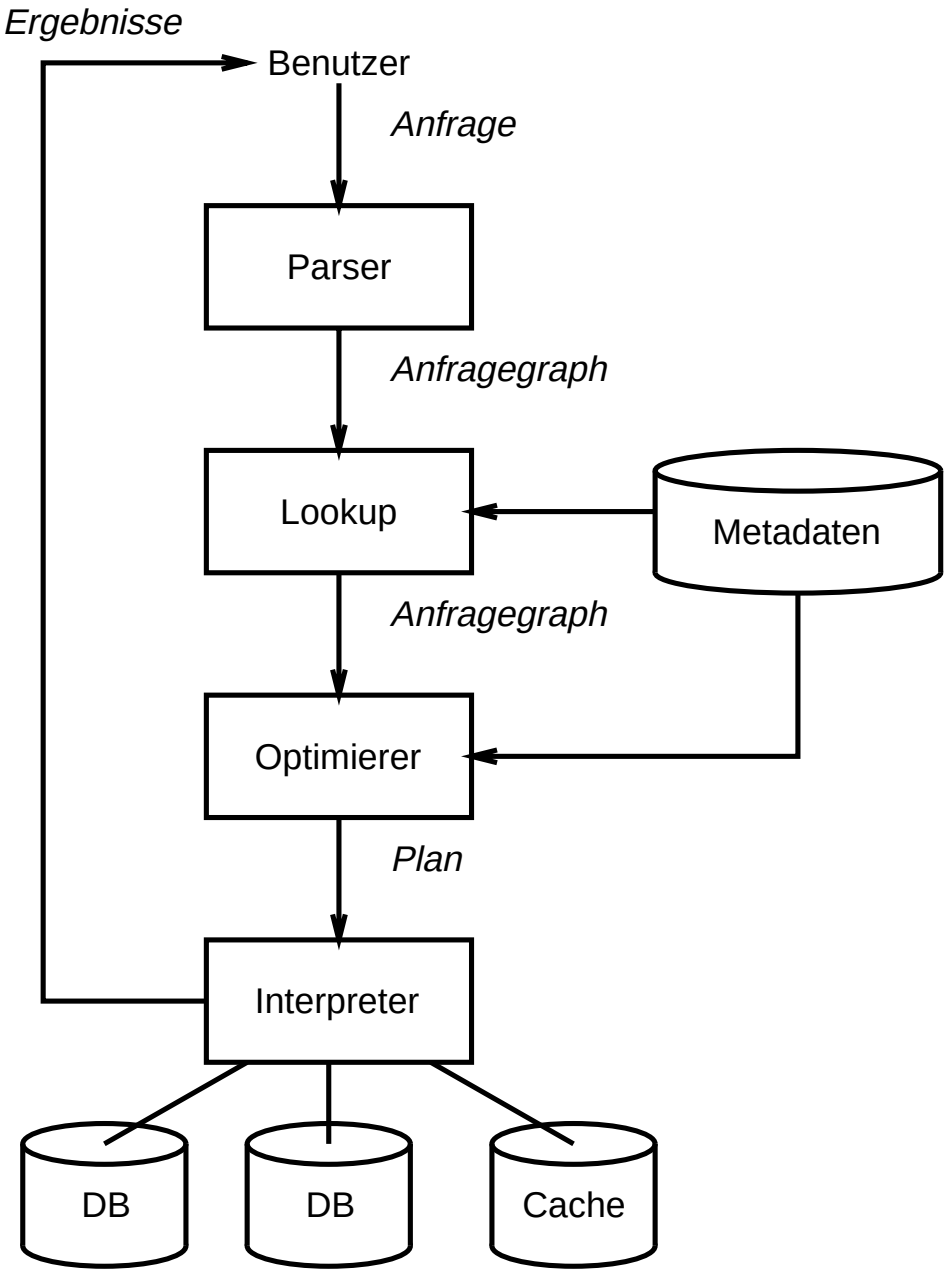
Anfragebearbeitung



Ziel

Finde kostengünstigsten *Plan*.

Anfrageoptimierung

Es gibt viele Möglichkeiten, eine Anfrage auszuführen

- Auswahl der Verbundreihenfolge

- Auswahl des Ausführungsknotens

- ...

Anfrageoptimierung

Grundprinzip eines Optimierers

1. zähle verschiedene Pläne auf
2. wende ein Kostenmodell an
3. wähle kostengünstigsten Plan

Algorithmen

1. Dynamische Programmierung [Selinger, 1979]
2. Zufallsgesteuerte Algorithmen [Ioannidis, 1990]
3. Iterierte Dynamische Programmierung [Kossmann, Stocker 1998]
4. viele weitere in den 80er und 90er Jahren

Beobachtungen

1. das Problem ist $\mathcal{NP}$-hart
2. Algorithmen unterscheiden sich durch Laufzeit und Qualität der Pläne
3. Unterschied zwischen einem guten und einem schlechten Plan beträgt *Stunden* oder gar *Jahre*
4. gänge Algorithmen haben unvorhersehbare Laufzeit

Dynamische Programmierung [Selinger, 1979]

Prinzip: Baue Pläne von unten nach oben auf

Dynamische Programmierung (Verteiltes System)

Dynamische Programmierung

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

1: **for** $i = 1$ **to** n **do** {
2: optPlan($\{R_i\}$) = accessPlans(R_i)
3: prunePlans(optPlan($\{R_i\}$)
4: }
5: **for** $i = 2$ **to** n **do** {
6: **for all** $S \subseteq \{R_1, \ldots, R_n\}$ such that $|S| = i$ **do** {
7: optPlan(S) = $\emptyset$
8: **for all** O, I such that $S = O \cup I$ and $O \cap I = \emptyset$ **do** {
9: optPlan(S) = optPlan(S) $\cup$ joinPlans(optPlan(O), optPlan(I))
10: }
11: prunePlans(optPlan(S))
12: }
13: }
14: finalPlans(optPlan($\{R_1, \ldots, R_n\}$)
15: prunePlans(optPlan($\{R_1, \ldots, R_n\}$)
16: **return** optPlan($\{R_1, \ldots, R_n\}$)

- Der meiste Code steckt in den *accessPlans*, *joinPlans*, *finalPlans*, und *prunePlans* Funktionen.
- 200.000 Zeilen Code bei IBM UDB

Bewertung: Dynamische Programmierung

Qualität der Pläne

- findet immer den besten Plan

Komplexität und Laufzeitverhalten

- Laufzeit: $\mathcal{O}(s^3 * 3^n)$
- Speicher: $\mathcal{O}(s^2 * 2^n)$
- scheitert für $n > 15$ bei $s = 1$ (zentral)
- scheitert für $n > 8$ bei $s > 1$ (verteilt)
- genaue Vorhersage aber nicht möglich
- liefert Pläne erst am Ende

Einsetzbarkeit

- wird in fast allen Datenbankprodukten eingesetzt
- Einsatz für heterogene verteilte Systeme möglich
- Erweiterbar für neue Techniken, Funktionen und Typen

Zufallsgesteuerte Algorithmen

Prinzip

1. finde einen (zufälligen) kompletten Plan für die Anfrage
2. transformiere iterativ den Plan in andere zufällige Pläne; merke dabei stets den besten Plan

Suchstrategien

- *hill climbing* oder *iterative improvement*
- *simulated annealing* oder *Metropolis*
- Kombinationen: II+SA [Ioannidis, 1990]

Bewertung

- von Anfang an gütlige Pläne (*graceful degradation*)
- keine Komplexitätsabschätzung möglich; braucht in der Regel ca. eine Minute für komplexe Anfragen
- keine Garantie über die Qualität der erzeugten Pläne
- Parameterauswahl schwierig
- kann nicht in bestehende Optimierer integriert werden
- wird gegenwärtig in **keinem** Produkt eingesetzt

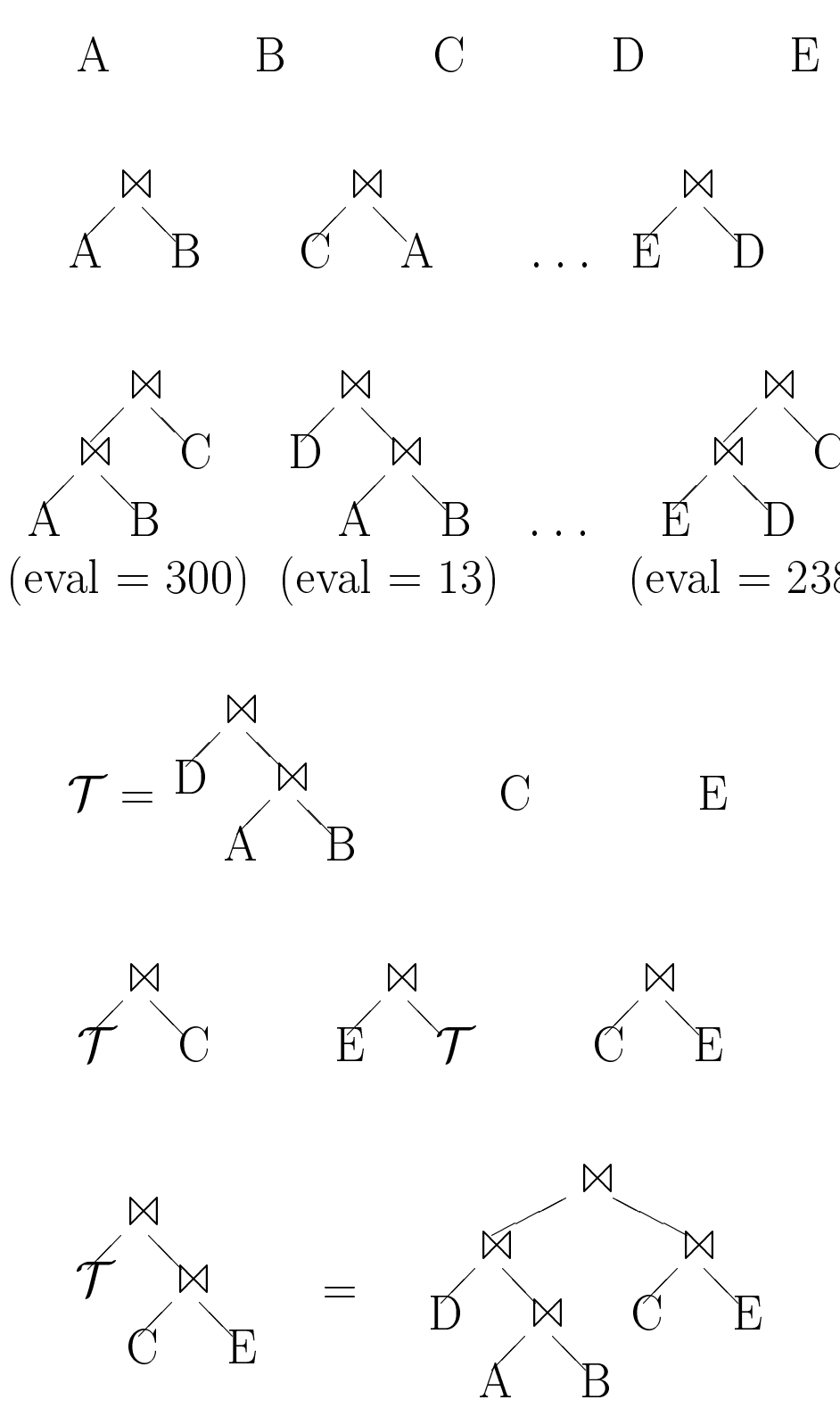
Iterierte Dynamische Programmierung (IDP)

Ziele

- **Qualität der Pläne**
  - so gut wie möglich
- **Laufzeitverhalten**
  - Beschränkung der Laufzeit und des Hauptspeichers
  - "Abbruch" des Optimierungsvorgangs jederzeit möglich (graceful degradation)
- **Einsetzbarkeit**
  - einfache Erweiterung bestehender (DP) Optimierer
  - selbe Erweiterbarkeit wie ein DP Optimierer

Prinzip

1. starte wie dynamische Programmierung
2. stoppe nach einem Time-Out, vollem Hauptspeicher (oder Prozeßorcache) oder Eingriff eines Benutzers
3. wähle einen Subplan aus und starte dynamische Programmierung von vorne


A B C D E
A B
C A
. . .
E D
C
A B
(eval = 300)
D
A B
(eval = 13)
. . .
C
E D
(eval = 2387)
T =
D
A B
C
E
T C
E T
C E
T
C E
=
D
A B
C E

IDP Algorithmus

Input: SPJ query q on relations $R_1, \ldots, R_n$, k
Output: A query plan for q

1: **for** $i = 1$ **to** n **do** {
2: optPlan($\{R_i\}$) = accessPlans(R_i)
3: prunePlans(optPlan($\{R_i\}$)
4: }
N_1: toDo = $\{R_1, \ldots, R_n\}$
N_2: **while** |toDo| > 1 **do** {
N_3: $k = \min\{k, |\text{toDo}|\}$
5': **for** $i = 2$ **to** k **do** {
6': **for all** $S \subseteq$ toDo such that $|S| = i$ **do** {
7: optPlan(S) = $\emptyset$
8: **for all** O, I such that $S = O \cup I$ and $O \cap I = \emptyset$ **do** {
9: optPlan(S) = optPlan(S)$\cup$joinPlans(optPlan(O), optPlan(I))
10: }
11: prunePlans(optPlan(S))
12: }
13: }
N_4: find P, V with $P \in V$, $V \subseteq$ toDo, $|V| = k$ such that
eval(P) = $\min\{\text{eval}(P') \mid P' \in W, W \subseteq \text{toDo}, |W| = k\}$
N_5: generate new symbol: $\mathcal{T}$
N_6: optPlan($\{\mathcal{T}\}$) = $\{P\}$
N_7: toDo = toDo – $V \cup \{\mathcal{T}\}$
N_8: }
14': finalPlans(optPlan(toDo)
15': prunePlans(optPlan(toDo)
16': **return** optPlan(toDo)

- aufwendige Funktionen bleiben unangetastet

Planbewertungsfunktionen

Wann ist $A \bowtie B$ *besser als* $C \bowtie B$*?*

Flaches Kriterium:

- Kosten
- Selektivität der Joinprädikate
- Kardinalität des Zwischenergebnisses
- ...

Ballooning:

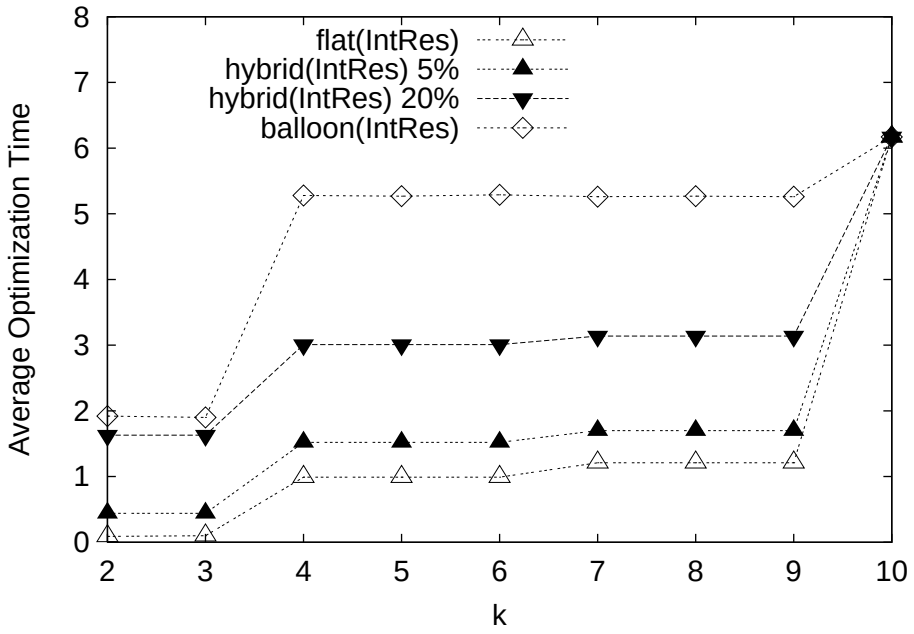
- verwende Greedy Heuristik, um Subplan zu komplettieren
- bewerte Subplan mit Kosten des kompletten Planes
- Vorteil: präzise
- Nachteil: aufwendig

Ergebnis (experimentell)

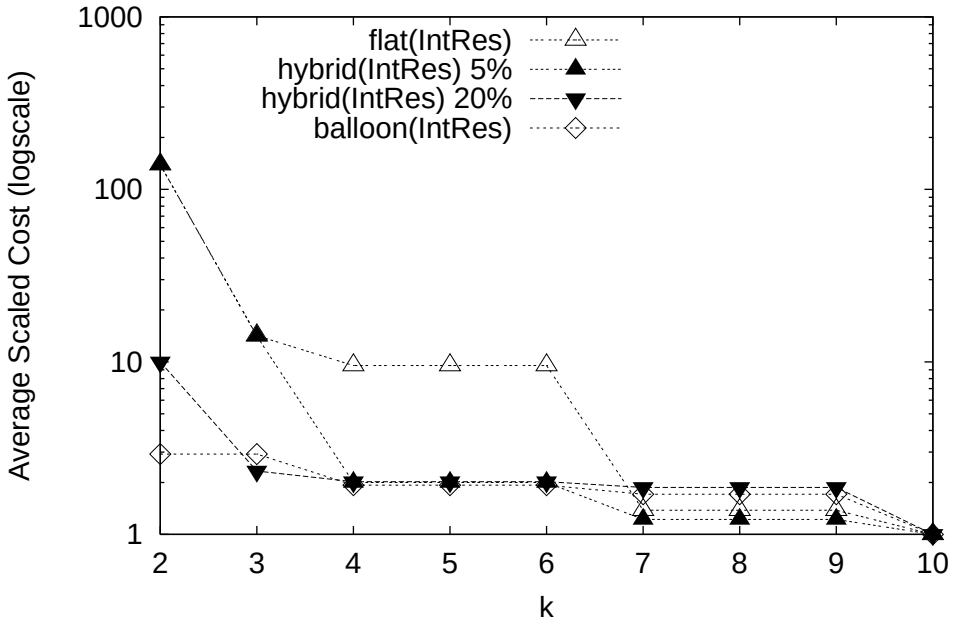
- "Zwischenergnis" gutes Kriterium
- Mischformen zwischen "flach" und "ballooning" zeigen das beste Preis-Leistungs-Verhältnis

Flach vs. Ballooning

Laufzeit



Qualität der Pläne



IDP Varianten: Standard vs. Bushy

Beobachtung:

Bei $k = 3$, kann IDP *a priori* diesen Plan nicht erzeugen.

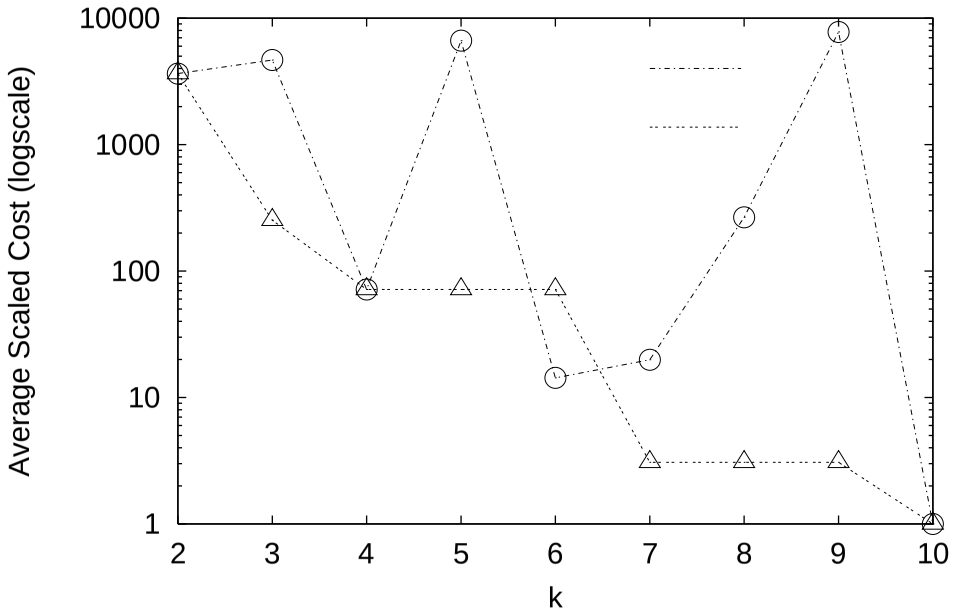
Beispiel: ausgewählter Subplan bei $k = 3$

Mögliche komplette Pläne:

Standard vs. Bushy

Lösung: Beschränke k

- k muß geradzahlig sein
- $k \leq \lceil \frac{l}{2} \rceil$
  (l: Anzahl Relationen in toDo Liste einer Iteration)



Fazit: Bushy Pläne sind wichtig!

Plan vs. Row vs. Rows

Hebt man einen oder mehrere Pläne auf?

BestPlan: hebe nur einen Plan auf

- gute Laufzeit – schlechte Pläne

BestRow: hebe ähnliche Pläne auf

- gute Laufzeit – gute Pläne

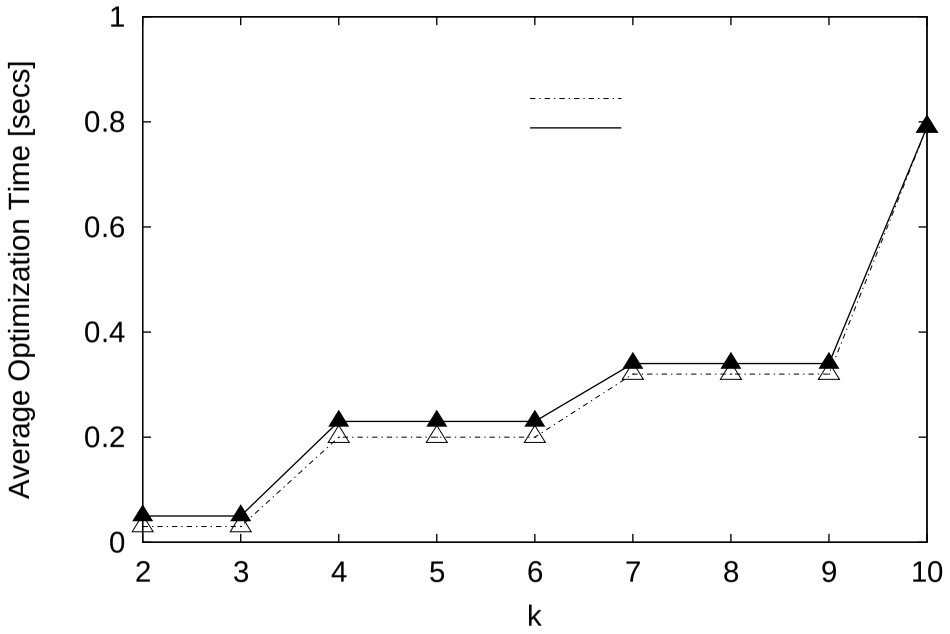
BestRows: hebe Mengen ähnlicher Pläne auf

- gute Pläne – schlechte Laufzeit

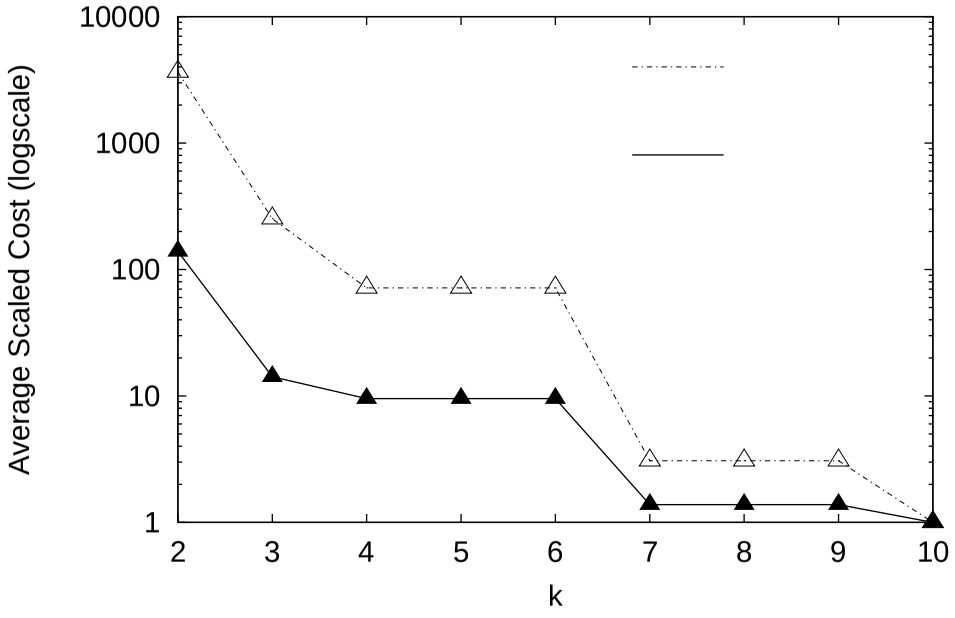
Fazit: BestRow ist Sieger!

Plan vs. Row vs. Rows

Laufzeit



Qualität der Pläne



IDP_1 vs. IDP_2

Alternativer Algorithmus (IDP_2)

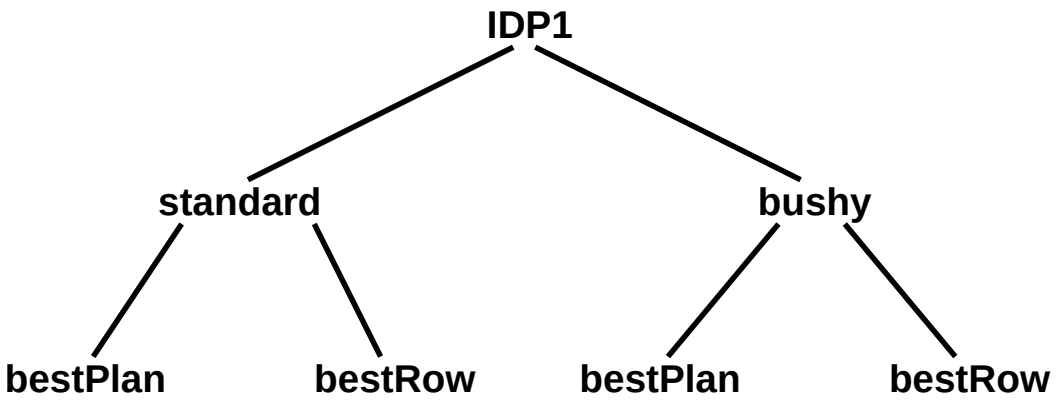
1. verwende greedy Heuristik, um einen Teilplan mit k Relationen zu erzeugen
2. wende dynamische Programmierung auf die Relationen dieses Teilplanes an
3. kontrahiere den Teilplan wie bei IDP_1
4. gehe zu Schritt 1; iteriere bis kompletter Plan erzeugt

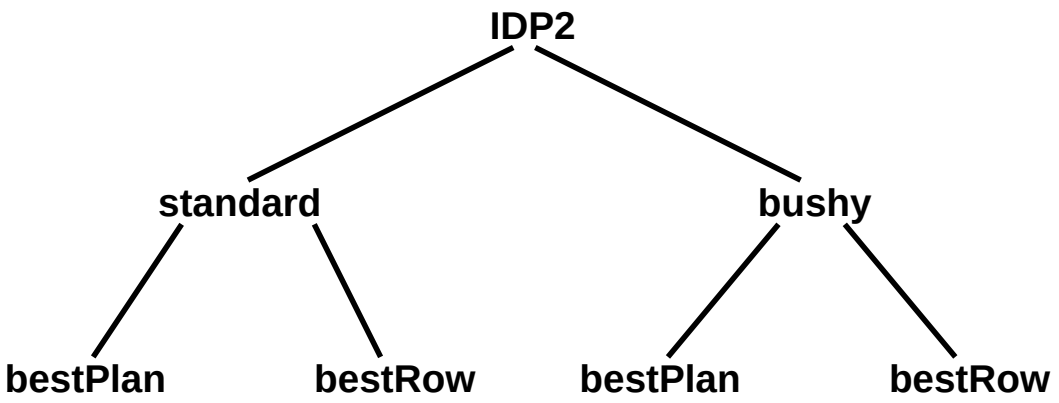
Anmerkungen

- *Bushy* und *BestRow* Varianten hier auch anwendbar
- Vorteil von IDP_2: geringere Komplexität
- Nachteil von IDP_2: nicht adaptiv

IDP_2 Beispiel mit $k = 3$

IDP Übersicht





Bewertung IDP

Vorteile

- erzeugt so gute Pläne wie möglich
- polynomiale Laufzeit: $\mathcal{O}(s^3 * n^k)$
  (weitere Verbesserung durch IDP_2)
- Beschränkung des Ressourcenbedarfs möglich
- in der Praxis sehr gut einsetzbar

Nachteile

- es gibt keine Garantien über die Qualität der Pläne
- hohe Laufzeit bei sehr großem n und s

Komplexitätsabschätzung: IDP_1

Schritt 1
Die erste Iteration von IDP_1 kann in $\mathcal{O}(n^k)$ Schritten ausgeführt werden:

$$\frac{(2 * m - 1)!}{(m-1)!} \quad \text{Pläne mit } m \text{ Tabellen}$$

$$\binom{n}{m} \quad \text{verschiedene } m \text{ Elementige Teilmengen von } n$$

Also, Anzahl der aufgezählten Pläne in der ersten Iteration:

$$A = n + \sum_{m=2}^{k} \binom{n}{m} * \frac{(2 * m - 2)!}{(m-1)!}$$

Ferner gilt

$$A \leq n + \frac{(2 * k - 2)!}{(k-1)!} * \sum_{m=2}^{k} \binom{n}{m} \in \mathcal{O}(n^k)$$

da

$$\binom{n}{m} = \frac{n * (n-1) * \ldots * (n-m)}{m!} \in \mathcal{O}(n^m)$$

und

$$\sum_{m=2}^{k} n^m \in \mathcal{O}(n^k)$$

Komplexitätsabschätzung: IDP_1

Schritt 2

Alle weiteren Iterationen von IDP_1 knnen in $\mathcal{O}(n^k)$ Schritten ausgeführt werden:

$$\lceil \frac{n-k}{k-1} \rceil \quad \text{weitere Iterationen}$$

$$n - i * (k-1) \quad \text{Tabellen in der } i\text{-ten Iteration}$$

Anzahl Pläne in der i-ten Iteration:

$$\sum_{m=2}^{k} \binom{n - i * (k-1)}{m} * \frac{(2*m-2)!}{(m-1)!}$$

Anzahl *alten* Pläne der i-ten Iteration:

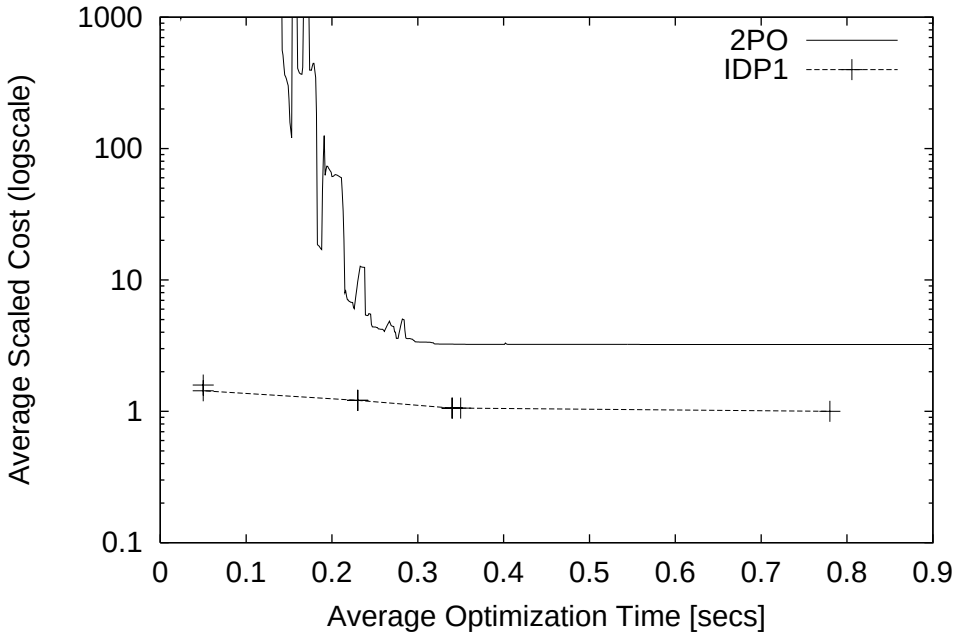
$$\sum_{m=2}^{k} \binom{n - i * (k-1) - 1}{m} * \frac{(2*m-2)!}{(m-1)!}$$

Anzahl der *neuen* Pläne in der i-ten Iteration liegt in $\mathcal{O}(n^{k-1})$.

Insgesamt können also alle Iterationen in $\mathcal{O}(n^k)$ Schritten ausgeführt werden.

Graceful Degradation

Chain, 10-fach Join, 3 Knoten, Replikation



Zusammenfassung und Ausblick

Zusammenfassung

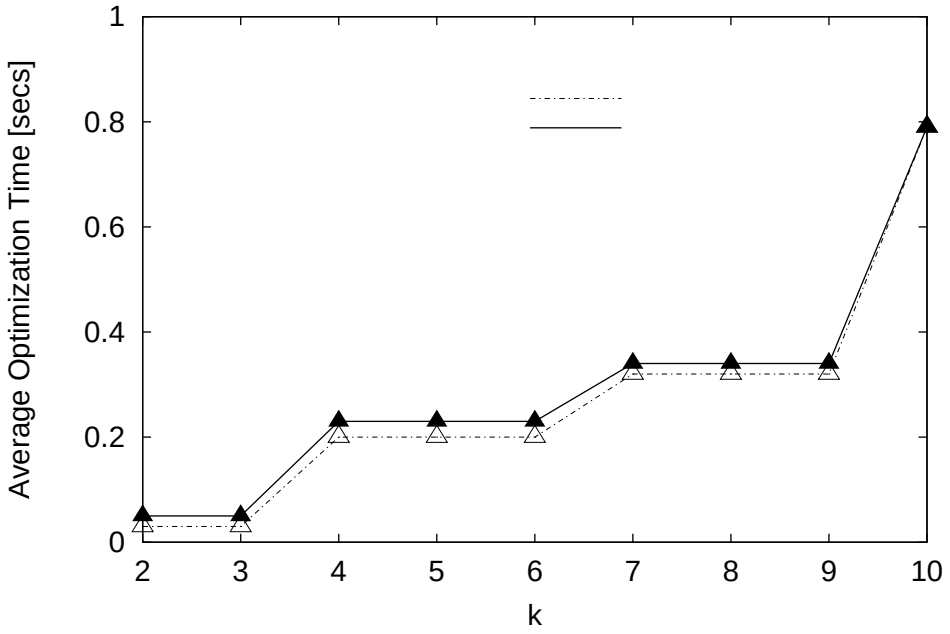
- neue Klasse von *adaptiven* Optimierungsalgorithmen
- funktioniert besonders gut im DB-Kontext
- besonders wichtig für erweiterbare oder verteilte/heterogene Datenbanken
- wichtiger Baustein im ObjectGlobe Projekt

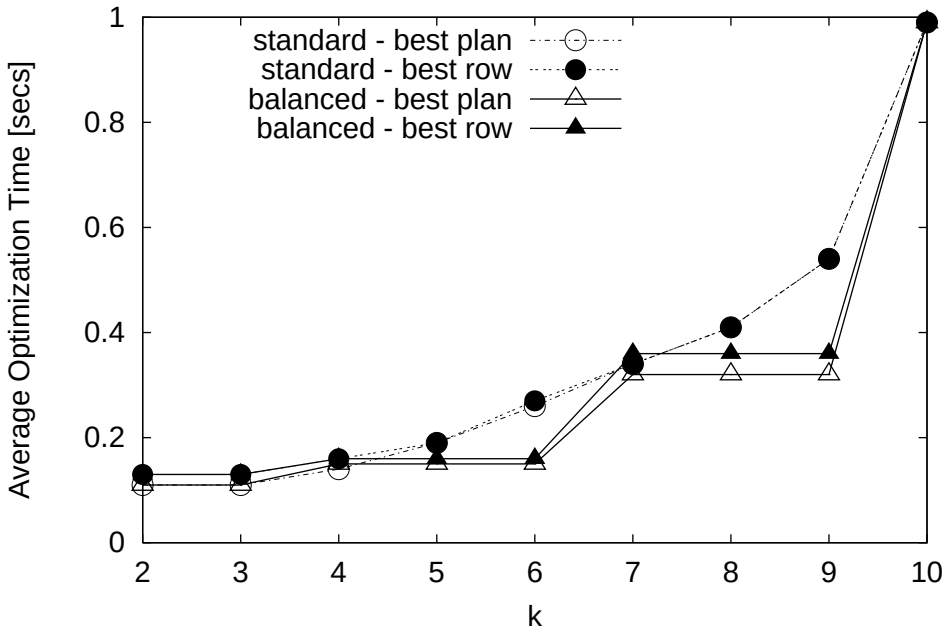
Ausblick

- Einbau in kommerzielle Systeme (z.B. IBM UDB)
- Einsatz für andere Optimierungsprobleme
- Realisierung von ObjectGlobe

Komplexitätsabschätzungen

	$\mathbf{IDP}_1$	$\mathbf{IDP}_2$
Laufzeit	$\mathcal{O}(s^3 * n^k)$	$\mathcal{O}(s^3 * \frac{n}{k} * 3^k)$
Speicher	$\mathcal{O}(s^2 * n^k)$	$\mathcal{O}(s^2 * \frac{n}{k} * 2^k)$





Laufzeit von IDP_1

Die erste Iteration ist teurer als alle weiteren zusammen:

k	*Total*	*1st Iteration*	*All Other Iterations*
2	0.01sec	40%	60%
3	0.09sec	64%	36%
4	0.41sec	74%	26%
5	1.25sec	95%	5%
6	2.96sec	98%	2%
7	5.45sec	99%	1%
8	7.89sec	100%	0%
9	9.21sec	100%	0%
10	9.51sec	100%	0%

Beobachtung zu $\mathbf{IDP}_2$

- die Laufzeit hat eine sehr hohe Varianz

Bewertung von Optimierern

Es gibt keinen Standardbenchmark, aber folgende Punkte sind allgemein anerkannt:

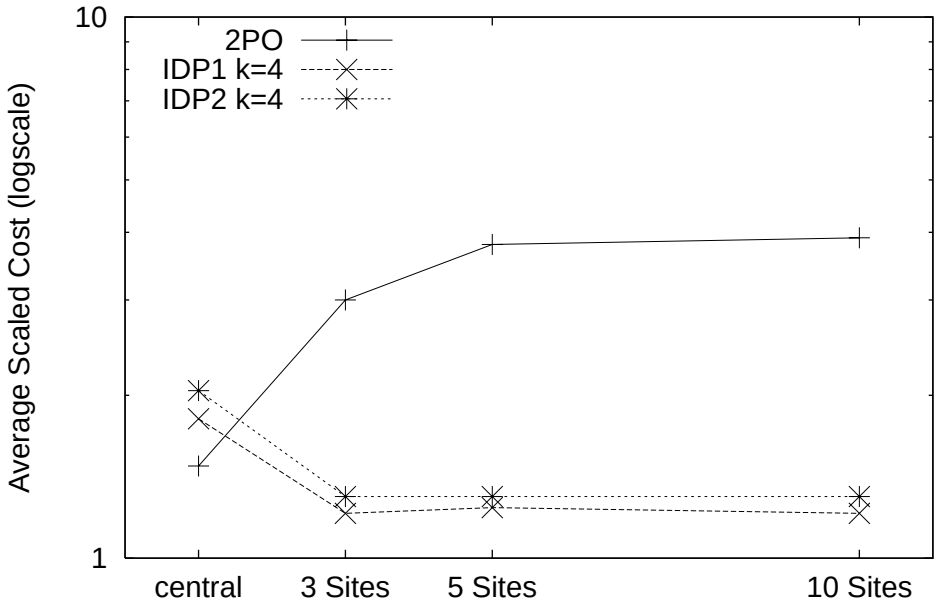
- betrachte Anfragen mit kleinen und großen Tabellen
- variiere Selektivität von Prädikaten
- betrachte verschiedene Anfragetypen
- bewerte möglichst viele Anfragen (~ 100); statistische Relevanz ist unmöglich
- messe *Laufzeit* der Algorithmen und *Qualität* der Pläne
- verwende möglichst realistisches Kostenmodell

Unser Benchmark

- vier Klassen von Tabellengrößen (S, M, L, XL)
- beliebige Selektivität von Prädikaten möglich
- STAR und CHAIN Anfragetypen
- 10-fach Joinanfragen und 20-fach Joinanfragen
- unterschiedliche Verteilungs- und Replikationsmuster im verteilten System
- für jeden Anfragetyp und jedes Verteilungsmuster, erzeuge 100 Anfragen mit zufälligen Tabellen und Prädikaten
- messe:
  - *durchschnittliche Laufzeit* der Algorithmen,
  - *durchschnittliche normierte Kosten* der Pläne
  - % *gute* Pläne (normierte Kosten < 2)
  - % *akzeptable* Pläne (normierte Kosten ≤ 10)
  - % *schlechte* Pläne (normierte Kosten > 10)
- Kostenmodell aus Literatur [Steinbrunn et al. 1997]

Ergebnisse: DP vs. 2PO vs. IDP

Chain, 10-fach Join, Variiere s, Replikation



	$S=1$			$S=3$			$S=10$		
	G	A	B	G	A	B	G	A	B
2PO	92	7	1	57	41	2	31	64	5
IDP_1, $k=2$	80	15	5	86	13	1	84	15	1
IDP_1, $k=4$	91	7	2	93	7	0	94	6	0
IDP_1, $k=7$	99	1	0	98	2	0	98	2	0
IDP_2, $k=2$	80	15	5	86	13	1	84	15	1
IDP_2, $k=4$	89	7	4	94	5	1	92	8	0
IDP_2, $k=7$	99	1	0	99	1	0	99	1	0

Zusammenfassung und Ausblick

Zusammenfassung

- IDP: neue Klasse von Suchstrategien
- IDP vs. Dynamische Programmierung
  - kein Unterschied bei einfachen Anfragen
  - funktioniert auch bei komplexen Anfragen
  - bestehender Optimierer kann leicht erweitert werden
- IDP vs. 2PO (und andere)
  - für $k = 2$ ist IDP immer schneller
  - für $k > 2$ und große n, s ist IDP oft langsamer
  - in jedem Fall erzeugt IDP bessere Pläne
- IDP_1 vs. IDP_2
  - IDP_1 auf jeden Fall zur Sicherheit einbauen
  - IDP_2 als "Optimierungslevel" einbauen

Ausblick

- Einbau in kommerzielle Anfrageoptimierer
- Metaoptimierer
- Multi-Anfrage-Optimierung
- andere Optimierungsprobleme (Traveling Salesman)