Datenbankanwendung

Wintersemester 2014/15

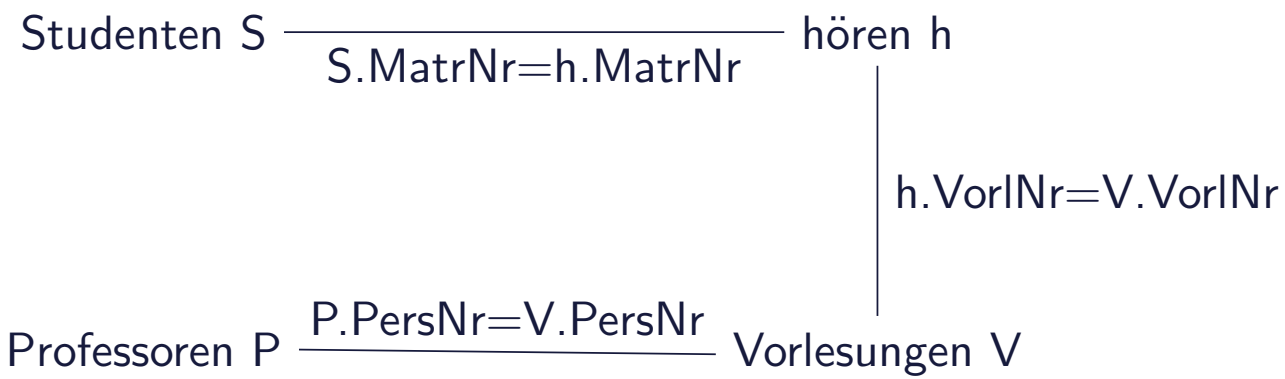
Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Wiederholung: Anfragegraph

Anfragen dieses Typs können als Graph dargestellt werden:

- Der Anfragegraph ist ein ungerichteter Graph mit $R_1, \ldots, R_n$ als Knoten
- Ein Pädiktat der Form $a_1 = a_2$, wobei $a_1 \in R_i$ und $a_2 \in R_j$ erzeugt eine Kante zwischen R_i und R_j, beschriftet mit dem Prädikat

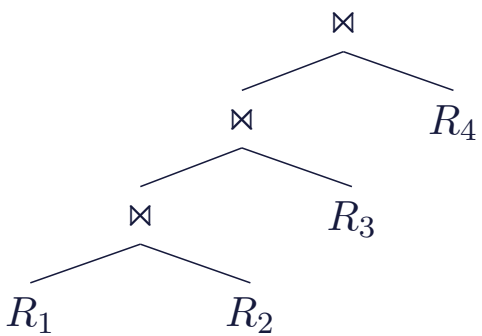


Für zwei Relationen, die nicht via einer Kante verbunden sind, kann nur ein Kreuzprodukt berechnet werden. Wir unterscheiden später ob Kreuzpodukte überhaupt zugelassen werden oder nicht.

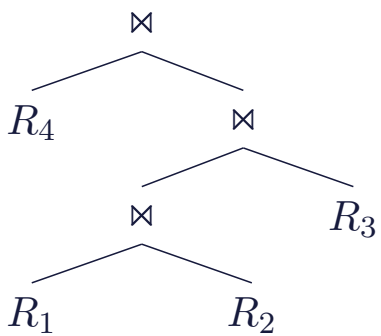
Wiederholung: Gestalt von Join-Bäumen

- links-tiefer Baum
- rechts-tiefer Baum
- zigzag Baum (mindestens eine Eingabe ist eine Relation)
- buschiger (bushy) Baum

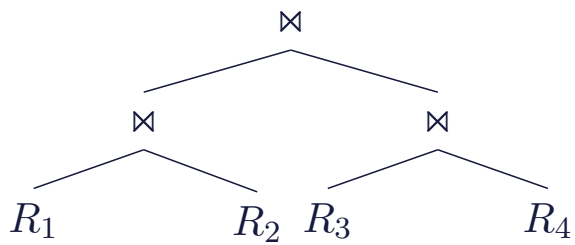
Die ersten drei Klassen werden auch zusammengefasst als lineare Bäume.



(links-tief)



(zigzag)



(buschig)

Klassifikation der Join-Ordering-Probleme

Die *hier* betrachteten Probleme können anhand der folgenden Kriterien klassifiziert werden:

1. Anfragegraph: *Kette*, *Cycle*, *Stern* und *Clique*
2. Struktur des Join-Baums: *links-tief*, *zigzag* oder *buschig* Bäume
3. Kreuzprodukte: *mit* oder *ohne* Kreuzprodukte

Wiederholung(?): Catalan-Zahl

Die Anzahl von Binärbäumen mit n Blättern ist gegeben durch $\mathcal{C}(n-1)$, wobei $\mathcal{C}(n)$ definiert ist durch

$$\mathcal{C}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{k=0}^{n-1} \mathcal{C}(k)\mathcal{C}(n-k-1) & \text{if } n > 0 \end{cases}$$

Dies kann in geschlossener Form geschrieben werden als

$$\mathcal{C}(n) = \frac{1}{n+1}\binom{2n}{n}$$

Die Catalan-Zahlen wachsen in der Ordnung von $\Theta(4^n/n^{\frac{3}{2}})$

Anzahl Join-Bäume mit Kreuzprodukte

links-tief		$n!$
rechts-tief		$n!$
zigzag		$n!2^{n-2}$
buschig		$n!\mathcal{C}(n-1)$
	$=$	$\frac{(2n-2)!}{(n-1)!}$

- Idee: Anzahl der Kombination der Blätter $(n!) * $ Möglichkeiten einen Baum zu bilden
- (bei zigzag im Vergleich zu links bzw. rechtstief: Vertauschen von Eingaben möglich, d.h. Join oder Relation links oder rechts)
- Wächst exponentiell
- und je flexibler die Baumstruktur ist, desto schneller

Ketten Anfragen, keine Kreuzprodukte

Wir bezeichnen mit $f(n)$ die Anzahl links-tiefer Bäume für eine Ketten Anfrage $R_1 - \ldots - R_n$

- Offensichtlich gilt $f(0) = 1, f(1) = 1$
- Für $n > 1$, R_n kann zu allen Bäumen für $R_1 - \ldots - R_{n-1}$ hinzugefügt werden
- R_n kann also an jede Position nach(!) R_{n-1} eingefügt werden
- Sei k (in $[1, n-1]$) die Position von R_{n-1} von unten betrachtet
- Es gibt $n - k$ Join-Bäume für Einfügen von R_n nach R_{n-1}
- Plus einen Weiteren für $k = 1$, da R_n auch vor R_{n-1} eingefügt werden kann
- Wenn R_{n-1} auf Platz k, dann $f(k-1)$ Bäume darunter.

Also insgesamt $(n > 1)$:

$$f(n) = 1 + \sum_{k=1}^{n-1} f(k-1) * (n-k)$$

Ketten-Anfragen, keine Kreuzprodukte (2)

Die Anzahl für links-tiefe Bäume für Ketten-Anfragen der Größe n ist

$$f(n) = \begin{cases} 1 & \text{if } n < 2 \\ 1 + \sum_{k=1}^{n-1} f(k-1) * (n-k) & \text{if } n \geq 2 \end{cases}$$

Kann in geschlossener Form geschrieben werden als

$$f(n) = 2^{n-1}$$

- Generalisierung für zigzag wie zuvor

Ketten-Anfragen, keine Kreuzprodukte (3)

Die Generalisierung zu **buschigen Bäumen** ist nicht so trivial:

- jeder Teilbaum muss eine Teil-Kette enthalten um Kreuzprodukte zu vermeiden
- Kette $R_1 - \ldots - R_n$ muss überall geteilt werden
- Kommutativität ist zu berücksichtigen (=2 Möglichkeiten)

Das führt zu folgender Formel:

$$f(n) = \begin{cases} 1 & \text{if } n < 2 \\ \sum_{k=1}^{n-1} 2f(k)f(n-k) & \text{if } n \geq 2 \end{cases}$$

In geschlossener Form:

$$f(n) = 2^{n-1}\mathcal{C}(n-1)$$

Stern-Anfragen, keine Kreuzprodukte

Betrachten wir eine Stern-Anfrage R_1 in der Mitte und $R_2, \ldots, R_n$ als "Satelliten" außen herum.

- Klar, der erste Join muss R_1 enthalten.
- Danach können die restlichen Relationen beliebig hinzugefügt werden

Das führt zu den folgenden Formeln:

- links-tief: $2 * (n-1)!$
- zigzag: $2 * (n-1)! * 2^{n-2} = (n-1)! * 2^{n-1}$
- buschig: Es sind keine buschigen Bäume möglich (R_1 muss immer vorhanden sein damit Join möglich), also gleiche Anzahl wie bei zigzag.

Clique-Anfragen, keine Kreuzprodukte

- In einer Clique-Anfrage kann jede Relation mit jeder anderen verbunden werden
- Es gibt hier gar keine Bäume die Kreuzprodukte enthalten
- Die Anzahl ist die gleiche wie mit Kreuzprodukten

Beispiel Zahlen, ohne Kreuzprodukte

	Ketten-Anfragen			Stern-Anfragen	
	Links-Tief	ZigZag	Buschig	Links-Tief	ZigZag/Buschig
n	2^{n-1}	2^{2n-3}	$2^{n-1}\mathcal{C}(n-1)$	$2(n-1)!$	$2^{n-1}(n-1)!$
1	1	1	1	1	1
2	2	2	2	2	2
3	4	8	8	4	8
4	8	32	40	12	48
5	16	128	224	48	384
6	32	512	1344	240	3840
7	64	2048	8448	1440	46080
8	128	8192	54912	10080	645120
9	256	32768	366080	80640	10321920
10	512	131072	2489344	725760	18579450

Beispiel Zahlen, mit Kreuzprodukte

n	Links-Tief $n!$	ZigZag $n!2^{n-2}$	Buschig $n!\mathcal{C}(n-1)$
1	1	1	1
2	2	2	2
3	6	12	12
4	24	96	120
5	120	960	1680
6	720	11520	30240
7	5040	161280	665280
8	40320	2580480	17297280
9	362880	46448640	518918400
10	3628800	968972800	17643225600

Greedy Heuristiken - Erster Algorithmus

- Suchraum für mögliche Join-Bäume ist sehr groß.
- Greedy Heuristiken produzieren brauchbare Join-Bäume sehr schnell
- Deshalb auch anwendbar für große Anfragen.

Für den ersten Algorithmus betrachten wir

- links-tiefe Bäume
- keine Kreuzprodukte
- Relationen sind anhand eines Gewichts geordnet (z.B. Kardinaliät)

Der Algoritmus produziert eine Sequenz von Relationen, welche dann als links-tiefer Baum interpretiert wird.

Greedy Heuristiken - Erster Algorithmus (2)

GreedyJoinOrdering-1($R = \{R_1, \ldots, R_n\}$,$w : R \to \mathbb{R}$)
Input: Eine Menge von Relationen R und Gewichtsfunktion w
Output: Eine Join-Ordnung
$S = \epsilon$
while $(|R| > 0)$ {
  $m = \arg\min_{R_i \in R} w(R_i)$
  $R = R \setminus \{m\}$
  $S = S \circ < m >$
}
return S

- Nachteil: Feste Gewichte
- Bereits ausgewählte Relationen beeinflussen Gewichte nicht

Greedy Heuristiken - Zweiter Algorithmus

GreedyJoinOrdering-2($R = \{R_1, \ldots, R_n\}$,$w : R, R^* \rightarrow \mathbb{R}$)
Input: Eine Menge von Relationen R und Gewichtsfunktion w
Output: Eine Join-Ordnung
$S = \epsilon$
while $(|R| > 0)$ {
$m = \arg\min_{R_i \in R} w(R_i, S)$
$R = R \setminus \{m\}$
$S = S \circ < m >$
}
return S

- Kann relative Gewichte berechnen/benutzen (siehe geänderte Definition von w)
- aber die erste Relation hat extrem großen Effekt

Greedy Heuristiken - Dritter Algorithmus

GreedyJoinOrdering-3($R = \{R_1, \ldots, R_n\}$,$w : R, R^* \rightarrow \mathbb{R}$)

Input: Eine Menge von Relationen R und Gewichtsfunktion w

Output: Eine Join-Ordnung

$S = \emptyset$

for each $R_i \in R$ {

  $R' = R \setminus \{R_i\}$

  $S' =< R_i >$

  while $(|R'| > 0)$ {

    $m = \arg\min_{R_j \in R'} w(R_j, S')$ /*Typischerweise: minimiere Selektivitäten*/

    $R' = R' \setminus \{m\}$

    $S' = S' \circ < m >$

  }

  $S = S \cup \{S'\}$

}

return $\arg\min_{S' \in S} w(S'[n], S'[1 : n-1])$

Greedy Operator Ordering (GOO)

- Die zuvor betrachteten Algorithmen erzeugen nur links-tiefe Bäume.
- Greedy Operator Ordering (GOO) **erzeugt buschige Bäume**.

Idee hinter GOO:

- Es sind auch Joins zwischen Join-Bäumen erlaubt.
- Es werden dafür greedy Join-Bäume verbunden (wobei Join-Bäume auch einfache Relationen sein können)
- Kombination der Join-Bäume wird so ausgewählt, dass Größe des (Zwischen)ergebnisses mininmal ist.

Greedy Operator Ordering (GOO) (2)

GOO($R = \{R_1, \ldots, R_n\}$)
Input: Eine Menge von Relationen R
Output: Ein Join-Baum
T = R
while $|T| > 1$ {
$(T_i, T_j) = \arg \min_{(T_i \in T, T_j \in T), T_i \neq T_j} |T_i \bowtie T_j|$
$T = (T \setminus \{T_i\}) \setminus \{T_j\}$
$T = T \cup \{T_i \bowtie T_j\}$
}
return $T_0 \in T$

- Erzeugt das Resultat "bottom up"
- Dabei werden Join-Bäume zu größeren Bäumen kombiniert
- Wählt dafür das Paar mit der geringsten Anzahl von Ergebnissen aus

Dynamische Programmierung (DP)

Grundlegende Ideen/Konzepte:

- Optimalitätsprinzip
  "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."
  (Bellman, 1957)
- Vermeidung von redundanten (mehrfach ausgeführten) Schritten

Eine sehr generische Klasse von Ansätzen:

- alle Kostenfunktionen (solange Optimalitätsprinzip gilt)
- links-tief/buschig/... mit oder ohne Kreuzprodukten
- findet die optimale Lösung.

Optimalitätsprinzip

Gegeben zwei Join-Bäume

$$(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) \bowtie R_5$$

und

$$(((R_3 \bowtie R_1) \bowtie R_2) \bowtie R_4) \bowtie R_5$$

- Falls wir wissen, dass $((R_1 \bowtie R_2) \bowtie R_3)$ günstiger ist als $((R_3 \bowtie R_1) \bowtie R_2)$, dann wissen wir, dass der erste Join-Baum günstiger ist als der zweite Join-Baum.
- D.h. wir brauchen diesen zweiten Join-Baum gar nicht zu berechnen, und würden trotzdem den optimalen Join-Baum nicht verpassen

Optimalitätsprinzip (2)

Formal ausgedrückt: Die Optimalität für das Join-Ordering Problem:

> *Sei T ein optimaler Join-Baum für die Relationen $R_1, \ldots, R_n$. Dann muss jeder Teilbaum S von T ein optimaler Join-Baum für die Relationen sein, die in S enthalten sind.*

Übersicht über DP Strategien

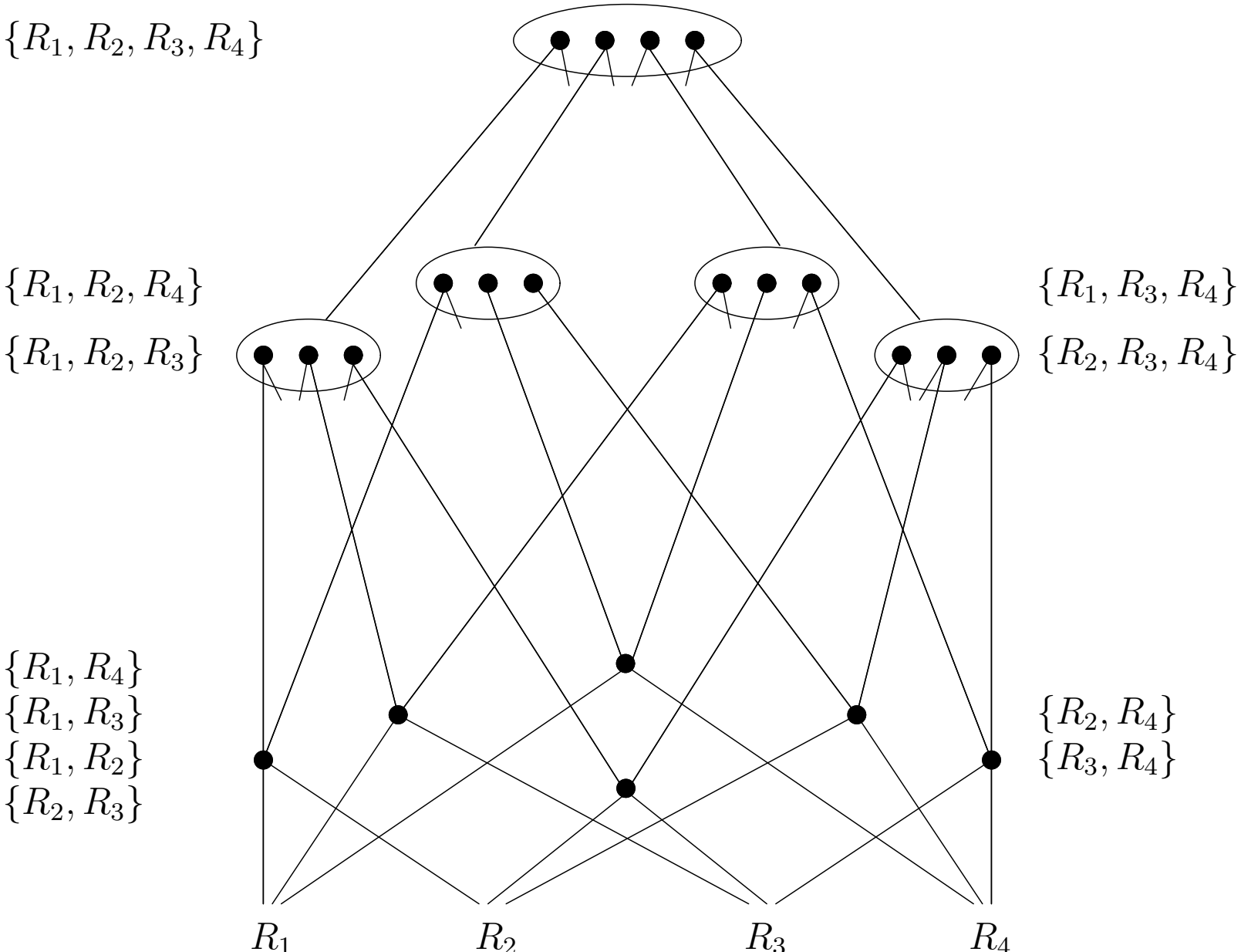
- Generiere optimale Join-Bäume "bottom up"
- Starte mit Join-Bäumen der Größe 1 (also Relationen)
- und baue größere Bäume durch Verwendung der bereits generierten Bäume (kleinerer Größe).

Um Algorithmen kompakter schreiben zu können: Hilfsmethode *CreateJoinTree*, die zwei Join-Bäume verbindet.

Hilfsmethode: CreateJoinTree

CreateJoinTree(T_1,T_2)
Input: Zwei (optimale) Join-Bäume T_1, T_2
für lineare Bäume: setze voraus, dass T_2 eine Relation ist
Output: ein (optimaler) Join-Baum für $T_1 \bowtie T_2$
$B = \emptyset$
for each $impl \in \{$ anwendbare Join-Implementierungen$\}$ $\{$
  if ¬rechts-tief only $\{$
    $B = B \cup \{T_1 \bowtie^{impl} T_2\}$
  $\}$
  if ¬links-tief only $\{$
    $B = B \cup \{T_2 \bowtie^{impl} T_1\}$
  $\}$
$\}$
return $\arg\min_{T \in B} C(T)$

Suchraum von Bäumen unter Optimalitätsprinzip



Generierung von linearen Bäumen

- Ein (links-tiefer) linearer Baum T mit $|T| > 1$ hat die Form $T' \bowtie R_i$, wobei $|T| = |T'| + 1$.
- Falls T optimal ist, dann muss auch T' optimal sein.
- Grundlegende Strategie: Finde optimalen Baum T durch joinen aller optimalen T' mit $T \setminus T'$

Generierung von linearen Bäumen (2)

DPLinear(R)

Input: Eine Menge von Relationen $R = \{R_1, \ldots, R_n\}$

Output: Einen optimalen links-tiefen (rechts-tiefen, zigzag) Join-Baum

$B =$ eine leere DP Tabelle $2^R \rightarrow$ Join-Baum

for each $R_i \in R$

  $B[\{R_i\}] = R_i$

for each $1 < s \leq n$ **ascending** $\{$

  for each $S \subset R, R_i \in R : |S| = s - 1 \wedge R_i \notin S$ $\{$

    if ("keine Kreuzprodukte" $\wedge S$ nicht verbunden mit R_i) **continue**

    $p_1 = B[S], p_2 = B[\{R_i\}]$

    if $p_1 = \epsilon$ **continue**

    $P =$ CreateJoinTree(p_1, p_2);

    if $B[S \cup \{R_i\}] = \epsilon \vee C(B[S \cup \{R_i\}]) > C(P)$

      $B[S \cup \{R_i\}] = P$

  $\}$

$\}$

return $B[\{R_1, \ldots, R_n\}]$

Reihenfolge der Generierung von Teilbäumen

Die Reihenfolge in der die Teilbäume generiert werden spielt keine Rolle, bis auf folgenden Fall.

> *Sei S eine Teilmenge von $\{R_1, \ldots, R_n\}$. Dann müssen bevor der Join-Baum für S generiert werden kann, die Join-Bäume für alle relevanten Teilmengen von S bereits generiert worden sein.*

- *Relevant* bedeutet hier, dass es valide Teilprobleme sind
- Das bedeutet normalerweise, dass die Teilmengen durch den Anfragegraphen verbunden sind (keine Kreuzprodukte)

Generierung von buschigen Bäumen

- Ein buschiger Baum T mit $|T| > 1$ hat die Form $T_1 \bowtie T_2$, wobei $|T| = |T_1| + |T_2|$.
- Falls T optimal ist müssen auch T_1 und T_2 optimal sein.
- Grundlegende Strategie: Finde den optimalen Baum T in dem alle Paare von optimalen Bäumen T_1 und T_2 verbunden werden.

Generierung von buschigen Bäumen (2)

DP(R)

Input: Eine Menge von Relationen $R = \{R_1, \ldots, R_n\}$

Output: Einen optimalen buschigen Join-Baum

$B =$ eine leere DP Tabelle $2^R \rightarrow$ Join-Baum

for each $R_i \in R$

  $B[\{R_i\}] = R_i$

for each $1 < s \leq n$ **ascending** {

  for each $S_1, S_2 \subset R$ mit $|S_1| + |S_2| = s \wedge (S_1 \cap S_2 = \emptyset)$ {

    if ("keine Kreuzprodukte" $\wedge$ S_1 und S_2 nicht verbunden) **continue**

    $p_1 = B[S_1], p_2 = B[S_2]$

    if $p_1 = \epsilon \vee p_2 = \epsilon$ **continue**

    $P =$ CreateJoinTree(p_1, p_2);

    if $B[S_1 \cup S_2] = \epsilon \vee C(B[S_1 \cup S_2]) > C(P)$

      $B[S_1 \cup S_2] = P$

  }

}

return $B[\{R_1, \ldots, R_n\}]$

Randomisierte Verfahren

- Vermeiden (greedy) Aufzählen des Suchraums
- Dazu werden einzelne Pläne als Startpunkt hergenommen, z.B. durch zufällige Generierung
- Diese werden dann schrittweise verbessert
- Z.B. durch Vertauschen benachbarter Operatoren
- Keine Garantie gegeben den optimalen Plan zu finden
- Experimente zeigen aber, dass diese Strategien in der Praxis brauchbare Pläne erzeugen
- insbesondere also nützlich im Fall von sehr großen Anfragen (Performanz!)

Das eigentliche Erzeugen von zufälligen Plänen ist ein anderes Problem.

Randomisierte Verfahren (2)

Hill Climbing

- Ausgehend von Startplan werden Nachbar-Pläne angeschaut.
- Stop, wenn keine Verbesserung mehr möglich.
- Probleme? Hängenbleiben in lokalen Minima

Iterative Improvement

- Starte von zufällig ausgewählten Plan
- Betrachte Nachbarpläne: (lokales) Minimum erreicht?
- Wähle neuen zufällig ausgewählten Plan und beginne von Neuem
- Abbruchkriterium z.B. Anzahl Durchläufe

Weitere

- Simulated Annealing
- Genetische Algorithmen