Seminar Business Intelligence Teil I: OLAP und Data Warehousing

Thema 4: Anfrageevaluierung und Optimierung

Vortrag von Philipp Breitbach

Übersicht

1. Motivation

2. Umsetzung des CUBE-Operators

3. Umsetzung von Bereichssummenanfragen

4. Zusammenfassung

Einordnung

Bisher: Multidimensionales Modell und OLAP-Anfragesprachen

Dadurch: Externe Sichtweise auf die Funktionalität der Operatoren

Aber: Keinerlei Information über die algorithmische Umsetzung

Jetzt: Umsetzung von multidimensionalen Operatoren

Hier: CUBE-Operator und Bereichssummenanfragen

2. Umsetzung des CUBE-Operators

Umsetzung des CUBE-Operators anhand des *PipeSort*-Algorithmus

Dazu: - Syntax und Ergebnis des CUBE-Operators

- Untragbarkeit einer „naiven" Berechnung des Würfels
- Einführung des Suchgitters zur Interndarstellung einer CUBE-Anfrage und der prinzipiellen Berechnungsstrategie
- Verwendung einer Sortierungsbasierten Berechnung einer einzelnen Group-By und die globale Bedeutung der Sortierungsreihenfolge
- Detaillierte Erläuterung des *PipeSort*-Algorithmus

Syntax

Aufruf durch:

```
SELECT Produkt, Jahr, Kunde, SUM(Umsatz)
FROM Verkäufe
CUBE-BY Produkt, Jahr, Kunde
```

Auf Tabelle Verkäufe: Produkt, Jahr, Kunde, Umsatz.

Notation: (Produkt(P), Jahr(J), Kunde(K))

```
SELECT Produkt, Jahr, Kunde, SUM(Umsatz)
FROM Verkäufe
GROUP BY Produkt, Jahr, Kunde
```

Ergebnis

Aufruf mit n Dimensionen

→ Berechnung und Vereinigung von 2^n Group-By's

Ergebnis des Cubes der vorigen Folie:

(P, J, K) ⟶ 3D-Hypercube

UNION (P, J) **UNION** (P, K) **UNION** (J, K) ⟶ 2D-Hypercubes

UNION (P) **UNION** (J) **UNION** (K) ⟶ 1D-Hypercubes

UNION () ⟶ Gesamtsumme oder 0D-Hypercube

Naiver Berechnungsansatz

Isolierte Berechnung der 2^n Group-By's

Vorteil: Einfacher und übersichtlicher Berechnungsalgorithmus

Nachteil: 2^n Scans der Basistabelle nötig!

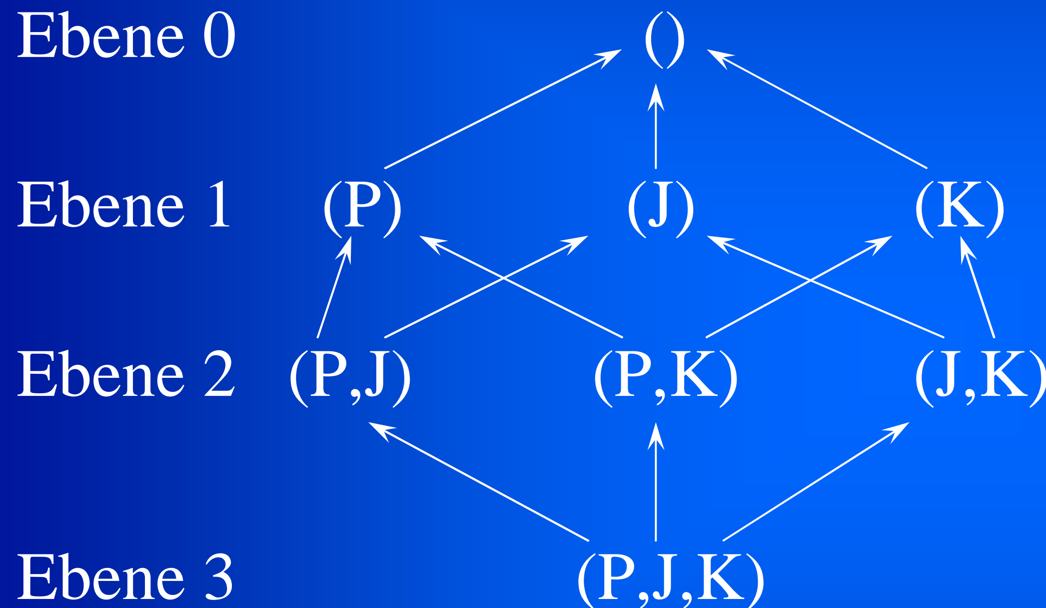
Bei Basistabelle mit 10^9 Tupeln, 1KB/Tupel, 4MB/Seite und 6 Dimensionen:

10^9KB/4MB ≈ 10^9KB/4000KB = 250000 Seiten
→ 2^6*250000 = 16 Millionen Seitenzugriffe!!!

Untragbar für OLAP-Analyse!!!

Lösung: Berechne (P) z.B. aus (P, J) anstatt aus der Basistabelle
→ Suchgitter ausnutzen

Suchgitter



Suchgitter ist ein gerichteter Graph mit:

- 2^n Group-By's als Knoten
- Kante von A nach B falls B aus A berechnet werden kann

Suchgitter als interne Darstellung einer CUBE-Anfrage

Prinzipielle Berechnungsstrategie

- Berechnung der Group-By aus Ebene n aus der Basistabelle (Rohdaten)

- Berechnung der restlichen Group-By‘s aus einem ihrer Väter im Suchgitter

→ Problem: Aus welchem Vater???

Optimale Lösung dieses Problems durch *PipeSort*, basierend auf Berechnungskosten für jede Kante im Suchgitter

Optimierungsstrategien

„Smallest-Parent“: Auswahl des kleinsten Vaters zur Berechnung einer Group-By

„Amortize-Scans“: Gleichzeitige Berechnung (in einem Scan) möglichst vieler Group-By‘s

„Cache-Results“: Cachen von Group-By‘s im Speicher zur Berechnung anderer Group-By‘s ohne Sekundärspeicherzugriff

„Share-Sorts“: Aufteilung der Sortierungskosten zwischen möglichst vielen Group-By‘s (bei Sortierungs-basierter Berechnung einer einzelnen Group-By

Globale Bedeutung der Sortierungsreihenfolge

Ergebnis der Sortierungsbasierten Berechnung von (P,J,K) liegt in bestimmter Sortierungsreihenfolge vor, z.B. P→J→K:

(P,J,K):

Produkt	Jahr	Kunde	Umsatz
Mixery	2002	Meier	50 €
Mixery	2003	Maier	100 €
Ur-Pils	2002	Maier	70 €
Ur-Pils	2003	Meier	30 €

Umsortierung nötig um (P,K) aus so sortierter (P,J,K)-Tabelle zu berechnen

→ Sortierungskosten müssen in das Kostenmodell zur Bestimmung eines optimalen Plans eingehen!

Kostenmodell

Unterscheidung von zwei Kostenarten:

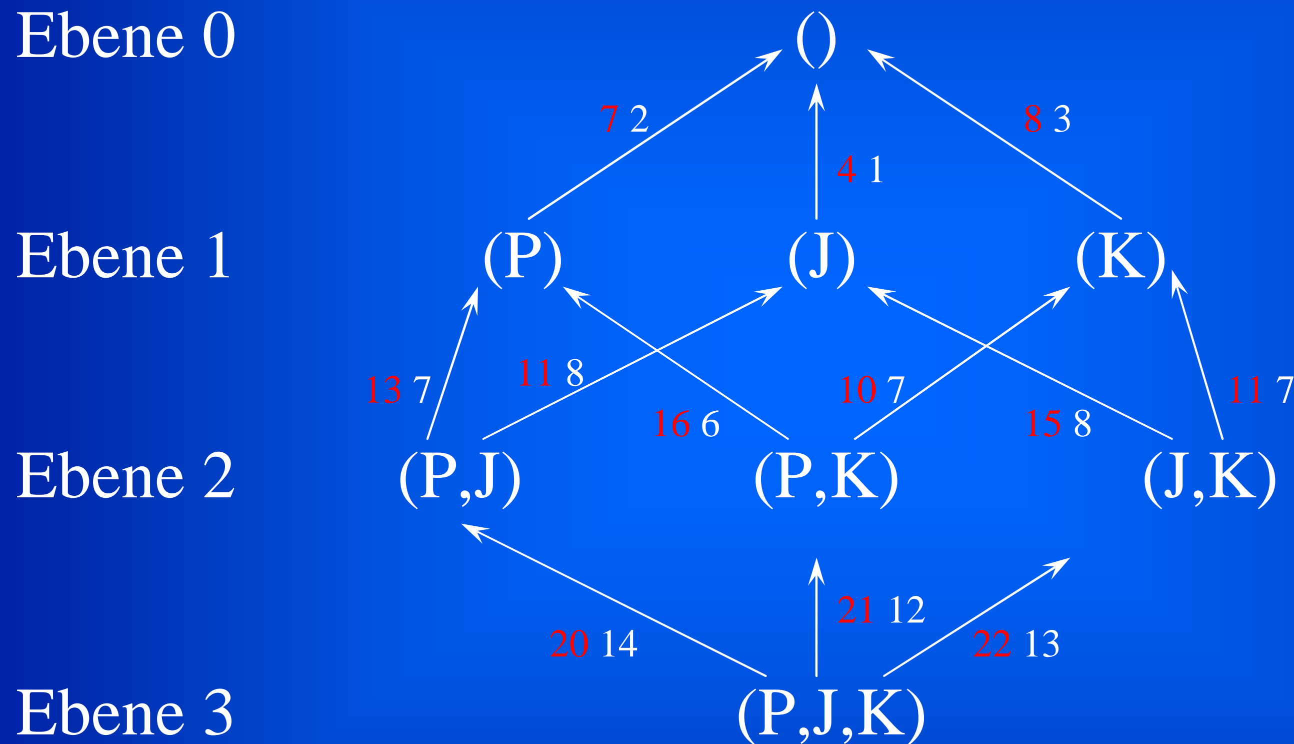
- Kosten um Group-By i aus Group-By j zu berechnen, falls Sortierung *nicht notwendig* (A-Kosten, <u>A</u>lready sorted)
- Kosten um Group-By i aus Group-By j zu berechnen, falls Sortierung *notwendig* (S-Kosten, <u>S</u>till to sort)

Gewinnung der A- und S-Kosten aufgrund von statistischen Schätzwerten über die Datenverteilung

→ Werden hier als vorhanden vorrausgesetzt

→A- und S-Kosten zu jeder Kante des Suchgitters hinzufügen: So modifiziertes Suchgitter ist Eingabe für *PipeSort*

Modifiziertes Suchgitter



- S-Kosten
- A-Kosten

PipeSort-Algorithmus (1)

Eingabe: Suchgitter mit A- und S-Kosten

Ausgabe: Teilgitter mit minimalen Kantenkosten
→ Optimaler Plan

Darauf basierend: Berechnung der Group-By's in Pipelines

PipeSort-Algorithmus (2)

Idee: Iteration von k=0 bis n-1 über Ebenen des Suchgitters:

∀k: Finden des besten Plans um Ebene k aus Ebene k+1 zu berechnen

Umsetzung: Reduktion der Suche nach dem optimalen Plan im Iterationsschritt auf das *weighted-bipartite-matching-problem*(WBMP)

PipeSort-Algorithmus (3)

WBMP:

Gegeben: Graph(hier: Suchgitter) aufgeteilt in 2 disjunkte Knotenmengen V_1(Ebenen unter Ebene k) und V_2(Rest des Suchgitters).
Außerdem Kantenmenge E, die V_1 und V_2 verbindet, mit Kantengewichten(A- oder S-Kosten) für jede Kante.

Gesucht: $E' \subseteq E$, so dass Knoten aus V_1 mit höchstens einem Knoten aus V_2 verbunden und umgekehrt
Und E' hat maximales Gewicht.

PipeSort-Algorithmus (4)

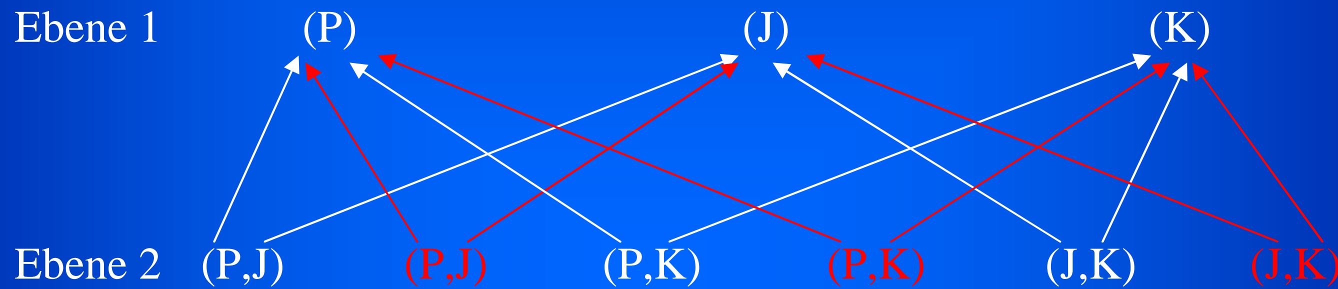
Reduktion des Iterationsschritts auf WBMP:

- Hinzufügen von k Kopien für jeden Knoten aus Ebene k+1
- Verbinden der Kopien mit denselben Knoten aus Ebene k wie das Original
- Zuteilen von A-Kosten für alle ausgehenden Kanten jedes Originals
- Zuteilen von S-Kosten für alle ausgehenden Kanten jeder Kopie
- Allerdings: Suche nach Kantenmenge mit *minimalen* Kosten

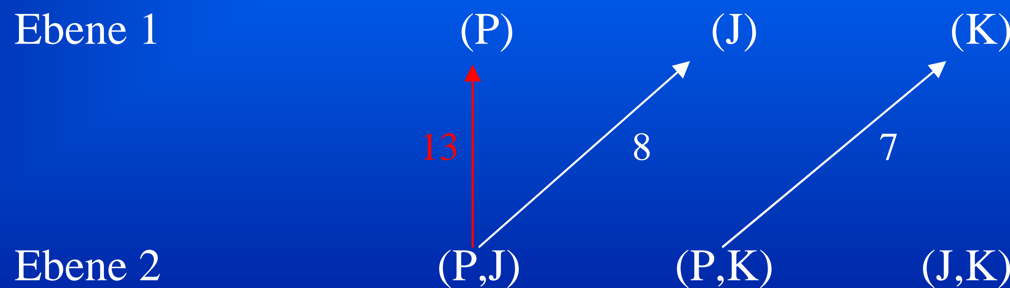
Zur Lösung des WBMP's wird der von Papadimitriou/Steiglitz entwickelte Algorithmus genutzt, der hier nicht näher erläutert wird.

PipeSort-Algorithmus (5)

Beispiel für Reduktion:



Transformation durch WBMP-Algortihmus



(P,J): Kopie

(P,J): Original

S-Kosten

A-Kosten

PipeSort-Algorithmus (6)

Nach Anwendung des WBMP-Algorithmus um Plan für Berechnung von Ebene k aus Ebene k+1 zu berechnen:

Jeder Knoten A aus Ebene k mit *genau einem* Knoten B aus Ebene k+1 verbunden. → 2 Fälle

1. Fall: A und B sind durch A-Kante verbunden

→ Setze die Sortierungsreihenfolge zur Berechnung von Group-By B auf die Sortierungsreihenfolge für A.

2. Fall: A und B sind durch S-Kante verbunden

→ B wird schon zur Berechnung eines anderen Ebene-k-Knoten verwendet und muss zur Berechnung von A umsortiert werden.

PipeSort-Algorithmus (7)

PipeSort in Pseudocode:

For Ebene k=0 to n-1

  GeneratePlan(k+1→k);

  Foreach Group-By g in Ebene k+1

    If eine A-Kante von g zu einem Knoten h aus Ebene k existiert

    Then setze die Sortierungsreihenfolge von g auf die von h;

GeneratePlan(k+1→k):

k Kopien für jeden Knoten aus Ebene k+1 machen;

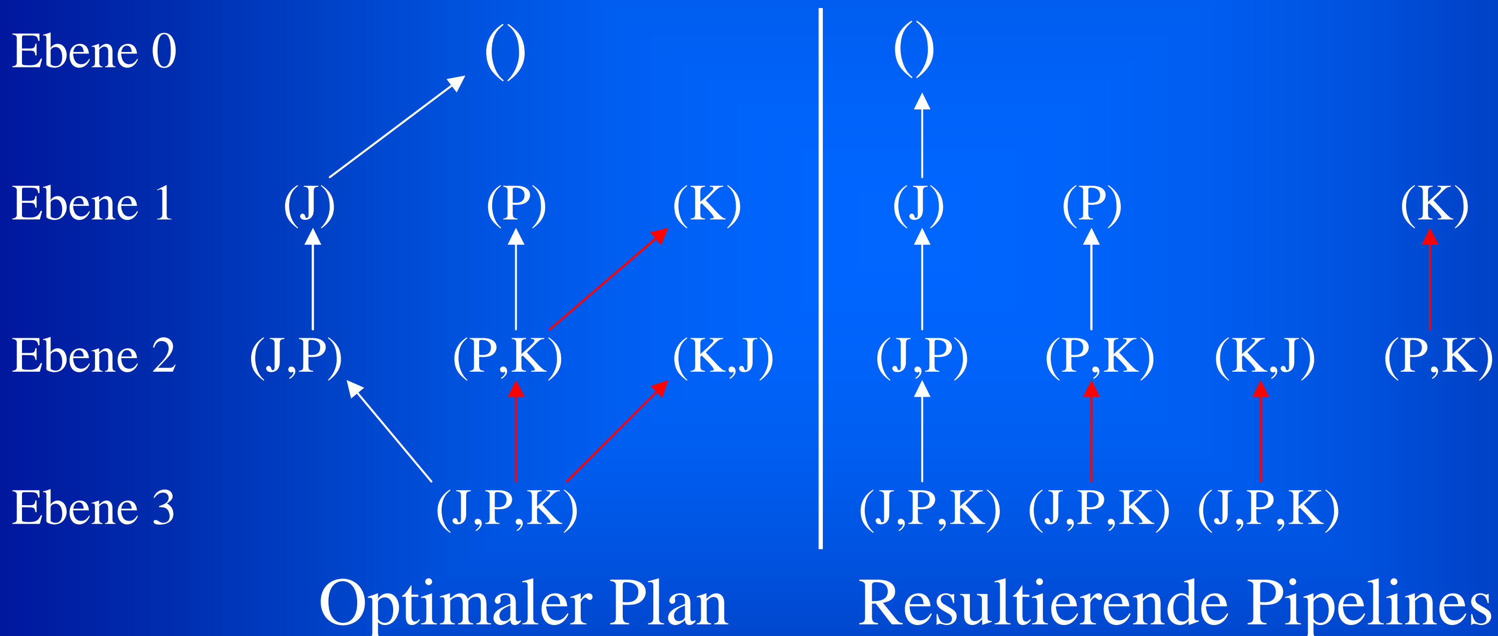
Jeden kopierten Knoten mit denselben Knoten wie Original verbinden;

Ordne Kanten A-Kosten für Originalknoten und S-Kosten für Kopien zu;

Finde minimale Kosten durch WBMP-Algorithmus;

PipeSort-Algorithmus (8)

Ausgabe von *PipeSort*:



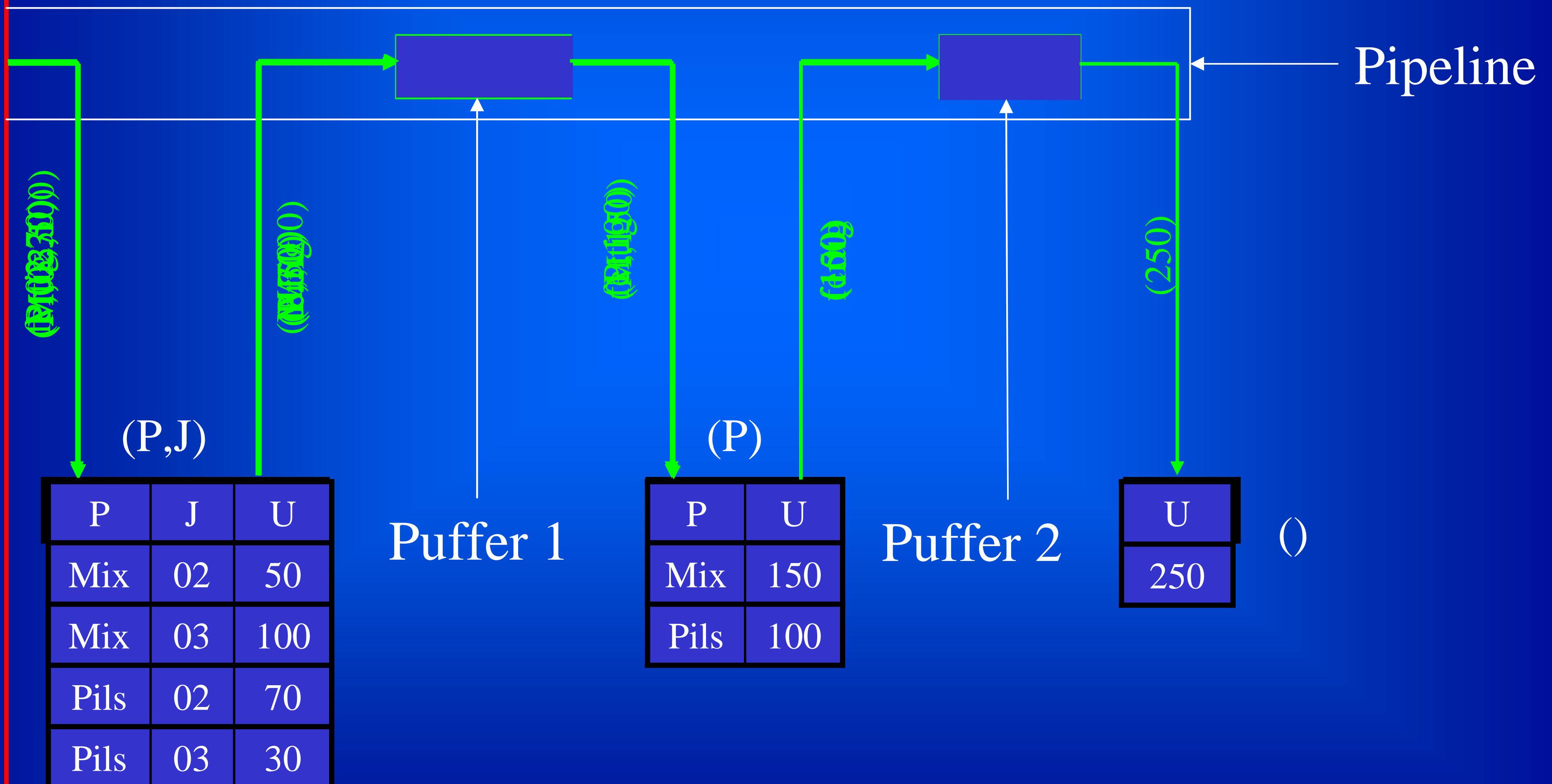
Optimaler Plan Resultierende Pipelines

A → B: Berechnung von Group-By B aus A, keine Neusortierung

A → B: Neusortierung nötig, um B aus A zu berechnen

Beispiel für Pipeline-Berechnung

Berechnung von ...(P,J) → (P) → ():



P	J	U
Mix	02	50
Mix	03	100
Pils	02	70
Pils	03	30

P	U
Mix	150
Pils	100

U
250

Umsetzung der Optimierungsstrategien

„smallest parent“: Durch die Auswahl anhand der A- und S-Kosten berücksichtigt.

„amortize-scans“: Berechnung mehrerer Group-By‘s in einer Pipeline während eines Scans

„cache-results“: Zwischenspeichern der aktuellen Aggregate in den Puffern einer Pipeline

„share-sorts“: Berechnung von durch A-Kanten verbundenen Group-By‘s ohne zusätzliche Sortierungskosten

→ Leistung von *PipeSort* kommt sehr nahe an geschätzte untere Schranken zur Berechnung des Datenwürfels heran!

3. Umsetzung von Bereichssummenanfragen

Letzter Teil: Berechnung des Datenwürfels

Dieser Teil: Anfragen auf dem Datenwürfel
genauer: Bereichssummenanfragen

Bereichssummen: Gesamtsummen eines zusammenhängenden Bereichs des Würfels

Im allgemeinen keine Deckung des Bereiches mit Klassifikationshierarchien der Dimensionen

Also nicht nur durch Roll-Up‘s berechenbar

Effiziente Berechnung durch vorberechnete Präfixsummen möglich

Modell

Seien D={1,2,...,d} Menge der Dimensionen des Würfels, $n_1, n_2, \ldots, n_d$ Beträge der Mengen der jeweiligen Dimensionselemente

Datenwürfel modelliert als d-dimensionales Array A der Grösse

$$N = \prod_{j=1}^{d} n_j$$

Bereichssummenanfrage modelliert durch:

$$Sum(l_1{:}h_1,\ldots,l_d{:}h_d) = \sum_{i_1=l_1}^{h_1} \cdots \sum_{i_d=l_d}^{h_d} A[i_1,\ldots,i_d],$$

wobei $l_j{:}h_j$ den ausgewählten Bereich der Dimension $j \in D$ bezeichne mit $l_j \leq i_j \leq h_j$

Präfixsummenarray

Präfixsummenarray P:

d-dimensionales Array der Größe $N = \prod_{j=1}^{d} n_j$

also derselben Kardinalität wie A

P speichert vorberechnete Präfixsummen:

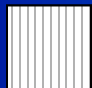
$\forall\ (0 \le x_j < n_j) \wedge (j \in D)$: $P[x_1,\ldots,x_d] = \mathit{Sum}(0{:}\ x_1,\ldots,0{:}\ x_d)$

$$= \sum_{i_1=l_1}^{h_1} \cdots \sum_{i_d=l_d}^{h_d} A[i_1,\ldots,i_d]$$

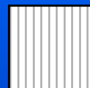
Beispiel für A und P

Array A						
Index	0	1	2	3	4	5
0	3	5	1	2	2	3
1	7	3	2	6	8	2
2	2	4	2	3	3	5

Array P						
Index	0	1	2	3	4	5
0	3	8	9	11	13	16
1	10	18	21	29	39	44
2	12	24	29	40	53	63



$21 = 3 + 5 + 1 + 7 + 3 + 2$



$53 = 3 + 5 + 1 + 2 + 2 + 7 + 3 + 2 + 6 + 8 + 2 + 4 + 2 + 3 + 3$

Range-Sum-Algorithmus

Berechnung von Bereichssummen auf A aus 2^d Elementen aus P:

$$\forall j \in D: \quad Sum(l_1:h_1,\ldots,l_d:h_d) = \sum_{\forall x_j \in \{l_j-1, h_j\}} \left\langle \left(\prod_{i=1}^{d} s(i)\right) * P[x_1,\ldots,x_d] \right\rangle,$$

$$\text{mit } s(i) = \begin{cases} 1, \mathit{falls} & x_j = h_j \\ -1, \mathit{falls} & x_j = l_j - 1 \end{cases}$$

und $P[x_1,\ldots,x_d] = 0$, falls $x_j = -1$ für ein $j \in D$.

Beispiel

Berechnungsvorschrift:

$$\forall j \in D: \quad Sum(l_1{:}h_1,\ldots,l_d{:}h_d) = \sum_{\forall x_j \in \{l_j - 1, h_j\}} \left\langle \left(\prod_{i=1}^{d} s(i)\right) * P[x_1,\ldots,x_d] \right\rangle,$$

$$\text{mit } s(i) = \begin{cases} 1, \; falls & x_j = h_j \\ -1, \; falls & x_j = l_j - 1 \end{cases}$$

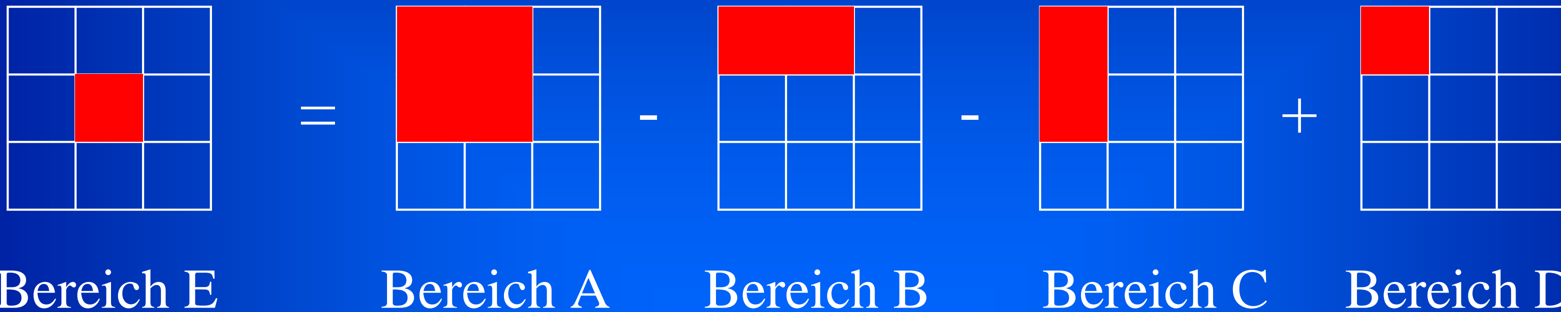
Berechnung von $Sum(l_1{:}h_1,l_2{:}h_2)$ mit d=2 aus 2^2=4 Elementen aus P:

$$Sum(l_1{:}h_1,l_2{:}h_2) = P[h_1,h_2] - P[h_1,l_2-1] - P[l_1-1,h_2] + P[l_1-1,l_2-1]$$

z.B. $Sum(2{:}3,1{:}2) = P[3,2] - P[3,0] - P[1,2] + P[1,0]$

$$= 40 - 11 - 24 + 8 = 13$$

Geometrische Veranschaulichung



Bereich E Bereich A Bereich B Bereich C Bereich D

Array A						
Index	0	1	2	3	4	5
0	3	5	1	2	2	3
1	7	3	2	6	8	2
2	2	4	2	3	3	5

Array P						
Index	0	1	2	3	4	5
0	3	8	9	11	13	16
1	10	18	21	29	39	44
2	12	24	29	40	53	63

Variationen des Algorithmus

Löschen von Array A nach Berechnung von P, denn:

$A[x_1,...,x_d] = \mathit{Sum}(x_1{:}x_1,...,x_d{:}x_d)$

→ Zugriff auf 2^d Elemente von P anstatt auf 1 Element von A bei Einzelelementanfragen

Speicherung von geblockten Präfixsummen in P auf gröberer Ebene mit Hilfe eines Blockfaktors b:

Speicherung von $P[x_1,...,x_d]$ nur, falls $x_i \bmod (b\text{-}1) = 0$ $(\forall i \in D)$
oder $x_i = n_i - 1$ für ein $i \in D$

→ Reduzierung des Speicherplatzes für P, aber Zugriffe auf A nötig bei Bereichen die nicht mit Blockstruktur übereinstimmen

4. Zusammenfassung

Anfrageevaluierung und Optimierung anhand des CUBE-Operators sowie Bereichssummenanfragen:

CUBE: Berechnung durch *PipeSort*-Algorithmus
→ Nahe an geschätzter unterer Schranke zur Berechnung des Würfels
Erläuterung der genutzten Optimierungen

Bereichssummen: Berechnung durch Range-Sum-Algorithmus
→ Berechnung auf Würfel mit d Dimensionen durch Zugriff auf 2^d Elemente eines vorberechneten Präfixsummenarrays
Flexible Möglichkeiten für Kompromiss zwischen Laufzeit und Speicherlast