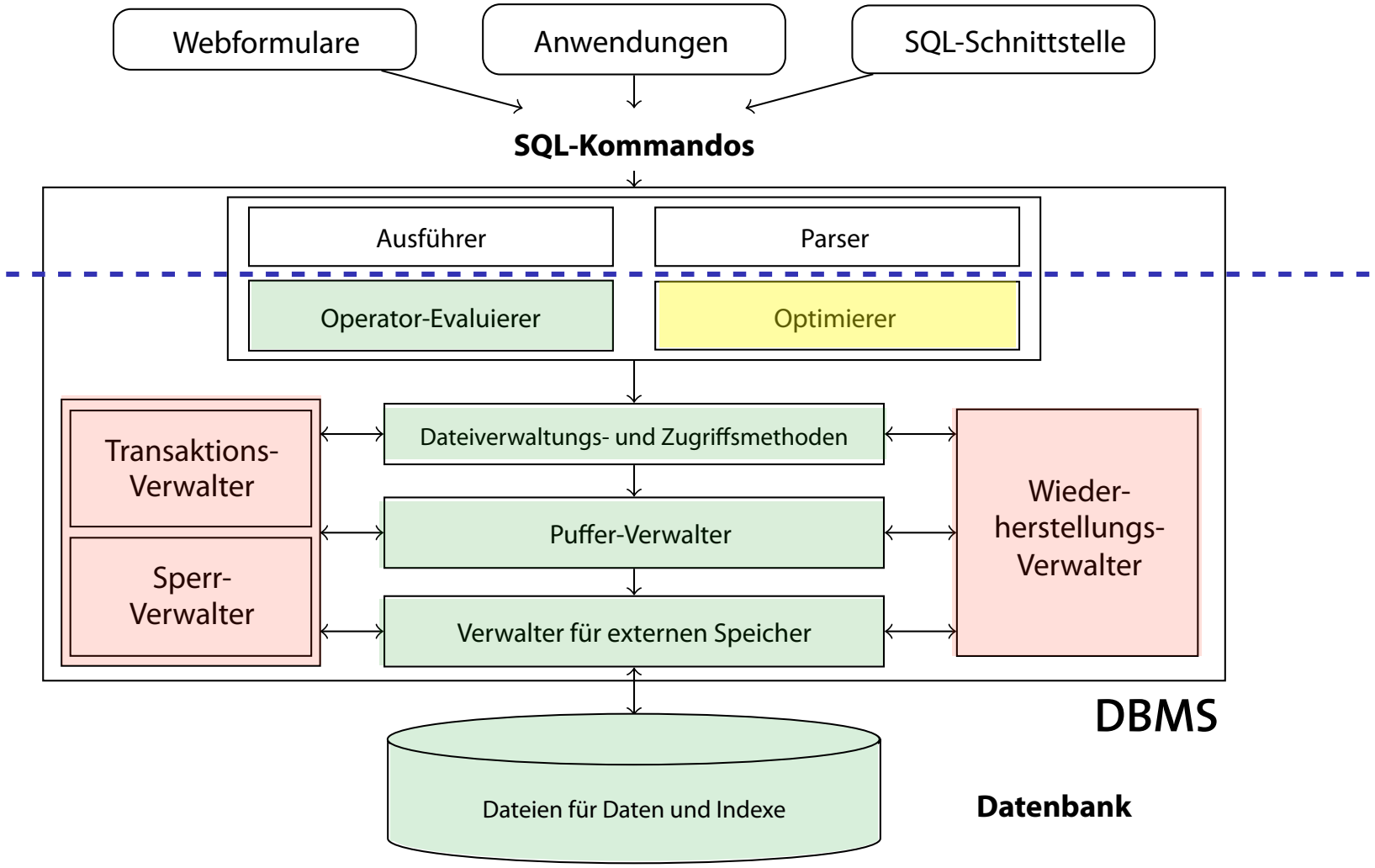
Datenbanken

Prof. Dr. Ralf Möller
Universität zu Lübeck
Institut für Informationssysteme

Marc Stelzner (Übungen)
Torben Matthias Kempfert (Tutor)
Maurice-Raphael Sambale (Tutor)

Anfrageoptimierung



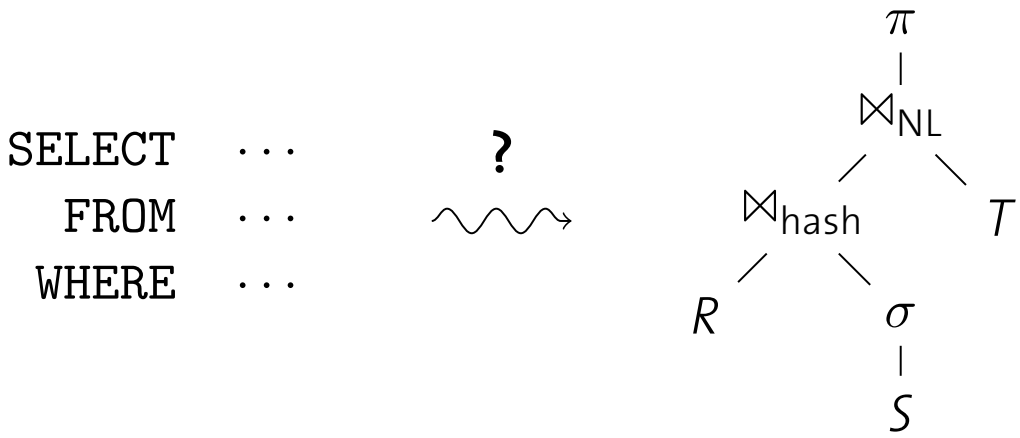
Danksagung

- Diese Vorlesung ist inspiriert von den Präsentationen zu dem Kurs:

  „Architecture and Implementation of Database Systems“ von Jens Teubner an der ETH Zürich

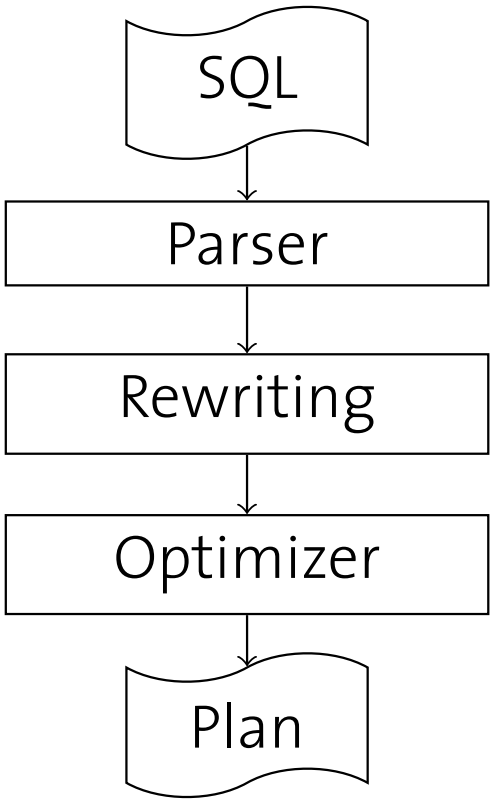
- Graphiken und Code-Bestandteile wurden mit Zustimmung des Autors (und ggf. kleinen Änderungen) aus diesem Kurs übernommen

Anfrageoptimierung



- Es gibt mehr als eine Art, eine Anfrage zu beantworten
  - Welche Implementation eines Verbundoperators?
  - Welche Parameter für Blockgrößen, Pufferallokation, ...
  - Automatisch einen Index aufsetzen?
- Die Aufgabe, den besten Ausführungsplan zu finden, ist der heilige Gral der Datenbankimplementierung

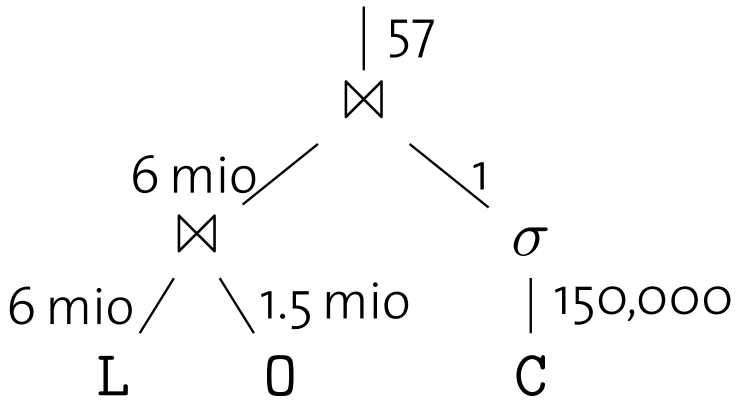
Generierung eines Ausführungsplans

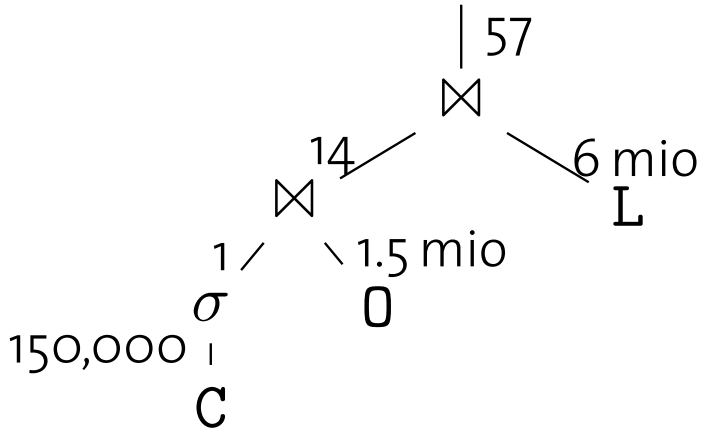


- Parser: syntaktische und semantische Analyse
- Umschreibung: Optimierung unabhängig von dem aktuellen Datenbankinhalt (Tabellengrößen, Verfügbarkeit eines Index, ...)
- Optimierer: Umformungen auf Basis eines Kostenmodells und Information zum DB-Zustand
- Resultierender Plan wird dann ausgeführt durch Ausführungsmaschine

Auswirkungen auf die Performanz

```
SELECT L.L_PARTKEY, L.L_QUANTITY, L.L_EXTENDEDPRICE
FROM LINEITEM L, ORDERS O, CUSTOMER C
WHERE L.L ORDERKEY = O.O ORDERKEY
      AND O.O_CUSTKEY = C.C_CUSTKEY
      AND C.C_NAME = 'IBM Corp.'
```



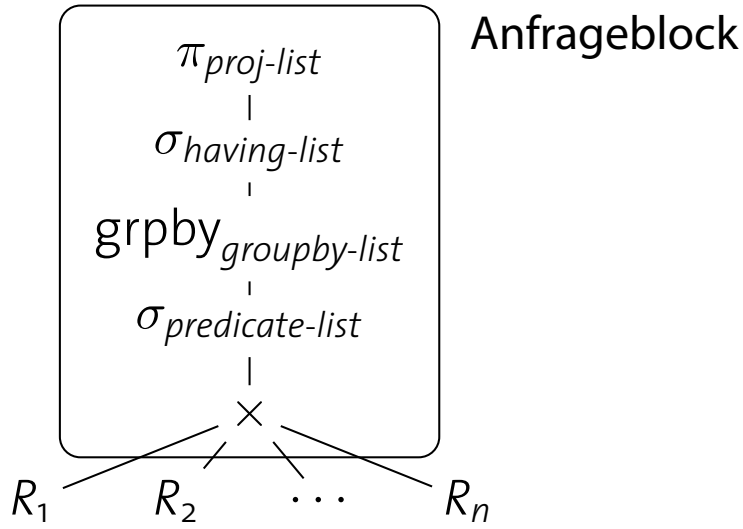


- Bezogen auf die Ausführungszeit können die Unterschiede „Sekunden vs. Tage“ bedeuten

Der SQL-Parser

- Neben der syntaktischen und semantischen Richtigkeit einer Eingabeanfrage erzeugt der Parser eine interne Repräsentation (Anfrageblock)

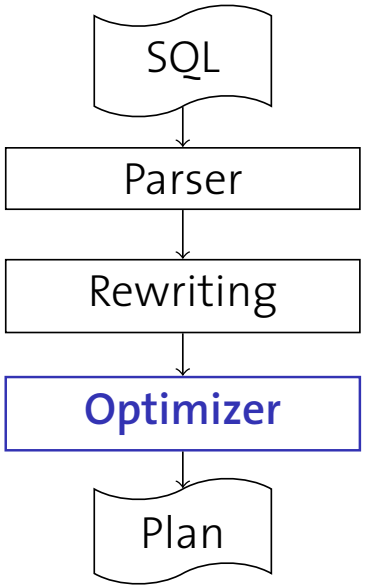
```
SELECT  proj-list
  FROM  R1, R2, ..., Rn
 WHERE  predicate-list
 GROUP  BY  groupby-list
HAVING  having-list
```



- Jedes R_i kann eine Basisrelation oder ein anderer Anfrageblock sein

Such nach dem „besten“ Ausführungsplan

- Ausgabe des Parsers wird durch Umschreibe-maschine verarbeitet, die einen äquivalenten Anfrageblock generiert
- Aufgabe des Optimierers: Finde optimalen Ausführungsplan
  - Zähle alle äquivalenten Pläne auf
  - Bestimme die Qualität (Kosten) eines jeden Plans
  - Wähle den/die besten als finalen Plan
- Bevor wir fortfahren müssen wir klären
  - Was ist überhaupt ein „guter“ Plan?

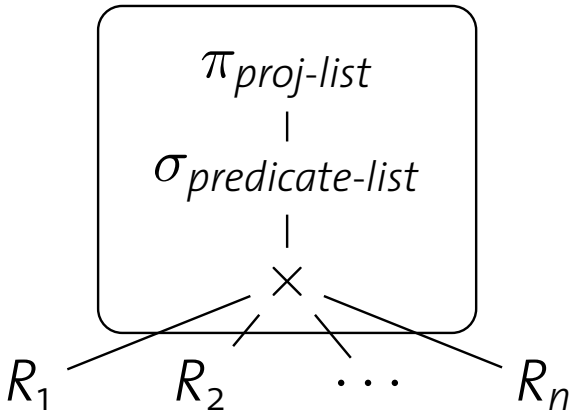


Kostenmetriken

- Kostenfaktoren für einen Plan
  - Anzahl der benötigten Platten-I/Os
  - Kosten für vorgesehene CPU-Operationen
  - Reaktionszeit bis zum ersten Tupel für den Benutzer
  - Gesamtausführungszeit
- Abschätzung der Kosten, so dass der beste („billigste“) Plan gefunden wird, ohne ihn tatsächlich auszuführen
  - Kritischer Faktor: Größe der Zwischenrepräsentationen
  - Ziel: Größe von Anfrageergebnissen möglichst genau abschätzen

Abschätzung der Ergebnisgröße

Betrachte Anfrageblock für Select-From-Where-Anfrage Q



Abschätzung der Ergebnisgröße von Q durch

- die Größe der Eingabetabellen $|R_1|, |R_2|, \ldots, |R_n|$ und
- die Selektivität *sel(predicate-list)*

$$|Q| = |R_1| \cdot |R_2| \cdot \ldots \cdot |R_n| \cdot sel(predicate\text{-}list)$$

Tabellenkardinalitäten

- Die Größe einer Tabelle ist über den Systemkatalog verfügbar (hier IBM DB2)

```
db2 => SELECT TABNAME, CARD, NPAGES
db2 (cont.) => FROM SYSCAT.TABLES
db2 (cont.) => WHERE TABSCHEMA = 'TPCH';
TABNAME         CARD                 NPAGES
--------------- -------------------- --------------------
ORDERS          1500000              44331
CUSTOMER        150000               6747
NATION          25                   2
REGION          5                    1
PART            200000               7578
SUPPLIER        10000                406
PARTSUPP        800000               31679
LINEITEM        6001215              207888
8 record(s) selected.
```

Abschätzung der Selektivität

... durch Induktion über die Struktur des Anfrageblocks

column = value

$$sel(\cdot) = \begin{cases} 1/|I| & \text{falls es einen Index } I \text{ auf Attribut } column \text{ gibt} \\ 1/10 & \text{sonst} \end{cases}$$

column$_1$ = *column*$_2$

$$sel(\cdot) = \begin{cases} \frac{1}{\max\{|I_1|,|I_2|\}} & \text{falls es Indexe auf beiden Spalten gibt} \\ \frac{1}{|I_k|} & \text{falls es einen Index nur auf } column \text{ gibt} \\ 1/10 & \text{sonst} \end{cases}$$

p_1 `AND` p_2

$$sel(\cdot) = sel(p_1) \cdot sel(p_2)$$

p_1 `OR` p_2

$$sel(\cdot) = sel(p_1) + sel(p_2) - sel(p_1) \cdot sel(p_2)$$

Verbesserung der Selektivitätsabschätzung

- Hinter den obigen Regeln stecken Annahmen
  - Gleichverteilung der Datenwerte in einer Spalte
  - Unabhängigkeit zwischen einzelnen Prädikaten

- Annahmen nicht immer gerechtfertigt

- Sammlung von Datenstatistiken (offline)
  - Speicherung im Systemkatalog
    - IBM DB2: RUNSTATS ON TABLE
  - Meistverwendet: Histogramme

Histogramme

```
SELECT SEQNO, COLVALUE, VALCOUNT
FROM SYSCAT.COLDIST
WHERE TABNAME = 'LINEITEM'
AND COLNAME = 'L_EXTENDEDPRICE'
AND TYPE = 'Q';
SEQNO COLVALUE           VALCOUNT
----- ----------------- --------
1     +0000000000996.01 3001
2     +0000000004513.26 315064
3     +0000000007367.60 633128
4     +0000000011861.82 948192
5     +0000000015921.28 1263256
6     +0000000019922.76 1578320
7     +0000000024103.20 1896384
8     +0000000027733.58 2211448
9     +0000000031961.80 2526512
10    +0000000035584.72 2841576
11    +0000000039772.92 3159640
12    +0000000043395.75 3474704
13    +0000000047013.98 3789768
```

SYSCAT.COLDIST enthält Informationen wie

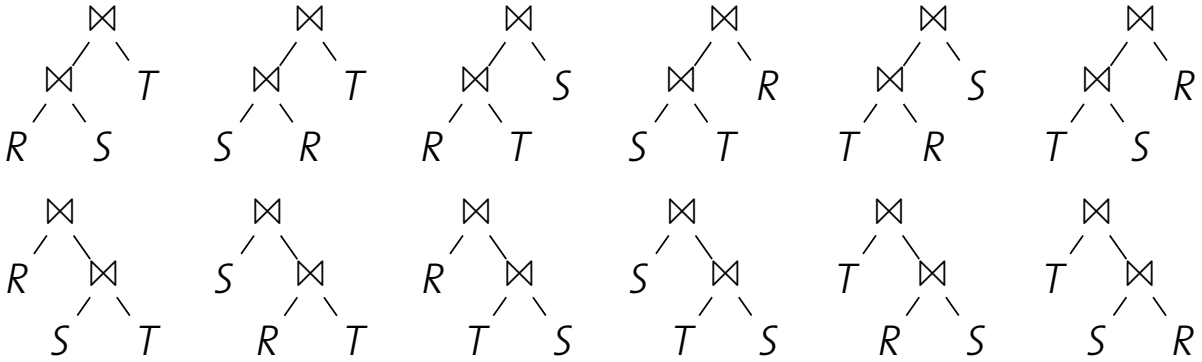
- n-häufigste Werte (und der Anzahl)
- Anzahl der verschiedenen Werte pro Histogramm-rasterplatz

Tatsächlich können Histogramme auch absichtlich gesetzt werden, um den Optimierer zu beeinflussen

Verbund-Optimierung

- Anfrage übersetzt in Graph von Anfrageblöcken
  - Sieht wie ein mehrfaches kartesischen Produkt aus mit einer zusätzlichen Selektion oben drauf
- Abschätzung der Kosten eines Ausführungsplans
  - Abschätzung der Ergebnisgrößen mit den Kosten von einzelnen Verbundalgorithmen für jeden Block

Auflistung der möglichen Ausführungspläne, d.h.
alle 3-Wege-Verbundkombinationen für jeden Block



Suchraum

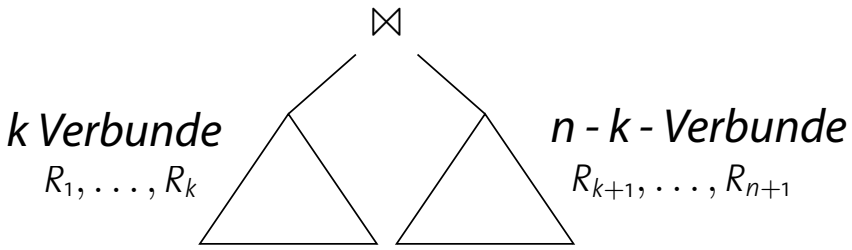
- Der sich ergebende Suchraum ist enorm groß:
  Schon bei 4 Relationen ergeben sich 120 Möglichkeiten

number of relations *n*	join trees
2	2
3	12
4	120
5	1,680
6	30,240
7	665,280
8	17,297,280
10	17,643,225,600

- Noch nicht berücksichtigt: Anzahl v der verschiedenen Verbundalgorithmen

Wie viele Kombinationen gibt es?

- Verbund über n+1 Relationen $R_1, ..., R_{n+1}$ benötigt n binäre Verbunde
- Der Wurzeloperator verbindet k und n-k-1 Verbundoperatoren ($0 \leq k \leq n-1$)



- Sei C_i die Anzahl der Möglichkeiten, um einen Binärbaum von i inneren Knoten (Verbunde) zu konstruieren:

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1}$$

Catalan-Zahlen

Rekurrenzgleichung erfüllt durch Catalan-Zahlen

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{(n+1)!n!}$$

zur Beschreibung der Anzahl der geordneten Binärbäume mit n+1 Blättern

Für jeden dieser Bäume können die Eingaberelationen $R_1, ..., R_{n+1}$ permutiert werden, und es ergeben sich

$$\frac{(2n)!}{(n+1)!n!} \cdot (n+1)! = \frac{(2n)!}{n!}$$

Möglichkeiten für einen (n+1)-Wege-Verbund

Suchraum

- Der sich ergebende Suchraum ist enorm groß:

Anzahl der Relationen n	C_{n-1}	#Verbundbäume
2	1	2
3	5	12
4	14	120
5	42	1,680
6	132	30,240
7	429	665,280
8	1,430	17,297,280
10	16,796	17,643,225,600

- Noch nicht berücksichtigt: Anzahl v der verschiedenen Verbundalgorithmen (ergibt neuen Faktor $v^{(n-1)}$)

Dynamische Programmierung

- Traditioneller Ansatz, einen großen Suchraum zu beherrschen (siehe AuD-Vorlesung)
  - Bestimme günstigsten Plan für n-Wege-Verbund in n Durchgängen
  - In jedem Durchgang k bestimme den besten Plan für alle k-Relationen-Unterabfragen
  - Konstruiere die Pläne aus Durchgang k aus den besten i-Relationen- und $(k-i)$-Relationen-Unterplänen aus früheren Durchgängen ($1 \leq i < k$)
- Annahme:
  - Zur Bestimmung des optimalen globalen Plans ist es ausreichend, die optimalen Pläne der Unterabfragen zu betrachten

Beispiel: 4-Wege-Verbund

Pass 1 (best 1-relation plans)

Find the best **access path** to each of the R_i individually (considers index scans, full table scans).

Pass 2 (best 2-relation plans)

For each **pair** of tables R_i and R_j, determine the best order to join R_i and R_j ($R_i \bowtie R_j$ or $R_j \bowtie R_i$?):

$$optPlan(\{R_i, R_j\}) \leftarrow \text{ best of } R_i \bowtie R_j \text{ and } R_j \bowtie R_i \ .$$

$\longrightarrow$ 12 plans to consider.

Pass 3 (best 3-relation plans)

For each **triple** of tables R_i, R_j, and R_k, determine the best three-table join plan, using sub-plans obtained so far:

$$optPlan(\{R_i, R_j, R_k\}) \leftarrow \text{ best of } R_i \bowtie optPlan(\{R_j, R_k\}),$$
$$optPlan(\{R_j, R_k\}) \bowtie R_i, \ \ R_j \bowtie optPlan(\{R_i, R_k\}), \ldots \ .$$

$\longrightarrow$ 24 plans to consider.

Beispiel (Fortsetzung)

Pass 4 (best 4-relation plan)

For each set of **four** tables R_i, R_j, R_k, and R_l, determine the best four-table join plan, using sub-plans obtained so far:

$$optPlan(\{R_i, R_j, R_k, R_l\}) \leftarrow \text{ best of } R_i \bowtie optPlan(\{R_j, R_k, R_l\}),$$
$$optPlan(\{R_j, R_k, R_l\}) \bowtie R_i, \quad R_j \bowtie optPlan(\{R_i, R_k, R_l\}), \ldots,$$
$$optPlan(\{R_i, R_j\}) \bowtie optPlan(\{R_k, R_l\}), \ldots \quad .$$

$\longrightarrow$ 14 plans to consider.

- Insgesamt: 50 (Unter-)Pläne betrachtet (anstelle von 120 möglichen 4-Wege-Verbundplänen, siehe Tabelle)
- Alle Entscheidungen basieren auf vorher bestimmten Unterplänen (keine Neuevaluierung von Plänen)

Algorithmus zur Planbestimmung

```
1  Function: find_join_tree_dp (q(R_1, ..., R_n))
2  for i = 1 to n do
3      optPlan({R_i}) ← access_plans (R_i) ;
4      prune_plans (optPlan({R_i})) ;
5  for i = 2 to n do
6      foreach S ⊆ {R_1, ..., R_n} such that |S| = i do
7          optPlan(S) ← ∅ ;
8          foreach O ⊂ S do
9              optPlan(S) ← optPlan(S) ∪
10                 possible_joins (optPlan(O), optPlan(S \ O));
11         prune_plans (optPlan(S)) ;
12 return optPlan({R_1, ..., R_n}) ;
```

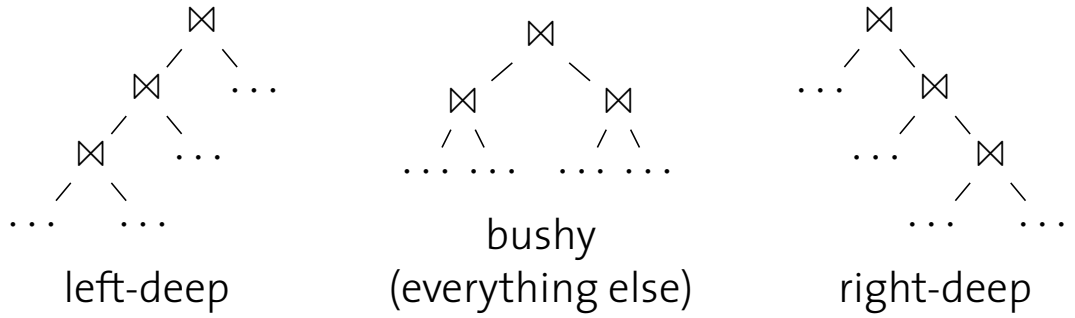
- possible_joins(R, S) enumeriert alle möglichen Pläne zwischen R und S (geschachtelte-Schleifen-Verbund, Misch-Verbund, ...)
- prune_plan(*set*) eliminiert alle außer den besten Plänen aus *set*

Diskussion: Dynamische Programmierung

- finf_join_tree_dp() eliminiert Plankandidaten früh
  - Im Beispiel wird in Durchgang 2 der Suchraum um den Faktor 2 und in Durchgang 3 um den Faktor 6 reduziert
- Heuristiken helfen, den Suchraum weiter zu reduzieren
  - Vermeide kartesische Produkte
  - Erzeuge links-tiefe Pläne (s. nächste Folie)
- Abwägung der Optimalität der Plan vs. Suchdauer nach dem optimalen Plan
  - DB2 UDB: SET CURRENT QUERY OPTIMIZATION = n

Links-tiefe vs. buschige Verbundplan-Bäume

Der Plangenerierungsalgorithmus exploriert alle möglichen Formen von Verbundplänen



Implementierte Systeme generieren meist links-tiefe Pläne[1]

- Die innere Relation ist immer eine Basisrelation
- Bevorzugung der Verwendung von Index-Verbunden
- Gut geeignet für die Verwendung von Pipeline-Verarbeitungsprinzipien

[1] So z.B. der Prototyp von System R (IBM)

Verbund über mehrere Relationen

- Dynamische Programmierung erzeugt immer noch exponentiellen Aufwand
  - Zeit: $O(3^n)$
  - Platz: $O(2^n)$
- Das kann zu teuer sein...
  - für Verbunde mit mehreren Relationen (10-20 und mehr)
  - für einfache Anfragen über gut-indizierte Daten, für die ein sehr guter Plan einfach zu finden wäre
- Neue Plangenerierungsstrategie: Greedy-Join-Enumeration

Greedy-Join-Enumeration

1 **Function:** `find_join_tree_greedy` $(q(R_1, \ldots, R_n))$
2 $worklist \leftarrow \varnothing$;
3 **for** $i = 1$ **to** n **do**
4 $\quad worklist \leftarrow worklist \cup$ `best_access_plan` (R_i) ;
5 **for** $i = n$ **downto** 2 **do**
$\quad$ // $worklist = \{P_1, \ldots, P_i\}$
6 $\quad$ find $P_j, P_k \in worklist$ and $\bowtie_{\ldots}$ such that $cost(P_j \bowtie_{\ldots} P_k)$ is minimal ;
7 $\quad worklist \leftarrow worklist \setminus \{P_j, P_k\} \cup \{(P_j \bowtie_{\ldots} P_k)\}$;
// $worklist = \{P_1\}$
8 **return** single plan left in $worklist$;

- Wähle in jeder Iteration den kostengünstigsten Plan, der über den verbliebenen Unterplänen zu realisieren ist

Diskussion

- Greedy-Join-Enumeration
  - Zeitkomplexität: $O(n^3)$
    - Schleife: $O(n)$
    - Pro Durchgang alle Paare von verbliebene Plänen betrachtet: $O(n^2)$
- Andere Verbund-Generierungstechniken:
  - Randomisierte Algorithmen
    - Zufälliges Umschreiben zusammen mit Hill-Climbing und Simulated Annealing
  - Genetische Algorithmen
    - Exploration des Suchraums durch Kombination von Plänen (Nachfolger) und Abänderung bestimmter Merkmale (Mutation)

Physikalische Planeigenschaften

Betrachten wir die Anfrage,

```
SELECT O.O ORDERKEY, L.L EXTENDEDPRICE
FROM ORDERS O, LINEITEM L
WHERE O.O ORDERKEY = L.L ORDERKEY
```

in der Tabelle `ORDERS` mit einem geclusterten Index `OK_IDX` auf der Spalte `O_ORDERKEY` versehen ist

- Mögliche Pläne für ORDERS
  - Absuchen der ganzen Tabelle: #I/Os = N_{ORDERS}
  - Absuchen des Index: #I/Os = $N_{OK\_IDX} + N_{ORDERS}$
- Mögliche Pläne für LINEITEM
  - Absuchen der ganzen Tabelle: #I/Os = $N_{LINEITEM}$

Physikalische Planeigenschaften

- Absuchen der Tabellen ist jeweils günstigster Plan
- Verbund-Planer wird diesen Plan als günstigsten 1-Relationenplan in Durchgang 1 wählen[1]
- Für den Verbund beider Ausgaben gibt es:
  - Geschachtelte-Schleifen-Verbund
  - Hash-Verbund
  - Sortierung + Misch-Verbund
- Hash-Verbund und Misch-Verbund meist bevorzugt
  - Kosten hierfür ca. $2 \cdot (N_{ORDERS} + N_{LINEITEM})$
- Ergibt Gesamtkosten:
  - $N_{ORDERS} + N_{LINEITEM} + 2 \cdot (N_{ORDERS} + N_{LINEITEM})$

[1] Dynamische Programmierung und Greedy-Join unterscheiden sich hier nicht

Verbesserung des Plans

Ein besserer Plan liegt nahe

1. Verwende Absuchen des Index als Zugriff auf `ORDERS` (hierdurch ist die Ausgabe sortiert nach `O_ORDERKEY`)
2. Sortierte `LINEITEM` und
3. Berechne Verbund mittels Mischen

Gesamtkosten: $\underbrace{(N_{OK\_IDX} + N_{ORDERS})}_{1.} + \underbrace{2 \cdot N_{LINEITEM}}_{2./3.}$

Obwohl teurer als Einzelplan, günstiger im Gesamtgefüge

Interessante Ordnungen

- Vorteil des Index-basierte Zugriffs of ORDERS:
  Gute physikalische Eigenschaften
- Optimierer verwenden Annotation für
  Kandidatenpläne, um auch dieses zu berücksichtigen
- System R hat Konzept der „interessanten Ordnungen“
  eingeführt, bestimmt durch
  - Angaben ORDER BY oder GROUP BY in der Anfrage
  - Verbundattribute für weitere Verbunde (Misch-Verbund)
- In prune_plans(), führe
  - günstigsten „ungeordneten“ Plan und
  - günstigsten Plan mit interessanter Ordnung

Anfrageumschreibung

- Verbundoptimierung nimmt Menge von Relationen und Menge von Verbundprädikaten und bestimmt die beste Verbundreihenfolge
- Durch Umschreiben von Anfragegraphen im Vorwege Verbesserung der Effektivität der Optimierung
- Anfrageumschreiber verwendet (heuristische) Regeln, ohne den Datenbankzustand zu berücksichtigen
  - Umschreibung von Prädikaten
  - Entschachtelung von Anfragen

Prädikatsvereinfachung

Beispiel: Schreibe

```
SELECT *
FROM LINEITEM L
WHERE L.L TAX * 100 < 5
```

um in

```
SELECT *
FROM LINEITEM L
WHERE L.L TAX < 0.05
```

- Prädikatsvereinfachung ermöglicht Verwendung von Indexen und Vereinfacht die Erkennung von effizienten Verbundimplementierungen

Zusätzliche Verbundprädikate

Implizite Verbundprädikate wie in

```
SELECT *
FROM A, B, C
WHERE A.a = B.b AND B.b = C.c
```

können explizit gemacht werden

```
SELECT *
FROM A, B, C
WHERE A.a = B.b AND B.b = C.c AND A.a = C.c
```

Hierdurch werden Pläne möglich wie $(A \bowtie C) \bowtie B$

Vorher schien $(A \bowtie C)$ ein kartesisches Produkt zu sein

Geschachtelte Anfragen

SQL bietet viele Wege, geschachtelte Anfrage zu schreiben

- Unkorrelierte Unteranfragen

```
SELECT *
FROM ORDERS O
WHERE O_CUSTKEY IN (SELECT C_CUSTKEY
                    FROM CUSTOMER
                    WHERE C_NAME = 'IBM Corp.')
```

- Korrelierte Unteranfragen

```
SELECT *
FROM ORDERS O
WHERE O_CUSTKEY IN (SELECT C.C_CUSTKEY
                    FROM CUSTOMER
                    WHERE C.C_ACCTBAL = O.O_TOTALPRICE)
```

Anfrageentschachtelung

- Schachtelung kann teuer sein
  - Ein unkorrelierte geschachtelte Anfrage braucht allerdings nicht immer wieder ausgewertet zu werden
- Häufig wird Schachtelung nur verwendet, um einen Verbund auszudrücken (oder einen Semi-Verbund)
- Anfrageumschreiber kann solche Verwendung entdecken und den Verbund explizit machen
- Dadurch kann Anfrageschachtelung durch Verbundoptimierung behandelt werden

Won Kim. On Optimizing an SQL-like Nested Query.
ACM TODS, vol. 7, no. 3, **1982**

Zusammenfassung

- Anfrageparser
  - Übersetzung der Anfrage in Anfrageblock
- Umschreiber
  - Logische Optimierung (unabhängig vom DB-Inhalt)
  - Prädikatsvereinfachung
  - Anfrageentschachtelung
- Verbundoptimierung
  - Bestimmung des „günstigsten“ Plan auf Basis
    - eines Kostenmodells (I/O-Kosten, CPU-Kosten) und
    - Statistiken (Histogramme) sowie
    - Physikalischen Planeigenschaften (interessante Ordnungen)
  - Dynamische Programmierung, Greedy-Join

Beim nächsten Mal...