Datenbanksysteme

Wintersemester 2016/17

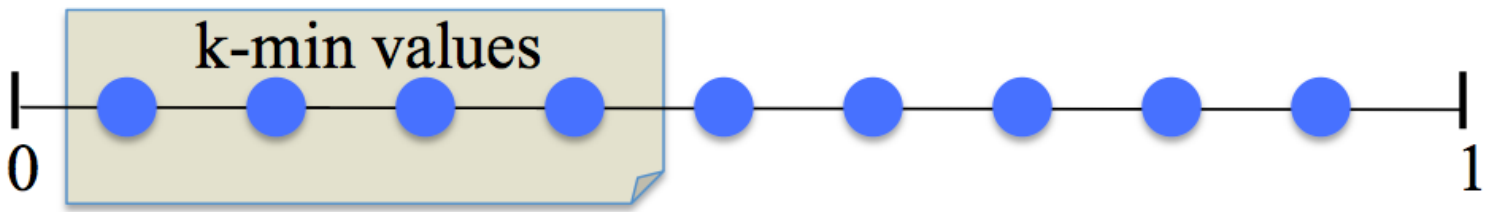
Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

K-Min Value (KMV) Sketch

Problemstellung wie zuvor: Gegeben eine Multimenge S von Elementen, finde die Anzahl n der unterschiedlichen (distinct) Elemente.

- Wende Hashfunktion auf Elemente an zur gleichverteilten Abbildung auf $[0,1]$
- Die KMV Synopse besteht dann aus den k kleinsten dieser Werte $L = \{U_{(1)}, U_{(2)}, \ldots, U_{(k)}\}$

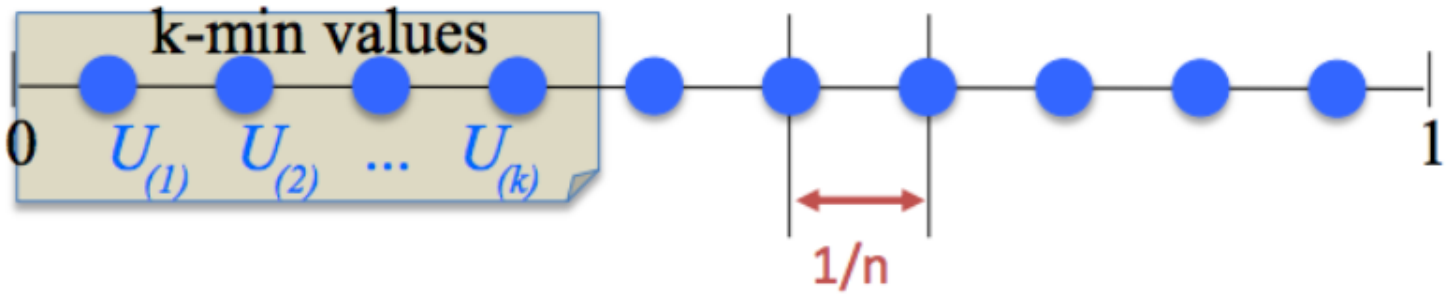


- Unbiased (Deutsch: erwartungstreuer) Schätzer erhalten wir durch

$$\hat{n}^{UB} = (k-1)/U_{(k)}$$

K-Min Value (KMV) Sketch (2)

Wieso funktioniert das?



- $L = \{U_{(1)}, U_{(2)}, \ldots, U_{(k)}\}$
- Distanz zwischen den Hashwerten ist $1/n$, ausgehend von Gleichverteilung.
- Wir möchten n abschätzen. Welche Größe kennen wir?
- Beobachtung

$$E[U_{(k)}] = k \times \frac{1}{n} \qquad \text{also} \qquad \hat{n}^{BE} = \frac{k}{U_{(k)}}$$

$\hat{n}^{BE}$ ist biased (nicht erwartungstreu).

K-Min Value (KMV) Sketch (3)

Beispiel

- MD5 als Hashfunktion, Abbildung auf $[0,1]$
- Hier, Elemente als einfache Buchstaben a–z.
- Die sortierten Hashwerte der 26 Buchstaben sind [0.043, 0.172, 0.281, 0.354, 0.382, 0.421, 0.443, 0.459, 0.463, 0.523, 0.556, 0.565, 0.569, 0.57, 0.59, 0.644, 0.652, 0.675, 0.682, 0.724, 0.818, 0.864, 0.89, 0.938, 0.994, 0.997]

	$\hat{n}^{UB}$	$U_{(k)}$
1	0.000	0.043
2	5.814	0.172
3	7.117	0.281
4	8.475	0.354
5	10.471	0.382
6	11.876	0.421
7	13.544	0.443
8	15.251	0.459
9	17.279	0.463
10	17.208	0.523
11	17.986	0.556
12	19.469	0.565
13	21.090	0.569
14	22.807	0.570
15	23.729	0.590
16	23.292	0.644
17	24.540	0.652
18	25.185	0.675
19	26.393	0.682

Allgemeine Beobachtungen

- Warum werden nur distinct Elemente gezählt? Abbildung von Duplikaten auf gleichen Wert.
- Gegeben zwei KMV Synopsen, für Multimengen A und B, kann man eine Synopse angeben für $A \cup B$? Ja, einfach kleinste der $2 * k$ Werte nehmen.

Count-Min-Sketch (CM-Sketch)

Gegeben eine Sequenz von Objekten

$$a_1, a_4, a_4, a_9, a_1, a_7$$

Wir möchten wissen wie oft Objekt a_i in der Sequenz vorkommt.

Idee hinter **CM-Sketch**:

- Führe einen Zähler für jede mögliche Ausgabe jeder Hashfunktion.
- Für Objekt a_i, wende Hashfunktion h_i auf a_i an und erhöhe Zähler zu $h_i(a_i)$ um 1.

Initial sind alle Zähler auf Null gesetzt.

Beispiel mit vier Hashfunktionen h_0, h_1, h_2 und h_3 mit Wertebereich $[0...7]$

	0	1	2	3	4	5	6	7
h_0	0	0	0	0	0	0	0	0
h_1	0	0	0	0	0	0	0	0
h_2	0	0	0	0	0	0	0	0
h_3	0	0	0	0	0	0	0	0

Count-Min-Sketch

Das zweidimensionale Array nennen wir *count*. Es hat d Zeilen, eine für jede Hashfunktion, und Breite w. Hashfunktionen $h_i : \{a_1, \ldots a_n\} \rightarrow \{1, \ldots w\}$

einfügen(a_i):

for each j do

$count[j,h_j(a_i)] + +$

end

Schätzwert für Auftreten von a_i ist dann gegeben durch

$$\hat{a}_i = min_j count[j,h_j(a_i)]$$

Paper: G. Cormode, S. Muthukrishnan: An improved data stream summary: the count-min sketch and its applications. J. Algorithms 55(1), 2005.
`http://sites.google.com/site/countminsketch/cm-latin.pdf`

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von p

	0	1	2	3	4	5	6	7
h_0	0	1	0	0	0	0	0	0
h_1	0	0	1	0	0	0	0	0
h_2	1	0	0	0	0	0	0	0
h_3	0	0	1	0	0	0	0	0

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von g

	0	1	2	3	4	5	6	7
h_0	0	1	0	0	0	0	1	0
h_1	0	0	1	0	1	0	0	0
h_2	1	1	0	0	0	0	0	0
h_3	0	0	1	0	0	1	0	0

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von w

	0	1	2	3	4	5	6	7
h_0	0	1	0	0	0	0	2	0
h_1	0	0	1	0	2	0	0	0
h_2	1	2	0	0	0	0	0	0
h_3	0	0	1	0	0	2	0	0

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von e

	0	1	2	3	4	5	6	7
h_0	1	1	0	0	0	0	2	0
h_1	1	0	1	0	2	0	0	0
h_2	1	2	0	1	0	0	0	0
h_3	0	0	1	1	0	2	0	0

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von w

	0	1	2	3	4	5	6	7
h_0	1	1	0	0	0	0	3	0
h_1	1	0	1	0	3	0	0	0
h_2	1	3	0	1	0	0	0	0
h_3	0	0	1	1	0	3	0	0

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von e

	0	1	2	3	4	5	6	7
h_0	2	1	0	0	0	0	3	0
h_1	2	0	1	0	3	0	0	0
h_2	1	3	0	2	0	0	0	0
h_3	0	0	1	2	0	3	0	0

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von j

	0	1	2	3	4	5	6	7
h_0	3	7	3	0	5	0	10	3
h_1	8	0	7	0	13	0	3	0
h_2	7	10	0	3	0	3	3	5
h_3	0	3	7	3	3	10	0	5

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von o

	0	1	2	3	4	5	6	7
h_0	3	7	3	0	5	0	11	3
h_1	8	0	7	0	14	0	3	0
h_2	7	11	0	3	0	3	3	5
h_3	0	3	7	3	3	11	0	5

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von g

	0	1	2	3	4	5	6	7
h_0	3	7	3	0	5	0	12	3
h_1	8	0	7	0	15	0	3	0
h_2	7	12	0	3	0	3	3	5
h_3	0	3	7	3	3	12	0	5

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von w

	0	1	2	3	4	5	6	7
h_0	3	7	3	0	5	0	13	3
h_1	8	0	7	0	16	0	3	0
h_2	7	13	0	3	0	3	3	5
h_3	0	3	7	3	3	13	0	5

p g w e w e p o g w p a a a o p k g j w p s s p o i j e p i j o g w q

Einfügen von q

	0	1	2	3	4	5	6	7
h_0	3	7	3	0	6	0	13	3
h_1	9	0	7	0	16	0	3	0
h_2	7	13	0	3	0	3	3	6
h_3	0	3	7	3	3	13	0	6

Der eben erzeugte Count-Min-Sketch:

	0	1	2	3	4	5	6	7
h_0	3	7	3	0	6	0	13	3
h_1	9	0	7	0	16	0	3	0
h_2	7	13	0	3	0	3	3	6
h_3	0	3	7	3	3	13	0	6

Wie oft kommt schätzungsweise 'a' vor? Wir schauen im Sketch an den durch die Hashfunktionen gegebenen Positionen und finden Zähler 6, 9, 6 und 6.

'a' kommt also höchstens 6 Mal vor.

Die Abbildung der Objekte bzgl. der Hashfunktionen:

	h_0	h_1	h_2	h_3
a	4	0	7	7
e	0	0	3	3
g	6	4	1	5
i	4	0	7	7
j	7	6	6	4
k	2	4	5	1
o	6	4	1	5
p	1	2	0	2
q	4	0	7	7
s	2	4	5	1
w	6	4	1	5

Count-Min-Sketch: Schätzer

Als Schätzwert für die Häufigkeit eines Objekts a wir der kleinste Zählerstand bzgl. der Hashfunktionen h_i verwendet.

Dieser Schätzwert ist eine obere Schranke für die tatsächliche Häufigkeit. Wieso?

Und wieso benutzt man nicht den größten Zählerstand oder den durchschnittlichen Zählerstand?

Count-Min-Sketch: Garantien

CM-Sketch mit Parametern (ε, δ) repräsentiert als zweidimensionales Array der Breite w und Tiefe d (also: w Spalten und d Zeilen), count$[1,1] \ldots$ count$[d,w]$, mit

- $w = \lceil \frac{e}{\varepsilon} \rceil$
- $d = \lceil ln \frac{1}{\delta} \rceil$

Der Schätzwert $\hat{a}_i$ hat folgende Garantien:

- $a_i \leq \hat{a}_i$ und
- mit Wahrscheinlichkeit von mindestens $1 - \delta$ gilt $\hat{a}_i \leq a_i + \varepsilon ||\mathbf{a}||_1$

e ist die Eulersche Zahl

Tuning von Datenbanken

- Statistiken (Histogramme, etc.) müssen (explizit) angelegt werden
- Andernfalls liefern die Kostenmodelle falsche Werte
- Oracle:
  - `analyze table Professoren compute statistics for table;`
  - Man kann sich auch auf approximative Statistiken verlassen
  - Anstatt compute verwendet man estimate
- DB2:
  - `runstats on table ....`
- Postgres:
  - `analyze`
  - `http://www.postgresql.org/docs/9.0/static/catalog-pg-statistic.html`

Statistiken in Postgresql

Tabelle analysieren mit:

```
analyze lineitem;
```

Daten werden in einer internen Tabelle (pg_statistic) abgelegt; pg_stats ist eine (besser zu lesende) Sicht darauf.

```
select *
from pg_stats
where tablename = 'lineitem';
```

Beispiel: Auszug aus pg_stats (in Postgresql)

Für Tabelle `lineitem` des TPC-H Datensatzes

(http://www.tpc.org/tpch/)

attname name	inheri boole	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray	most_common_freqs real[]	histogram_bounds anyarray	correlation real
l_orderkey	f	0	4	396485	{73190,578534,	{0.0001,0.0001,	{3,64448,1278	1
l_partkey	f	0	4	189666	{5893,21347,14	{0.000133333,0.	{2,2005,4000,5	0.003532
l_suppkey	f	0	4	10018	{9118,7099,747	{0.0004,0.00036	{2,96,200,303,	0.007977
l_linenumber	f	0	4	7	{1,2,3,4,5,6,7	{0.2474,0.21653		0.178991
l_quantity	f	0	5	50	{41.00,28.00,4	{0.0219,0.02163		0.022648
l_extendedprice	f	0	8	-0.12923	{73119.15,1216	{0.000133333,0.	{930.00,1474.4	0.004723
l_discount	f	0	4	11	{0.08,0.09,0.0	{0.0952,0.0922,		0.083595
l_tax	f	0	4	9	{0.07,0.06,0.0	{0.116433,0.115		0.104229
l_returnflag	f	0	2	3	{N,A,R}	{0.508433,0.247		0.371031
l_linestatus	f	0	2	2	{O,F}	{0.502267,0.497		0.499684
l_shipdate	f	0	4	2510	{1993-04-21,19	{0.000833333,0.	{1992-01-06,19	-0.00355
l_commitdate	f	0	4	2458	{1993-01-31,19	{0.0008,0.0008,	{1992-02-04,19	-0.00370
l_receiptdate	f	0	4	2522	{1994-03-23,19	{0.000966667,0.	{1992-01-10,19	-0.00361
l_shipinstruct	f	0	26	4	{"DELIVER IN PI	{0.2548,0.25283		0.241248
l_shipmode	f	0	11	7	{"SHIP ",	{0.1479,0.14653		0.137974
l_comment	f	0	27	0.153266	{" furiously",	{0.000233333,0.	{"about the ac	0.012145

Erläuterung zu pg_stats

http://www.postgresql.org/docs/9.2/static/view-pg-stats.html

- **null_frac**: Fraction of column entries that are null
- **n_distinct**: If greater than zero, the estimated number of distinct values in the column. If less than ...
- **most_common_vals**: A list of the most common values in the column. (Null if no values seem to be more common than any others.)
- **most_common_freqs**: A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows. (Null when most_common_vals is.)

Erläuterung zu pg_stats (2)

http://www.postgresql.org/docs/9.2/static/view-pg-stats.html

- **histogram_bounds**: A list of values that divide the column's values into groups of approximately equal population. The values in most_common_vals, if present, are omitted from this histogram calculation.
- **correlation**: Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk.

Beobachtungen

Korrelation zwischen physischer und logischer Speicherung

- l_orderkey hat Korrelationswert von 1 (!)
- Was bedeutet dies bzw. könnte bedeuten?

Frequent Values und Distinct Values

- l_shipinstruct ist "DELIVER IN PERSON" in ca. 25% aller Tupel.
- Es gibt ohnehin nur 4 unterschiedliche Werte für diese Spalte

Was bedeuten diese Beobachtungen bzgl. Notwendigkeit Indexe anzulegen bzw. evtl. vorhandene Indexe zu benutzen?

Histogramm

Für Spalte l_extendedprice in Tabelle lineitem (6001215 Tupel)

Die ersten 36 Werte aus histogram_bounds

930	9625,12	18639,95
1474,44	10529,84	19346,36
1943,94	11282,1	20020,8
2876,78	12023,41	20742,2
3540,48	12838,14	21406,91
4366,08	13624,3	22120,91
5158,89	14327,04	22799,66
5816,1	15079,54	23495,52
6600,48	15850,9	24180,42
7387,1	16550,1	24868,61
8064,56	17200,04	25609,44
8919,2	17909,52	26433

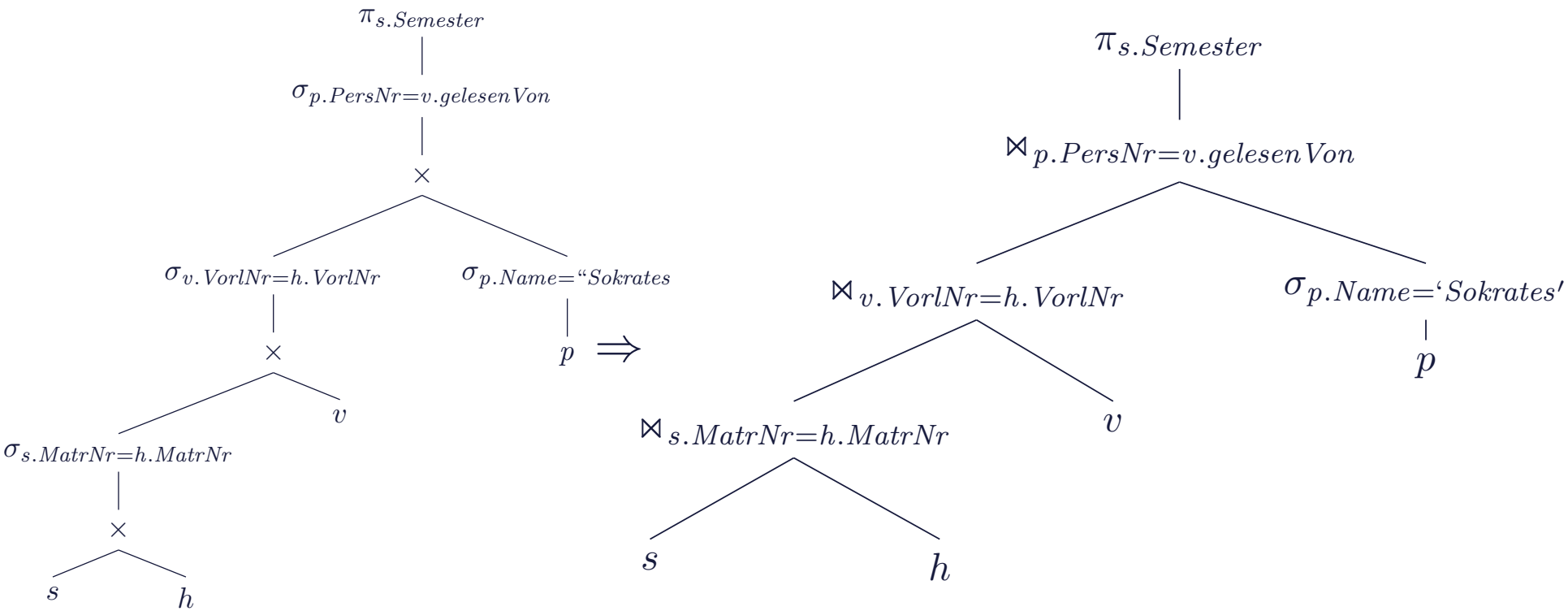
Equi-Depth-Histogramm mit 100 Zellen. Jede Histogrammzelle also beschreibt/umfasst rund 60 000 Werte.

Wie viele Tupel haben l_extendedprice < 1600?

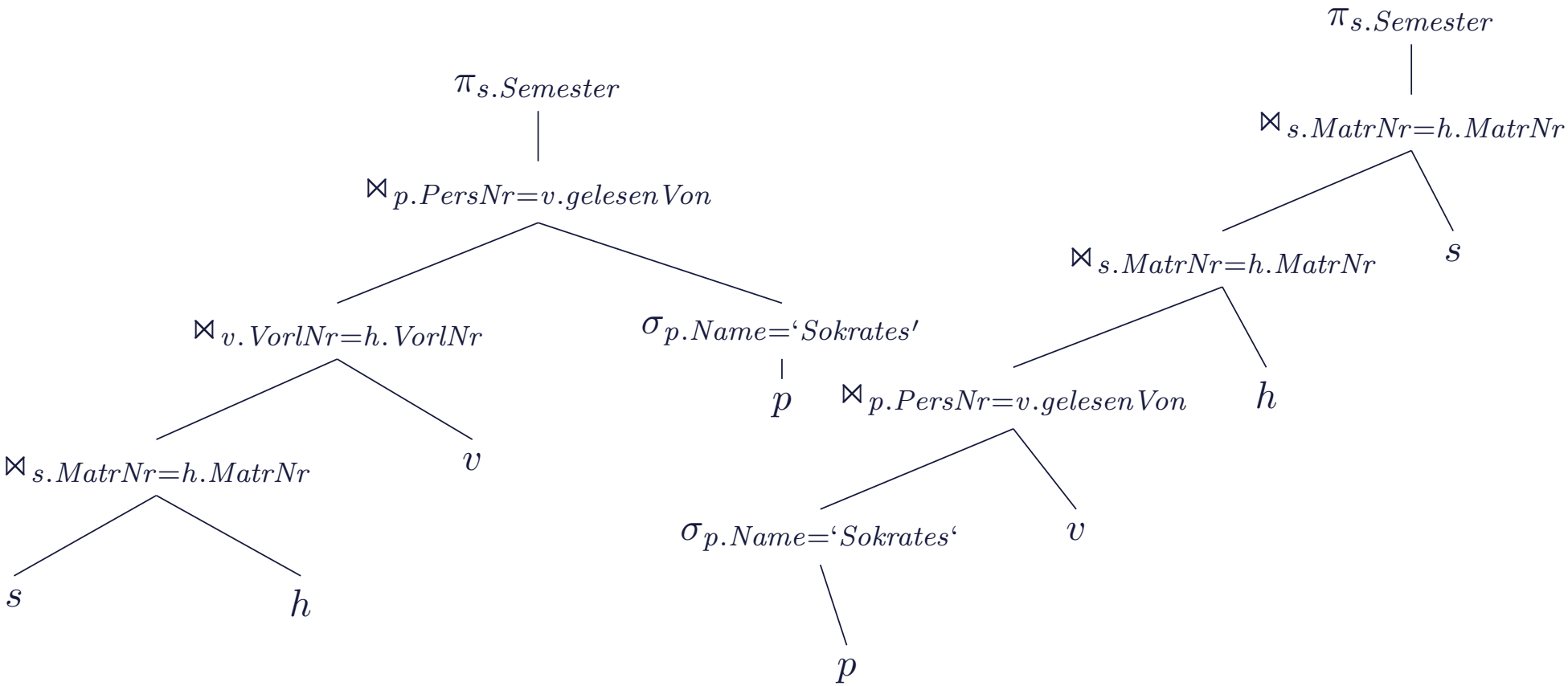
Anfrageoptimierung - Join Ordering

Literatur hierzu, Übersicht im Buch (Under Construction): "Building Query Compilers" von Guido Moerkotte (Uni Mannheim) (enthält Verweise auf Originalarbeiten). Großteil der Folien im Folgenden basierend auf Folien von Thomas Neumann (TUM) – basierend auf diesem Buch.

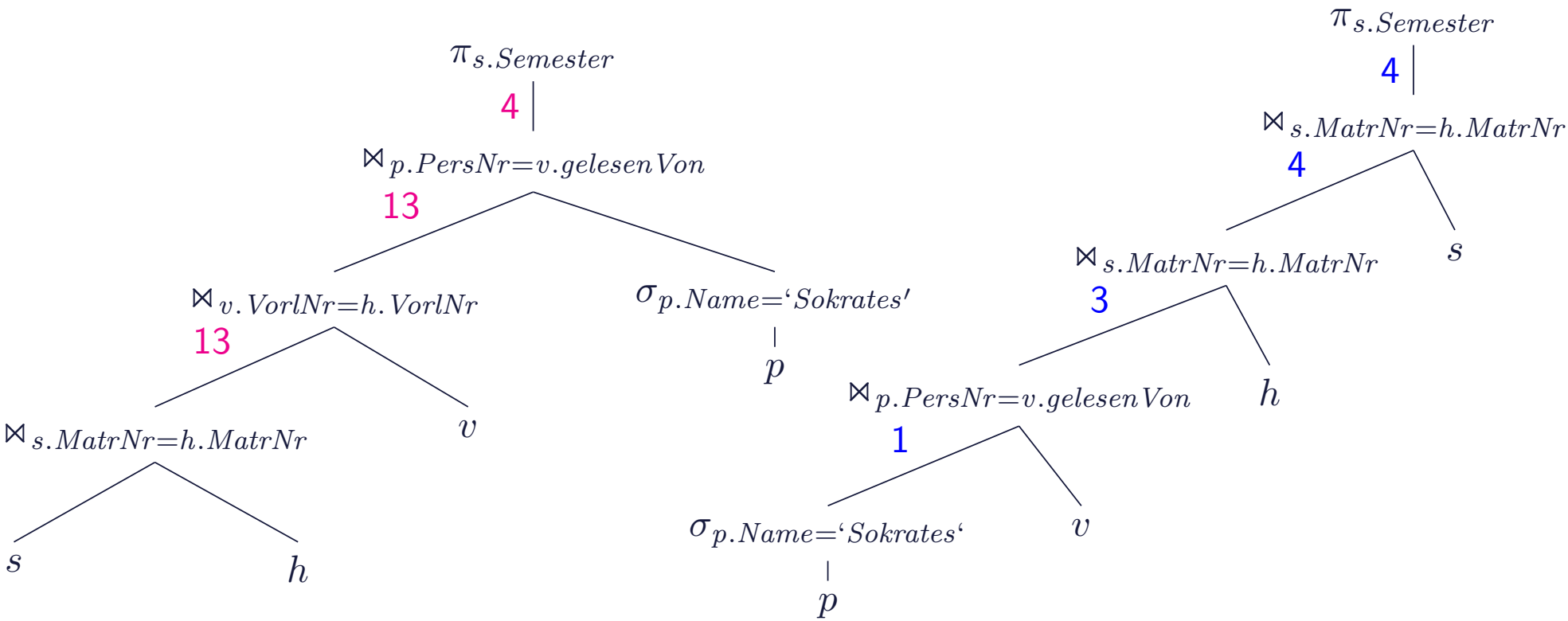
Wiederholung: Beispiel - Zusammenfassung von Selektionen und Kreuzprodukten zu Joins



Wiederholung: Beispiel - Optimierung der Joinreihenfolge



Wiederholung: Beispiel - Effekt: Reduzierung der Zwischenergebnisse



Diese Zwischenkosten müssen natürlich geschätzt werden. Der Optimierer kann dann den günstigsten Plan bzgl. dieser geschätzten Kosten auswählen .

Problemstellung und Setup

- Wir haben bereits gesehen, dass der **Join-Operator kommutativ und assoziativ** ist, d.h. $R \bowtie S = S \bowtie R$ und $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- Wir betrachten nun in welcher Reihenfolge die Joins eines Anfrageplans ausgeführt (geordnet) werden sollen bzw. können.

Die beteiligten Relationen sind $R_1, \ldots, R_n$ und wir betrachten Anfragen der Form:

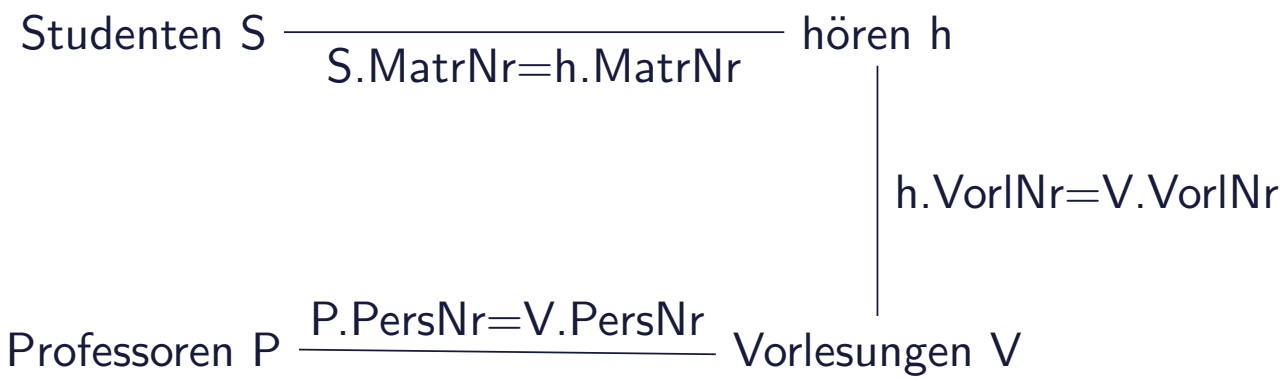
- Selektionen sind Konjunktionen über einfachen Prädikate der Form $x = y$

select ...
from $R_1, \ldots, R_n$
where $R_1.a = R_2.b$ **and** $R_1.a = R_3.c$ ...

Anfragegraph

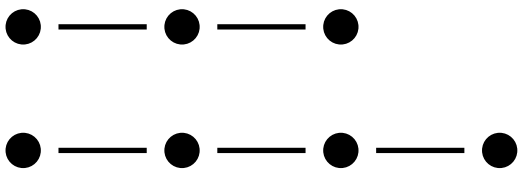
Anfragen dieses Typs können als Graph dargestellt werden:

- 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

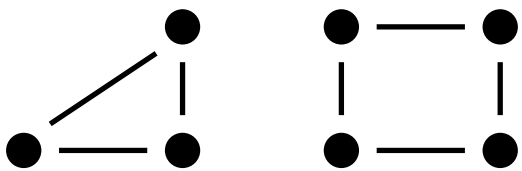


Für zwei Relationen, die nicht via einer Kante verbunden sind, kann nur ein Kreuzprodukt berechnet werden. Wir unterscheiden später ob Kreuzpodukte überhaupt zugelassen werden oder nicht.

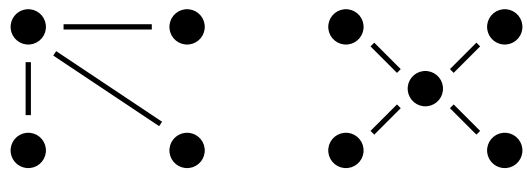
Formen von Anfragegraphen



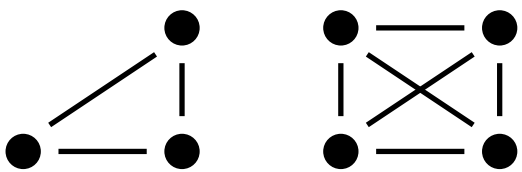
Ketten (chains)



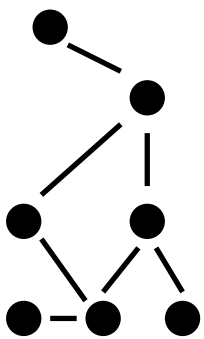
Ringe (cycles)



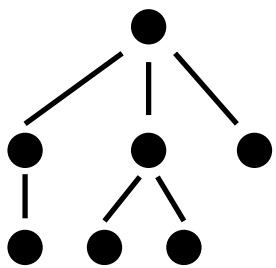
Sterne (stars)



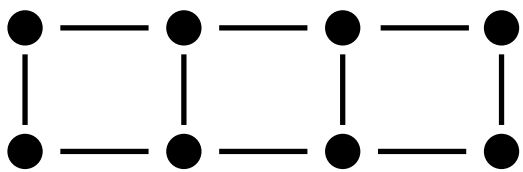
(cliques)



(cyclic)



Baum (tree)



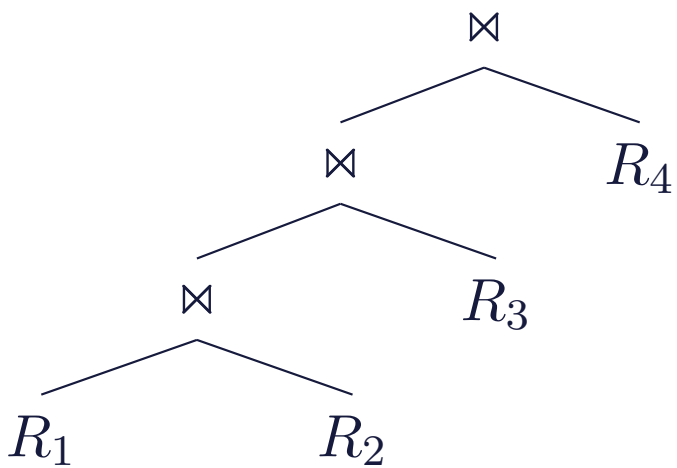
(grid)

Join-Baum

Ein **Join-Baum ist ein Binärbaum** mit

- Join-Operatoren als innere Knoten
- Relationen als Blätter

Beispiel:

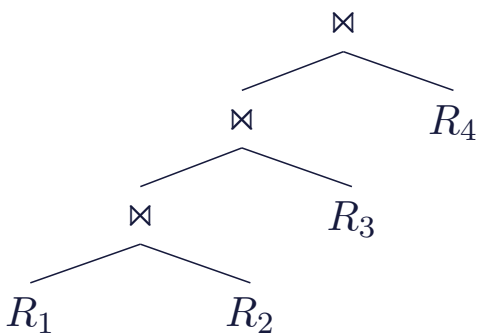


Beschreibt eine tatsächliche Realisierung (Ordnung!) des Joins über den beteiligten Relationen eines Anfragegraphen.

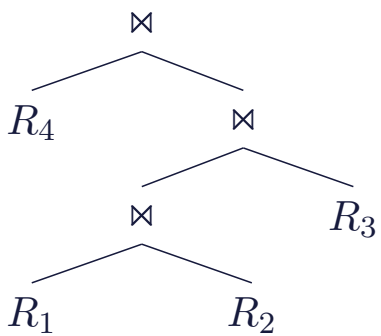
Gestalt von Join-Bäumen

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

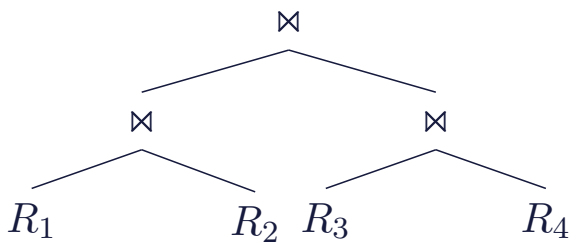
Die ersten drei Klassen werden auch zusammengefasst als **lineare Bäume**.



(links-tief)



(zigzag)



(buschig)

Selektivität von Joins

Eingabe:

- Kardinalitäten $|R_i|$
- Selektivitäten $f_{i,j}$: falls $p_{i,j}$ das Join-Präditkat zwischen R_i und R_j ist dann definieren wir

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

Berechne:

- Kardinalität des Ergebnisses:

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

Idee dahinter: Die Selektivität kann (idealerweise) recht einfach berechnet/geschätzt werden.

Kardinalität für Join-Bäume

Gegeben ein Join-Baum T, die **Kardinalität des Anfrageergebnisses** $|T|$ kann rekursiv berechnet werden durch

$$|T| = \begin{cases} |R_i| & \text{falls } T \text{ ein Blatt } R_i \text{ ist} \\ \left(\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}\right) * |T_1| * |T_2| & \text{falls } T = T_1 \bowtie T_2 \end{cases}$$

- Erlaubt einen einfache Berechnung der Kardinalität eines Joins
- Benötigt nur die Kardinalitäten der zugrunde liegenden Relationen und Selektivitäten
- **Setzt Unabhängigkeit der Prädikate voraus**

Kostenfunktion

Gegeben ein Join-Baum T, dann ist die **Kostenfunktion** C_{out} definiert als

$$C_{out}(T) = \begin{cases} 0 & \text{falls } T \text{ ist ein Blatt } R_i \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{falls } T = T_1 \bowtie T_2 \end{cases}$$

- **Addiert die Größen der (Zwischen)ergebnisse auf**
- **Idee dahinter: Größere Zwischenergebnisse erfordern mehr Arbeit**
- Die Kosten der einzelnen Relationen werden hier weggelassen (da sie sowieso gelesen werden müssen)

Kostenfunktion für Joins

Für einzelne Joins

$$\begin{aligned}
\textbf{Nested-Loops Join:} C_{nlj}(e_1 \bowtie e_2) &= |e_1||e_2| \\
\textbf{Hash-Join:} C_{hj}(e_1 \bowtie e_2) &= 1.2|e_1| \\
\textbf{Sort-Merge-Join:} C_{smj}(e_1 \bowtie e_2) &= |e_1|\log(|e_1|) + |e_2|\log(|e_2|)
\end{aligned}$$

Für Sequenzen von Join-Operatoren $s = s_1 \bowtie \ldots \bowtie s_n$**:**

$$\begin{aligned}
C_{nlj}(s) &= \sum_{i=2}^{n} |s_1 \bowtie \ldots \bowtie s_{i-1}||s_i| \\
C_{hj}(s) &= \sum_{i=2}^{n} 1.2|s_1 \bowtie \ldots \bowtie s_{i-1}| \\
C_{smj}(s) &= \sum_{i=2}^{n} |s_1 \bowtie \ldots \bowtie s_{i-1}|\log(|s_1 \bowtie \ldots \bowtie s_{i-1}|) + \sum_{i=2}^{n} |s_i|\log(|s_i|)
\end{aligned}$$

Anmerkungen zu diesen Kostenfunktionen

- Kostenfunktionen sehr einfach
- Join-Implementierungen sind sehr einfach modelliert (z.B. Faktor 1.2, kein n-way sort/merge)
- Designed für links-tiefe Bäume
- C_{hj} und C_{smj} funktionieren nicht für Kreuzprodukte (Korrektur dafür: Betrachte Kardinalität der Ausgabe, d.h C_{nl})
- Kostenfunktionen nehmen an, dass die gleiche Join-Implementierung (Algorithmus) für den gesamten Join-Baum benutzt wird

Beispiel Statistiken und Hinweis zu Kreuzprodukten

Als im Folgenden verwendetes Beispiel nehmen wir an:

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

- Daraus folgt der Anfragegraph $R_1 - R_2 - R_3$

Hinweis zu Kreuzprodukten

- Relationen, die nicht durch Kanten verbunden sind können nur mit Hilfe eines Kreuzproduktes (d.h. $f_{i,j} = 1$) berechnet werden.
- Das Zulassen von Kreuzprodukten vergrößert den Suchraum. Ob diese zugelassen werden oder nicht ist später essenziell bei Algorithmen und Betrachtung der Größe des Suchraums.

Beispiel für Kostenberechnung

	C_{out}	C_{nl}	C_{hj}	C_{smj}
$R_1 \bowtie R_2$	100	1000	12	697.61
$R_2 \bowtie R_3$	20000	100000	120	10630.26
$R_1 \times R_3$	10000	10000	10000	10000.00
$(R_1 \bowtie R_2) \bowtie R_3$	20100	101000	132	11327.86
$(R_2 \bowtie R_3) \bowtie R_1$	40000	300000	24120	32595.00
$(R_1 \times R_3) \bowtie R_2$	30000	1010000	22000	143542.00

Beobachtungen:

- Kosten variieren sehr stark
- Join-Bäume mit Kreuzprodukten sind sehr teuer
- Join-Ordnung essenziell

Weitere Beispiele

Für $|R_1| = 1000, |R_2| = 2, |R_3| = 2, f_{1,2} = 0.1, f_{1,3} = 0.1$
haben wir die Kosten

	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 der Baum mit Kreuzprodukt am besten**
- Aber nur weil die Kardinalitäten von $|R_2|$ und $|R_3|$ sehr klein
- Kann daher durchaus (im Allgemeinen) eine attraktive Lösung sein, eben wenn Kardinalitäten klein

Weitere Beispiele

Für $|R_1| = 10, |R_2| = 20, |R_3| = 20, |R_4| = 10, f_{1,2} = 0.01, f_{2,3} = 0.5, f_{3,4} = 0.01$ haben wir die Kosten

	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

- **Der buschige Baum ist hier besser als alle anderen Möglichkeiten**