FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Datenbanksysteme II

Prof. Dr. Viktor Leis

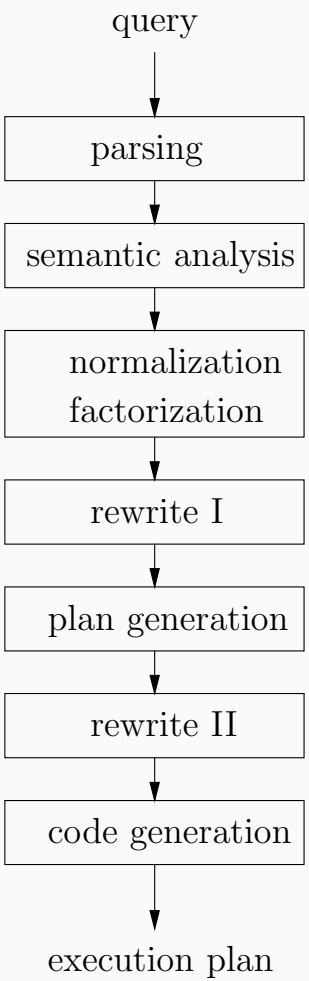
Professur für Datenbanken und Informationssysteme

Anfrageoptimierung

Anfrageoptimierung

- SQL ist eine deklarative Sprache
- Datenbanksystem kann/muss entscheiden wie die Anfrage ausgeführt werden soll
- unterschiedliche Ausführungspläne können um sich um Größenordnungen unterscheidende Laufzeiten zur Folge
- insbesondere die Joinreihenfolge ist entscheidend
- der Abfrageoptimierer (“query optimizer”) ist die Komponente, die versucht einen möglichst guten Plan zu finden

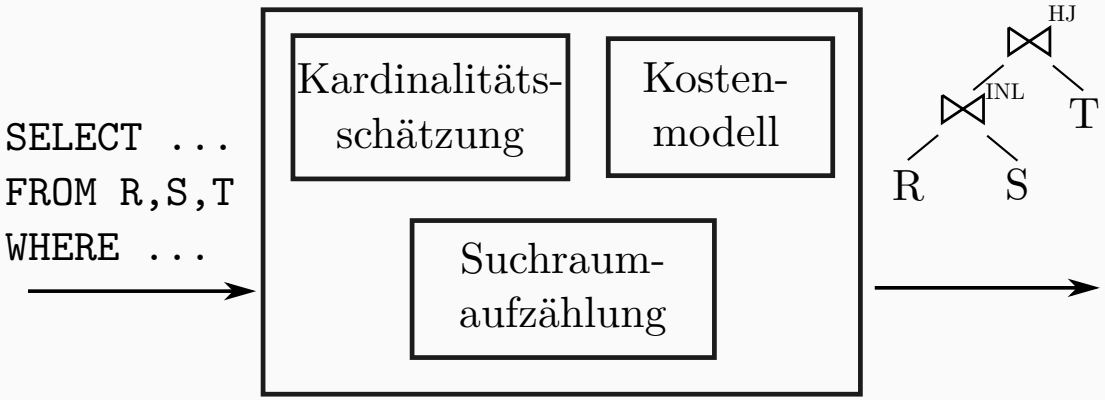
Übersicht Datenbankfrontend



1. parsing, AST
2. schema lookup, variable binding, type inference
3. normalization, factorization, constant folding etc.
4. view resolution, unnesting, deriving predicates etc.
5. constructing the execution plan
6. refining the plan, pushing group by etc.
7. producing the imperative plan

rewrite I, plan generation und rewrite II sind zusammen der Abfrageoptimierer

Anfrageoptimierer



Statistiken

- um Kardinalitäten zu schätzen werden Statistiken über die gespeicherten Daten vorgehalten
- diese werden üblicherweise nicht ganz aktuell gehalten, sondern werden periodisch oder wenn sich die Daten substanziell geändert haben neu erzeugt
- damit die Erzeugen/Aktualisierung von Statistiken nicht zu lange dauert, werden sie oft auf Basis von Stichproben generiert

Statistiken (2)

- Statistiken werden meist pro Attribut vorgehalten:
  - Histogramme (gut für Bereichsanfragen)
  - Ausreißer für sehr häufige Werte
  - Domänengröße (Anzahl)
  - manchmal Stichproben
- manche Systeme erlauben auch mehrdimensionale Statistiken, dies muss aber meist explizit angefordert werden

Kardinalitätsschätzung

- auf Basis von statistischen Annahmen und den gespeicherten Statistiken werden Kardinalitäten geschätzt
- Gleichverteilung:

$$|\sigma_{R.m=\text{'BMW'}}(R)| = \frac{|R|}{\text{dom}(R.m)}$$

- Unabhängigkeit:

$$|\sigma_{R.m=\text{'BMW'} \wedge R.n=\text{'M5'}}(R)| = |\sigma_{R.m=\text{'BMW'}}(R)| \cdot |\sigma_{R.n=\text{'M5'}}(R)| : |R|$$

- ad hoc Konstanten:

$$|\sigma_{R.m \ \text{LIKE} \ \text{'BM\%'}}(R)| = 0.1|R|$$

- Inklusion (Joins):

$$|R \bowtie_{R.x=S.y} S| = \frac{|R||S|}{\max(\text{dom}(R.x), \text{dom}(S.y))}$$

Anfragen

Wir fokussieren uns auf die Bestimmung der Joinreihenfolge und folgende Anfragen:

- innere Joins
- Selektionen auf Basistabellen
- Joinprädikate haben die Form $a_1 = a_2$ wobei a_1 und a_2 Attribute sind

Wir verbinden Relationen $R_1, \ldots, R_n$, wobei R_i folgendes sein kann:

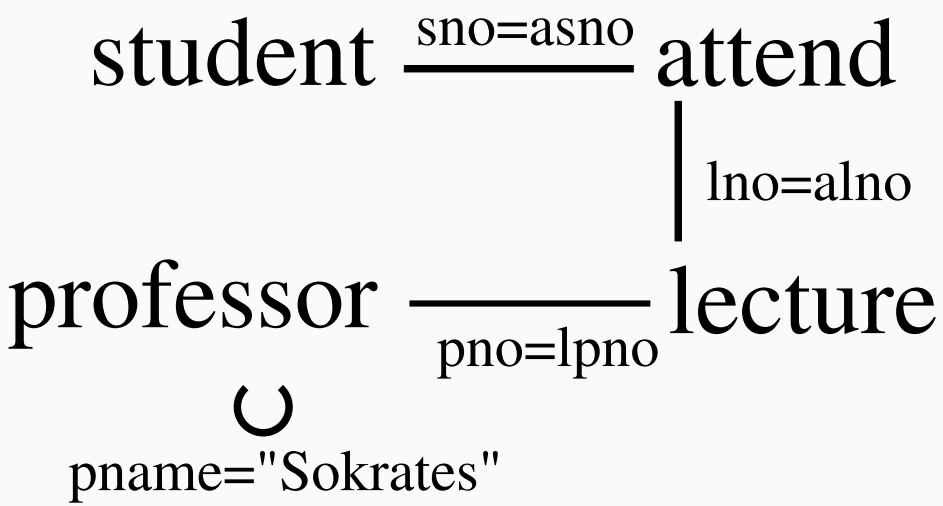
- eine Basisrelation
- eine Basisrelation inklusive Selektionen
- ein komplexer Teilbaum oder Index-Scan

Query Graph

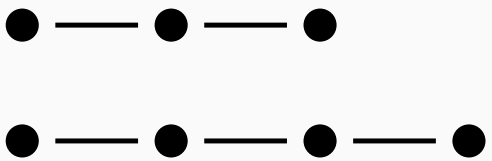
Solche Anfragen können als Query Graph dargestellt werden:

- ungerichteter Graph mit Relationen $R_1, \ldots, R_n$ als Knoten
- ein Prädikat der Form $a_1 = a_2$ mit $a_1 \in R_i$ und $a_2 \in R_j$ ergibt eine Kante zwischen R_i and R_j

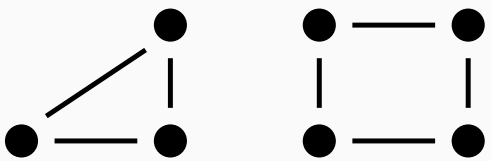
Beispiel Query Graph



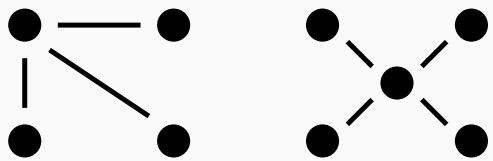
Arten von Query Graphen



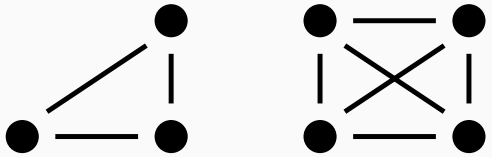
chains



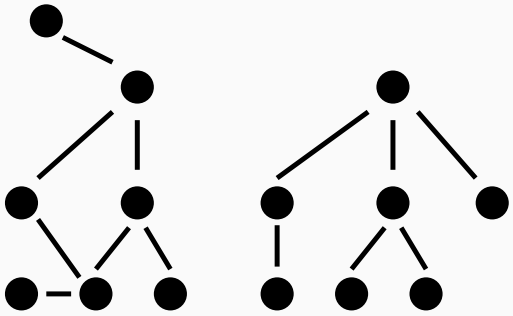
cycles



stars

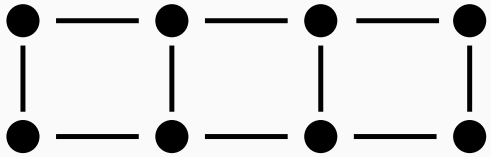


cliques



cyclic

tree



grid

- “echte” Query Graphen liegen irgendwo zwischen diesen idealisierten Formen

Joinbäume

Ein Joinbaum ist ein binärer Baum mit

- Joinoperatoren als inneren Knoten
- Relationen als Blattknoten

Manche Optimierungsalgorithmen erzeugen Joinbäume mit Kreuzprodukten, andere ohne

Formen von Joinbäumen

- links-tief
- rechts-tief
- zickzack
- bushy

Die ersten drei Varianten sind lineare Bäume

Join Selektivität

Eingabe:

- Kardinalitäten $|R_i|$
- Selektivitäten $f_{i,j}$: falls $p_{i,j}$ ein Join-Prädikat zwischen R_i and R_j ist, definieren wir:

$$f_{i,j} = \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i \times R_j|}$$

Berechnung:

- Ergebniskardinalität:

$$|R_i \bowtie_{p_{i,j}} R_j| = f_{i,j}|R_i||R_j|$$

Kardinalität von Joinbäumen

Gegeben einen Joinbaum T, lässt sich die Ergebniskardinalität $|T|$ rekursiv berechnen:

$$|T| = \begin{cases} |R_i| & \text{if } T \text{ is a leaf } R_i \\ (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}) |T_1||T_2| & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

- einfach zu berechnen
- nimmt Unabhängigkeit der Prädikate an
- diese Annahme ist in der Praxis oft falsch[1]

[1] Blog post zum Thema: `https://wp.sigmod.org/?p=1075`

Beispiel

$$
\begin{aligned}
|R_1| &= 10 \\
|R_2| &= 100 \\
|R_3| &= 1000 \\
f_{1,2} &= 0.1 \\
f_{2,3} &= 0.2
\end{aligned}
$$

- impliziert Query Graph $R_1 - R_2 - R_3$
- für alle anderen Kombinationen ist $f_{i,j} = 1$

Eine einfache Kostenfunktion

Gegeben einen Joinbaum T, ist die Kostenfunktion C_{out} definiert als

$$C_{out}(T) = \begin{cases} 0 & \text{if } T \text{ is a leaf } R_i \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

- Summe der Zwischenergebnisgrößen
- Idee: größere Zwischenergebnisse verursachen mehr Arbeit
- die Kosten der Basisrelationen werden ignoriert, da diese immer gelesen werden müssen

Einfache joinspezifische Kostenfunktionen

$$
\begin{aligned}
C_{nlj}(e_1 \bowtie e_2) &= |e_1||e2| \\
C_{smj}(e_1 \bowtie e_2) &= |e_1| \log(|e_1|) + |e_2| \log(|e_2|)
\end{aligned}
$$

Beispiel: Kostenberechnung

$|R_1| = 10, |R_2| = 100, |R_3| = 1000, f_{1,2} = 0.1, f_{2,3} = 0.2$

	C_{out}	C_{nl}	C_{smj}
$R_1 \bowtie R_2$	100	1000	697.61
$R_2 \bowtie R_3$	20000	100000	10630.26
$R_1 \times R_3$	10000	10000	10000.00
$(R_1 \bowtie R_2) \bowtie R_3$	20100	101000	11327.86
$(R_2 \bowtie R_3) \bowtie R_1$	40000	300000	32595.00
$(R_1 \times R_3) \bowtie R_2$	30000	1010000	143542.00

- große Kostenunterschiede zwischen den Joinbäumen
- unterschiedliche K.funktionen haben unterschiedliche Kosten
- der günstigste Plan ist in diesem Fall immer gleich, aber die relative Reihenfolge unterscheidet sich
- Joinbäume mit Kreuzprodukten sind sehr teuer
- Joinreihenfolge ist entscheidend für alle Kostenfunktionen

Weitere Beispiele

$|R_1| = 1000, |R_2| = 2, |R_3| = 2, f_{1,2} = 0.1, f_{1,3} = 0.1$

	C_{out}
$R_1 \bowtie R_2$	200
$R_2 \times R_3$	4
$R_1 \bowtie R_3$	200
$(R_1 \bowtie R_2) \bowtie R_3$	240
$(R_2 \times R_3) \bowtie R_1$	44
$(R_1 \bowtie R_3) \bowtie R_2$	240

- hier ist das Kreuzprodukt von Vorteil
- aber nur weil $|R_2|$ und $|R_3|$ sehr klein sind

Weitere Beispiele (2)

$|R_1| = 10, |R_2| = 20, |R_3| = 20, |R_4| = 10$
$f_{1,2} = 0.01, f_{2,3} = 0.5, f_{3,4} = 0.01$

	C_{out}
$R_1 \bowtie R_2$	2
$R_2 \bowtie R_3$	200
$R_3 \bowtie R_4$	2
$((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$	24
$((R_2 \times R_3) \bowtie R_1) \bowtie R_4$	222
$(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$	6

- hier ist der bushy Baum ist am besten

Klassifikation des Joinreihenfolgeproblems

1. Query Graph Klasse: *chain*, *cycle*, *star* und *clique*
2. Joinbaumstruktur: *links-tief*, *zickzack* oder *bushy*
3. Joinkonstruktion: *mit* oder *ohne* Kreuzprodukte

Katalanische Zahlen

$\mathcal{C}(n-1)$ ist die Anzahl der Binärbäume mit n Blattknoten, wobei $\mathcal{C}(n)$ definiert ist als

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

In geschlossener Form:

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

Asymptotisches Wachstum: $\Theta(4^n/n^{\frac{3}{2}})$

Anzahl von Joinbäumen mit Kreuzprodukten

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

- Idee: Anzahl der Blattkombinationen ($n!$) $\times$ Anzahl Bäume
- wächst exponentiell
- je flexibler die Baumstruktur, desto mehr Möglichkeiten

Beispielzahlen mit Kreuzprodukten

n	links-tief $n!$	zickzack $n!2^{n-2}$	bushy $n!\mathcal{C}(n-1)$
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

Beispielzahlen ohne Kreuzprodukte

	Chain Anfragen			Star Anfragen	
	links-tief	zickzack	bushy	links-tief	zickzack/Bushy
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

Problemkomplexität

Graph	Joinbaum	Kr.produkte	Kostenfunktion	Komplexität
general	links-tief	nein	ASI	NP-hart
tree/star/chain	links-tief	nein	ASI, 1 joint.	P
star	links-tief	nein	NLJ+SMJ	NP-hart
general/tree/star	links-tief	ja	ASI	NP-hart
chain	links-tief	ja	-	offen
general	bushy	nein	ASI	NP-hart
tree	bushy	nein	-	offen
star	bushy	nein	ASI	P
chain	bushy	nein	any	P
general	bushy	ja	ASI	NP-hart
tree/star/chain	bushy	ja	ASI	NP-hart

Adjacent Sequence Interchange (ASI) ist eine Eigenschaft, die einige (eher einfache) Kostenfunktionen haben

Greedy Heuristik - Erster Algorithmus

- Suchraum der Joinbäume ist sehr groß
- Greedy Heuristiken können sinnvolle Joinbäume sehr schnell berechnen
- sinnvoll für sehr große Anfragen (mit sehr vielen Joins)

Der erste Algorithmus hat folgende Eigenschaften:

- berechnet eine Sequenz von Relationen, was einem links-tiefen Baum entspricht
- keine Kreuzprodukte
- Relationen sind nach Gewichtsfunktion sortiert (z.B. Kardinalität)

Greedy Heuristik - Erster Algorithmus (2)

GreedyJoinOrdering-1($R = \{R_1, \ldots, R_n\}$,$w : R \to \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \epsilon$
while $(|R| > 0)$ {
  $m = \textbf{arg min}_{R_i \in R}\, w(R_i)$
  $R = R \setminus \{m\}$
  $S = S \circ < m >$
}
return S

- Nachteil: feste Gewichtsfunktion
- gewählte Relationen haben keinen Einfluss auf Gewicht
- es ist z.B. nicht möglich die Zwischenergebnisgröße zu minimieren

Greedy Heuristik - Zweiter Algorithmus

GreedyJoinOrdering-2($R = \{R_1, \ldots, R_n\}$,$w : R, R^* \rightarrow \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \epsilon$
while $(|R| > 0)$ {
  $m = \textbf{arg min}_{R_i \in R} w(R_i, S)$
  $R = R \setminus \{m\}$
  $S = S \circ < m >$
}
return S

- kann relative Gewichte berechnen
- aber: erste Relation hat sehr großen Effekt, hat aber wenig verfügbare Informationen

Greedy Heuristik - Dritter Algorithmus

GreedyJoinOrdering-3($R = \{R_1, \ldots, R_n\}$,$w : R, R^* \rightarrow \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \emptyset$
for each $R_i \in R$ {
  $R' = R \setminus \{R_i\}$
  $S' =< R_i >$
  while $(|R'| > 0)$ {
    $m = \arg\min_{R_j \in R'} w(R_j, S')$
    $R' = R' \setminus \{m\}$
    $S' = S' \circ < m >$
  }
  $S = S \cup \{S'\}$
}
return $\arg\min_{S' \in S} w(S'[n], S'[1 : n-1])$

- üblicherweise minimiert man die Selektivitäten (*MinSel*)

Greedy Operator Ordering

- bisher wurden nur links-tiefe Bäume konstruiert
- Greedy Operator Ordering (GOO) berechnet bushy Bäume

Idee:

- alle Relationen müssen irgendwo gejoined werden
- aber Joins können auch zwischen Teilbäumen geschehen
- wir kombinieren deswegen in jedem Schritt jene Joinbäume (die auch Relationen sein können), bei denen das Zwischenergebnis am kleinsten ist

Greedy Operator Ordering (2)

GOO($R = \{R_1, \ldots, R_n\}$)
Input: a set of relations to be joined
Output: a join tree
T = R
while $|T| > 1$ {
  $(T_i, T_j) = \textbf{arg min}_{(T_i \in T, T_j \in T), T_i \neq T_j} |T_i \bowtie T_j|$
  $T = (T \setminus \{T_i\}) \setminus \{T_j\}$
  $T = T \cup \{T_i \bowtie T_j\}$
}
return $T_0 \in T$

- erzeugt Ergebnis “bottom-up”
- Joinbäume werden zu größeren Joinbäumen kombiniert
- in jedem Schritt wird die kleinste Zwischenergebnisgröße gewählt

Dynamische Programmierung

Grundlegende Annahmen Dynamische Programmierung:

- optimale Teilstruktur
- überlappende Teilprobleme

Sehr generischer Ansatz:

- beliebige Kostenfunktionen (solange optimale Teilstruktur erfüllt ist)
- links-tief/bushy, mit/ohne Kreuzprodukte
- ermittelt die optimale Lösung

Konkrete Algorithmen können spezialisiert sein.

Optimale Teilstruktur

Gegeben die Joinbäume

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

und

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

- wenn wir wissen, dass $((R_1 \bowtie R_2) \bowtie R_3)$ billiger als $((R_3 \bowtie R_1) \bowtie R_2)$ ist, wissen wir dass der erste Baum billiger als der zweite ist
- wir brauchen den zweiten Baum also gar nicht zu generieren, finden aber trotzdem die optimale Lösung

Optimale Teilstruktur (2)

Definition des Optimalitätsprinzips für das Joinreihenfolgeproblem:

> *In einem optimalen Joinbaum T für die Relationen $R_1, \ldots, R_n$, ist jeder Teilbaum S ein optimaler Joinbaum für die in S enthaltenen Relationen.*

Übersicht Dynamische Programmierung

- erzeuge optimale Joinbäume “bottom up”
- beginne mit optimalen Joinbäumen der Größe eins (Relationen)
- konstruiere größere Joinbäume durch Wiederverwendung von diesen kleineren Größen

Um die Algorithmen einfach zu halten, benutzen wir eine Hilfsfunktion *CreateJoinTree*, die zwei Joinbäume kombiniert (joined).

Erzeugung von Joinbäumen

CreateJoinTree(T_1,T_2)

Input: two (optimal) join trees T_1, T_2
for linear trees: assume that T_2 is a single relation

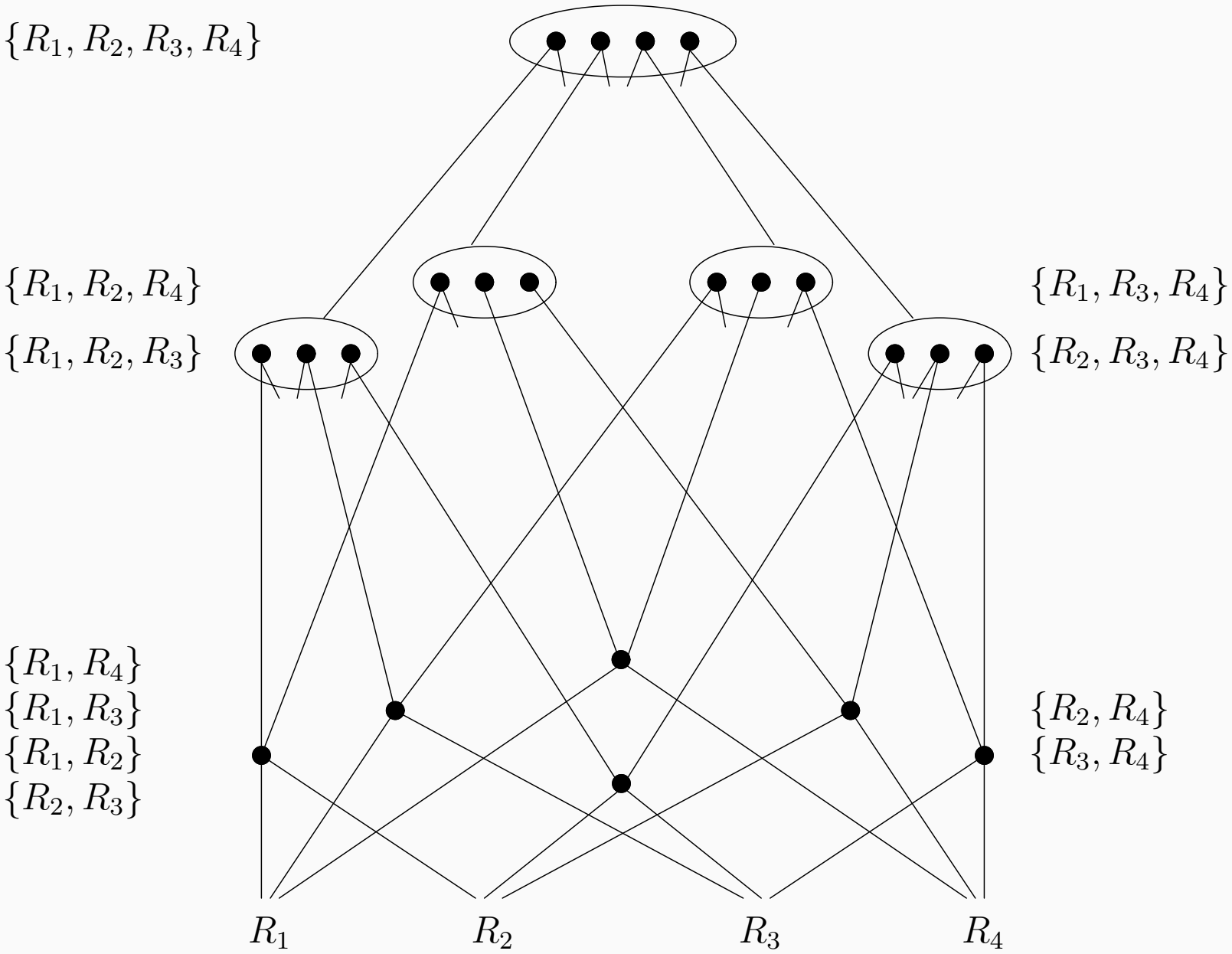
Output: an (optimal) join tree for $T_1 \bowtie T_2$

$B = \emptyset$

for each $impl \in$ { applicable join implementations } {
  if ¬right-deep only {
    $B = B \cup \{T_1 \bowtie^{impl} T_2\}$
  }
  if ¬left-deep only {
    $B = B \cup \{T_2 \bowtie^{impl} T_1\}$
  }
}

return $\arg\min_{T \in B} C(T)$

Suchraum mit Wiederverwendung



Erzeugung Linearer Bäume

- ein links-tiefer (linearer) Baum T mit $|T| > 1$ hat die Form $T' \bowtie R_i$, mit $|T| = |T'| + 1$
- wenn T optimal ist, muss T' auch optimal sein
- grundlegende Idee: finde des optimale T durch joinen aller optimalen T' mit $T \setminus T'$

Die Aufzählungsreihenfolge ist je nach Algorithmus unterschiedlich

Erzeugung Linearer Bäume (2)

DPsizeLinear(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep (right-deep, zig-zag) join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
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 $\neg$cross products $\wedge \neg S$ connected to 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 $B[\{R_1, \ldots, R_n\}]$

Aufzählungsreihenfolge

Die Reihenfolge in der Teilbäume erzeugt werden ist unwichtig solange die folgende Bedingung eingehalten wird:

> *Wenn S eine Teilmenge von $\{R_1, \ldots, R_n\}$ ist, dann müssen, bevor ein Joinbaum für S generiert werden kann, die Joinbäume für alle relvanten Teilmengen von S bereits verfügbar sein.*

- *relevant* heißt hier, dass alle für den Algorithmus validen Teilprobleme berücksichtigt werden
- typischerweise bedeutet diese verbundene Teilprobleme (keine Kreuzprodukte)

Generierung in Integerreihenfolge

000	$\{\}$
001	$\{R_1\}$
010	$\{R_2\}$
011	$\{R_1, R_2\}$
100	$\{R_3\}$
101	$\{R_1, R_3\}$
110	$\{R_2, R_3\}$
111	$\{R_1, R_2, R_3\}$

- kann sehr effizient implementiert werden
- Mengen werde als Integer repräsentiert

Erzeugung Linearer Bäume (3)

DPsubLinear(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep (right-deep, zig-zag) join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < i \leq 2^n - 1$ **ascending** {
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \, mod \, 2) = 1\}$
  for each $R_j \in S$ {
    if ¬cross products ∧¬$S \setminus \{R_j\}$ connected to R_j **continue**
    $p_1 = B[S \setminus \{R_j\}], p_2 = B[\{R_j\}]$
    if $p_1 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  }
}
return $B[\{R_1, \ldots, R_n\}]$

Erzeugung von Bushy Bäumen

- ein Bushy Baum T mit $|T| > 1$ hat die Form $T_1 \bowtie T_2$ mit $|T| = |T_1| + |T_2|$
- wenn T optimal ist, müssen T_1 und T_2 auch optimal sein
- grundlegende Idee: finde das optimale T durch joinen aller optimalen Paare T_1 und T_2

Erzeugung von Bushy Bäumen (2)

DPsize(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < s \leq n$ **ascending** {
  for each $S_1, S_2 \subset R : |S_1| + |S_2| = s$ {
    if ($\neg$cross products $\wedge \neg S_1$ connected to S_2) $\vee$ ($S_1 \cap S_2 \neq \emptyset$) **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 $B[\{R_1, \ldots, R_n\}]$

Erzeugung von Bushy Bäumen (3)

DPsub(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < i \leq 2^n - 1$ **ascending** {
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \, mod \, 2) = 1\}$
  for each $S_1 \subset S, S_2 = S \setminus S_1$ {
    if ¬cross products $\wedge \neg S_1$ connected to S_2 **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 $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  }
}
return $B[\{R_1, \ldots, R_n\}]$

Effiziente Teilmengenerzeugung

Wenn wir Integer benutzen um Mengen zu repräsentieren, können wir alle Teilmengen von S wie folgt aufzählen:

$S_1 = S\&(-S)$
do {
  $S_2 = S - S_1$
  // Do something with S_1 and S_2
  $S_1 = S\&(S_1 - S)$
} **while** ($S_1 != S$)

Bemerkungen

- DPsize/DPsizeLinear können ohne den Test auf $p_1 = \epsilon$ implementiert werden
- man braucht hierzu nur eine Liste der Pläne pro Größe
- Kandidaten können so sehr schnell gefunden werden
- in manchen Fällen erhält man so polynomielle Laufzeit
- DPsub/DPsubLinear sind jedoch effizienter wenn das Problem nicht polynomiell ist

Memoisation

- “top-down” Formulierung von dynamischer Programmierung
- rekursive Erzeugung von Joinbäumen
- bereits erzeugte Teilbäume werden gecached und wiederverwendet um unnötige Arbeit zu sparen
- einfachere Implementierung
- in einigen Fällen auch effizienter (pruning)
- aber typischerweise langsamer als dynamische Programmierung

Memoisation (2)

Memoization(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
MemoizationRec(B, R)
return $B[\{R_1, \ldots, R_n\}]$

- Initialisierung der DP/Memo Tabelle und Start der Rekursion
- eigentliche Arbeit in der Rekursion

Memoisation (3)

MemoizationRec(B,S)
Input: a DP table B and a set of relations S to be joined
Output: an optimal bushy join tree for the subproblem
if $B[S] = \epsilon$ {
  for each $S_1 \subset S, S_2 = S \setminus S_1$
    $p_1 =$MemoizationRec(B, S_1), $p_2 =$MemoizationRec(B, S_2)
    $P =$CreateJoinTree(p_1, p_2)
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  }
}
return $B[S]$

hier: ohne Prüfung auf Verbundenheit

Dynamische Programmierung - Verbundene Teilgraphen

- DP ist eine sehr generelle Strategie
- typisches Anwendungsszenario: bushy, ohne Kreuzprodukte
- DPsize und DPsub möglich, sind aber (asymptotisch) nicht optimal
- die Aufzählungsreihenfolge berücksichtigt nicht den Querygraphen
- viele aufgezählte Paare werden sofort ignoriert (weil sie Kreuzprodukte enthalten sind)
- besonders problematisch bei DPsub

Lösung: Berücksichtigung der Querygraph-Struktur während des Aufzählens

Asymptotischer Suchraum: DPsize

- DP Tabelle wird nach Größe organisiert
- Problem: wenige DP Einträge, trotzdem werden viele Paare betrachtet

Guter Algorithmus für Chains, schlecht für Cliquen:

	chains	cycles	stars	cliques
pairs	$O(n^4)$	$O(n^4)$	$O(4^n)$	$O(4^n)$

Asymptotischer Suchraum: DPsub

- DP Tabelle wird nach Mengendarstellung organisiert
- Problem: es gibt immer 2^n Einträge in der DP Tabelle und eine feste Aufzählungsreihenfolge

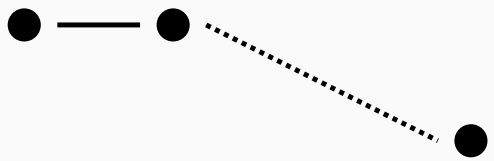
guter Algorithmus für Cliquen, aber passt sich schlecht an:

	chains	cycles	stars	cliques
pairs	$O(2^n)$	$O(n2^n)$	$O(3^n)$	$O(3^n)$

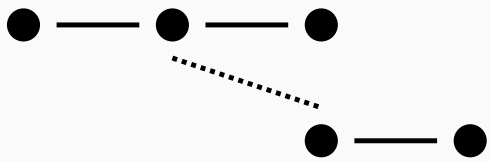
Beobachtung

DPsize und DPsub erzeugen viele Paare, die sofort ignoriert werden.

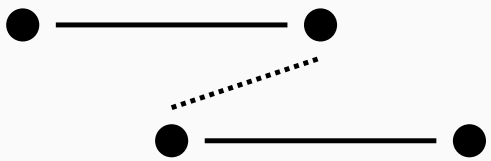
Typische ignorierte Paare (Chain mit 4 Relationen):



nicht verbunden

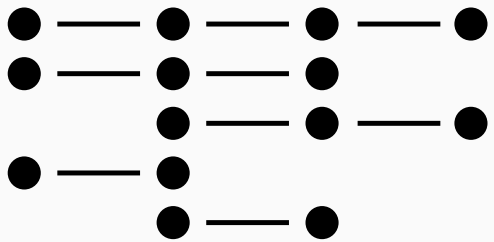


Überlappung



ungültige Teilprobleme

letztes Beispiel $\Rightarrow$ jeder Joinpartner muss verbundener Teilgraph sein:



...

Graph-Theoretischer Ansatz

- Formulierung als Graph-theoretisches Problem:
- zähle alle verbundenen Teilgraphen des Anfragegraphen auf
- für jeden Teilgraph zähle alle verbundenen Teilgraphen auf, die nicht überlappenden und die mit dem ursprünglichen Teilgraphen verbunden sind
- jeder Paar des verbundenen Teilgraphen und des Komplementpaares (ccp) können gejoined werden
- Aufzählung in einer für DP korrekten Reihenfolge

Algorithmus passt sich automatisch der Graphstruktur an:

	chains	cycles	stars	cliques
pairs	$O(n^3)$	$O(n^3)$	$O(n2^n)$	$O(3^n)$

#ccp ist eine untere Schranke für alle DP Algorithmen

DP Algorithmus mit Verbundenen Teilgraphen

Wenn es möglich wäre alle verbundenen Teilgraphen und dazugehörenden Komplementpaare aufzuzählen:

DPccp(R)
Input: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
for $\forall$ csg-cmp-pairs (S_1, S_2), $S = S_1 \cup S_2$ {
  $p_1 = B[S_1]$, $p_2 = B[S_2]$
  $P =$ CreateJoinTree(p_1, p_2);
  if $B[S] = \epsilon \vee C(B[S]) > C(P)$
    $B[S] = P$
}
return $B[\{R_0, \ldots, R_{n-1}\}]$

Die Schwierigkeit ist also die Aufzählung (csg-cmp-pairs).

Einfluss auf den Suchraum

Anzahl der erzeugten Paare:

	Chain			Star		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	20	84	73	32	130	110
10	165	3,962	1,135	2,304	38,342	57,888
15	560	130,798	5,628	114,688	9,533,170	57,305,929
20	1,330	4,193,840	17,545	4,980,736	2,323,474,358	59,892,991,338
	Cycle			Clique		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	40	140	120	90	180	280
10	405	11,062	2,225	28,501	57,002	306,991
15	1,470	523,836	11,760	7,141,686	14,283,372	307,173,877
20	3,610	22,019,294	37,900	1,742,343,625	3,484,687,250	309,338,182,241

Bemerkungen

- DPsize ist gut für Chains, DPsub for Cliquen
- Implementierung von DPccp ist komplizierter
- jede Enumeration muss sehr schnell sein (am besten $O(1)$, maximal $O(n)$, wobei n die Anzahl der Relationen ist)
- aber Gewinne sind sehr groß
- DPccp passt sich dem Anfragegraphen an
- betrachten minimale Anzahl von Paaren
- insbesondere für “mittelkomplexe” Anfragen (z.B. stars) viel schneller

Meta-Heuristiken

- generelle Optimierungsstrategien
- für viele unterschiedliche Probleme anwendbar
- funktionieren auch auf sehr großen Problemen

Wir betrachten einige Meta-Heuristiken, die für die Lösung des Joinreihenfolgeproblems vorgeschlagen wurden.

Zufällige Joinbäume

Zufällige Joinbäume können sehr nützlich sein:

- stichprobenartige Ermittlung der Kosten
- randomisierte Optimierungsstrategien
- Basis für Simulated Annealing, Iterative Improvement etc.

Quick Pick

- effiziente Erzeugung von Psuedo-zufälligen Joinbäumen
- Idee: wähle zufällige Kante im Joingraph
- erweitere Joinbaum durch gewählte Kante

Nicht gleichverteilt, aber sehr schnell

Quick Pick (2)

QuickPick(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a bushy join tree
$E' = E$;
Trees = $\{R_1, \ldots, R_n\}$;
while |Trees| > 1 {
  choose a random $e \in E'$
  $E' = E' \setminus \{e\}$
  if e connects two relations in different subtrees $T_1, T_2 \in$ Trees
    Trees = Trees$\setminus\{T_1, T_2\} \cup$CreateJoinTree$(T_1, T_2)$
}
return $T \in$Trees

- kann mehrmals ausgeführt werden um einen guten Plan zu finden

Iterative Improvement

- Starte mit einem zufälligen Joinbaum
- Benutze Regel, die den Joinbaum verbessert
- Stoppe wenn keine weitere Verbesserung möglich ist

Iterative Improvement (2)

IterativeImprovementBase(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
do {
  JoinTree = random tree
  JoinTree = IterativeImprovement(JoinTree)
  if cost(JoinTree) $<$ cost(BestTree) {
    BestTree = JoinTree
  }
} **while** (time limit not exceeded)
return BestTree

Iterative Improvement (3)

```
IterativeImprovement(JoinTree)
Input:  a join tree
Output: improved join tree
do {
   JoinTree' = randomly apply a transformation from the rule set
   if (cost(JoinTree') < cost(JoinTree)) {
      JoinTree = JoinTree'
   }
} while local minimum not reached
return JoinTree
```

Simulated Annealing

- II: bleibt im lokalen Minimum hängen
- SA: erlaubt auch Änderungen, die in teureren Joinbäumen resultieren
- mit der Zeit werden solche riskante Änderungen immer seltener durchgeführt

Simulated Annealing (2)

SimulatedAnnealing(Query Graph G)

Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$

Output: a join tree

BestTreeSoFar = random tree

Tree = BestTreeSoFar

Simulated Annealing (3)

```
do {
   do {
      Tree' = apply random transformation to Tree
      if (cost(Tree') < cost(Tree)) {
         Tree = Tree'
      } else {
         with probability e^−(cost(Tree')−cost(Tree))/temperature
            Tree = Tree'
      }
      if (cost(Tree) < cost(BestTreeSoFar)) {
         BestTreeSoFar = Tree'
      }
   } while equilibrium not reached
   reduce temperature
} while not frozen
return BestTreeSoFar
```

Simulated Annealing (4)

Vorteile:

- kann aus lokalen Minima entkommen
- empirisch bessere Ergebnisse als II

Probleme:

- Parametertuning nötig
- initiale Temperatur
- wann und wie soll die Temperatur gesenkt werden

Tabu Suche

- Wähle günstigsten erreichbaren Nachbar (auch wenn er teurer ist)
- Tabu-Menge verhindert, dass man im Kreis läuft

Tabu Suche (2)

TabuSearch(Query Graph)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
Tree = random join tree
BestTreeSoFar = Tree
TabuSet = $\emptyset$
do {
  Neighbors = all trees generated by applying a transformation to Tree
  Tree = cheapest in Neighbors \ TabuSet
  if cost(Tree) $<$ cost(BestTreeSoFar)
    BestTreeSoFar = Tree
  if (|TabuSet| $>$ limit) remove oldest tree from TabuSet
  TabuSet = TabuSet$\cup${Tree}
}
return BestTreeSoFar

Kombinationen

- Meta-Heuristiken werden oft nicht alleine eingesetzt
- sie können dazu benutzt werden andere Heuristiken zu verbessern oder zu beschleunigen
- Two Phase Optimization:
  1. Erzeuge zufällige Bäume, führe Iterative Improvement aus um das lokale Minimum zu finden.
  2. Dann wird Simulated Annealing gestartet um einen besseren Plan in der Nachbarschaft zu finden.
  3. Die initiale Temperatur kann höher gewählt werden als in der selbstständigen Variante.
- GOO-II: erst GOO, dann II

Abschlussbemerkungen

- Queryoptimierung und insbesondere die Joinreihenfolge hat einen sehr großen Einfluss auf die Laufzeit
- die meisten Systeme setzen DPsize/bushy ein (z.B. für bis zu 12 Relationen) und dann Heuristiken
- DPccp ermöglicht es auch größere Anfragen optimal zu lösen
- Heuristiken erlauben es auch für sehr große Anfragen akzeptable Pläne zu finden
- in der Praxis ist Kardinalitätsschätzung ein großes Problem und oft die Ursache für zu langsame Anfragen