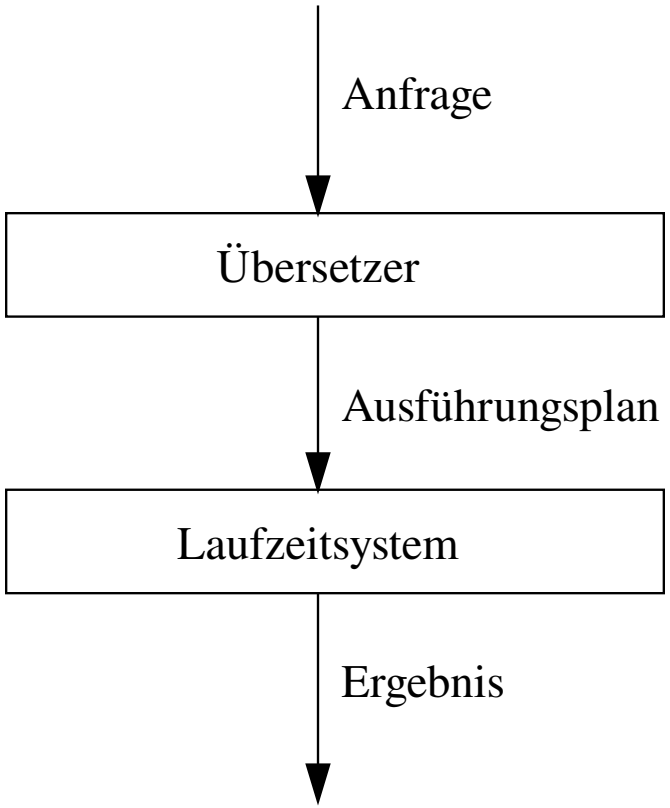
Anfragebearbeitung

Anfragebearbeitung

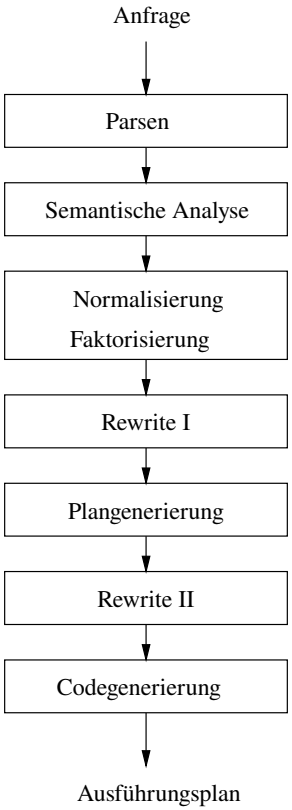
Übersicht


Anfrage
Übersetzer
Ausführungsplan
Laufzeitsystem
Ergebnis

Übersetzung

- SQL ist deklarativ, irgendwann muss Anfrage aber für Laufzeitsystem in etwas Prozedurales übersetzt werden
- DBMS übersetzt SQL in eine interne Darstellung
- Ein weit verbreiteter Ansatz ist die Übersetzung in eine relationale Algebra

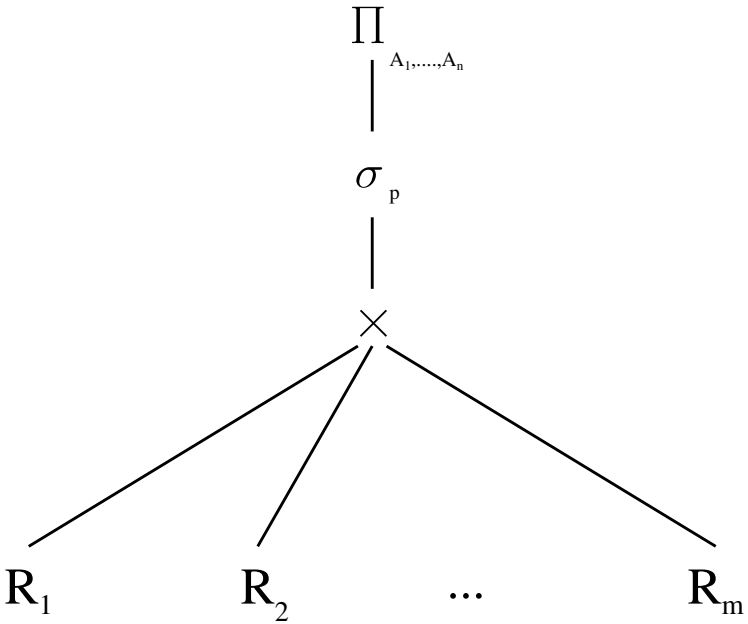
Übersetzung(2)



Kanonische Übersetzung

- Es gibt eine Standardübersetzung von SQL in relationale Algebra
- Algebraausdrücke werden oft auch graphisch repräsentiert

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



Erweiterte Übersetzung

Erweiterungen zur “klassischen” kanonischen Übersetzung:

- `select a, sum(d) as s from ... group by a, b, c`
  - $\Pi_{a,s}\left(\Gamma_{a,b,c;s:\sum(d)}(C)\right)$
  - wobei C die kanonische Übersetzung des inneren Teils ist
  - auf die Projektionsklausel achten!
- `select ... having p`
  - $\sigma_p(C)$, wobei C die Übersetzung inklusive group by ist
- `select ... order by a, b, c`
  - $sort_{a,b,c}(C)$, wobei C die Übersetzung inklusive having ist
  - $sort$ tauchte in relationaler Algebra nicht auf, weil Relationen unsortiert sind

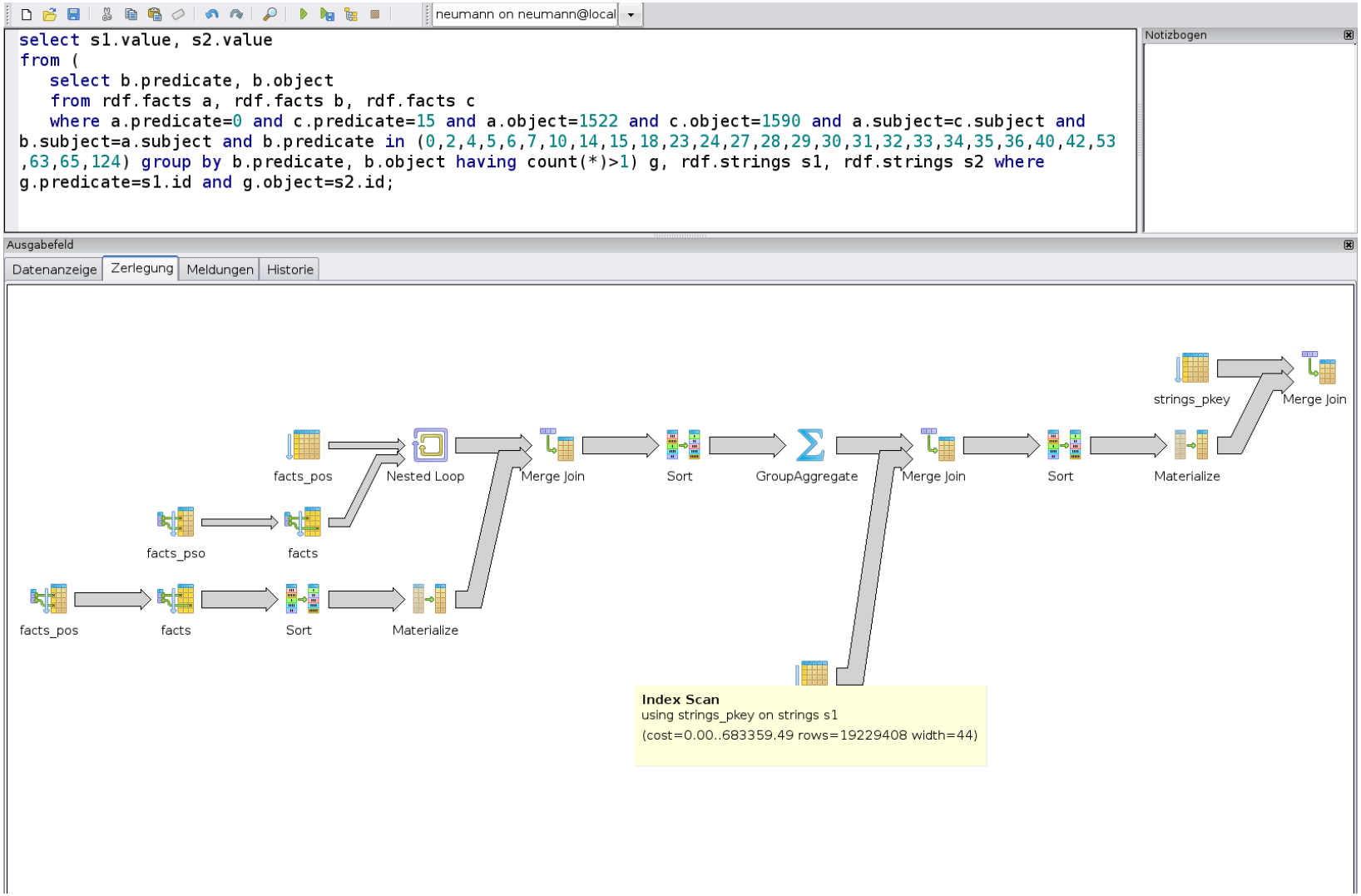
Optimierung

- Kanonischer Plan ist nicht sehr effizient (z.B. enthält er Kreuzprodukte)
- DBMS besitzt Optimierer, um einen Plan in eine effizientere Form zu überführen
- Das Finden eines Plans ist ein sehr schwieriges Problem, immer noch Gegenstand aktueller Forschung

Optimierung(2)

- Was hat ein gewöhnlicher Benutzer mit Anfrageoptimierung zu tun?
- DBMS Optimierer produziert manchmal suboptimale Pläne
- Die meisten DBMSe geben dem Benutzer Einblick in die generierten Pläne
- Benutzer kann generierten Plan analysieren und gegebenenfalls die Anfrage umbauen oder dem DBMS Hinweise zur Ausführung geben

Visualisierung von Plänen



Anfrageoptimierung 1x1

- DBMS kann die Kosten für die Ausführung eines Operators mit Hilfe von Kostenmodellen und Statistiken abschätzen
- Bei der Optimierung eines Plans werden Heuristiken angewandt, alle möglichen Pläne anzuschauen ist viel zu teuer
- Optimierung kann auf verschiedenen Ebenen stattfinden:
  - Logische Ebene
  - Physische Ebene

Logische Ebene

- Ausgangspunkt ist relationaler Algebraausdruck, der nach kanonischer Übersetzung entstanden ist
- Optimierung: Trans-formation relationaler Algebraausdrücke in äquivalente Ausdrücke (die zu schnellerem Ausführungsplan führen)
- Umformungen sollten so gewählt werden, dass die Ausgaben der einzelnen Operatoren möglichst klein werden

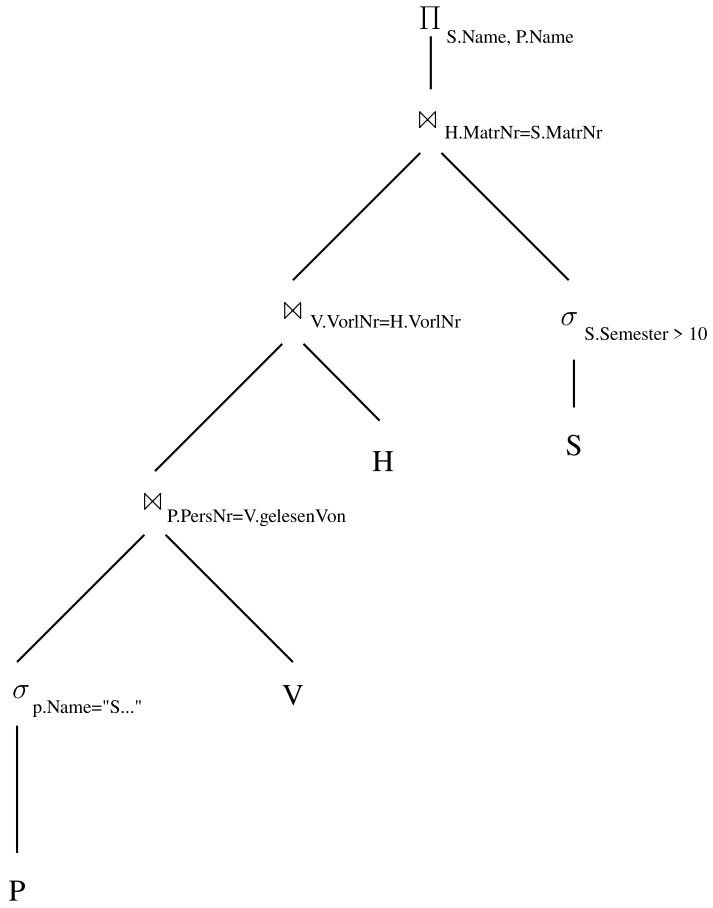
Logische Ebene(2)

- Grundlegende Techniken:
  - Aufbrechen von Selektionen
  - Verschieben von Selektionen nach “unten” im Plan
  - Zusammenfassen von Selektionen und Kreuzprodukten zu Joins
  - Bestimmung der Joinreihenfolge
  - Einfügen von Projektionen
  - Verschieben von Projektionen nach “unten” im Plan

Beispielanfrage

```
select S.Name, P.Name
from   Studenten S, hören H, Vorlesungen V, Professoren P
where  S.MatrNr = H.MatrNr
and    H.VorlNr = V.VorlNr
and    V.gelesenVon = P.PersNr
and    S.Semester > 10
and    P.Name = 'Sokrates';
```

(Optimierter) Anfrageplan



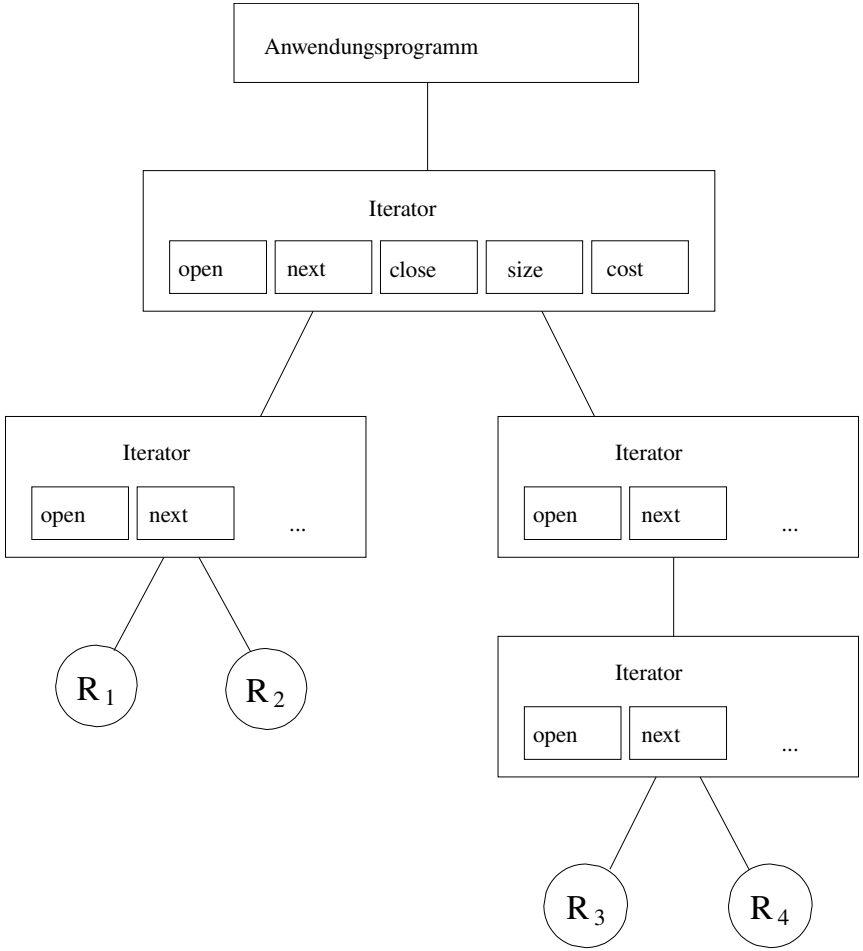
Auszug aus Regeln

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

Physische Optimierung

- Man unterscheidet zwischen logischen Algebraoperatoren und physischen Algebraoperatoren
- Physische Algebraoperatoren stellen die Realisierung der logischen dar
- Es kann mehrere physische für einen logischen Operator geben
- Optimierung auf der physischen Ebene bedeutet, einen dieser Operatoren auszuwählen, zu entscheiden, ob Indexe benutzt werden sollen, Zwischenergebnisse zu materialisieren, etc.

Iteratorkonzept



Implementierung Selektion

* **iterator** Scan_p

- **open**
  - Öffne Eingabe
- **next**
  - Hole solange nächstes Tupel, bis eines die Bedingung p erfüllt
  - Gib dieses Tupel zurück
- **close**
  - Schließe Eingabe

Implementierung Selektion(2)

iterator IndexScan$_p$

- **open**
  - Schlage im Index das erste Tupel nach, das die Bedingung erfüllt
  - Öffne Eingabe
- **next**
  - Gib nächstes Tupel zurück, falls es die Bedingung p noch erfüllt
- **close**
  - Schließe Eingabe

Implementierung Join

- Mengendifferenz und -durchschnitt können analog zum Join implementiert werden
- hier nur Equi-Joins

Nested-Loop-Join:

for each $r \in R$

  for each $s \in S$

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

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

Implementierung Join(2)

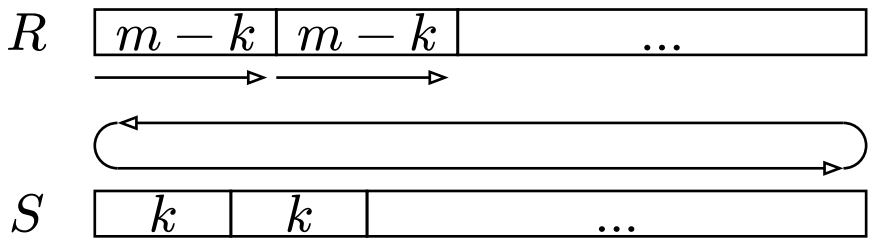
iterator NestedLoop$_p$

- **open**
  - Öffne die linke Eingabe
- **next**
  - Rechte Eingabe geschlossen?
    - Öffne sie
  - Fordere rechts solange Tupel an, bis Bedingung p erfüllt ist
  - Sollte zwischendurch rechte Eingabe erschöpft sein
    - Schließe rechte Eingabe
    - Hole nächstes Tupel der linken Eingabe
    - Starte **next** neu
  - Gib den Verbund von aktuellem linken und aktuellem rechten Tupel zurück
- **close**
  - Schließe beide Eingabequellen

Verfeinerter Joinalgorithmus

- Relationen sind *seitenweise* abgespeichert
- Es stehen m Pufferrahmen im Hauptspeicher zur Verfügung:
  - k für die innere Schleife des Nested Loop
  - $m - k$ für die äußere

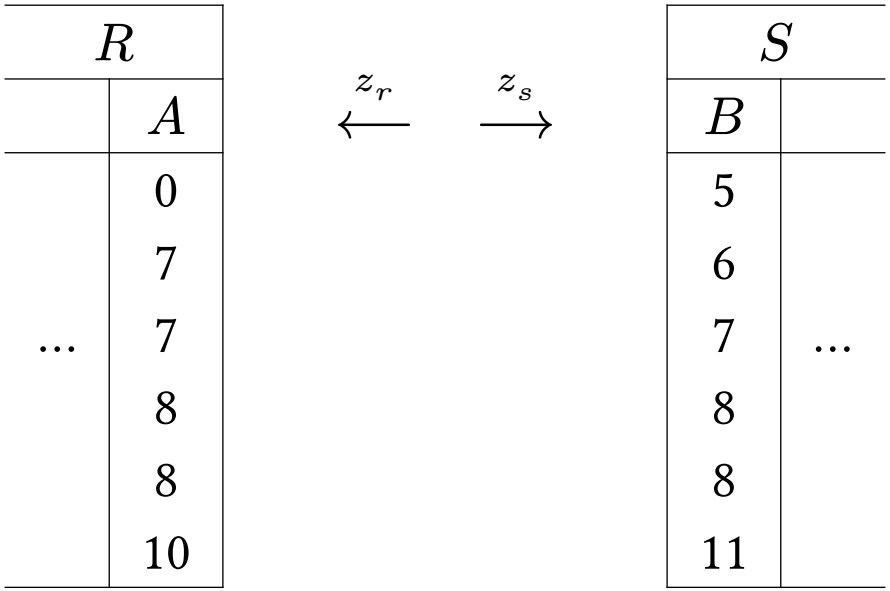
Join von R und S:



(Sort-)Merge-Join

- Voraussetzung: R und S sind sortiert (notfalls vorher sortieren)

Beispiel:



(Sort-)Merge-Join(2)

iterator MergeJoin$_p$

- **open**
  - Öffne beide Eingaben
  - Setze akt auf linke Eingabe
  - Markiere rechte Eingabe
- **close**
  - Schließe beide Eingabequellen

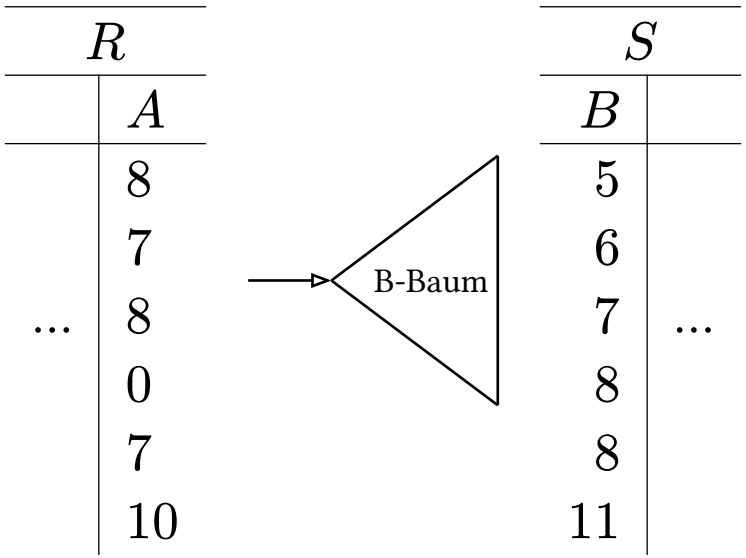
iterator MergeJoin$_p$

- **next**
  - Solange Bedingung nicht erfüllt
    - Setze akt auf Eingabe mit dem kleinsten anliegenden Wert im Joinattribut
    - Rufe **next** auf akt auf
    - Markiere andere Eingabe
  - Verbinde linkes und rechtes Tupel
  - Bewege andere Eingabe vor
  - Ist Bedingung nicht mehr erfüllt oder andere Eingabe erschöpft?
    - Bewege akt vor
    - Wert des Joinattributes in akt verändert?

      $\Rightarrow$ Nein, dann setze andere Eingabe auf Markierung zurück

      $\Rightarrow$ Ansonsten markiere die andere Eingabe

Index-Join



Index-Join(2)

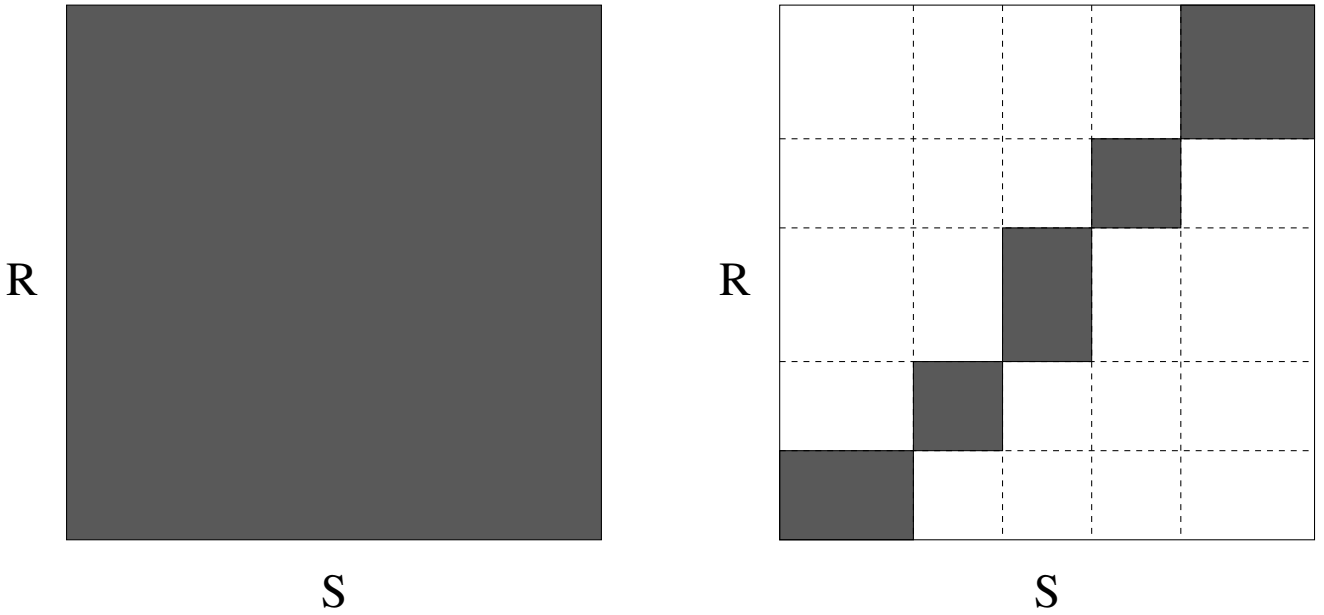
iterator IndexJoin$_p$

- **open**
  - Öffne die linke Eingabe
  - Hole erstes Tupel aus linker Eingabe
  - Schlage Joinattributwert im Index nach
- **next**
  - Bilde Join, falls Index (weiteres) Tupel zu diesem Wert liefert
  - Ansonsten bewege linke Eingabe vor und schlage Joinattributwert im Index nach
- **close**
  - Schließe die Eingabe

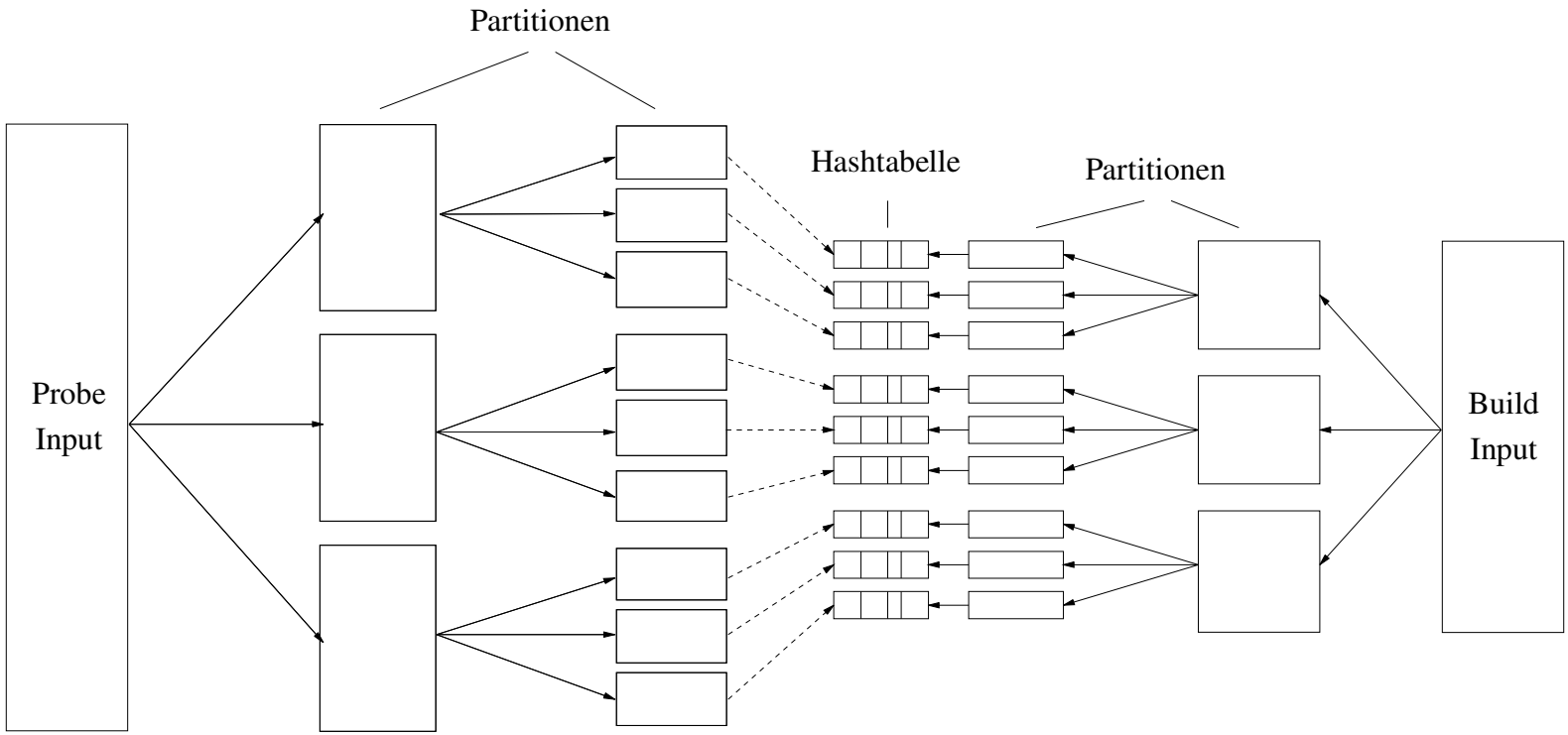
Index-Join(3)

- Nachteile des Index-Joins:
  - auf Zwischenergebnissen existieren keine Indexstrukturen
  - temporäres Anlegen i.A. zu aufwendig
  - Nachschlagen im Index i.A. zu aufwendig

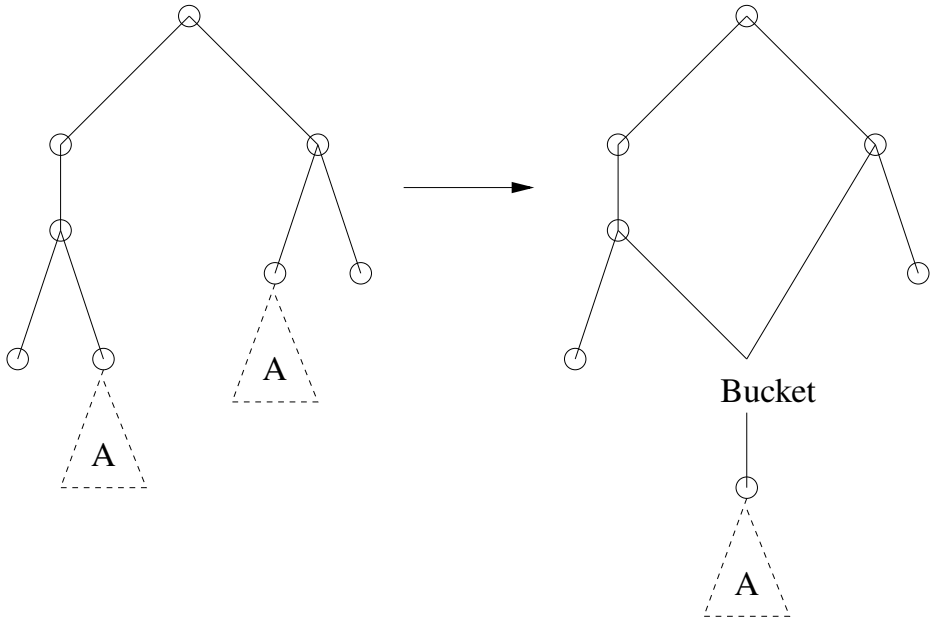
- Idee: Partitionieren der Relationen
- Anlegen von *Hauptspeicher*-Indexstrukturen je Partition



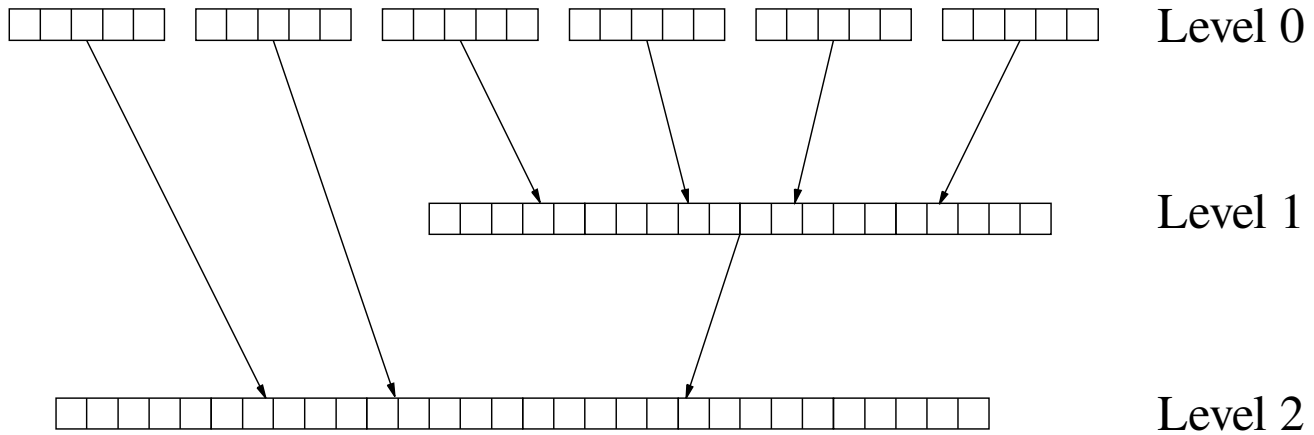
Partitionierung von Relationen



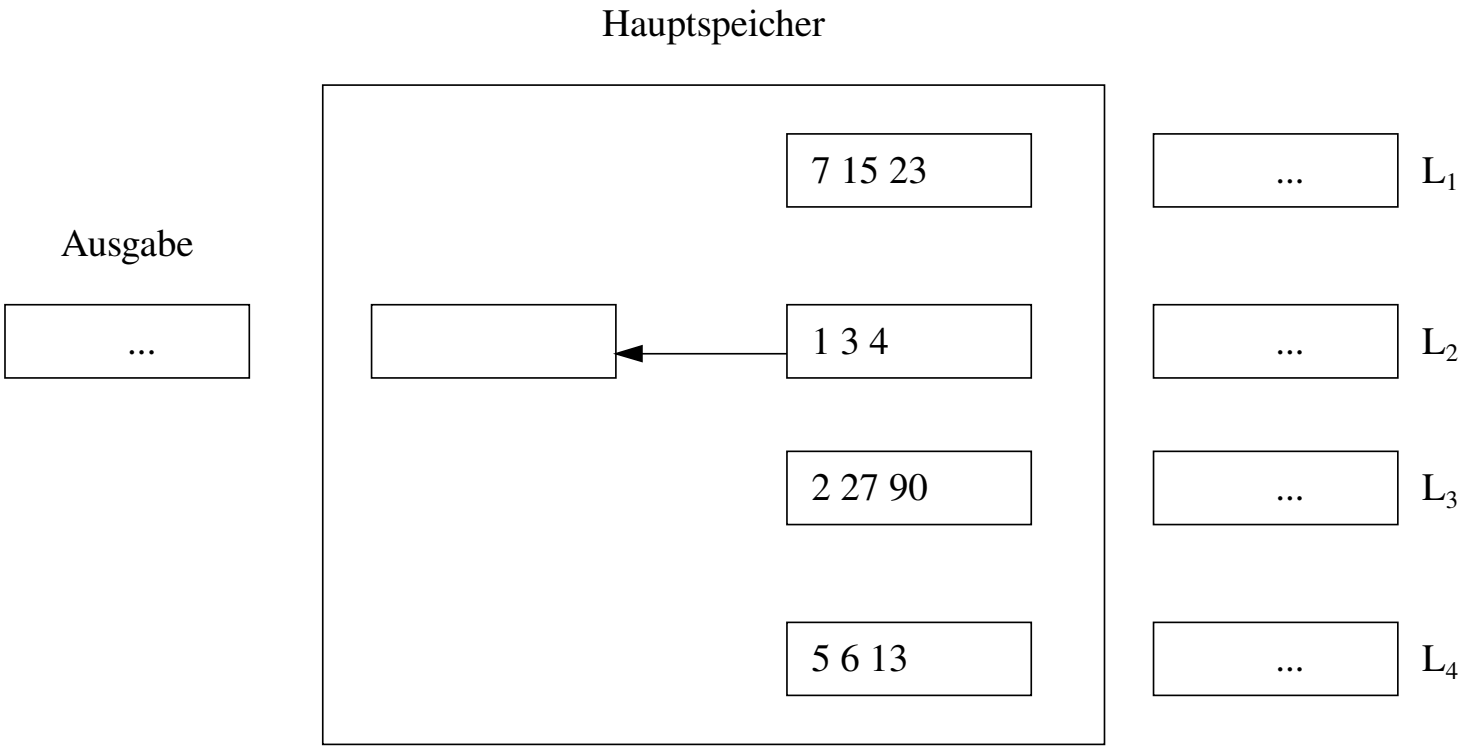
- Wenn mehrere Operationen mit hohem Hauptspeicherverbrauch vorkommen (z.B. Hash-Join)
- Wenn gemeinsame Teilausdrücke eliminiert werden sollen



Mergesort

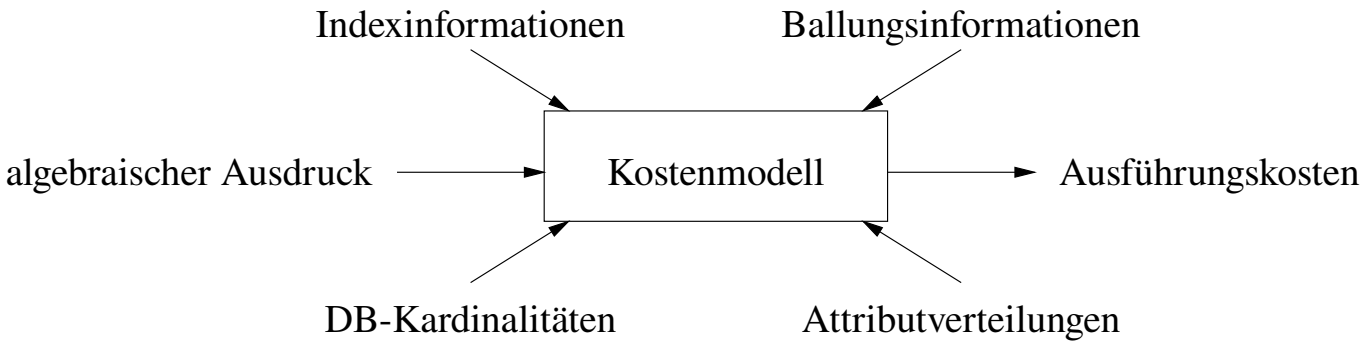


Ein Mischvorgang



Replacement Selection

Ausgabe					Speicher				Eingabe						
					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	30	(16)	30	40	73	26	33	50	31		
	10	20	25	30	40	(16)	(26)	40	73	33	50	31			
10	20	25	30	40	73	(16)	(26)	(33)	73	50	31				
					16	26	31	33	50						


Indexinformationen
Ballungsinformationen
algebraischer Ausdruck
Kostenmodell
Ausführungskosten
DB-Kardinalitäten
Attributverteilungen

Selektivitäten

- Anteil der qualifizierenden Tupel einer Operation
- Selektion mit Bedingung p:

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

- Join von R mit S:

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

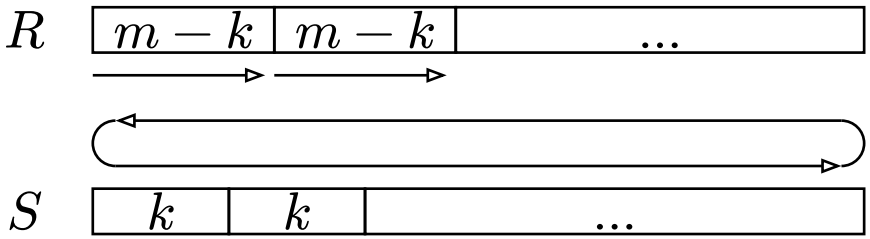
Selektivitäten(2)

- Abschätzung der Selektivität:
  - $sel_{R.A=C} = \frac{1}{|R|}$ falls A Schlüssel von R
  - $sel_{R.A=C} = \frac{1}{i}$ falls i die Anzahl der Attributwerte von $R.A$ ist (Gleichverteilung)
  - $sel_{R.A=S.B} = \frac{1}{|R|}$ bei Equijoin von R mit S über Fremdschlüssel in S
- Ansonsten z.B. Stichprobenverfahren

Abschätzung Selektionskosten

- Brute Force: Lesen aller Seiten von R
- B$^+$-Baum-Index: $t + \lceil sel_{Athetac} \cdot b_R \rceil$
  - Absteigen in der Indexstruktur
  - Lesen der qualifizierenden Tupel
- Hash-Index: für jeden die Bedingung erfüllenden Wert einen Look-up

Abschätzung Joinkosten



- Durchlaufen aller Seiten von R: b_R
- Durchläufe der inneren Schleife: $\lceil b_R/(m-k) \rceil$
- Insgesamt: $b_R + k + \lceil b_R/(m-k) \rceil \cdot (b_S - k)$
- minimal, falls $k = 1$ und R die kleinere Relation

Joinreihenfolge

Eine der wichtigsten Optimierungen ist die Wahl der Joinreihenfolge

- Joins kommen sehr häufig vor (fast immer)
- Joins sind relativ teuer
- Joins verändern die Zahl der Tupel
- die Joinreihenfolge hat enormen Einfluss auf die Laufzeit

Praktisch alle Datenbanksysteme optimieren die Joinreihenfolge. Das Problem ist NP-hart $\Rightarrow$ häufig nur Heuristiken

Wir betrachten als Beispiel den Nested-Loop-Join. Vereinfachte Kostenfunktion:

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

Statistiken einer Beispielanfrage:

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

Beispiel(2)

Kosten für mögliche Pläne:

	$C_{\bowtie^{nl}}$
$R_1 \bowtie R_2$	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

- riesige Unterschiede zwischen den Kosten
- Rückschlüsse von Teilen auf die Gesamtkosten schwierig
- $R_1 \bowtie R_3$ scheint zunächst besser zu sein als $R_2 \bowtie R_3$ usw.

Komplexes Optimierungsproblem!

Greedy-Heuristiken

- Bestimmung der optimalen Joinreihenfolge ist NP-hart
- für große Anfragen sehr aufwendig
- deshalb häufig Heuristiken

Einfacher Ansatz:

- beginne mit einer Relation
- joine die “günstigste” Relation dazu
- wiederhole, bis alle gejoined sind

Vermeidet die größten Fehler

Greedy-Heuristiken (2)

Einfache Strategie, um $R = \{R_1, ..., R_n\}$ anzuordnen:

1. wähle $R_i \in R$ mit $\mid R_i \mid$ minimal
2. $S = R_i$ und $R = R \setminus \{R_i\}$.
3. solange $\mid R \mid > 0$
   3.1. wähle $R_i \in R$ mit $C(S \bowtie R_i)$ minimal.
   3.2. $S = S \bowtie R_i$. $R = R \setminus \{R_i\}$
4. liefere S als Joinreihenfolge

stark vereinfacht, liefert nur links-tiefe Bäume, oft suboptimal

Greedy-Heuristiken (3)

Greedy Kosten zu optimieren oft nicht gut:

- Joins beeinflussen nachfolgende Operatoren
- beliebte Strategie: minimiere $sel_{\bowtie}$
- funktioniert oft gut

Alternative:

- minimiere $\frac{1-sel_{\bowtie}}{C_{\bowtie}}$
- Beschränkungen durch die Joinprädikate
- “richtiger” Algorithmus komplexer als Greedy
- in manchen Fällen optimal

Noch sehr viele Varianten.

Standardtechnik für die optimale Joinreihenfolge: Dynamisches Programmieren (DP)

> Optimalitätsprinzip: Die optimale Lösung des Gesamtproblems kann aus optimalen Lösungen von Teilproblemen zusammengesetzt werden.

Generelles Vorgehen:

- löse zunächst einfache Teilprobleme optimal
- löse immer kompliziertere Probleme und nutze dabei bekannte Lösungen

Dynamisches Programmieren (2)

1 erzeuge eine leere DP Tabelle
2 trage die Basisrelationen als optimale Lösungen der Größe 1 ein
3 für alle Problemgrößen s von 2 bis n
4 für alle gelösten Probleme (l, r), so dass $\mid l \mid + \mid r \mid = s$
5 ist $l \bowtie r$ möglich?
6 wenn nein, weiter bei 4
7 $T = \{x \mid x \text{ Relation in } l \text{ oder in } r\}$
8 $p = l \bowtie r$
9 ist $dpTabelle[T]$ leer oder $C(p) < C(dpTabelle[T])$?
10 wenn ja, $dpTabelle[T] = p$
11 liefere $dpTabelle[\{x \mid x \text{ ist Basisrelation}\}]$ als optimale Lösung

- betrachtet alle relevanten Relationenkombinationen
- Zeile 5 verhindert ungültige Joins, typischerweise auch Kreuzprodukte

Dynamisches Programmieren (3)

- findet die optimale Lösung
- aber im worst case exponentielle Laufzeit
- für sehr große/komplexe nicht mehr durchführbar
- es gibt aber sehr effiziente DP Algorithmen (komplexer als der hier vorgestellte)

- grundlegende Annahme: Kostenmodell stimmt
- erfordert oft ein `run stats`, `analyze` o.ä.!
- selbst dann noch große Schätzfehler möglich

"Echte" Algorithmen wählen nicht nur die Reihenfolge, sondern auch die physischen Operatoren

Zusammenfassung

- Anfragebearbeitung und -optimierung sind wichtige Aufgaben eines DBMS
- Es hilft auch, als Benutzer Ahnung davon zu haben, da Entwurfsentscheidungen und Anfrageformulierung einen Einfluss auf die Performanz eines DBMS haben