Anfragebearbeitung 3

VU Datenbanksysteme vom 23.11.2015

Reinhard Pichler

Arbeitsbereich Datenbanken und Artificial Intelligence
Institut für Informationssysteme
Technische Universität Wien

Kostenmodelle und Tuning

- Grundidee der Optimierung
- Größe der Zwischenergebnisse: Selektivität
- Kostenabschätzung der Operationen
- Optimierungsverfahren
- Tuning

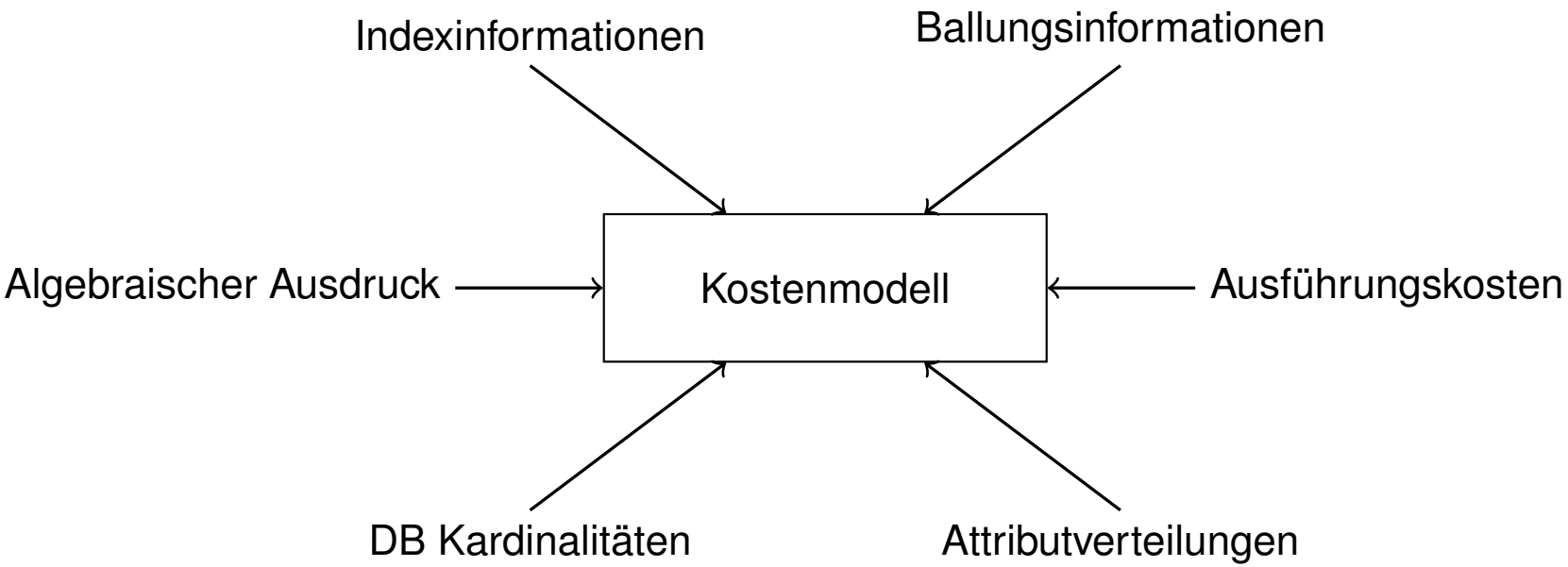
Kostenbasierte Optimierung

Idealvorstellung:

- Generiere *alle* denkbaren Auswertungspläne
- Bewerte deren Kosten
  - Kostenmodell
  - Informationen über Datenbestand: Statistiken und Histogramme
  - Informationen über den verwendeten Rechner (z.B. verfügbarer Speicher)
  - Optimierungsziel (z.B.: Durchsatz maximierend, Antwortzeit minimierend)
- Behalte den billigsten Plan

In Praxis: Beschränkung auf einen Teil der Auswertungspläne

Kostenmodelle



Selektivität

Definition

Die *Selektivität* eines Suchprädikats ist die Anzahl der qualifizierenden Tupel relativ zur Gesamtanzahl der Tupel in der Relation.

Mittels Schätzung der Selektivität wird die Anzahl der Tupel in den Zwischenergebnissen geschätzt.

Beispiel

Die Selektivität einer Anfrage, die das Schlüsselattribut einer Relation R spezifiert, ist $1/|R|$.

Beispiel

Wenn ein Attribut A spezifiziert wird, für das es i verschiedene Werte gibt, so wird üblicherweise die Selektivität als $1/i$ abgeschätzt.

Selektivitäten

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

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

- Join von R mit S:

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

Abschätzung der Selektivität

Einfache Fälle

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

Ansonsten z.B. Stichprobenverfahren

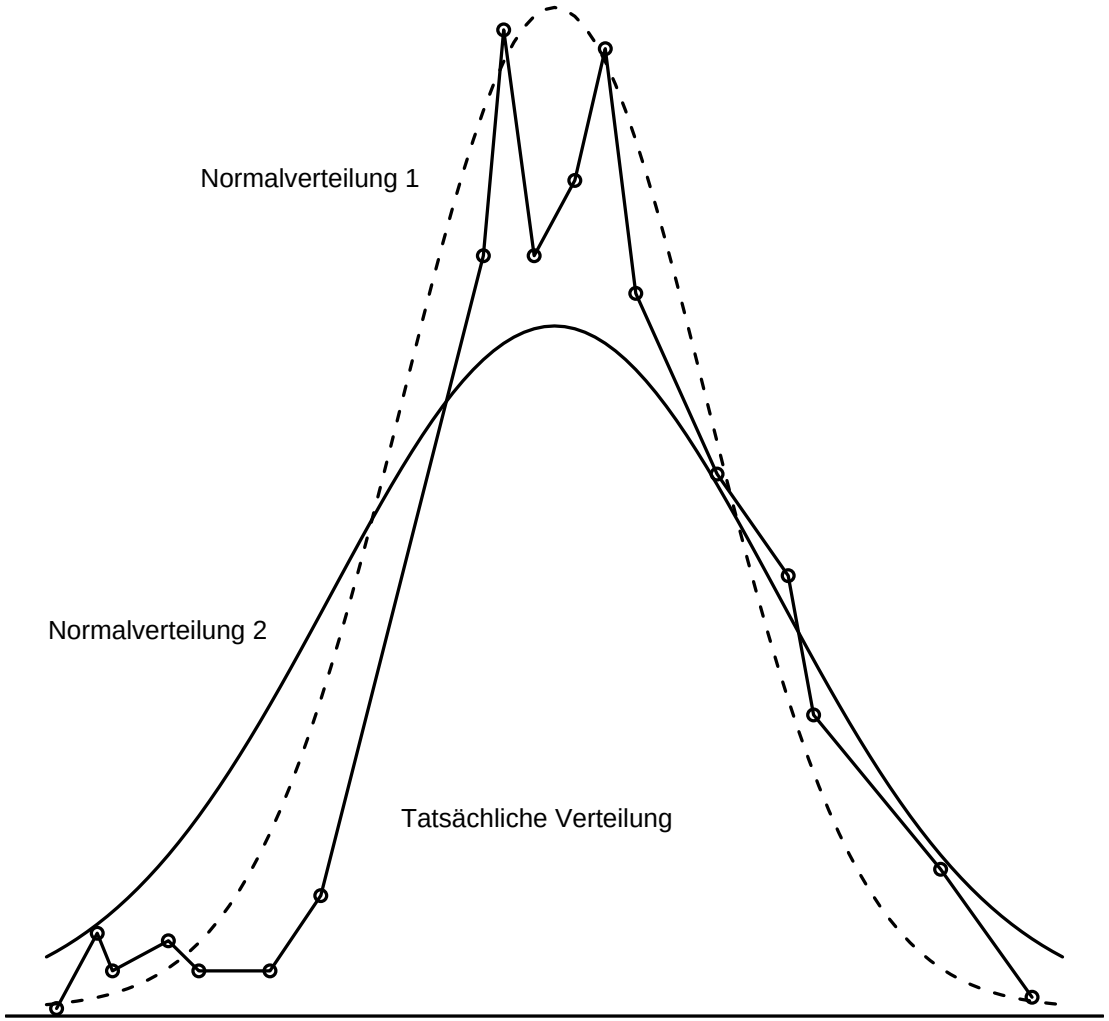
Abschätzung der Selektivität

Weitere Methoden

- Stichprobenverfahren:
  - Berechne exakte Selektivität für eine Stichprobe des Inputs
  - Verallgemeinerung auf gesamten Input
- Histogramme:
  - Unterteile den Wertebereich eines Attributs in Teilbereiche
  - Berechne die relative Häufigkeit dieser Teilbereiche
  - Unterteilung: equi-width (d.h. Intervalle gleich groß) oder equi-depth (Intervalle mit gleicher rel. Haufigkeit)
- Parametrisierte Verteilungen:
  - Annahmen über die Verteilung, z.B. Normalverteilung
  - Parameterbestimmung mittels Stichproben

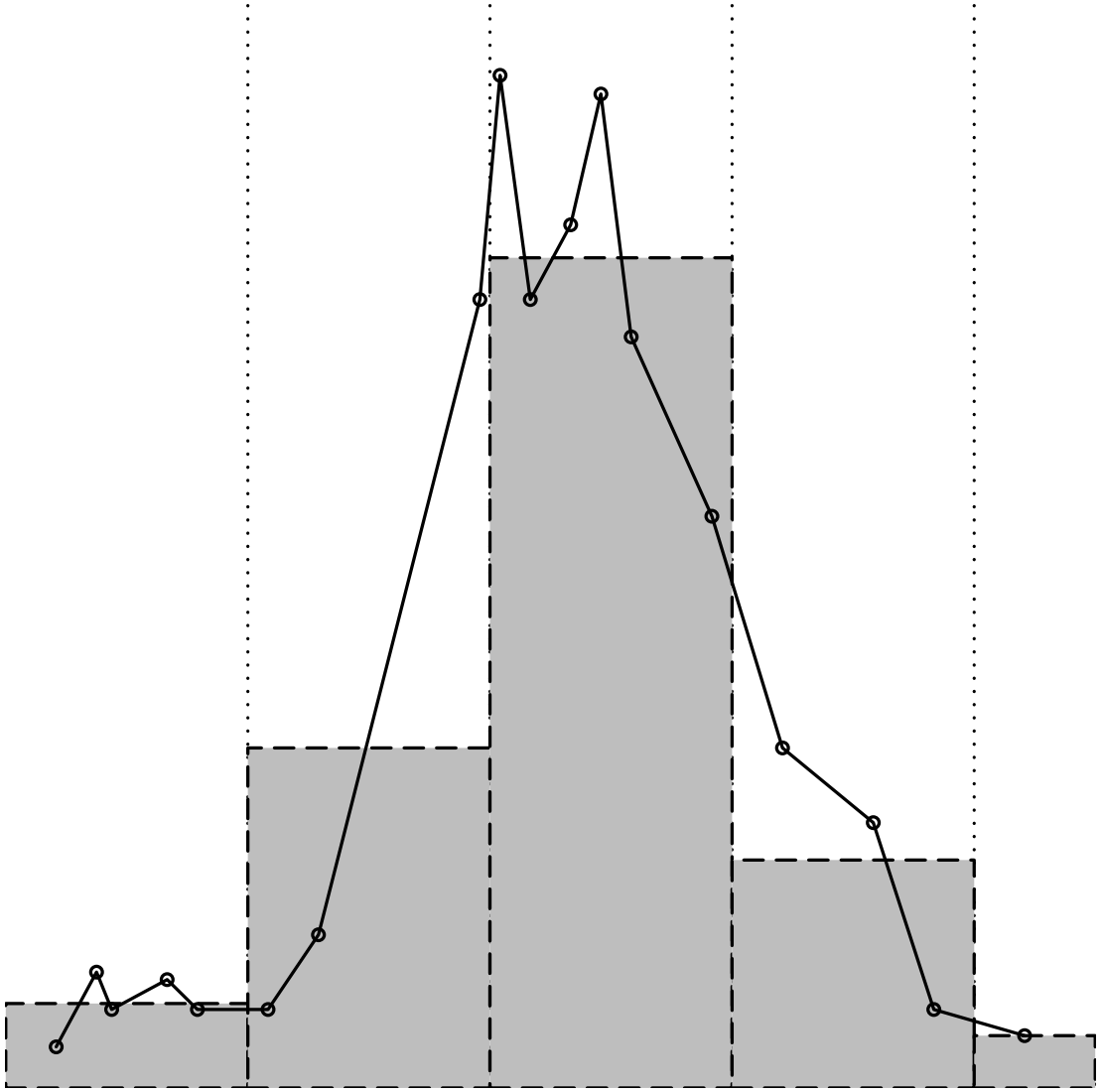
Abschätzung der Selektivität

Parametrisierte Verteilung



Abschätzung der Selektivität

Histogramm



Abschätzung der Größe der Zwischenergebnisse

Beispiel (ehemalige Prüfungsfrage):

- Uni Datenbank mit Relationen Professoren (kurz *prof*), Studenten (*s*), Vorlesungen (*v*) und Prüfen (*prf*).
- SQL Anfrage:

```
SELECT *
FROM Professoren prof, Studenten s,
     Vorlesungen v, pruefen prf
WHERE prof.PersNr = prf.PersNr AND v.VorlNr = prf.VorlNr
AND s.MatrNr = prf.MatrNr AND s.Semester = 1
AND v.titel = 'Datenbanksysteme'
```

d.h.: Prüfungen von Studenten im 1. Semester über Vorlesungen, deren Titel 'Datenbanksysteme' lautet.

Die relationale Uni DB

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

Studenten		
MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schoppenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Vorlesungen			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

hoeren	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

Assistenten			
PersNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2126

voraussetzen	
Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

pruefen			
MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

Beispiel

Fortsetzung

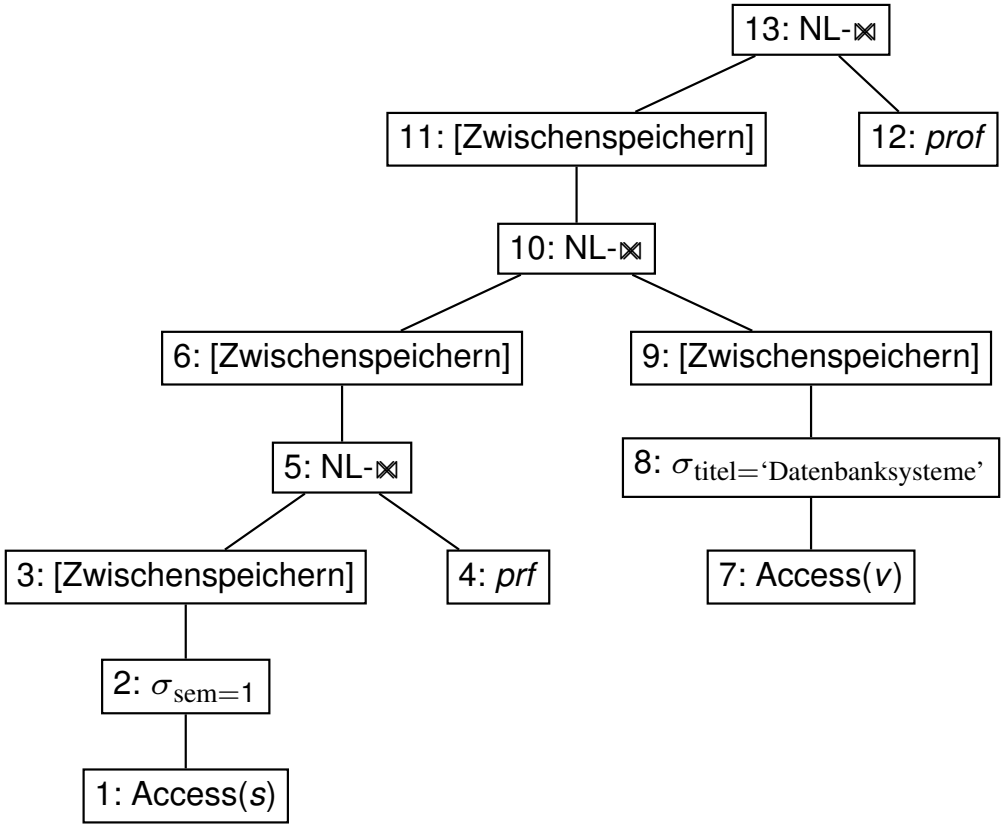
- Informationen über Größe dieser Relationen: $|prof| = 800$, $|s| = 38000$, $|v| = 2000$, und $|prf| = 400000$.
- Abschätzung einiger Selektivitäten:

$$
\begin{aligned}
\mathrm{sel}_{prof/prf} &= 1/800 = 0.00125 && \text{(Fremdschlüssel)} \\
\mathrm{sel}_{v/prf} &= 1/2000 = 0.0005 && \text{(Fremdschlüssel)} \\
\mathrm{sel}_{s/prf} &= 1/38000 \approx 2.63 \cdot 10^{-5} && \text{(Fremdschlüssel)} \\
\mathrm{sel}_{s.\mathrm{Semester}} &= 0.1 \\
\mathrm{sel}_{s.\mathrm{Titel}} &= 0.001
\end{aligned}
$$

Beispiel

Fortsetzung

Operatorbaum:



Beispiel

Fortsetzung

Geschätzte Anzahl der Tupel in den Zwischenergebnissen:

Knotennummer	Anzahl Tupel
1	38000
2	3800
3	3800
4	400000
5	40000
6	40000
7	2000
8	2
9	2
10	40
11	40
12	800
13	40

Kostenabschätzung der Operationen

- Idee
  - hauptsächlich: I/O-Kosten
  - CPU-Kosten in erträglichem Rahmen halten, z.B.: in-memory Hash Table in Probe Phase eines Hash Join oder bei Nested Loop Join.
- Notation:
  - m: Anzahl der Seitenrahmen im Datenbank-Puffer
  - b_R, b_S: Anzahl der Seiten (am Hintergrundspeicher) für die Relationen R bzw. S
  - M_R, M_S: Anzahl der Tupel von R bzw. S
  - p_R, p_S: Anzahl der Tupel pro Seite (von R bzw. S)

Kostenabschätzung

Selektion

- Falls Input der Selektion von einem anderen Operator kommt, entstehen durch die Selektion keine zusätzlichen Kosten.
- Selektion $\sigma_p(R)$, Tabelle R auf der Platte, ohne Index: alle Seiten lesen, d.h. Kosten b_R
- Selektion $\sigma_{A=c}(R)$, Tabelle R auf der Platte, A ist ein Schlüssel (d.h.: max. 1 qualifizierendes Tupel):
  - mit B^+-Index (realistische Annahme: Tiefe des Baums max. 4, Wurzel des Baums befindet sich im Hauptspeicher): Kosten für Finden des Blattknotens ≤ 4
  - mit Hash-Index: statisches Hashing (ohne Überlauf): 1 I/0, erweiterbares Hashing: 1 zusätzlicher I/O für Indirektion
  - Falls Index nur TIDs enthält: 1 weiterer I/O für Tupel

Kostenabschätzung

Selektion

Selektion $\sigma_{A=c}(R)$, Tabelle R auf der Platte, Selektivität $\mathrm{sel}_{A=c}$

- mit ungeballtem Index:
  - Suche erstes qualifizierendes Tupel wie zuerst: Kosten $\approx$ 1 bis 5 I/O (je nach Index-Art).
  - Die weiteren Indexeinträge sind „in der Nähe“, aber die Tupel sind „zufällig“ verteilt.
    - Kosten pro Tupel: 1 I/O
    - Kosten für alle Tupel: $M_R \cdot \mathrm{sel}_{A=c}$
- mit geballtem Index:
  - Suche erstes qualifizierendes Tupel wie zuerst.
  - Ab dort sequentielle Suche aller weiteren Treffer:
    - Anzahl der gesuchten Tupel: $M_R \cdot \mathrm{sel}_{A=c}$
    - Anzahl der benötigten Seiten: $b_R \cdot \mathrm{sel}_{A=c}$
    - Kosten für alle Tupel: $b_R \cdot \mathrm{sel}_{A=c}$

Kostenabschätzung

Sortierung

- Für Erzeugung der Level 0 Runs werden alle Seiten einmal eingelesen (im Puffer sortiert) und wieder ausgeschrieben.
  Kosten für Erzeugung der Level 0 Runs: $2b_R$
- Länge der Level 0 Runs: m Seiten
  Anzahl der Level 0 Runs: $i = \lceil b_R/m \rceil$
- Bei jedem Pass werden $m-1$ Runs zu einem gemerged.
  Anzahl der benötigten Passes: $l = \lceil \log_{m-1}(i) \rceil$
- Bei jedem Pass werden alle Seiten einmal eingelesen und wieder ausgeschrieben, d.h. $2b_R$ I/O pro Pass
- Gesamtkosten:
  $2b_R + l \cdot 2b_R = 2b_R \cdot (1 + l) = 2b_R \cdot (1 + \lceil \log_{m-1}(\lceil b_R/m \rceil) \rceil)$

Bemerkung: Bei der Kostenformel im Buch (Kap. 8.3.4) wurden die Kosten für das Erstellen der Level 0 Runs vernachlässigt.

Kostenabschätzung

Joins

- Join Methoden:
  - (Block) Nested Loop Join
  - Sort Merge Join
  - Index Nested Loop Join
  - (Hybrid) Hash Join

Beispiel

Vorlesungen $\bowtie_{\text{gelesenVon}=\text{PersNr}}$ Professoren
(R = Vorlesungen, S = Professoren)

$b_R = 1000$	$b_S = 500$	(Anzahl der Seiten)
$M_R = 100000$	$M_S = 50000$	(Anzahl der Tupel)
$p_R = 100$	$p_S = 100$	(Tupel pro Seite)
$m = 100$		(Seitenrahmen im DB Puffer)

(Simple) Nested Loop Join

I/O-Kostenformel:

- Jede Seite von R (= äußere Relation) wird einmal gelesen: b_R I/Os
- Für jedes Tupel von R muss jede Seite von S einmal gelesen werden: $M_R \cdot b_S$ I/Os
- Gesamtkosten: $b_R + M_R \cdot b_S$ I/Os

Beispiel

Gesamtkosten: $1000 + 100000 \cdot 500 = 50001000$ I/Os
Bei 10ms pro I/O: ca 140 Stunden!

Pagewise Nested Loop Join

I/O-Kostenformel:

- Jede Seite von R (= äußere Relation) wird einmal gelesen: b_R I/Os
- Für jede Seite von R muss jede Seite von S einmal gelesen werden: $b_R \cdot b_S$ I/Os
- Gesamtkosten: $b_R + b_R \cdot b_S$ I/Os

Beispiel

Gesamtkosten: $1000 + 1000 \cdot 500 = 501000$ I/Os
Bei 10ms pro I/O: ca 1,4 Stunden!

Block Nested Loop Join

I/O-Kostenformel:

- Jede Seite von R wird einmal gelesen: b_R I/Os
- Für jeden Block aus $(m - k - 1)$ Seiten von R muss jede Seite von S einmal gelesen werden. Ab dem zweiten Durchlauf von S stehen die ersten k Seiten bereits im Puffer.
  - 1. Durchlauf von S: b_S I/Os
  - Weitere Durchläufe von S: jeweils $(b_S - k)$ I/Os
  - Gesamtanzahl der Durchläufe: $\lceil b_R/(m - k - 1) \rceil$
- insgesamt: $b_R + k + \lceil b_R/(m - k - 1) \rceil \cdot (b_S - k)$ I/Os
- für $k = 1$: $b_R + 1 + \lceil b_R/(m - 2) \rceil \cdot (b_S - 1)$ I/Os

Bemerkung: Die I/O-Kosten sind minimal, wenn

- R die kleinere Relation ist (d.h.: weniger Seiten als S) und
- $k = 1$ gewählt wird.

Block Nested Loop Join

Beispiele

- Gesamtkosten für $k = 32$:
  $1000 + 32 + \lceil 1000/(100 - 33) \rceil \cdot (500 - 32) = 8052$
- Gesamtkosten für $k = 16$:
  $1000 + 16 + \lceil 1000/(100 - 17) \rceil \cdot (500 - 16) = 7308$
- Gesamtkosten für $k = 1$:
  $1000 + 1 + \lceil 1000/(100 - 2) \rceil \cdot (500 - 1) = 6490$

Bemerkung: Im Buch von Kemper (Kap. 8.3.3) wurde der Output-Puffer vernachlässigt. Dann erhält man die (unvollständige) Formel $b_R + k + \lceil b_R/(m - k) \rceil \cdot (b_S - k)$ I/Os. Diese Formel stimmt nur, wenn das Ergebnis tupelweise (und nicht seitenweise) weiterverarbeitet wird.

Index Nested Loop Join

I/O-Kostenformel:

- Jede Seite von R wird einmal gelesen: b_R I/Os
- Für jedes Tupel in R: Zugriff auf qualifizierende Tupel in S ist „normale“ Selektion, d.h. (je nach Art des Index) Kosten für erstes qualifizierendes Tupel in S: 1 bis 5 I/O.
- Insgesamt, falls es für jedes Tupel in R maximal 1 qualifizierendes Tupel in S gibt (d.h. B Schlüssel in S): $b_R + c \cdot M_R$ I/O mit $c \in [1, 5]$ (je nach Art des Index).
- Insgesamt, falls es für jedes Tupel in R mehrere qualifizierende Tupel in S geben kann:
  geballter Index: ca. $b_R + M_r \cdot (c + b_S \cdot \mathrm{sel}_{RS})$
  ungeballter Index: ca. $b_R + M_r \cdot (c + M_S \cdot \mathrm{sel}_{RS})$
  mit $c \in [1, 5]$

Index Nested Loop Join

Beispiel

- Hash-Index auf Attribut $S.B$.
  Annahme: Auslesen des **TID** in durchschnittlich **1,2** I/O.
- In unserem Beispiel: B (= „PersNr“) ist Schlüssel in S, d.h. 1 Selektion in S pro Tupel von R
- Gesamtkosten:
  $b_R + (\mathbf{1,2} + \mathbf{1}) \cdot M_R = 1000 + 2,2 \cdot 100000 = 221000$

Beispiel

- B^+-Index auf Attribut $S.B$.
  Annahme: Auslesen eines **Tupels** in durchschnittlich **4** I/O
- Gesamtkosten: $b_R + \mathbf{4} \cdot M_R = 1000 + 4 \cdot 100000 = 401000$

Sort Merge Join

I/O-Kostenformel:

- Kosten für das Sortieren von R: $2b_R \cdot (1 + l_R)$ mit $i_R = \lceil b_R/m \rceil$ und $l_R = \lceil \log_{m-1}(i_R) \rceil$
- Kosten für das Sortieren von S: $2b_S \cdot (1 + l_S)$ mit $i_S = \lceil b_S/m \rceil$ und $l_S = \lceil \log_{m-1}(i_S) \rceil$
- Kosten für Merge Join: Falls entweder A in R oder B in S ein Schlüssel ist, genügt je 1 Durchlauf von R und S, d.h.: $b_R + b_S$ I/Os (zähle nur Kosten fürs Lesen).
- Kosten für Merge Join, falls es für jedes Tupel in R mehrere qualifizierende Tupel in S haben kann und umgekehrt: Merge Join kann letztlich zu Nested Loop Join entarten, wenn (fast) alle Werte von $R.A$ und $S.B$ gleich sind.

Sort Merge Join

Beispiel

- Kosten für das Sortieren:

$$i_R = \lceil b_R/m \rceil = 1000/100 = 10 \qquad l_R = \lceil \log_{99}(10) \rceil = 1$$
$$i_S = \lceil b_S/m \rceil = 500/100 = 5 \qquad l_R = \lceil \log_{99}(5) \rceil = 1$$

  Kosten für Sortieren von R: $2 \cdot 1000 \cdot (1 + 1) = 4000$
  Kosten für Sortieren von S: $2 \cdot 500 \cdot (1 + 1) = 2000$
- Kosten für Merge Join (B ist ein Schlüssel in S):
  $1000 + 500$
- Gesamtkosten für Sort Merge Join:
  $4000 + 2000 + 1000 + 500 = 7500$

Sort Merge Join

Bemerkung: Idee der Kostenberechnung für das Sortieren:

- Pass 0: Mit 100 Seitenrahmen im Puffer werden die 1000 Seiten von R in 10 Level 0 Runs aufgeteilt.
  Kosten: $2 \cdot 1000$ (für je einmal Lesen und Schreiben von R)
- Mit 100 Seitenrahmen im Puffer können diese 10 Runs in einem einzigen weiteren Pass zusammengeführt werden.
  Kosten: $2 \cdot 1000$ (für je einmal Lesen und Schreiben von R)
- Gesamtkosten für das Sortieren von R: $4 \cdot 1000 = 4000$.
- Analog die Kosten für das Sortieren von S: $4 \cdot 500 = 2000$.

Hash Join

I/O-Kostenformel:

- Build-Phase (Annahme: Die Buckets sind klein genug, so dass die Buckets nicht rekursiv noch einmal gehasht werden müssen): Kosten von $2 \cdot (b_R + b_S)$ I/Os (d.h.: je einmal lesen und schreiben).
- Probe-Phase: Jede Seite von R und S wird je ein Mal durchlaufen: $b_R + b_S$ I/Os (zähle nur Kosten fürs Lesen)
- Gesamtkosten: $3 \cdot (b_R + b_S)$ I/Os

Beispiel

Gesamtkosten: $3 \cdot (1000 + 500) = 4500$ I/Os

Hybrid Hash Join

Annahme:

- Unterteilung von R und S in n (fast) gleich große Buckets.
- Es passen k Buckets von S in den Puffer.

I/O-Kostenformel:

- Build-Phase: Je k Buckets von R und S müssen nicht ausgeschrieben werden. Ersparnis: $(k/n) \cdot (b_R + b_S)$
- Probe-Phase: Je k Buckets von R und S müssen nicht mehr eingelesen werden. Ersparnis: $(k/n) \cdot (b_R + b_S)$
- Gesamtkosten: $(3 - 2k/n) \cdot (b_R + b_S)$
- Idealfall: Eine der beiden Relationen passt zur Gänze in den Puffer. D.h.: $k = n$ und deshalb $k/n = 1$
  Gesamtkosten: $(3 - 2) \cdot (b_R + b_S) = (b_R + b_S)$

Hybrid Hash Join

Beispiel

- Annahme: Hashing in 16 ca. gleich große Buckets. D.h., 2 Buckets (zu je 32 Seiten) von S passen in den Puffer. Bemerkung: 3 Buckets haben nicht Platz, da man noch 1 Seite für den Input und je 1 Seite für die restlichen Buckets braucht: $3 \cdot 32 + 1 + 13 > 100$.
- Build-Phase: R und S werden zur Gänze eingelesen. Aber nur 14 (von 16) Buckets werden ausgeschrieben:
  Kosten: $(1 + 14/16) \cdot (1000 + 500) \approx 2820$ I/Os
- Probe-Phase: Nur noch 14 (von 16) Buckets von R und S müssen eingelesen werden:
  Kosten: $14/16 \cdot (1000 + 500) \approx 1320$ I/Os
- Gesamtkosten: 4140 I/Os [$\approx (3 - 4/16) \cdot (1000 + 500)$]

Kosten der übrigen Operationen

Projektion:

- Keine Duplikatelimination in der physischen Algebra.
- Projektion wird üblicherweise mit einem anderen Schritt kombiniert, i.e. I/O-Kosten der Projektion: 0.

Weitere Operationen:

- Die anderen Operationen (Duplikatelimination, Gruppierung, die meisten Mengenoperationen) werden üblicherweise mittels Sortierung oder Hashing implementiert.
- Kostenabschätzungen dieser Operationen erhält man auf ähnliche Weise wie für Sort Merge Join bzw. Hash Join.

Fehler bei den Schätzungen

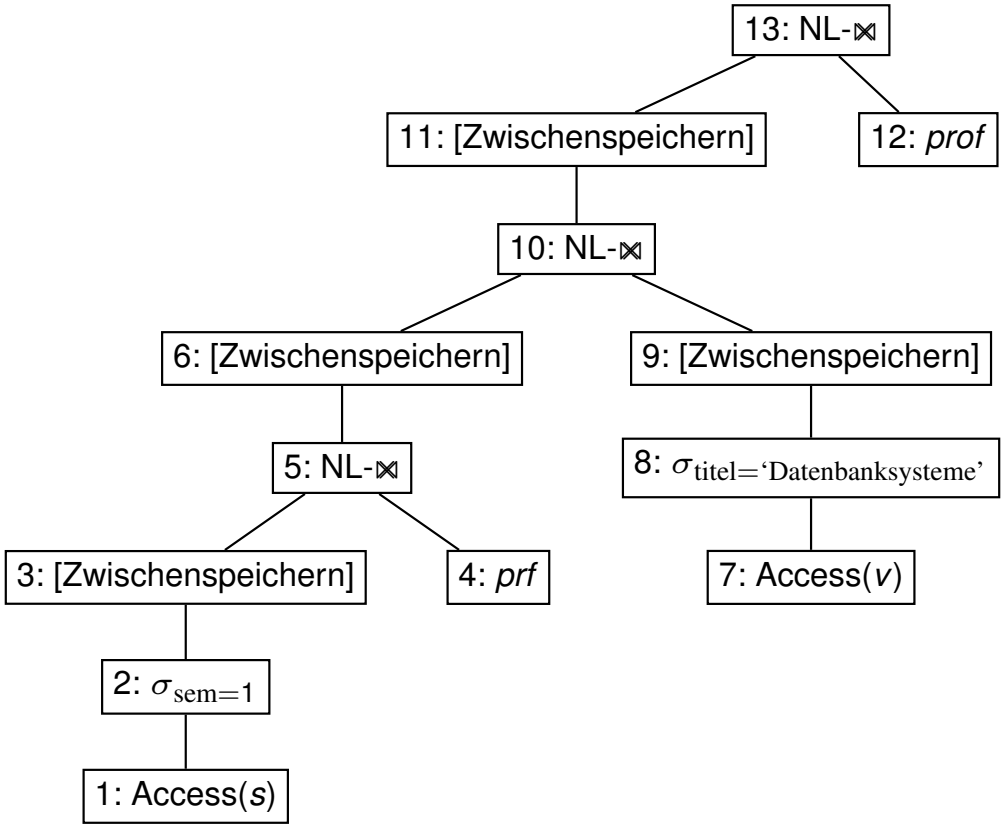
Die Kostenabschätzungen liefern nur **ungefähre Werte**.

- Vereinfachende Annahmen, z.B.:
  - Annahme der Gleichverteilung (Selektivität)
  - Effekt von Random I/O vs. Chained I/O ignoriert
- Vernachlässigte Kosten, z.B.:
  - Bei einigen Methoden ist In-Memory Hash-Tabelle erforderlich (die etwas zusätzlichen Speicher braucht)
- „Kleine Ungenauigkeiten“, z.B.:
  - Alle Zahlen müssen aufgerundet werden.
  - Block NL Join im Kemper-Buch: Output-Puffer „vergessen“.

ABER: Diese Ungenauigkeiten ändern nichts an den **grundsätzlichen Aussagen** beim Vergleich der Methoden!

Kostenabschätzung

Beispiel (ehemalige Prüfungsfrage, Fortsetzung):



Beispiel

Fortsetzung

Geschätzte Anzahl der Tupel in den Zwischenergebnissen:

Knotennummer	Anzahl Tupel
1	38000
2	3800
3	3800
4	400000
5	40000
6	40000
7	2000
8	2
9	2
10	40
11	40
12	800
13	40

Beispiel

Fortsetzung

- Durchschnittliche Tupelgrößen: *prof*: 50 Bytes, *s*: 50 Bytes, *v*: 100 Bytes, *prf*: 25 Bytes
- Computer: Seitengröße: 1024 Bytes, DB Puffer: 20 Seiten
- (Vereinfachende) Annahmen:
  - Tupelgröße bei Join von 2 Relationen: Summe der einzelnen Tupelgrößen
  - Pro Seite stehen 1000 Bytes für die Tupel zur Verfügung.

Gesucht für jeden Knoten: Tupelgröße, Anzahl der Seiten, I/O-Kosten, Kostenformel

Beispiel

Fortsetzung

Knoten	Tupel	Größe	Seiten b_i	Kostenformel	Page I/O
1	38000	50	1900		1900
2	3800	50	190		0
3	3800	50	190		190
4	400000	25	10000		0
5	40000	75	3000	$b_3 + 1 + \lceil b_3/18 \rceil \cdot (b_4 - 1)$	110180
6	40000	75	3000		3000
7	2000	100	200		200
8	2	100	1		0
9	2	100	1		1
10	40	175	7	$b_6 + 1 + \lceil b_6/18 \rceil \cdot (b_9 - 1)$	3001
11	40	175	7		7
12	800	50	40		0
13	40	225	9	$b_{11} + 1 + \lceil b_{11}/\mathbf{18} \rceil \cdot (b_{12} - 1)$	47

Gesamtkosten: 118526

Bemerkung: Falls das Endergebnis tupelweise ausgegeben wird (z.B. direkt am Bildschrim), dann gilt in Schritt 13: $b_{11} + 1 + \lceil b_{11}/\mathbf{19} \rceil \cdot (b_{12} - 1)$.

Optimierungsverfahren

Zerlegung in Teilprobleme:

- Zerlegung der SQL Query in „Blöcke“ mit genau einer select-from-where Klausel (+ eventuell group by, having)
- Getrennte Optimierung der Auswertung der Blöcke
- Einzelner Block: zuerst select-from-where optimieren, dann Rest auswerten (group by, having, Aggregatfunktionen)
- Nested Subqueries:
  - Falls Ergebnis einzelner Wert: auswerten und einsetzen
  - Falls Ergebnis Menge von Tupeln, unkorrelierte Subquery: auswerten und zwischenspeichern, dann (Block) Nested Loop Join (mit Ergebnis der Subquery als innere Relation)
  - Falls Ergebnis Menge von Tupeln, korrelierte Subquery: für jedes Tupel der äußeren Relation auswerten und joinen (vergleichbar mit Simple Nested Loop Join)

Optimierungsverfahren

Einschränkung des Suchraums (select-from-where):

- Betrachte nur „left-deep trees“, d.h.: bei jeder Join Operation ist die rechte Relation eine Basistabelle.
- Für die linke Relation eines Joins wird zunächst Pipelining angenommen (falls die Joinmethode Zwischenspeichern erfordert, wird dies zu den Kosten des Join gezählt).
- Projektionen und Selektionen werden so weit wie möglich nach unten geschoben. Ihre Auswertung erfolgt entweder als Teil des Zugriffs oder als Teil des Joins.

Optimierungsverfahren:

- In erster Linie nur noch Festlegung der Join-Reihenfolge.
- Standardverfahren in heutigen relationalen DB Systemen: **dynamische Programmierung**

Optimierung durch Dynamische Programmierung

1. Schritt:
   - Betrachte nur **Pläne für eine einzige Relation**.
   - Identifiziere alle Selektionen, die sich nur auf Attribute dieser Relation beziehen.
   - Identifiziere alle Attribute dieser Relation, die nirgendwo gebraucht werden, i.e. definiere passende Projektionen.
   - Die Selektionen und Projektionen können (aber müssen nicht) ausgeführt werden.
   - Berechne für jede Relation den billigsten Plan, z.B.: Zugriff auf Basistabellen mit unterschiedlichen Indexen oder ohne Index.
   - Für jede Relation werden alle Pläne außer dem billigsten gelöscht.

Optimierung durch Dynamische Programmierung

2. Schritt:

- Betrachte nur **Pläne für zwei Relationen**.
- Dabei werden alle im 1. Schritt behaltenen Pläne mit einer zweiten Relation gejoined.
- Bestimme alle möglichen Projektionen.
- Die Selektionen und Projektionen können (aber müssen nicht) ausgeführt werden.
- Berechne für jede Kombination aus 2 Relationen den billigsten Plan, z.B.: unterschiedliche Joinmethoden.
- Für jede Kombination aus 2 Relationen werden alle Pläne außer dem billigsten gelöscht.

Optimierung durch Dynamische Programmierung

3. Schritt:

- Betrachte nur **Pläne für drei Relationen**.
- Dabei werden alle im 2. Schritt behaltenen Pläne mit einer dritten Relation gejoined.
- et cetera

Dieser Prozess wird solange wiederholt, bis schließlich **Pläne für alle Relationen** dieser Anfrage erzeugt und bewertet werden.

Datenbank-Tuning

Analyse der Workload

- Datenbankanfragen: Zugriff auf welche Tabellen, Join/Selektions-Attribute, Projektion auf welche Attribute)
- Datenänderungen: Operationen (update/insert/delete), Selektionsbedingungen, von Änderungen betroffene Attribute
- Häufigkeit der verschiedenen Statements
- Spezielle Zeitanforderungen (z.B.: besonders zeitkritische Statements bzw. Transaktionen)

Datenbank-Tuning

Entscheidungen des physischen Datenbankdesigns

- Die wichtigste Frage ist: welche Indexe?
  Ziel: Abarbeitung bestimmter Statements ohne Durchlaufen der ganzen Tabelle.
  - welche Attribute bzw. Attributkombinationen?
  - Index geballt oder nicht?
  - Index-Typ: Hash Index (Punktanfragen) oder B^+-Tree
- Entscheidungen des logischen Datenbankdesigns;
  Ziel: Vermeide Joins bzw. Zugriff auf “unnötige” Attribute
  - Denormalisierung: 3NF-Verletzung ist eventuell gerechtfertigt, um häufigen Join zu vermeiden.
  - Alternative: materialized view
  - vertikale Partitionierung einer Tabelle (im Extremfall: Schlüssel + 1 Attribut pro Tabelle).
  - horizontale Partitionierung einer Tabelle (d.h.: Aufsplitten einer Tabelle in mehrere schema-gleiche Tabellen).

Anlegen von Datenbank-Statistiken

- Statistiken (Histogramme, etc.) müssen explizit angelegt werden. Andernfalls liefern die Kostenmodelle falsche Werte.
- In Oracle ...
  - ANALYZE **TABLE** Professoren COMPUTE STATISTICS FOR **TABLE**
  - Man kann auch approximative Statistiken verwenden: anstatt COMPUTE verwendet man ESTIMATE.
- In DB2 ...
  - RUNSTATS **ON TABLE** ...
- In PostgreSQL ...
  - ANALYZE bzw. VACUUM ANALYZE

Analyse von Abarbeitungsplänen

- DBMSe erlauben die Ausgabe der Abarbeitungspläne, um Performance-Probleme zu analysieren.
- in PostgreSQL: EXPLAIN Kommando.

```
EXPLAIN [ ( option [, ...] ) ] statement

Optionen:
    ANALYZE [ boolean ]      (default: OFF)
    VERBOSE [ boolean ]      (default: OFF)
    COSTS [ boolean ]        (default: ON)
    BUFFERS [ boolean ]      (default: OFF)
    TIMING [ boolean ]       (default: ON)
    FORMAT { TEXT | XML | JSON | YAML }
            (default: TEXT)
```

Parameter beim EXPLAIN-Kommando

- ANALYZE ON: Ausgabe der tatsächlichen Kosten bei Ausführung des Statements (ansonsten nur Schätzungen).
- VERBOSE ON: detailliertere Informationen
- COSTS ON: Ausgabe der (geschätzten) Kosten für *jeden Knoten* (inklusive Anzahl der Tupel, Größe der Tupel, etc.)
- BUFFERS ON: Informationen zur Puffer-Verwendung
- TIMING OFF: damit kann man die Zeitnehmung für die einzelnen Knoten im Plan ausschalten
- FORMAT: spezifiert das Ausgabeformat.

Analyse von Abarbeitungsplänen

Visual Explain

Visualisierung von

- Zugriffsplänen
- Datenbankoperationen
- und deren geschätzten Kosten.

In der Praxis meist GUIs mit visual explain Komponente:

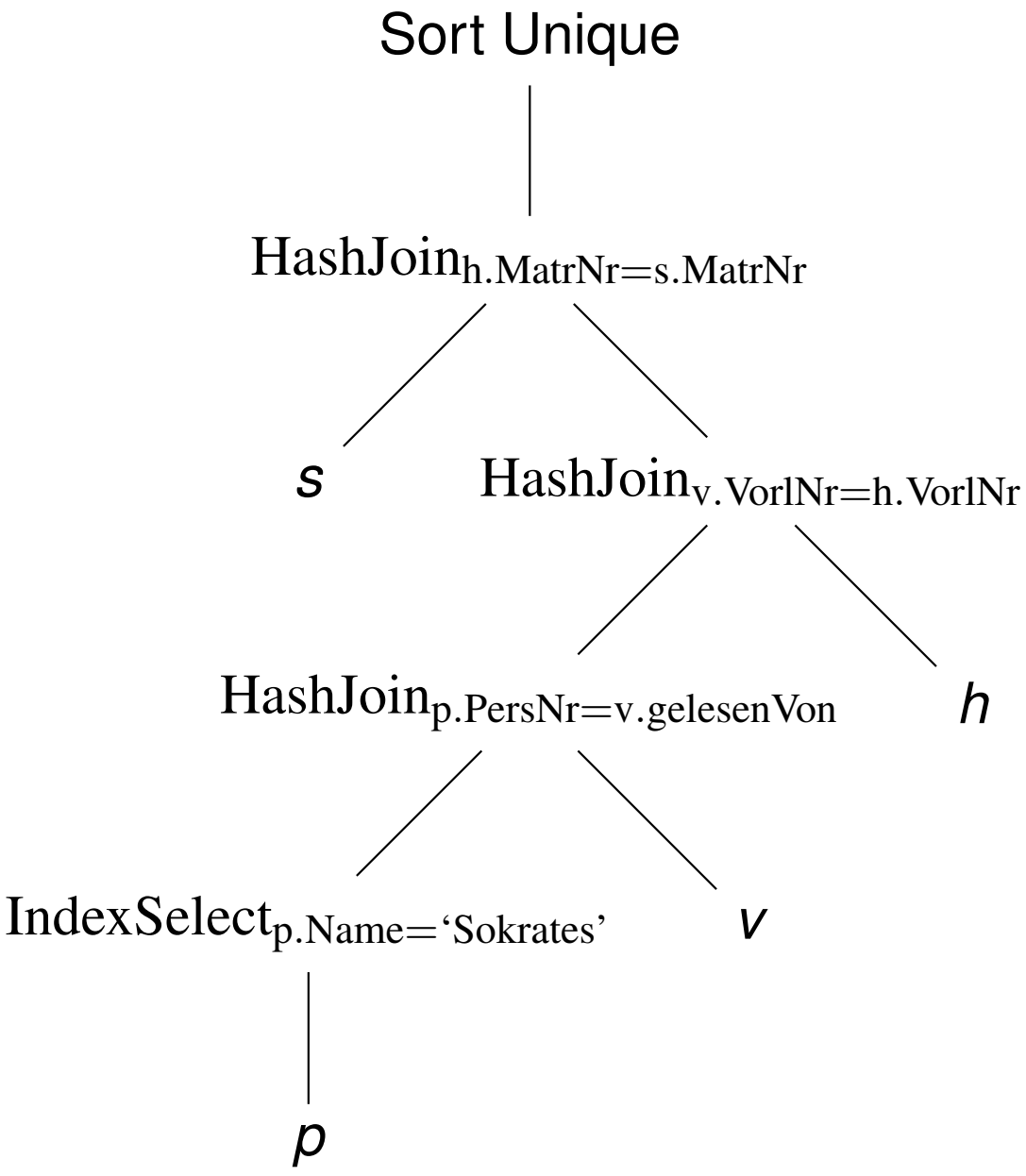
- In PostgreSQL . . .
  - pgAdmin
- In DB2 . . .
  - IBM Data Studio

EXPLAIN PLAN: Oracle Beispiel

```
EXPLAIN PLAN FOR
SELECT DISTINCT s.Semester
FROM Studenten s, hoeren h,
     Vorlesungen v, Professoren p
WHERE p.Name = 'Sokrates' AND
      v.gelesenVon = p.PersNr AND
      v.VorlNr = h.VorlNr AND
      h.MatrNr = s.MatrNr
```

```
SELECT STATEMENT   Cost = 37710
  SORT UNIQUE
    HASH JOIN
      TABLE ACCESS FULL STUDENTEN
      HASH JOIN
        HASH JOIN
          TABLE ACCESS BY ROWID PROFESSOREN
            INDEX RANGE SCAN PROFNAMEINDEX
          TABLE ACCESS FULL VORLESUNGEN
        TABLE ACCESS FULL HOEREN
```

Baumdarstellung



Anfrage-Tuning

Mögliche Maßnahmen zur Performance-Verbesserung

- Manche Anfrage-Optimierer verwenden bei arithmetischen Ausdrücken keinen Index, z.B.:
  ersetze `where salary / 365 > 200`
  durch `where salary > 365 * 200`.
- Duplikatelimination: ist diese aus Anwendungssicht wirklich nötig? Ist die `DISTINCT` Anweisung eventuell redundant? Verwendung von `UNION ALL` vs. `UNION`, ...
- Wenn es mehrere Möglichkeiten gibt, JOIN-Bedingungen auszudrücken: Wähle nach Möglichkeit eine JOIN-Bedingung mit einem geballten Index und vermeide String-Vergleiche, z.B. (Annahme: `name` ist `UNIQUE`)
  ersetze `employee.name = student.name`
  durch `employee.ssn = student.ssn`, falls Attribut `ssn` einen geballten Index in einer der Tabellen hat.

Anfrage-Tuning

Mögliche Maßnahmen zur Performance-Verbesserung

- Bei manchen Anfrage-Optimierern hat die Reihenfolge der Tabellen in der `FROM`-Klausel möglicherweise einen Einfluss auf die tatsächlich verwendete JOIN-Reihenfolge.
- Häufig werden für den Datenzugriff einer Applikation eigene Views mit den für diese Applikation relevanten Daten definiert. Vorsicht bei Verwendung von Views, die mittels JOIN definiert sind: Sind die JOINs der View für die konkrete Anfrage wirklich nötig? Wäre Formulierung der Anfrage mittels Zugriff auf die Basistabellen möglich?

Anfrage-Tuning

Vorsicht mit geschachtelten SELECTs

Beispiel

```
SELECT ssn
FROM Employee E
WHERE salary = SELECT MAX(salary)
               FROM Employee M
               WHERE M.dept_no = E.dept_no
```

- Möglicherweise wird diese Anfrage mittels nested loops ausgewertet, so dass für *jedes* Tupel von E die innere Anfrage ausgewertet wird.
- besser: berechne alle Kombinationen ⟨`dept_no`, `MAX(salary)`⟩ vorab mittels `WITH`-Statement.

Anfrage-Tuning

Vorsicht mit geschachtelten SELECTs

Beispiel

```
SELECT ssn
FROM Employee
WHERE dept_no IN (SELECT d_no FROM Department
                  WHERE mgr_ssn = 123456)
```

- Bei geschachtelten Anfragen mit `IN`-Operator wird häufig kein Index für den Zugriff auf die innere Tabelle verwendet (wodurch letztlich eine Art von nested loop join entsteht).
- besser: ersetze geschachtelte Anfrage durch ungeschachtelte Anfrage
  (hier: `FROM Employee, Department ...`)