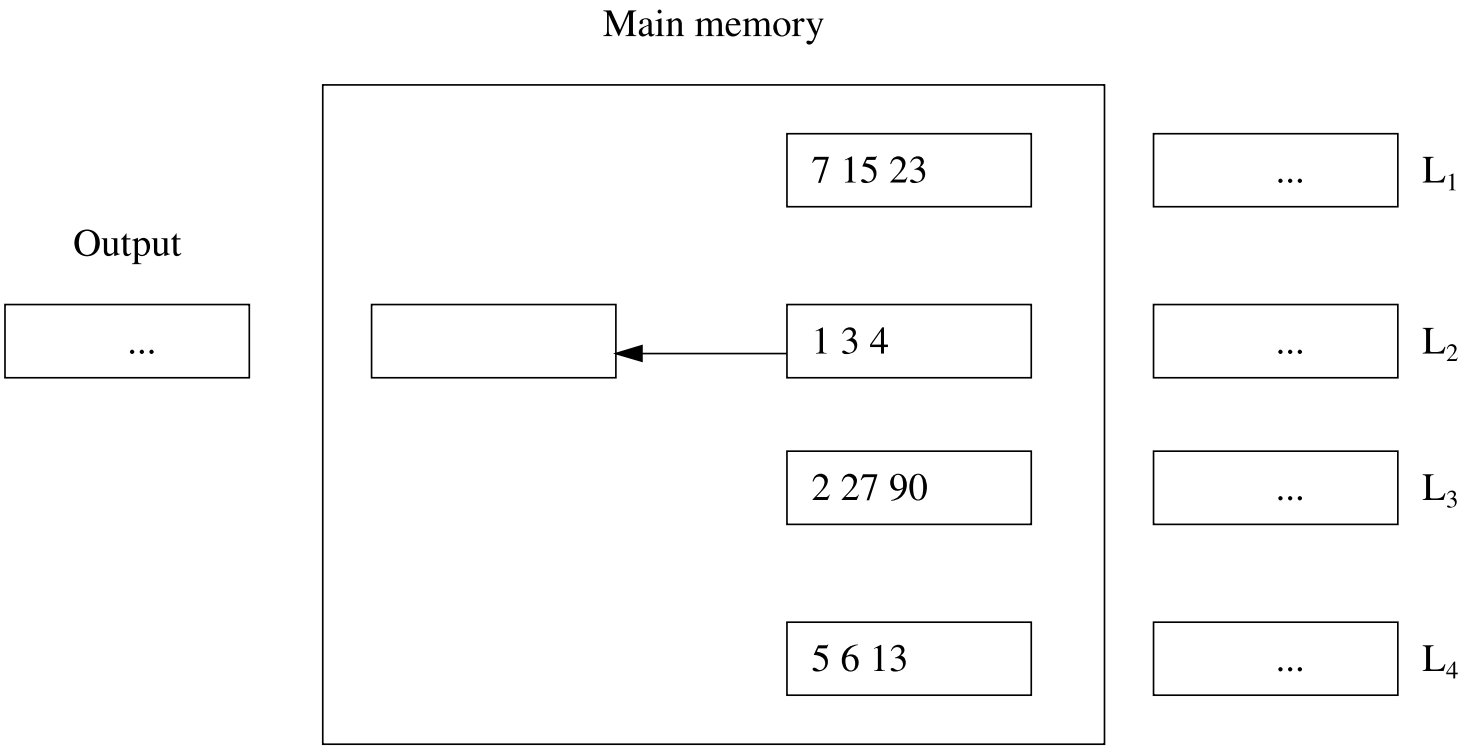
- When several operations with high main-memory consumption occur (e.g., hash join)
- When common subexpressions are to be eliminated



Mergesort



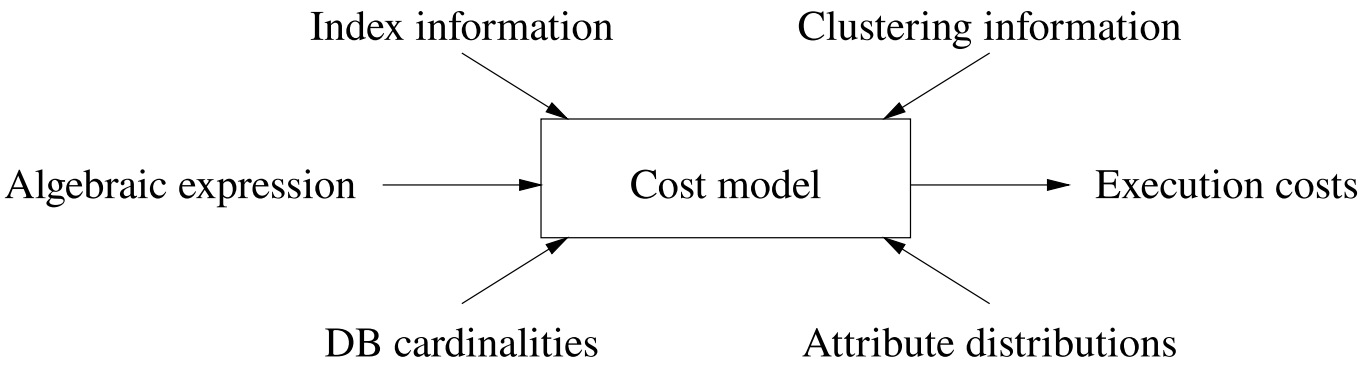
One Merge Step



Replacement Selection

Output						Memory				Input					
						10	20	30	40	25	73	16	26	33	50	31
					10	20	25	30	40	73	16	26	33	50	31	
				10	20	25	30	40	73	16	26	33	50	31		
			10	20	25	(16)	30	40	73	26	33	50	31			
		10	20	25	30	(16)	(26)	40	73	33	50	31				
	10	20	25	30	40	(16)	(26)	(33)	73	50	31					
10	20	25	30	40	73	(16)	(26)	(33)	(50)	31						
					16	26	31	33	50							

Cost Models


Index information
Clustering information
Algebraic expression
Cost model
Execution costs
DB cardinalities
Attribute distributions

Selectivities

- Fraction of qualifying tuples of an operation
- Selection with condition p:

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

- Join of R with S:

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

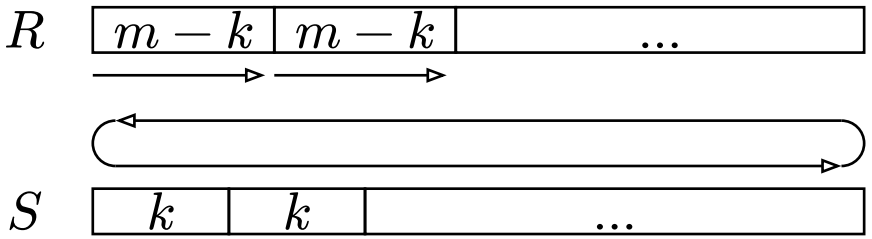
Selectivities(2)

- Estimating the selectivity:
  - $sel_{R.A=C} = \frac{1}{|R|}$ if A is a key of R
  - $sel_{R.A=C} = \frac{1}{i}$ if i is the number of attribute values of $R.A$ (uniform distribution)
  - $sel_{R.A=S.B} = \frac{1}{|R|}$ for an equi-join of R with S via a foreign key in S
- Otherwise, e.g., sampling

Estimating Selection Costs

- Brute force: reading all pages of R
- B$^+$-tree index: $t + \lceil sel_{Athetac} \cdot b_R \rceil$
  - Descending the index structure
  - Reading the qualifying tuples
- Hash index: one lookup for each value satisfying the condition

Estimating Join Costs



- Scanning all pages of R: b_R
- Iterations of the inner loop: $\lceil b_R/(m-k) \rceil$
- Total: $b_R + k + \lceil b_R/(m-k) \rceil \cdot (b_S - k)$
- minimal if $k = 1$ and R is the smaller relation

Join Order

One of the most important optimizations is the choice of the join order

- Joins occur very frequently (almost always)
- Joins are relatively expensive
- Joins change the number of tuples
- the join order has an enormous impact on the runtime

Practically all database systems optimize the join order. The problem is NP-hard $\Rightarrow$ often only heuristics

Example

As an example, we consider the nested-loop join. Simplified cost function:

$$C(e_1 \bowtie^{NL} e_2) = \mid e_1 \mid \mid e_2 \mid$$

Statistics of an example query:

$$\begin{aligned} \mid R_1 \mid &= 10 \\ \mid R_2 \mid &= 100 \\ \mid R_3 \mid &= 1000 \\ sel_{R_1 \bowtie R_2} &= 0.1 \\ sel_{R_1 \bowtie R_3} &= 1 \\ sel_{R_2 \bowtie R_3} &= 0.2 \end{aligned}$$

Example(2)

Costs of possible plans:

	$C_{\bowtie^{nl}}$
$R_1 \bowtie R_2$	1000
$R_2 \bowtie R_3$	100000
$R_1 \bowtie R_3$	10000
$(R_1 \bowtie R_2) \bowtie R_3$	101000
$(R_2 \bowtie R_3) \bowtie R_1$	300000
$(R_1 \bowtie R_3) \bowtie R_2$	1010000

- huge differences between the costs
- inferring the total cost from parts is difficult
- $R_1 \bowtie R_3$ initially seems to be better than $R_2 \bowtie R_3$ etc.

Complex optimization problem!

Greedy Heuristics

- Determining the optimal join order is NP-hard
- very expensive for large queries
- therefore often heuristics

Simple approach:

- start with one relation
- join the “cheapest” relation to it
- repeat until all are joined

Avoids the biggest mistakes

Greedy Heuristics (2)

Simple strategy to order $R = \{R_1, ..., R_n\}$:

1. choose $R_i \in R$ with $\mid R_i \mid$ minimal
2. $S = R_i$ and $R = R \setminus \{R_i\}$.
3. while $\mid R \mid > 0$
   3.1. choose $R_i \in R$ with $C(S \bowtie R_i)$ minimal.
   3.2. $S = S \bowtie R_i$. $R = R \setminus \{R_i\}$
4. return S as join order

greatly simplified, only produces left-deep trees, often suboptimal

Greedy Heuristics (3)

Greedily optimizing costs is often not good:

- Joins influence subsequent operators
- popular strategy: minimize $sel_{\bowtie}$
- often works well

Alternative:

- minimize $\frac{1-sel_{\bowtie}}{C_{\bowtie}}$
- restrictions due to the join predicates
- “proper” algorithm more complex than greedy
- optimal in some cases

Many more variants exist.

Standard technique for the optimal join order: dynamic programming (DP)

> Principle of optimality: The optimal solution of the overall problem can be composed of optimal solutions of subproblems.

General procedure:

- first solve simple subproblems optimally
- solve increasingly complex problems, using known solutions

Dynamic Programming (2)

1 create an empty DP table
2 enter the base relations as optimal solutions of size 1
3 for all problem sizes s from 2 to n
4 for all solved problems (l, r) such that $\mid l \mid + \mid r \mid = s$
5 is $l \bowtie r$ possible?
6 if not, continue with 4
7 $T = \{x \mid x \text{ relation in } l \text{ or in } r\}$
8 $p = l \bowtie r$
9 is $dpTable[T]$ empty or $C(p) < C(dpTable[T])$?
10 if so, $dpTable[T] = p$
11 return $dpTable[\{x \mid x \text{ is base relation}\}]$ as optimal solution

- considers all relevant combinations of relations
- line 5 prevents invalid joins, typically also cross products

Dynamic Programming (3)

- finds the optimal solution
- but exponential runtime in the worst case
- no longer feasible for very large/complex queries
- but there are very efficient DP algorithms (more complex than the one presented here)

- basic assumption: the cost model is correct
- often requires a `run stats`, `analyze`, or similar!
- even then large estimation errors are possible

"Real" algorithms choose not only the order but also the physical operators

Summary

- Query processing and optimization are important tasks of a DBMS
- It also helps users to know about them, since design decisions and query formulation have an impact on the performance of a DBMS