Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

"Essentially, all models are wrong, but some are useful."
(George E. P. Box)

(c) xkcd.com

Probabilistic (Approximate) Counting

$V(R, A)$ ist die Anzahl der verschiedenen Attributsausprägungen für Attribut A.
Wie kann diese Größe berechnet werden?

- Klar, via Duplikat-Eliminierung durch Sortieren oder durch Hashing
- Oder durch probabilistische Methoden (Schätzer)

Flajolet Martin (FM) Sketch (aka. Hash Sketch)

Vorgeschlagen von Flajolet und Martin in 1985[1]

- Erzeuge einen leeren Bitvektor B der Länge $m = log(N)$
- Scan über Eingabedaten: Dabei wird für jedes Objekt eine Position im Bitvektor berechnet und auf "1" gesetzt:
  - Hashing eines Objekts i in eine m-bit Zahl $h(i)$
  - Berechne Position k des am wenigsten signifikanten "1" Bits von h(i)
  - Setze bit $B[k]$ auf "1"

Beispiel

Eingabe: 17, 5, 19, 211, 17, 5, 31
Annahme h(17)=010100, dann ist das am wenigsten signif. 1 Bit = 3
Annahme h(5)=000101, dann ist das am wenigsten signif. 1 Bit = 1

[1] Philippe Flajolet, G. Nigel Martin: Probabilistic Counting Algorithms for Data Base Applications. J. Comput. Syst. Sci. 31(2): 182-209 (1985)

Schätzer

- Am Ende sieht B dann z.B. so aus: $B = 111010$
- Betrachte die Position t des am weitesten links stehenden "0" Bits, hier im Beispiel $t = 4$.
- Dann ergibt sich die Schäzung für die tatsächliche Anzahl n als

$$\hat{n} = 2^t / 0.7735$$

in unserem Beispiel mit $t = 4$: $\hat{n} = 2^4 / 0.7735 \approx 20.685$

Verbesserung der Schätzung

Verwendung mehrerer Bitvektoren B (entsprechend mit unterschiedlichen Hashfunktionen h) und Berechnung eines Durchnittlichen Werts von t.

Idee/Intuition

- $B[0]$ wird ungefähr $n/2$ mal gesetzt
- $B[1]$ wird ungefähr $n/4$ mal gesetzt

...

Also:

- $B[i] = 0$ falls $i >> log_2(n)$
- $B[i] = 1$ falls $i << log_2(n)$
- "Mischung" aus 1s und 0s um $i \approx log_2(n)$ herum

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 Datasets (`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?

Schätzung mit Histogrammen

- Obwohl man für diskrete Daten auch Punktanfragen bearbeiten kann, ist dies im kontinuierlichen Fall nicht möglich.
- Es liegen unendlich viele Werte in jeder Histogrammzelle (mit Häufigkeit 0)
- D.h. es kann im kontinuierlichen Fall "nur" nach Häufigkeiten für Intervalle gefragt werden.

Fehlermaße

- Ist für einen exakten Wert x eine Schätzung (Näherungswert) $\hat{x}$ gegeben,
  - so heisst $|\hat{x} - x|$ absoluter Fehler und
  - $\frac{|\hat{x} - x|}{x}$ im Fall $x \neq 0$ relativer Fehler
- Für mehrere solcher Beobachtungen: Sum Squared Error (SSE), Mean Absolute Error (MAE), Mean Squared Error (MSE)

Selektivitätsschätzung: Zusammenfassung

- Kosten der Operatoren hängen (neben Implementierng) von Größe der Eingaben ab
- Für einen Anfrageplan kann so die Anzahl der erwarteten Ergebnisse (Tupel) sowie die erwartete Größe der anfallenden Zwischenergebnisse berechnet werden.
- Je nach verfügbarkeit an Statistiken können Schätzungen sehr grob oder recht genau sein, vlg. Schätzung von $|\sigma_{A<c}(R)| = 1/3 * |R|$ mit Wert von Histogrammen.

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.

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:

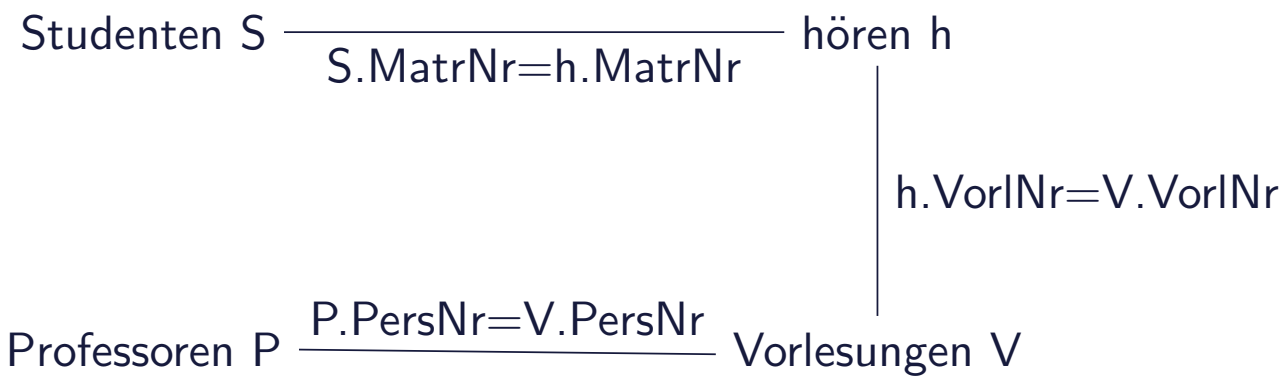
- Selektionen sind Konjunktionen über
- einfachen Prädikate der Form $x = y$

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

Anfragegraph

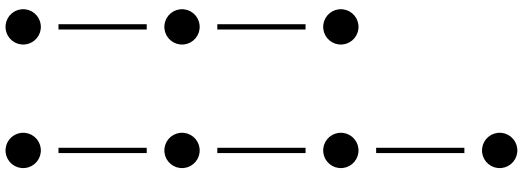
Anfragen dieses Typs können als Graph dargestellt werden:

- 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

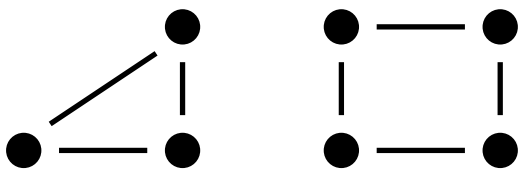


Für zwei Relationen, die nicht via einer Kante verbunden sind, kann nur ein Kreuzprodukt berechnet werden. Wir unterscheiden später ob Kreuzpodukte überhaupt zugelassen werden oder nicht.

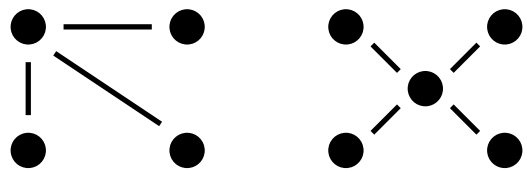
Formen von Anfragegraphen



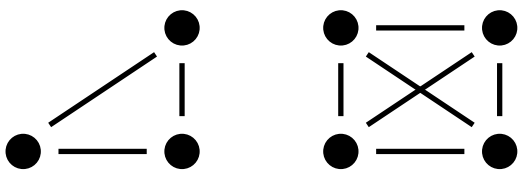
Ketten (chains)



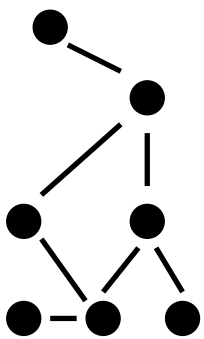
Ringe (cycles)



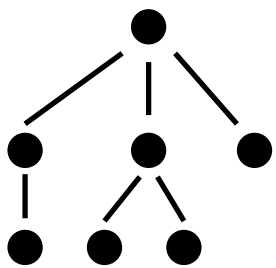
Sterne (stars)



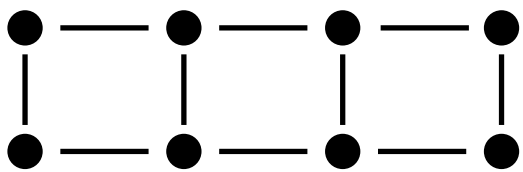
(cliques)



(cyclic)



Baum (tree)



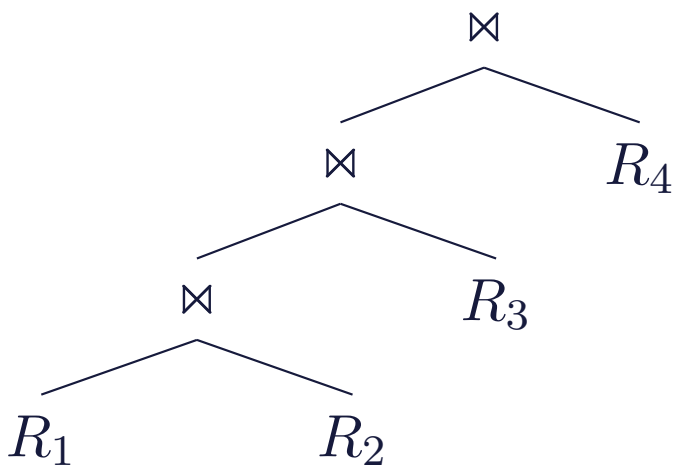
(grid)

Join-Baum

Ein Join-Baum ist ein Binärbaum mit

- Join Operatoren als innere Knoten
- Relationen als Blätter

Beispiel:

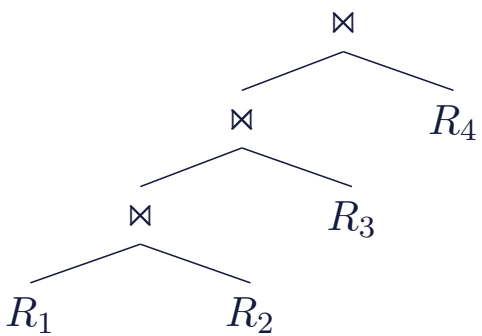


Beschreibt eine tatsächliche Realisierung (Ordnung!) des Joins über den beteiligten Relationen eines Anfragegraphen.

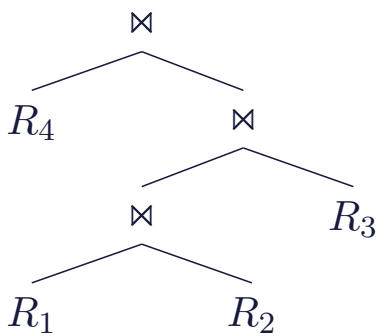
Gestalt von Join-Bäumen

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

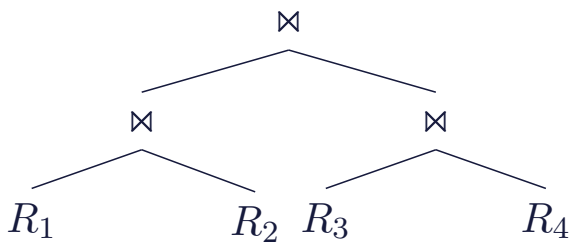
Die ersten drei Klassen werden auch zusammengefasst als lineare Bäume.



(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

Beispiel Statistiken

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 $R_1 - R_2 - R_3$
- Für alle anderen Selektivitäten nehmen wir $f_{i,j} = 1$ an

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 af
- 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

$$\begin{aligned} \text{Nested-Loops Join: } C_{nlj}(e_1 \bowtie e_2) &= |e_1||e2| \\ \text{Hash-Join: } C_{hj}(e_1 \bowtie e_2) &= 1.2|e_1| \\ \text{Sort-Merge-Join } C_{smj}(e_1 \bowtie e_2) &= |e_1|\log(|e_1|) + |e_2|\log(|e_2|) \end{aligned}$$

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 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 besser als alle anderen Möglichkeiten

Klassifikation der Join-Ordering-Probleme

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

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

Wiederholung(?): Catalan-Zahl

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

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

Dies kann in geschlossener Form geschrieben werden als

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

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

Anzahl Join-Bäume mit Kreuzprodukte

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

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