Data Warehousing und Data Mining

Implementierung von OLAP Operationen

Ulf Leser

Wissensmanagement in der Bioinformatik

Inhalt dieser Vorlesung

- Wiederholung: OLAP Operationen
- Implementierung der Gruppierung
- Implementierung von Cube & Iceberg-Cube

ROLLUP Beispiel

```
SELECT    T.year_id, T.month_id, T.day_id, sum(...)
FROM      sales S, time T
WHERE     T.day_id = S.day_id
GROUP BY  ROLLUP(T.year_id, T.month_id, T.day_id)
```

1997	Jan	1	200
1997	Jan	...	
1997	Jan	31	300
1997	Jan	ALL	31.000
1997	Feb	...	
1997	March	ALL	450
1997	...	...	
1997	ALL	ALL	1.456.400
1998	Jan	1	100
1998	...	...	
1998	ALL	ALL	45.000
...	...	...	
ALL	ALL	ALL	12.445.750

Multidimensionale Aggregation

Verkäufe nach Produktgruppen und Jahren

	1998	1999	2000	Gesamt
Weine	15	17	13	45
Biere	10	15	11	36
Gesamt	25	32	24	81

- `sum() ... GROUP BY pg_id, year_id`
- `sum() ... GROUP BY pg_id`
- `sum() ... GROUP BY year_id`
- `sum()`

- Besser: CUBE Operator

Inhalt dieser Vorlesung

- Wiederholung: OLAP Operationen
- Implementierung der Gruppierung
  - Sortieren oder hashen?
  - Algebraische Optimierung
- Implementierung von Cube & Iceberg-Cube

GROUP BY

- Gruppierung
  - Partitioniere die Tupel der Relation nach den Attributen G der GROUP BY Klausel
    - Gruppierungsattribute: G
  - Berechne die Aggregatfunktion für jede Teilmenge
  - Für jede Gruppe entsteht eine Zeile in der Ergebnisrelation
- Wie macht man das bei sehr vieler Tupeln?
  - Tupelstrom kann nicht im Hauptspeicher gehalten werden

Hashbasierte Implementierung

- Implementierung über Hashing
  - Lese Tupelstrom und hashe Werte von G
    - Kollisionsfrei hashen
    - Erzeugt einen Bucket pro Wertekombination in G
    - Größe der Buckets vorher unklar – schwieriges Speichermgt
      - Überläufe führen zu Paging – ganz schlecht
  - Wende Aggregatfunktion an – aber wann?
- Distributive (und algebraische) Aggregationsfunktionen
  - Bei jedem neuen Tupel wird sofort aggregiert
    - SUM: Addiert; AVG: count und sum; ...
  - Pro Bucket nur ein / wenige Werte notwendig
- Holistischen Funktionen: Erst, wenn Gruppe komplett ist
  - Aber man weiß nicht, wann das letzte Tupel einer Gruppe kam
  - Also muss man effektiv alle Tupel im Speicher halten

Sortierung

- Implementierung durch Sortierung
  - Sortiere den Tupelstrom nach G
    - Im Hauptspeicher oder mit externer Sortierung
  - Lies den sortieren Tupelstrom
    - Tupel puffern (holistisch) oder sofort (distributiv, algebraisch) aggregieren
    - Wenn sich Werte von G ändern beginnt eine neue Partition
    - Pipelining: Aggregierter Wert kann sofort weitergereicht werden, wenn die Eingabe schon sortiert ist
- Vorteil: Benötigt im schlimmsten Fall nur so viel Platz wie die größte Partition
  - Bei distributiven Funktionen sogar nur konstant viel Platz
- Nachteil: Erfordert Sortierung

Fazit

- Wenn alle Tupel in den Speicher passen – Hashen
- Bei distributiven / algebraischen Funktionen - hashen
- Wenn Eingabe sortiert ist – „Sortieren“ (und pipelinen)
- Wie kann man Speicherbedarf vorher abschätzen?
  - Naive Abschätzung
    - COUNT DISTINCT einzeln auf alle Attribute in G machen
    - Anzahl Werte von G = Produkt aller Counts
    - Worst Case: Nimmt an, dass alle Kombinationen vorkommen
  - Abzählen
    - COUNT DISTINCT auf G
    - Dann kann man (fast) gleich die Gruppierung ausrechen (echt?)
  - Besser: Sampling

Multi-Phase Gruppierung

- Erinnerung: Externes SORT muss Daten ca. 4 mal lesen
- Multi-Phase GROUP-BY
  - COUNT DISTINCT auf G (einmal lesen)
  - Größe und Zahl aller Gruppen bekannt
  - Man kann exakt bestimmen, mit welchen Gruppen man den Speicher genau ausfüllt
  - Gruppen entsprechend partitionieren und pro Partition (=Menge von GROUP-BY Kombinationen) erneut scannen
- Wann kann das lohnen?
  - Oft reicht der Speicher – aber nur, wenn man pro Gruppe genau richtig viel Platz reservieren kann
  - Auch bei SUM, wenn man über mehrere Attribute mit hoher Kardinalität gruppiert und es theoretisch mehr Kombinationen als Speicher gibt

Algebraische Optimierung

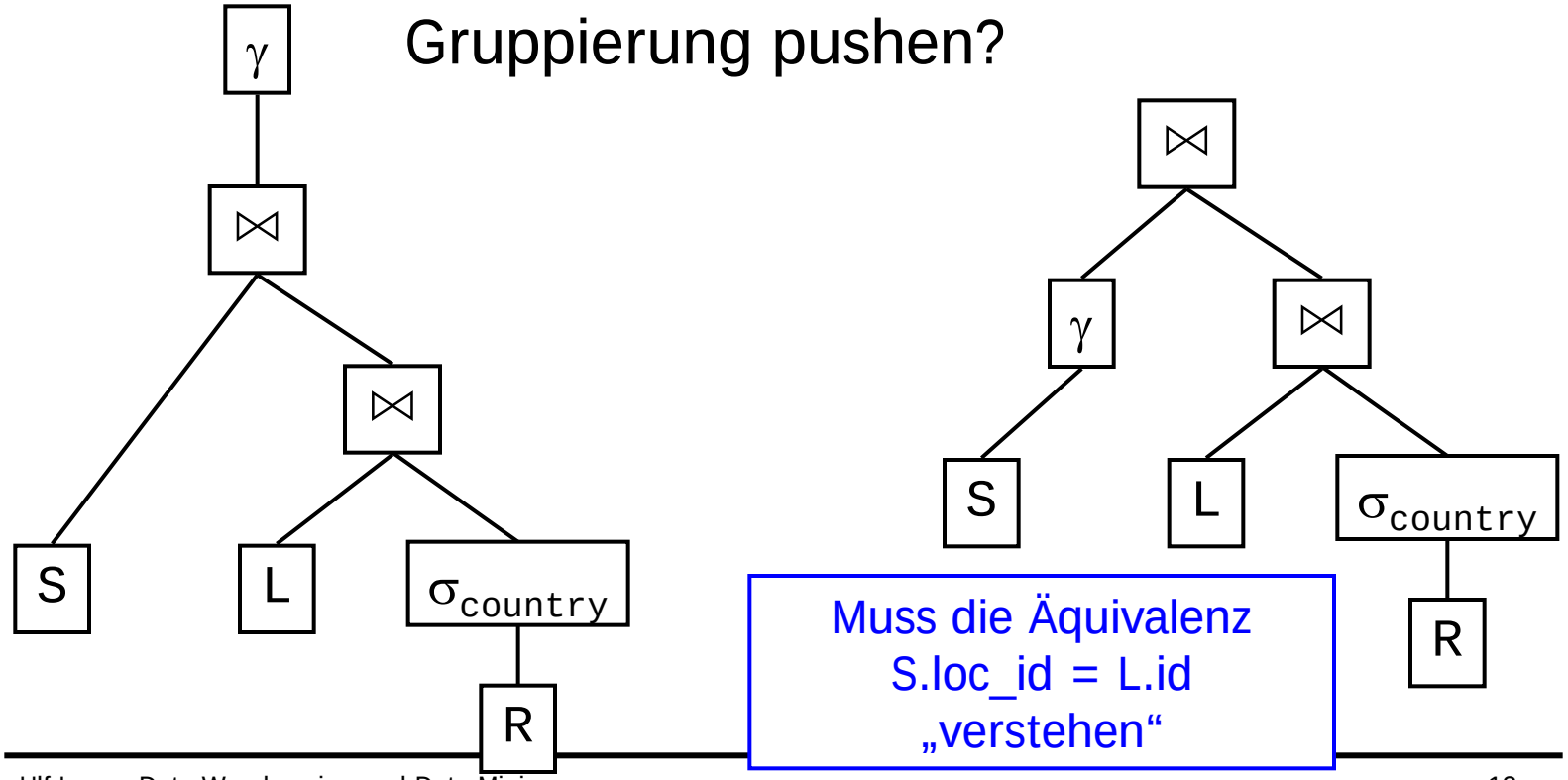
- Bisher nehmen wir an, dass ein GROUP BY als vorletzte Operation ausgeführt wird
  - Vor ORDER-BY (und Auswertung HAVING)
- Aber: Gruppierung reduziert die Anzahl Tupel
- Daher ist es sinnvoll, Gruppierung möglichst zu pushen
- Beispiel (Snowflake-Schema)

```
SELECT S.product_id, L.id, SUM(amount)
FROM   sales S, location L, region R
WHERE  S.loc_id = L.id AND
       L.region_id = R.id AND
       R.country = ‚BRD‘
GROUP BY S.product_id, L.id
```

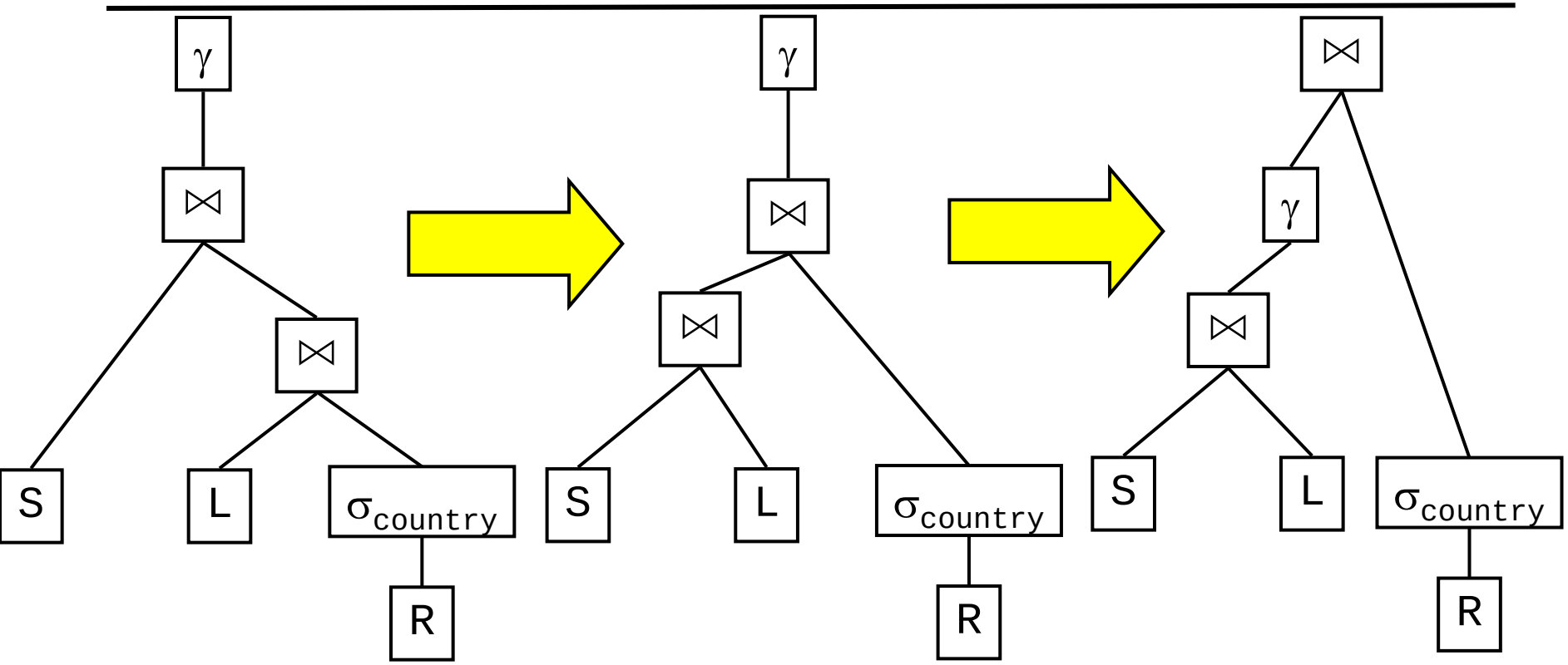
Erster Versuch

```
SELECT S.product_id, L.id, SUM(amount)
FROM   sales S, location L, region R
WHERE  S.loc_id = L.id AND
       L.region_id = R.id AND
       R.country = ‚BRD‘
GROUP BY S.product_id, L.id
```

Gruppierung pushen?



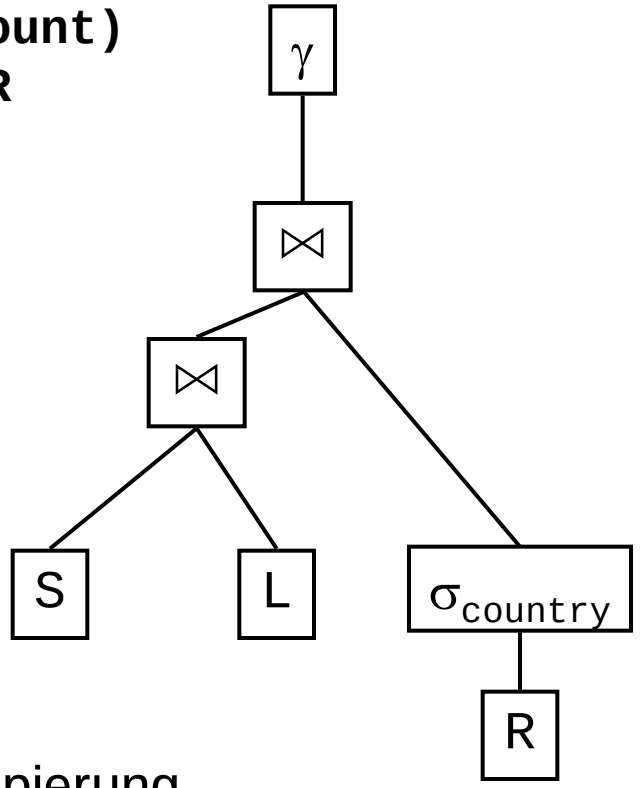
Alternative



- Bedingung: **`L.region_id`** muss im Ergebnis der Gruppierung sein
  - Z.B Umschreiben in **`SELECT S.product_id, L.id, max(l.region_id)`**
- Pushen oder nicht? Kostenbasierte Entscheidung
  - Der Join S⋈L ist größer im dritten Plan, da Filter auf ‚BRD' erst später erfolgt
  - Dafür gibt es weniger Tupel vor dem Join mit Region

Zweites Beispiel

```
SELECT S.product_id, L.type, SUM(amount)
FROM   sales S, location L, region R
WHERE  S.loc_id = L.id AND
       L.region_id = R.id AND
       R.country = ‚BRD‘
GROUP BY S.product_id, L.type
```



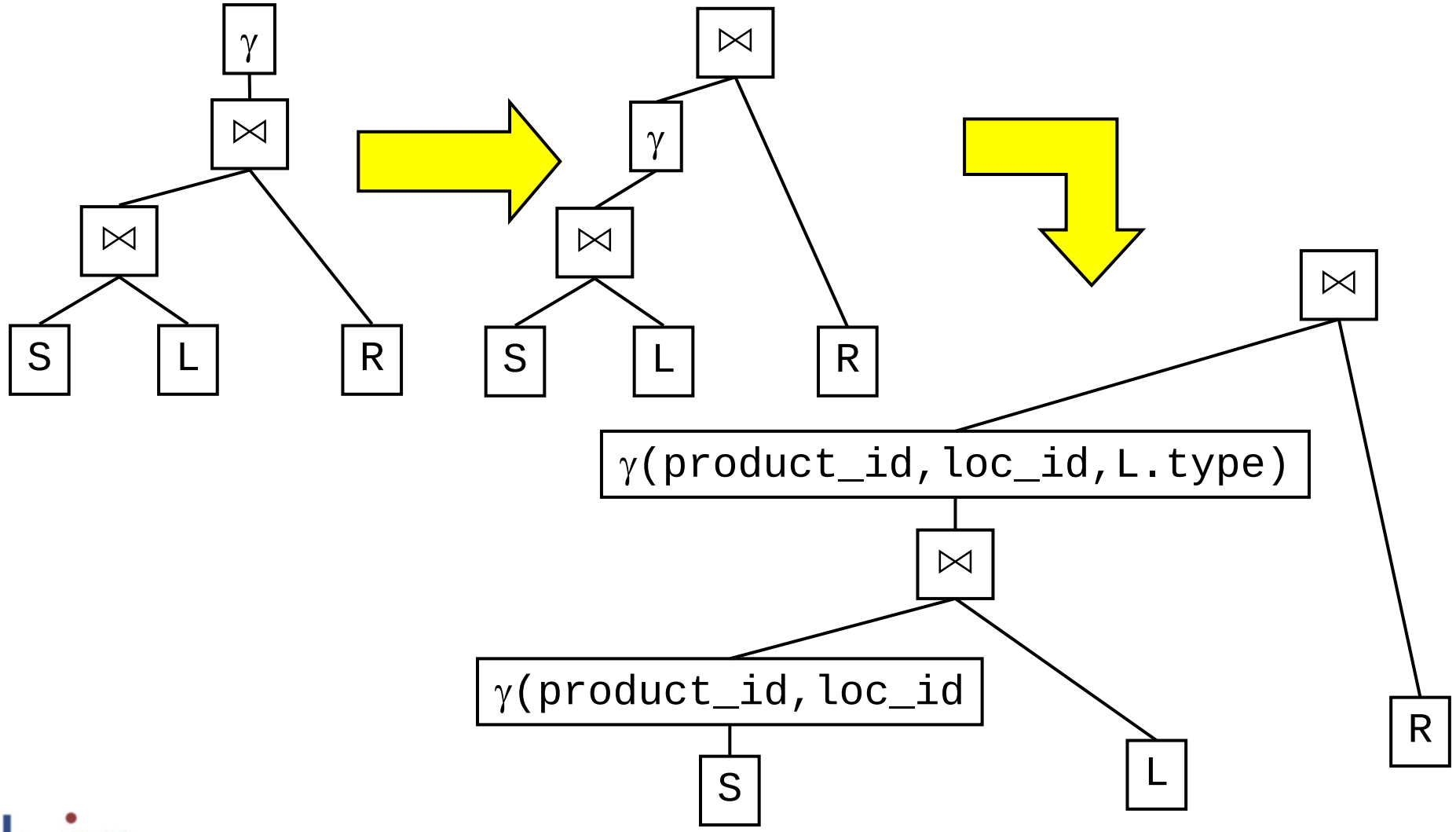
- Annahme: Tabelle **location** hat ein Attribut **type**
- Gruppierung pushen?
  - Geht nicht: Es gibt viele Fremdschlüssel L.region_id pro Gruppierung
- Allgemein: Pushen nur möglich, wenn
  - Alle Joinattribute sind Teil der GROUP-BY Attribute
  - Keine Attribute anderer Tabellen sind Teil des GROUP-BY

Präaggregation

- Möglich (und oftmals sinnvoll): Präaggregation
  - Mehr Operationen
  - Aber Reduktion der Zwischenergebnisse
- Beispiel
  - ```
    SELECT  S.product_id, L.type, L.id, max(R.r_name)
            SUM(amount), COUNT(amount)
    FROM    sales S, location L, region R
    WHERE   S.loc_id = L.id AND
            L.region_id = R.id
    GROUP BY S.product_id, L.type, L.id
    ```
    ```

Präaggregation

```
SELECT  S.product_id, L.type, L.id, max(R.r_name)
        SUM(amount), COUNT(amount)
FROM    sales S, location L, region R
WHERE   S.loc_id = L.id AND …
GROUP BY S.product_id, L.type, L.id
```



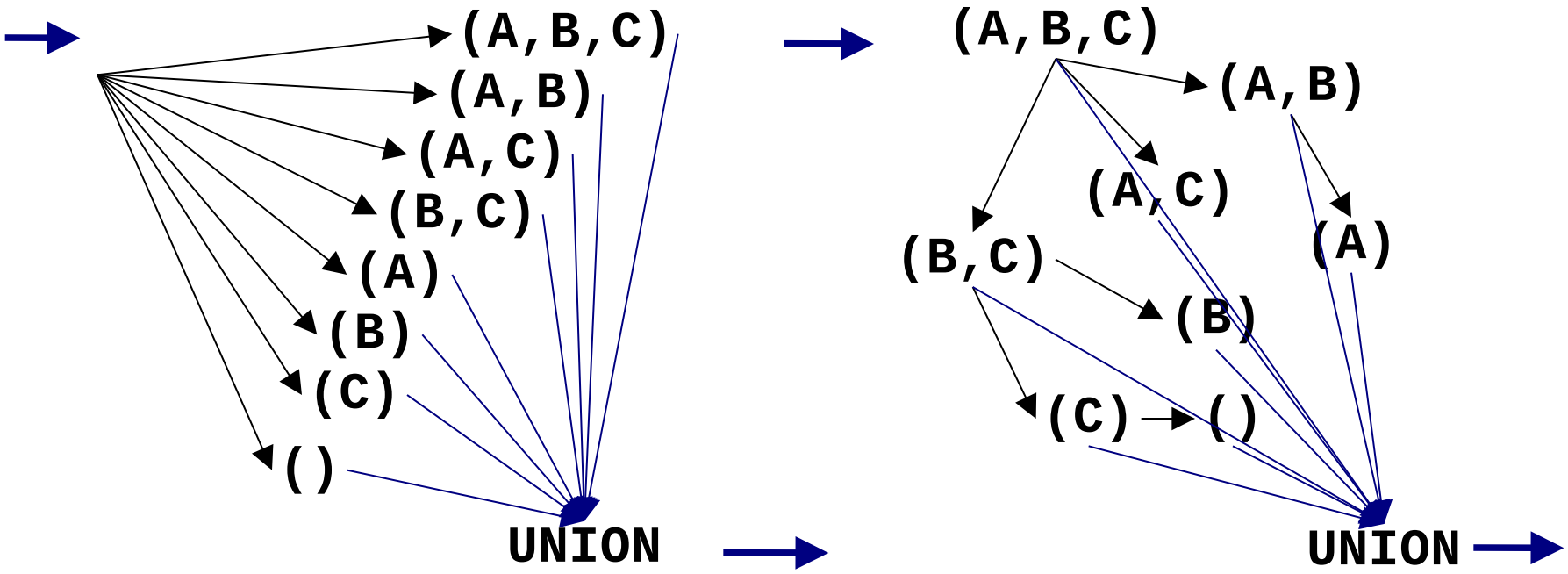
Inhalt dieser Vorlesung

- Wiederholung: OLAP Operationen
- Implementierung der Gruppierung
- Implementierung von Cube & Iceberg-Cube
  - Problem und Potential
  - Cube Algorithmen
  - Iceberg Cubes

Problem und Potential

- CUBE Operator berechnet Gruppierung auf allen Teilmengen seiner Attribute
  - CUBE(A,B,C) = GROUP BY (),(A),(B),(C),(AB),(AC),(BC),(ABC)
- Problem
  - Es gibt 2^n Gruppierungen
  - Jede Gruppierung einzeln (mit ggf. mehrmaligem Sortieren / Scan des Tupelstroms) ausführen dauert zu lange
- Potential
  - Gruppierungen können andere Gruppierungen als Präaggregate nutzen
  - Beispiel: (ABC) $\rightarrow$ (AB) $\rightarrow$ (B) $\rightarrow$ ()
  - Das lohnt sich: Je weniger Gruppierungsattribute, desto weniger Werte im Ergebnis

Realisierung CUBE



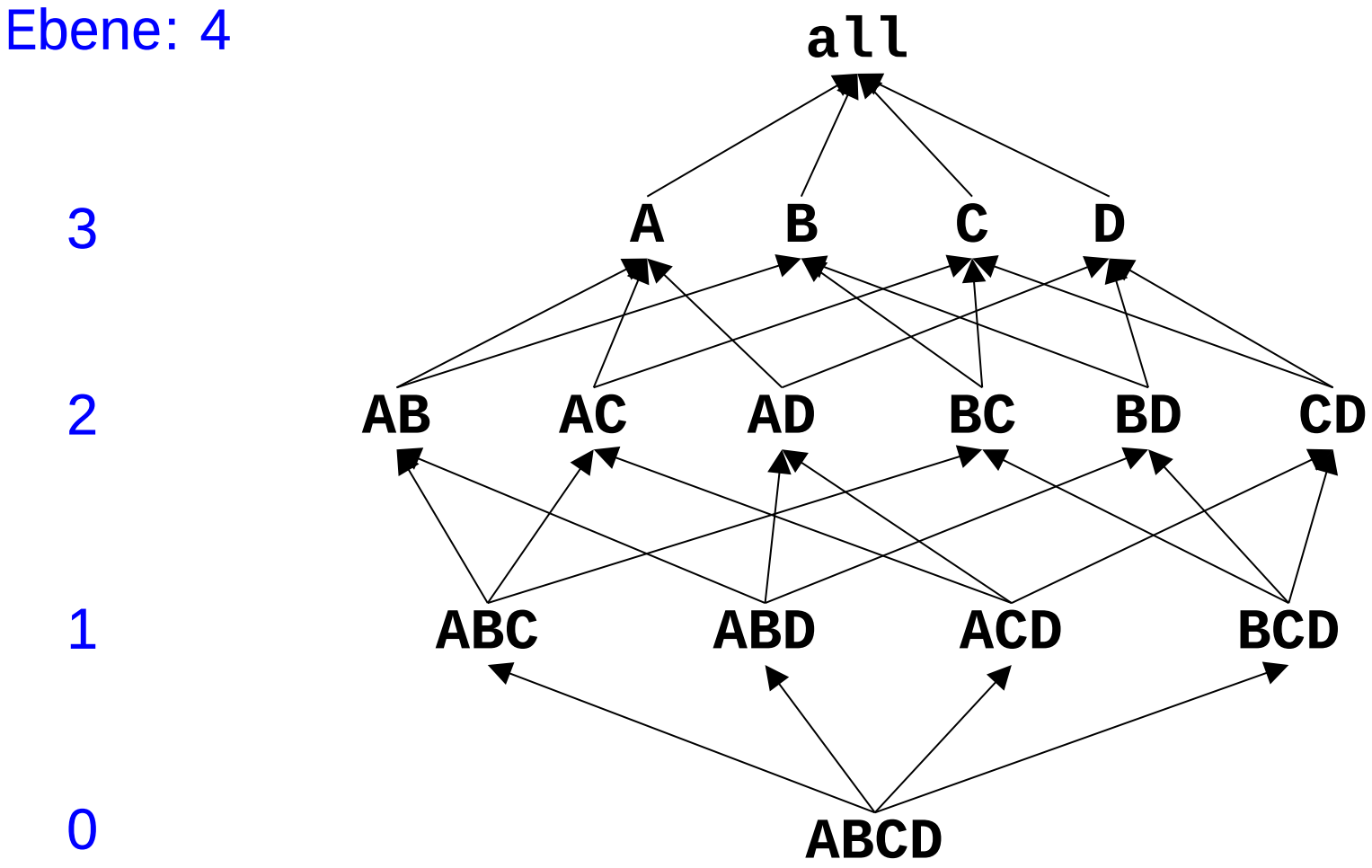
Naiver Ansatz
(auch 2^D Algorithmus genannt)

Ausnutzen von
(direkter) Ableitbarkeit

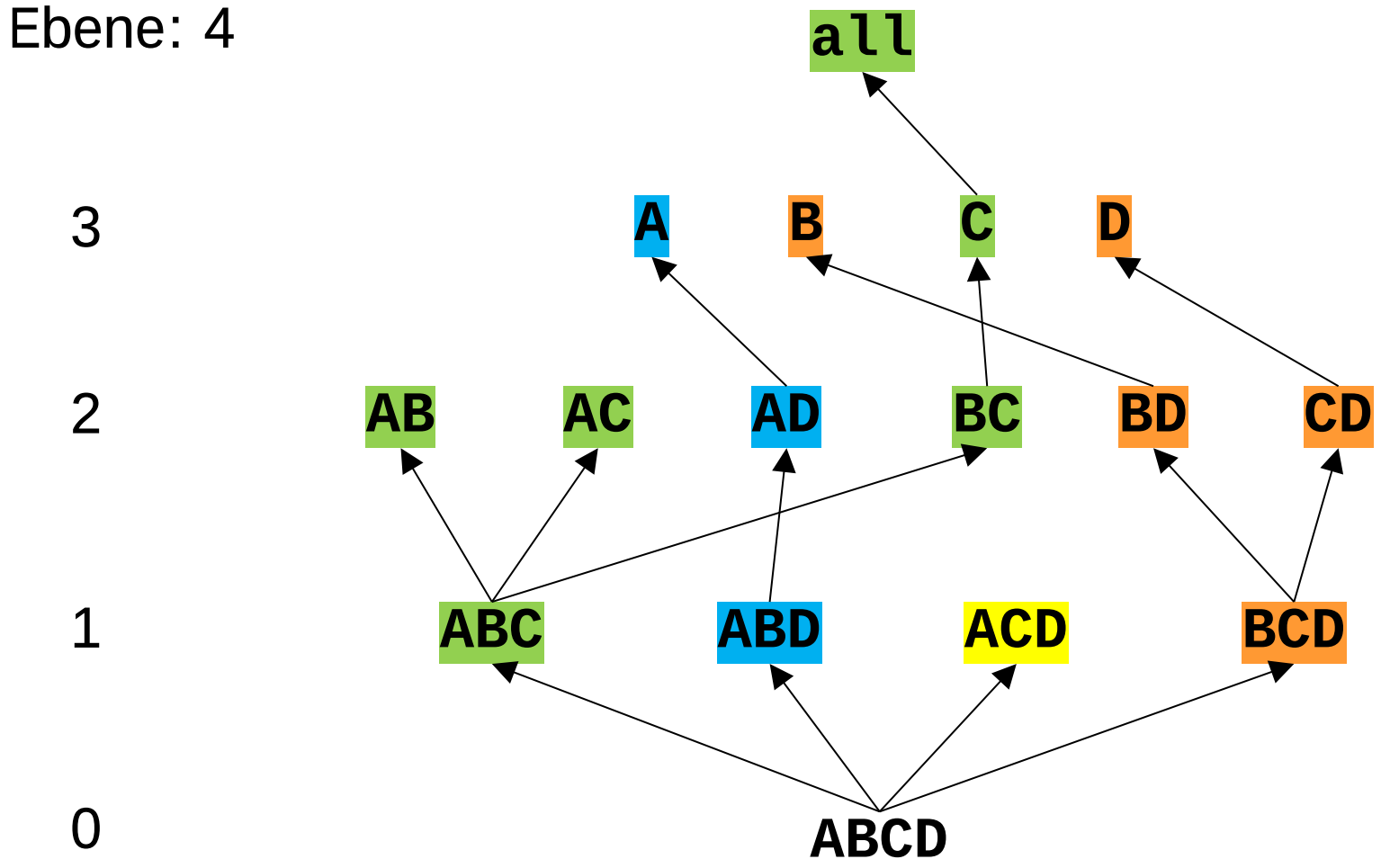
Szenario

- Definition
  Eine Gruppierung G ist aus einer Gruppierung H ableitbar, geschrieben H→G, wenn G ⊆ H. G ist direkt ableitbar aus H, wenn außerdem |G|=|H|-1.
- Vorgehen
  - Gruppierung mit höchster Auflösung wurde berechnet und liegt auf der Platte (unumgänglich)
  - In welcher Reihenfolge berechnet man am besten die weiteren Gruppierungen?
  - Sehr viele Varianten möglich
- Randbedingungen
  - Es gibt exponentiell viele Gruppierungen
  - Hauptspeicher ist beschränkt

Aggregationsgitter



Mögliche Reihenfolge



Optimale Berechnung von CUBE

- Es ergibt sich ein komplexes Optimierungsproblem
  - Begrenzter Hauptspeicher
  - Ableitbarkeit über alle Ebenen nutzbar
  - Gruppen sind von vielen Eltern ableitbar
  - Gruppierung kann über Hash oder Sort implementieren werden
  - Sortierung wiederverwenden für ableitbare Gruppen
  - Man kann Gruppierungen auch auf Partitionen berechnen
    - Damit geht vielleicht der ganze Graph in den Speicher, aber eben nicht für alle Daten
  - ...
- Notation (für die folgenden Beispiele)
  - (ABC): Gruppierung ohne festgelegte Sortierung
  - <ABC>: Gruppierung mit Sortierung nach A,B,C

Formales Problem

- Ein Gruppierungsplan ist eine Vorschrift zum Laden, Berechnen und Partitionieren einer Tabelle so, dass im Ergebnis alle verlangten Gruppierung berechnet sind
- Optimierung: Finde den Plan, der bei gegebenem Speicher in schnellster Zeit (am wenigsten IO) fertig ist
- Das Problem ist NP-schwer in der Zahl der Gruppen
- Wir brauchen Heuristiken

Heuristiken 1

1. Smallest-parent
   - Berechne Gruppierung aus dem Elternteil mit wenigsten Tupeln
   - Beispiel: (A) kann aus (AB), (AC) oder (ABC) berechnet werden
     - (ABC) hat sicher am meisten Tupel, aber |(AB)| ? |(AC)|
   - Gewinn: Weniger Rechenaufwand, eventuell können abgeschlossene Gruppierungen früher verdrängt werden
2. Cache-results
   - Berechne Ketten von ableitbaren Gruppierungen
   - Halte dazu Ergebnisse im Hauptspeicher, um IO zu sparen
   - Beispiel: (ABC) $\rightarrow$ (AB) $\rightarrow$ (B) $\rightarrow$ ()
     - Dazu muss (ABC) und (AB) in den Hauptspeicher passen
     - ... oder wir sortieren und pipelinen
   - Gewinn: Weniger IO

Heuristiken 2

3. Amortize-scans
   - Wenn schon eine Gruppierung geladen werden muss, berechne möglichst viele der Kinder gleichzeitig
   - Beispiel: Aus (ABC) berechne mit einem Scan (AB), (BC), (AC)
   - Braucht viel Platz
   - Gewinn: Weniger Scans, weniger IO
4. Share-partitions
   - Wenn eine hashbasierte Gruppierung aus Speichermangel in Partitionen erfolgt, dann benutze dieselben Partitionen für alle ableitbaren Gruppierungen
   - Berechnet mehrere GROUP BYs auf einer Menge von Partitionen
   - Beispiel: (ABC) für Werte A<10, dann dito $\rightarrow$ (AB) $\rightarrow$ (A) $\rightarrow$ ()
   - Gewinn: Weniger IO, weniger Hashen

Heuristiken 3

5. Share-sorts

- Wenn eine Gruppierung G sortiert gelesen werden kann, berechne alle Gruppierungen mit derselben Attributreihenfolge
  - Alle Gruppierungen, deren Attribute ein Präfix von G sind
- Beispiel: <ABC> $\rightarrow$ <AB> $\rightarrow$ <A>, aber nicht <ABC> $\rightarrow$ <BC>
- Gewinn: Keine zusätzlichen Sortierungen (und Gruppierung mit Sortierung war speicherplatzeffizienter als Hashen)

Inhalt dieser Vorlesung

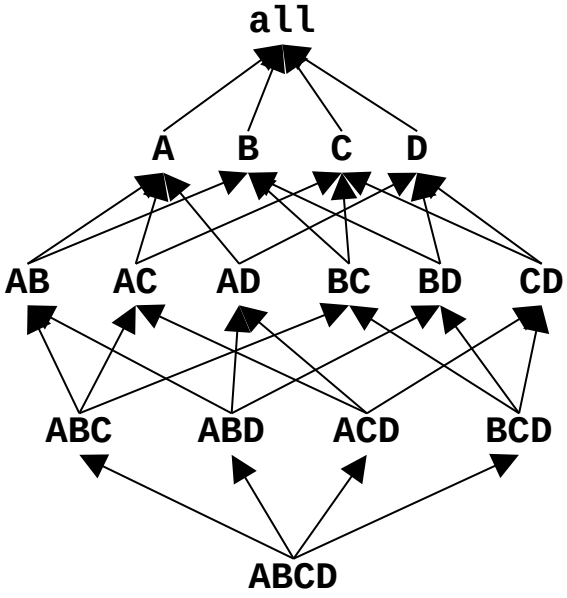
- Wiederholung: OLAP Operationen
- Implementierung der Gruppierung
- Implementierung von Cube & Iceberg-Cube
  - Problem und Potential
  - Cube Algorithmen
  - Iceberg Cubes

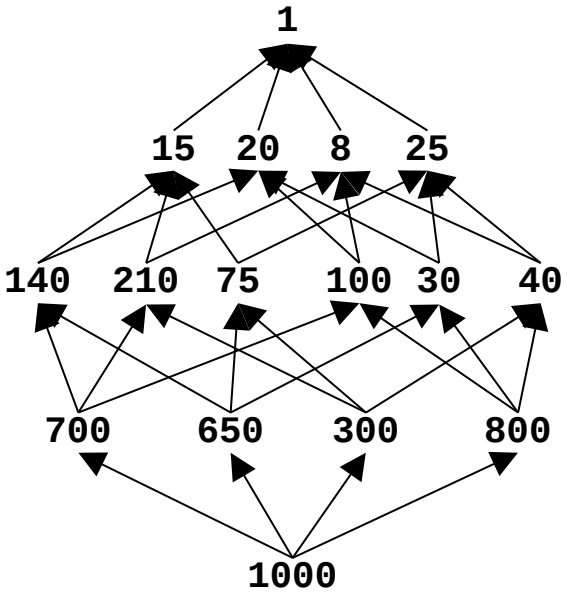
Algorithmen

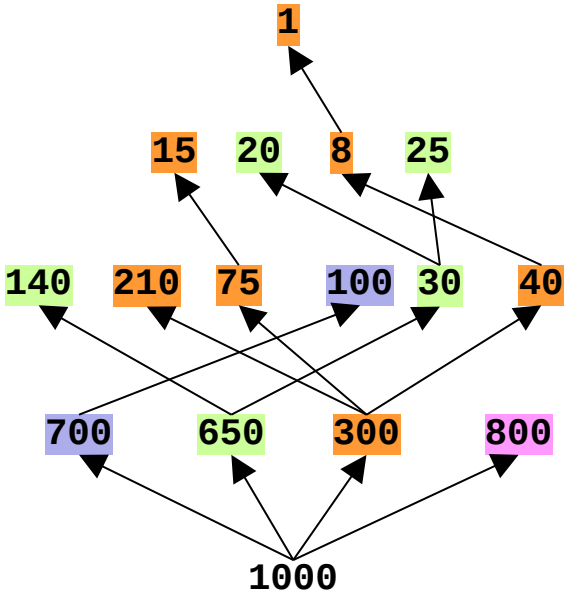
- Die verschiedenen Heuristiken stehen offensichtlich in Konflikt
  - Der kleinste Vorfahr muss nicht die gleiche Sortierung haben
  - In einem Scan alle Kinder berechnen, obwohl man nicht der kleinste Elternteil ist
  - ...
- Konkrete Verfahren konzentrieren sich auf einige der Tricks
- Wir sehen uns zwei Verfahren an: GBLP, PipeSort
  - 2^D ist ein drittes

GBLP [Gray, Bosworth, Layman, Pirahesh et al. 1997]

- Berechne Kinder immer vom kleinsten Elternknoten
  - Einfach, schnell
  - Keine Festlegung, wie konkret berechnet wird (Platz ist knapp!)
    - Sortieren oder hashen, Reihenfolge, ...
  - Benötigt Schätzungen der Kardinalitäten







PipeSort [AAD+96]

- Eingabe ist das Aggregationsgitter der Gruppierung
- Berechnet Reihenfolge und Sortierung der Gruppierungen
- Benutzt Share-Sorts und Smallest-Parent
- Benötigt Schätzungen über Kardinalität der Gruppierungen

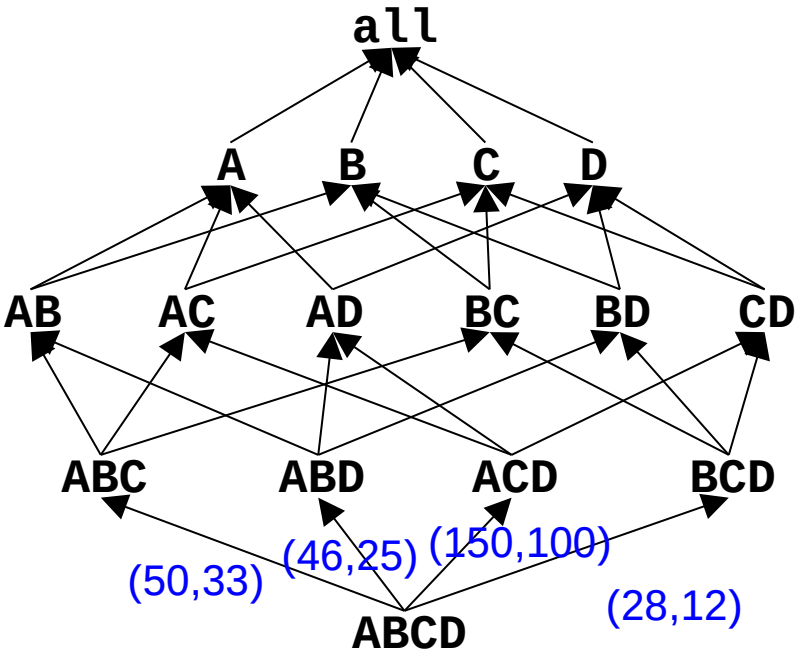
Grundidee

- Bei den Kosten der Berechnung gehen mit ein
  - Kosten für Sortierung – wenn notwendig
  - Kosten für Gruppierung – abhängig von der Größe des Elternknoten
- Kosten werden als Kantengewichte modelliert
- PipeSort geht ebenenweise durch das Gitter
  - Start bei der Gruppierung mit der höchsten Auflösung (Ebene 0)
  - Jede Gruppierung wird von einem unmittelbaren Vorfahr abgeleitet (Heuristik zur Komplexitätsreduktion)
- Für jeden Ebenenwechsel wird eine lokale Lösung durch bipartites Matching berechnet

Gewichtetes Aggregationsgitter

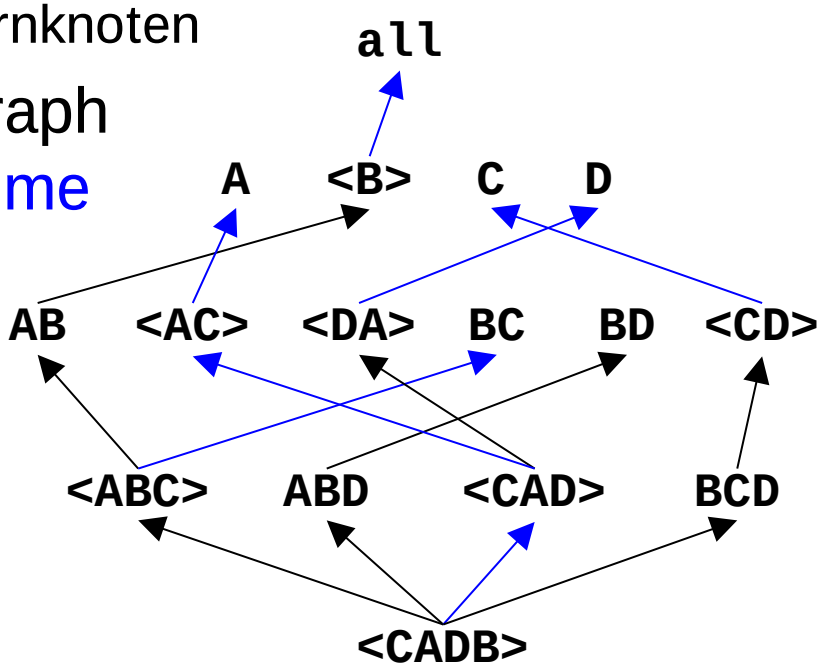
- Jede Kanten $G_i \rightarrow G_j$ hat zwei Kosten
  - S_{ij}: Berechnung von G_j aus G_i mit Umsortierung
  - A_{ij}: Berechnung von G_j aus G_i ohne Umsortierung

In den Gruppen ist (noch) keine Sortierung festgelegt



Ziel

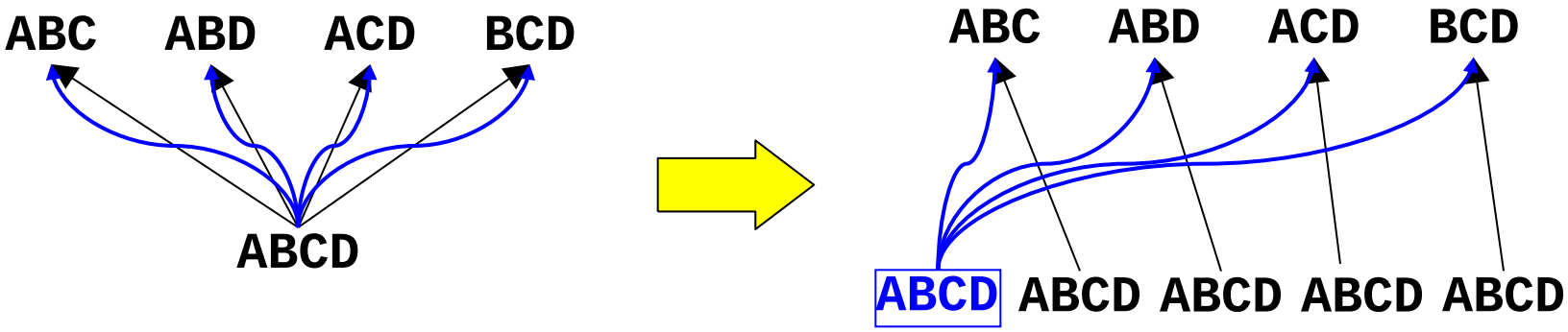
- Gegeben das Aggregationsgitter A mit doppelten Kanten
- Ein Subgraphen G von A ist erfüllbar, wenn
  - G enthält die gleiche Knotenmenge wie A
  - Jeder Knoten hat höchstens eine festgelegte Sortierung
  - Jeder Knoten hat nur eine eingehende Kante mit Gewicht passend zur Sortierung des Elternknoten
- Gesucht: Der erfüllbare Subgraph mit der kleinesten Kantensumme
  - Problem ist auch NP-schwer
- Bemerkung
  - Da nur direkte Ableitungen ausgenutzt werden, kann nur ein Nachfahre die ganze Sortierung „erben“



Ebene für Ebene

- Für jeden Ebenenübergang suche optimale Teillösung, die
  - Für jeden Kindknoten genau eine eingehende Kante festlegt
  - Die Sortierung aller Elternknoten festlegt
- Reduktion auf Bipartites Matching
  - Für jeden Elternknoten mit n ableitbaren Gruppen
    - Lege 1 Kopie an, die alle A-Kanten behält (keine Umsortierung)
    - Lege n Kopien an, die jeweils eine S-Kante behalten (Umsortierung)
  - Dies ist ein bipartiter Graph

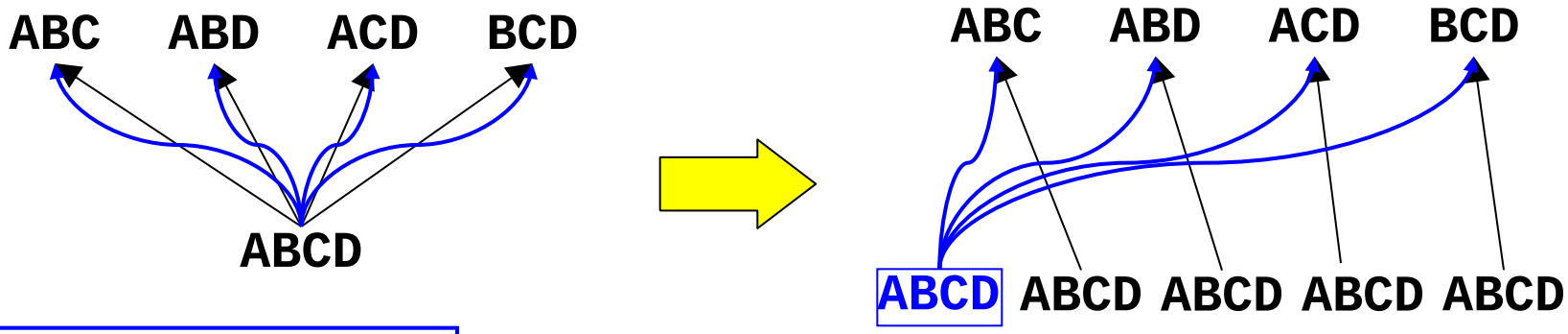
Beispiel – Level 1



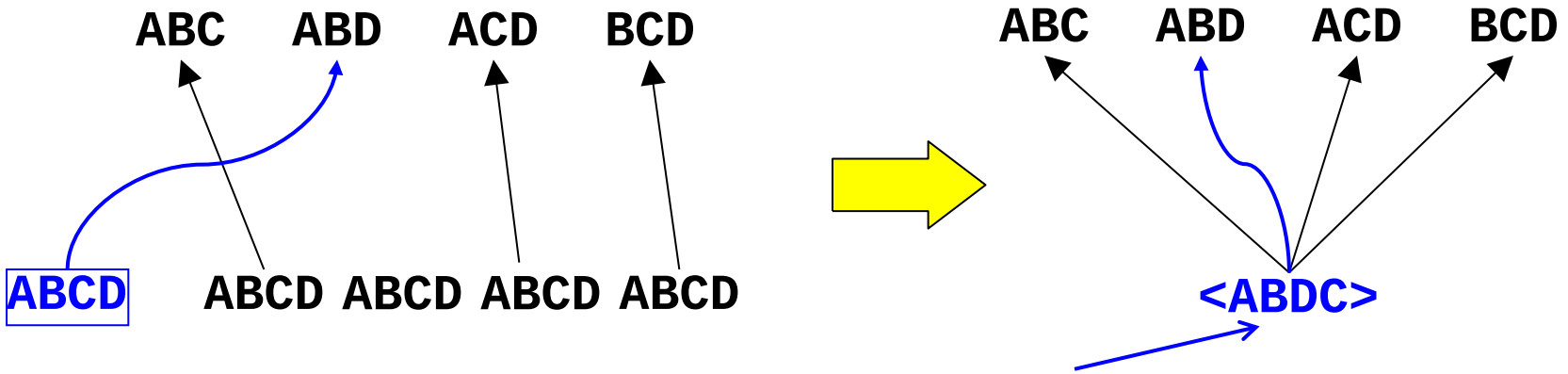
Ebene für Ebene

- Für jedes Ebenenpaar
  - Berechne optimales bipartites Matching
    - Eine maximal „leichte“ Menge von Kanten so, dass nie zwei Kanten einen Knoten gemeinsam haben und alle Kindknoten durch genau eine Kante erreicht werden
  - Verschmelze gleiche Knoten wieder
  - Lege Sortierreihenfolge der Eltern durch ausgehende A-Kante fest
    - Davon kann es nur noch eine geben

Beispiel – Level 1

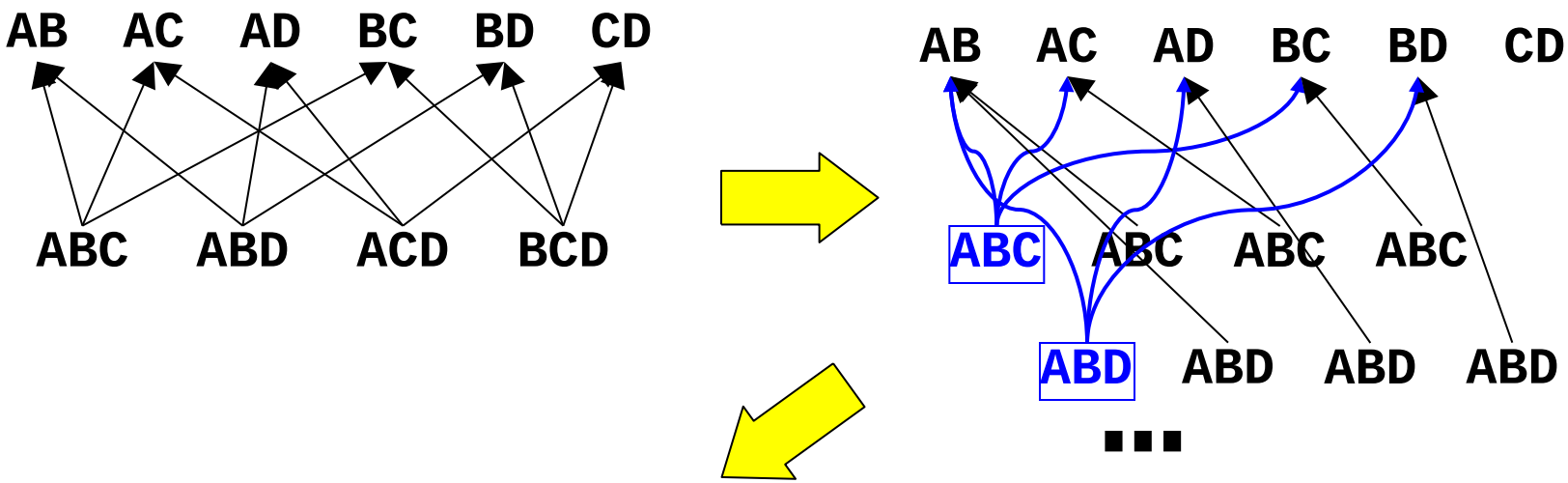


Nur eine A-Kante kann gewählt werden – Sortierung steht fest



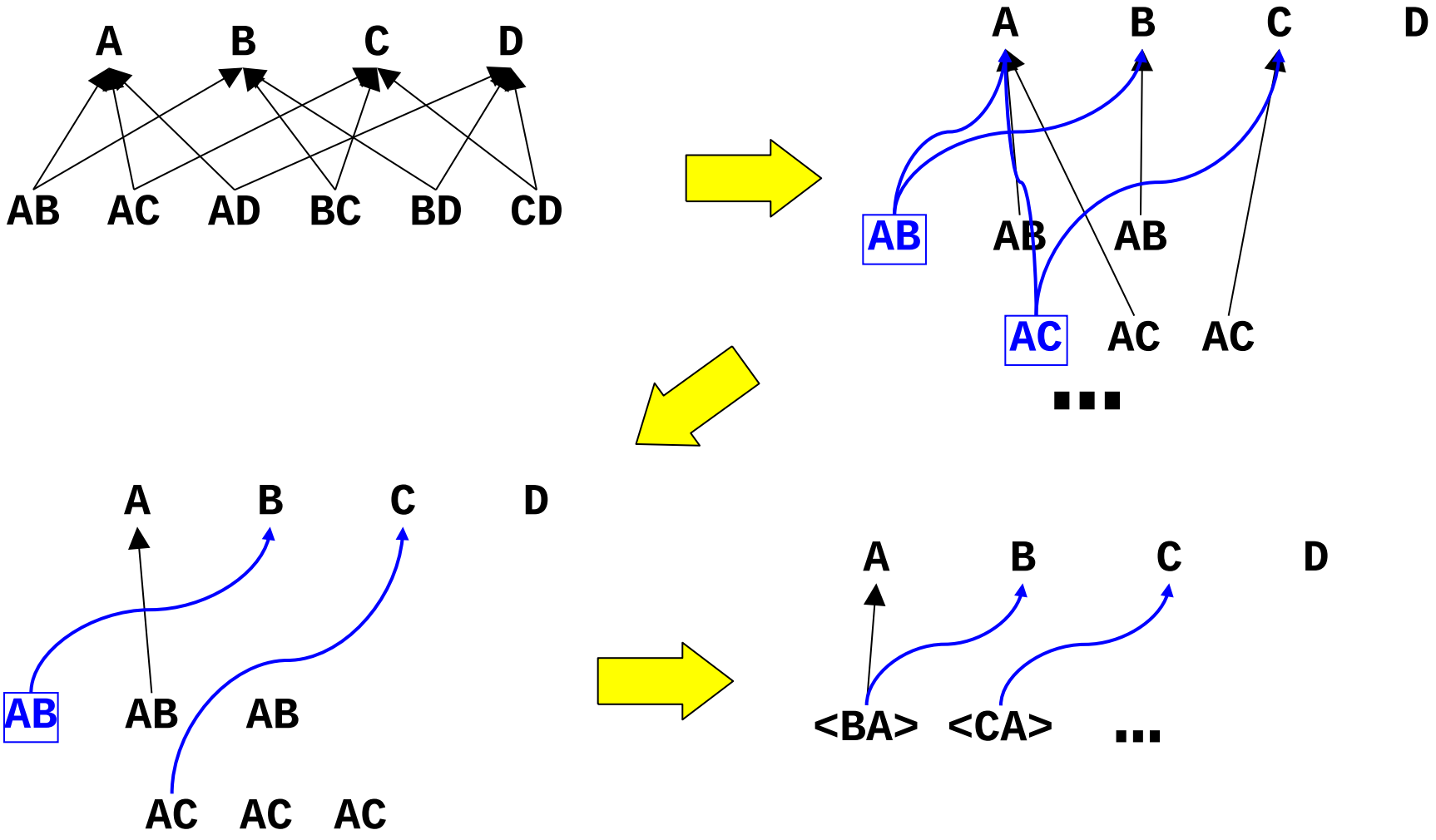
Sortierung durch A-Kante festgelegt

Beispiel – Level 2

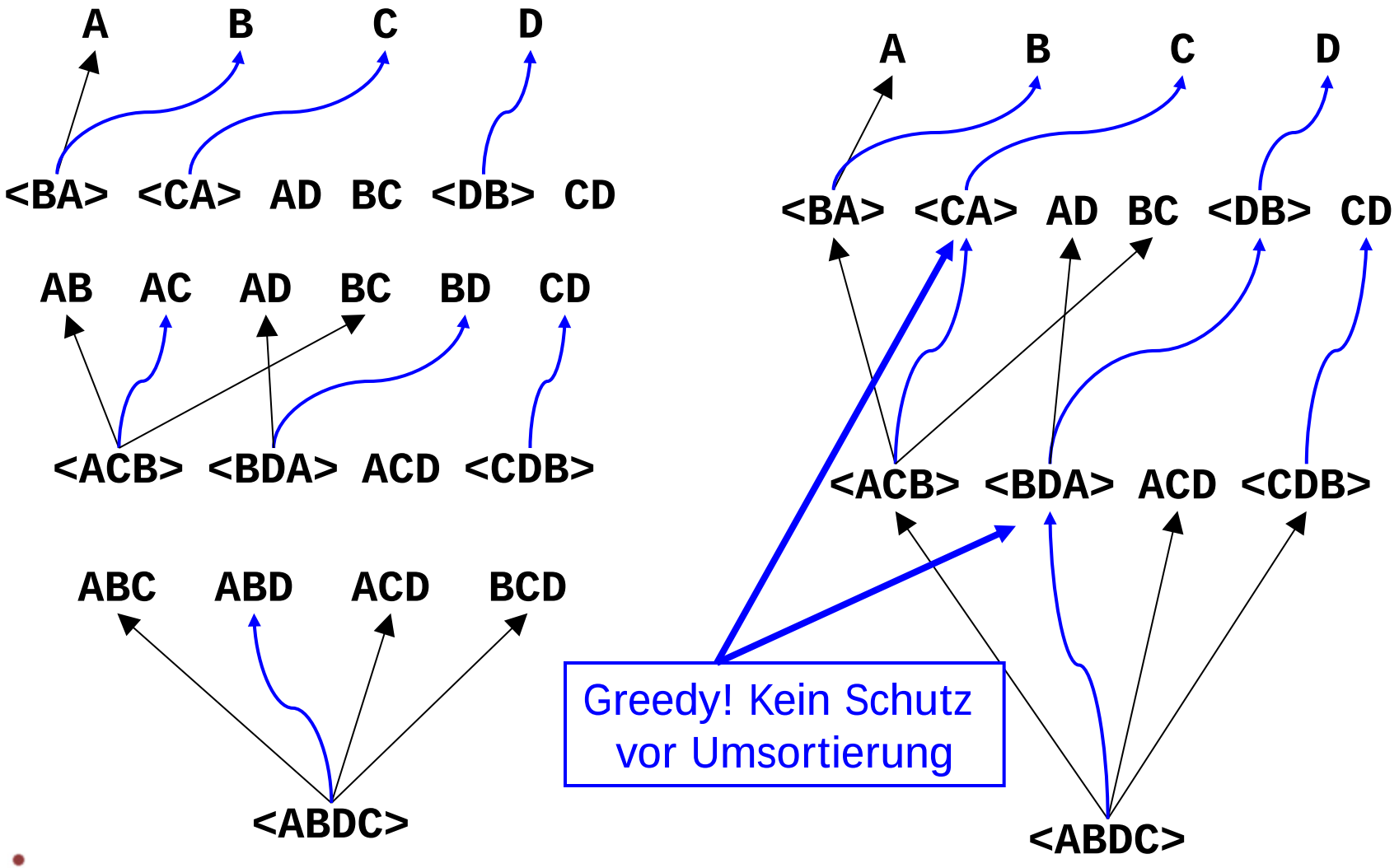


Von kleinen Eltern kann die Ableitung auch mit Umsortierung billiger sein als von großen Eltern

Level 3

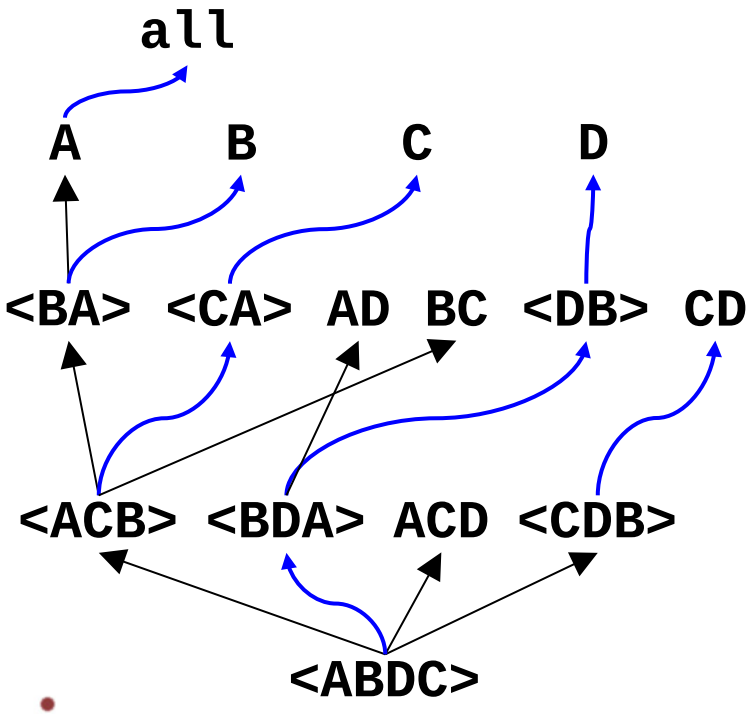


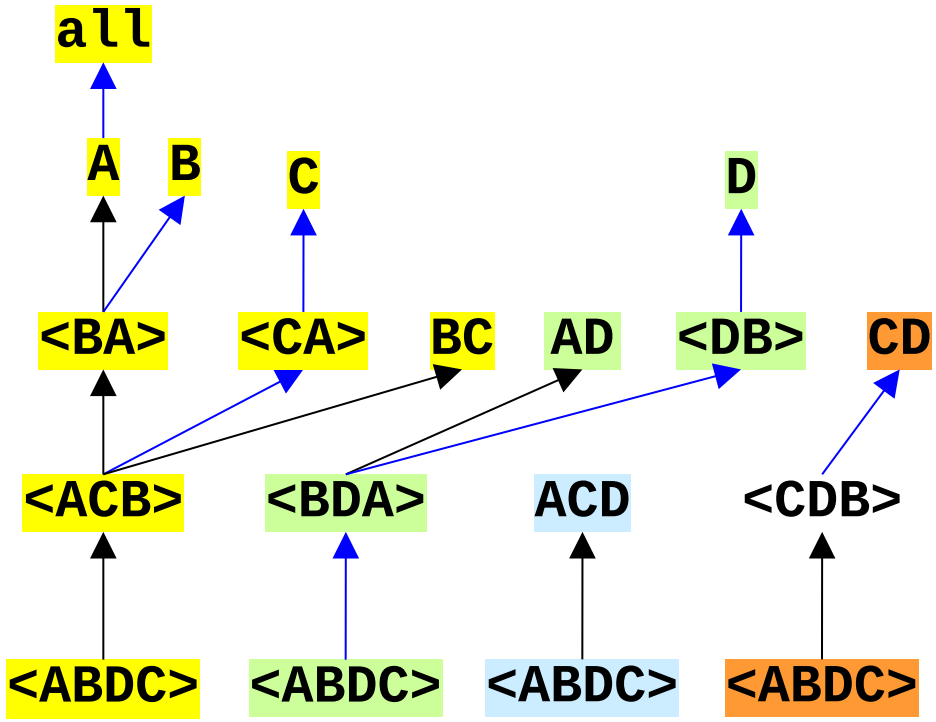
Ergebnis


A B C D
<BA> <CA> AD BC <DB> CD
AB AC AD BC BD CD
<ACB> <BDA> ACD <CDB>
ABC ABD ACD BCD
<ABDC>
A B C D
<BA> <CA> AD BC <DB> CD
<ACB> <BDA> ACD <CDB>
<ABDC>
Greedy! Kein Schutz vor Umsortierung

Tatsächliche Berechnung

- Ebenenübergänge werden bottom-up festgelegt
- Dann werden (maximal in den Speicher passende) Teilbäume in jeweils einem Lauf berechnet





Komplexität

- PipeSort löst für jeden Ebenenübergang ein bipartites Matching-Problem
  - Sei d die Zahl Attribute im CUBE Operator
  - Jede Ebene i enthält d über (d-i+1) Knoten (mal 2 plus 1)
    - Das sind im Worst-Case O(d!/(d/2)!)
  - Jeder Knoten hat (d-i+1) ausgehende Kanten
  - Bipartites Matching ist O(n*m)
    - Wenn n die Zahl Knoten und m die Zahle der Kanten ist
  - Es gibt d Ebenenübergänge
  - Zusammen: O(d* (d!/(d/2)!)*d)
  - Sehr pessimistische Abschätzung
- Effizient ist anders
  - Aber besser als alle passenden Subgraphen aufzuzählen

Bewertung PipeSort

- Viele Heuristiken, keine optimale Lösung
  - Keine Beachtung von Sortierungseffekte über mehrere Ebenen hinweg
  - Keine Garantie für die Anzahl notwendiger Sortierungen
  - ...
- Platz: Wenn ein Teilbaum nur einmal sortiert werden muss, braucht man ggf. nur ein Tupel pro Gruppe
  - Bei distributiven Aggregatfunktionen, sonst ...
  - Tupel werden sortiert durch eine Pipeline von Gruppierungen geschickt
- Viele weitere Vorschläge: PipeHash, Multiway Agg., ...

Inhalt dieser Vorlesung

- Wiederholung: OLAP Operationen
- Implementierung der Gruppierung
- Implementierung von Cube & Iceberg-Cube
  - Problem und Potential
  - Cube Algorithmen
  - Iceberg Cubes (Sketch)

Iceberg Cubes

- Immer noch: Es gibt exponentiell viele Gruppierungen in einem CUBE Operator
  - 10 Dimensionen a 10 Ausprägungen = potentiell 1024 Gruppen mit maximal 10^{10} Partitionen
- Die meisten interessieren meistens nicht
- Iceberg Cubes
  - Finde Gruppen mit Aggregaten, die größer sind als ein Schwellwert

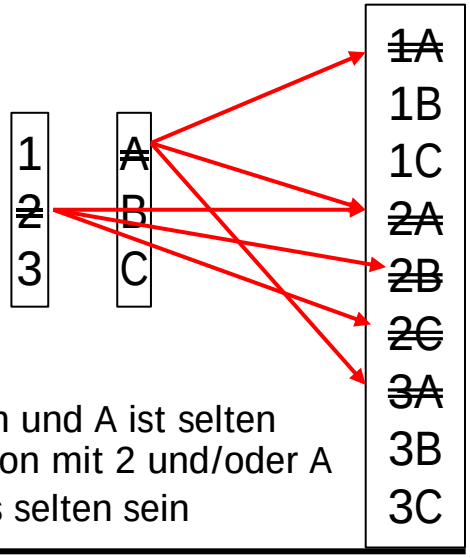
```
SELECT  product_id, day_id, shop_id, SUM(amount)
FROM    sales S
GROUP BY CUBE(product_id, day_id, shop_id)
HAVING COUNT(*) > threshold
```

Berechnung

- Naive: CUBE berechnen, „leere“ Partitionen filtern
  - Ineffizient
- Beobachtung
  - Nur distributive und monoton wachsende Aggs, z.B. CNT, SUM
  - Wenn für Partition P einer Gruppierung G und Aggregatfunktion f gilt, dass f(P)<t, dann muss für alle Partitionen P‘ jeder Gruppierung G‘ mit G⊆G‘ gelten: P‘⊆P und f(P‘)<t
    - A-Priori Eigenschaft
    - Durch die Hinzunahme weiterer Attribute werden die Partitionen kleiner (oder, selten, bleiben gleich gross)
    - Wenn schon die größere Partition den Schwellwert nicht schafft, schafft es die kleinere auch nicht
- Das kann man zum Prunen ausnutzen

Beispiel

- Gruppierungsattribute (year, product)
- Partitionen (1990,Wedding), (1991, Wedding), ...
- Wenn die Verkäufe 1991 im Wedding ingesamt unter 10.000 Euro lagen, dann liegen sie auch für jedes Produkt 1991 im Wedding unter 10.000 Euro
- Bei Berechnung von (year, shop, product) aus (year, product) kann man also Partitionen, die (1991, Wedding) enthalten, ignorieren
- Problem: Das ist „Falschrum“: von grob zu fein



2 ist selten und A ist selten
⇒ jede Partition mit 2 und/oder A muss selten sein

Konstruktion von Iceberg Cubes [BR99]

- Baut die Gruppierungen von ALL nach (ABCD)
- Iteriere über alle Dimensionen d
  - Partitioniere nach d (Gruppierung über Einzelattribute)
  - Iteriere über alle Partitionen P in d
    - Wenn COUNT(P) < t: P verwerfen
      - Das prunt implizit alle P‘ mit P⊆P‘ prunen in allen G‘ mit G‘⊆G
    - Sonst: Gib (P, COUNT(P)) aus; steige rekursiv zu weiteren Dimensionen ab (das verfeinert die Partitionen)
      - Siehe A-Prior Algorithmus später
- Beobachtung
  - Benötigt viele Scans der gesamten Datenbasis (äußere Schleife)
  - Geschickte Implementierungen „switchen“ ab einem variablen Punkt in jeder Dimension auf Hauptspeichervarianten
- Viele weitere Algorithmen

Literatur

- [AAD+96] Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J. F., Ramakrishnan, R. and Sarawagi, S. (1996). "On the Computation of Multidimensional Aggregates". 22nd Conference on Very Large Data Bases, Bombay, India. pp 506-521.
- [BR99] K. Beyer, R. Ramakrishnan (1999). „Bottom-Up Computation of Sparse and Iceberg CUBEs“, ACM SIGMOD, pp. 359-370.
- [MKIK0z] Konstantinos, M., Stratis, K., Yannis, I. and Nikolaos, K. (2007). "ROLAP implementations of the data cube." *ACM Computing Surveys 39(4): 12.*

Selbsttest

- Welche Strategien gibt es für die Implementierung eines GROUP-BY?
- Was kann man tun, wenn schon die größte Partition bei einer holistischen Aggregatfunktion nicht in den Speicher passt?
- Wie viel Platz braucht man bei folgenden Daten ... und der Aggregatsfunktion SUM mindestens, um 100M Tupel zu gruppieren?
- Was macht der 2^D Algorithmus?
- Wie funktioniert PipeSort?
- Was ist ein Ice-Berg Cube?
- Wie kann man dabei die A-Prior-Eigenschaft ausnutzen?